Algorithms for Programming Contests

Kevin Kauffman
Duke University

February 8, 2025

Abstract

learn to do programming!

Contents

1 Introduction **8**
1.1 Why Did I Write this Book? . . . 8
1.2 Who is this Book For? . . . 9
1.3 Why Do Competitive Programming? . . . 10
1.4 How to Solve Problems . . . 11
1.4.1 Using Others' Solutions . . . 13
1.5 Which Language Should You Use? . . . 15
1.6 What Other Resources Are There? . . . 15

2 Basic Techniques **17**
2.1 Runtime Estimation . . . 17
2.2 Native Types . . . 19
2.2.1 Converting Between Types . . . 19
2.2.2 Literals . . . 20
2.3 Basic Structures . . . 21
2.3.1 Static Arrays . . . 21
2.3.2 Dynamic Arrays . . . 22
2.3.3 Lists . . . 23
2.3.3.1 Queues . . . 23
2.3.3.2 Stacks . . . 24
2.3.4 Hashed Structures . . . 24
2.3.4.1 Hash Sets . . . 24
2.3.4.2 Hash Maps . . . 25
2.3.5 Ordered Structures . . . 26
2.3.5.1 Tree Sets . . . 26
2.3.5.2 Tree Maps . . . 28
2.3.5.3 Heaps . . . 28
2.3.6 Tuples . . . 29
2.4 IO . . . 30
2.4.1 C++ . . . 30
2.4.1.1 Input . . . 30
2.4.1.2 Output . . . 32
2.4.1.3 Problem Template . . . 33
2.4.2 Java . . . 35

2.4.2.1 Input 36
2.4.2.2 Output 37
2.4.2.3 Problem Template 37
2.4.2.4 Fast Input 38
2.5 Language Tools 39
2.5.1 Custom Comparation 39
2.5.2 C++ Specific Tools 40
2.5.2.1 Iterators 40
2.5.2.2 Next Permutation 41
2.5.2.3 String Access 42
2.5.2.4 Multi Maps 42
2.5.3 Java Specific Tools 42
2.5.3.1 BigInteger and BigDecimal 42
2.5.3.2 Geometry Package 43
2.5.3.3 Fancy Output Formatting 43
2.6 Basic Techniques 44
2.6.1 Brute Force Search 44
2.6.1.1 Fixing a Variable 45
2.6.2 Simulation 46
2.6.3 Recursion and Backtracking 46
2.6.4 Bit manipulation 48
2.6.4.1 Basic Operators 48
2.6.4.2 Mask and Shift 49
2.6.4.3 Other Hacks 50
2.6.4.4 Cumulative Bit Count 50

3 Math and Geometry **51**
3.1 Binary Search 51
3.2 Ternary Search 51
3.3 Catalan Numbers 51
3.4 GCD LCM and Euclid 51
3.5 Probability 52
3.5.1 combinations 52
3.5.2 permutations 52
3.5.3 stars and bars 52
3.6 Primality and Factoring 53
3.7 Convex Hull 53
3.8 cross product 53
3.9 Center of Gravity and Balances 54
3.10 Convolution and FFT 54
3.11 Coordinate Transformation 54
3.12 Matrices 54
3.12.1 Fast Exponentiation 54
3.13 rational math 57

4 Graphs **58**
4.1 Types and Terminology 58
4.1.1 Graph Classification 58
4.1.1.1 Directionality 59
4.1.1.2 Weighted-ness 59
4.1.1.3 Cycles 60
4.1.1.4 Density 61
4.1.1.5 Connectivity 62
4.1.1.6 Bipartite 62
4.2 Representation 63
4.2.1 Adjacency Matrix 63
4.2.2 Adjacency List 64
4.2.2.1 Multiple Edges 65
4.2.3 Implied Graphs 66
4.2.3.1 ASCII Grids 68
4.2.4 Linked Data Structures 68
4.2.4.1 Binary Trees 69
4.2.4.2 General Trees 70
4.2.5 Advanced Representations 72
4.2.5.1 Map-less Adjacency Lists 72
4.2.5.2 Compressed Adjacency Matrices 72
4.3 Algorithms 73
4.3.1 Search 73
4.3.1.1 DFS 73
4.3.1.2 BFS 76
4.3.1.3 Dijkstra 86
4.3.1.4 MST 104
4.3.1.5 Other Graph Search Topics 108
4.3.2 Flow and Match 112
4.3.2.1 Maximum Flow 112
4.3.2.2 Faster Max-Flow Implementations 121
4.3.2.3 Min-Cut 159
4.3.2.4 Bipartite Matching 160
4.3.2.5 Extended Matching: Using Flows to Solve Problems 165
4.3.2.6 In-Out Nodes 168
4.3.2.7 Node Disjoint Path Cover 170
4.3.2.8 Min-Cost-Max-Flow 174
4.3.2.9 Hungarian Matching 174
4.3.3 Componentization 175
4.3.3.1 SCC 175
4.3.3.2 BCC 197
4.3.3.3 2SAT 204
4.4 State Explosion 212
4.5 FSMs 212
4.6 DAGs 212

4.6.1 Topological Sort 212
4.7 Trees 212
4.7.1 HLD 212
4.7.2 Binary Lifting 212
4.7.3 Lowest Common Ancestor 214
4.7.3.1 Standard Algorithm 214
4.7.3.2 Unbalanced Trees 215
4.7.3.3 Constant Time 216
4.8 counting loops 216
4.9 Union-Find 216

5 Greedy Algorithms 218

6 Dynamic Programming 219
6.1 Fibonacci Numbers 219
6.1.1 Recursive Computation 219
6.1.2 Saving work with Memoization 220
6.1.2.1 Runtime 221
6.1.3 Eliminating the Stack 221
6.2 Building Blocks 222
6.3 Multiple Dimensions 223
6.3.0.1 Runtime 225
6.3.1 Benefits of Recursion 225
6.4 Standard DP Problems 226
6.4.1 Knapsack 226
6.4.1.1 As Stated: Greedy 226
6.4.2 Inequal Serving Amounts 226
6.4.2.1 Item Limitations 230
6.4.2.2 All or Nothing 237
6.4.2.3 Another Example 238
6.4.3 Forward-Looking Iteration 240
6.4.4 Travelling Salesman 241
6.4.5 Prefix Sum 244
6.4.6 Inclusion/Exclusion 244
6.4.7 Mini-Max 246
6.4.7.1 Simplifying 248
6.4.7.2 Simple Game Victor 248
6.4.8 Expected Value 250
6.4.8.1 Application to other DP types 251
6.4.9 Grid Tiling 252
6.5 Optimizations 256
6.5.1 Memory Reduction 256
6.5.2 Dimension Swapping 257
6.5.3 Interesting Points 258
6.5.4 Dimension Elimination 259
6.5.5 Secondary DP 260

6.5.6 Restricted Search 260
6.5.7 Divide and Conquer Optimization 260
6.5.8 Knuth's Optimization 260
6.5.9 Convex-Hull Optimization 260
6.6 Identifying DP 261

7 NP Completeness 262

8 Data Structures 263
8.1 segment trees 263
8.2 interval trees 263
8.3 Fenwick Trees 263
8.3.1 Understanding the Structure 263
8.3.1.1 Building a Sum 265
8.3.1.2 Updating a value 267
8.3.1.3 Building the tree 268
8.3.2 Implementation 268
8.3.2.1 Bithacks 270
8.4 Trie 271

9 Strings 272
9.1 Pattern Matching 272
9.1.1 Z Function 272
9.1.1.1 Applications 272
9.1.1.2 Implementation 273
9.1.2 Knuth-Morris-Pratt 275
9.1.2.1 Intuition 275
9.1.2.2 Preprocessing: Defining the Overlap Function . 277
9.1.2.3 Preprocessing: Computing the Overlap Function 279
9.1.2.4 Alignment 283
9.1.2.5 Implementation 284
9.1.3 Multi-pattern Matching/Aho-Corasick 285
9.1.3.1 Single-Pass Matching 285
9.1.3.2 Preprocessing: Defining Overlap Edges 290
9.1.3.3 Preprocessing: Computing Overlap Edges 294
9.1.3.4 Alignment 295
9.1.3.5 Corner Case: Patterns Containing Other Patterns 310
9.1.3.6 Implementation 313
9.2 Palindromic Substrings 316
9.2.1 Manacher's Algorithm 317
9.2.1.1 Example 318
9.2.1.2 Implementation 321
9.3 Regexes 323
9.4 Suffix Trees 326
9.4.1 Building Suffix Trees 326
9.4.1.1 The Naive Tree 326

9.4.1.2 Optimization 1: Compression 328
9.4.1.3 Optimization 2: Indexing 331
9.4.1.4 Optimization 3: Per character, not per suffix . . 332
9.4.1.5 Optimization 4: Open Ended Substrings 334
9.4.1.6 Optimization 5: Non-Committal Suffixes 335
9.4.1.7 Optimization 6: Suffix Links 344
9.4.1.8 Optimization 7: Fast Traversal 350
9.4.2 Rough Proof of Linearity 351
9.4.3 Complete Example 351
9.4.4 Implementation 362
9.4.4.1 Helper Functions 366
9.4.5 Usage 368
9.4.5.1 Pattern Existence 368
9.4.5.2 Finding All Instance of a Pattern 369
9.4.5.3 Longest Repeated Substring 371
9.4.5.4 Longest Common Substring 371
9.4.5.5 Longest Common Extension 373
9.4.5.6 Longest Palindromic Substring 373
9.4.5.7 Minimum Rotation 375
9.4.5.8 Counting Unique Substrings 375
9.4.6 Suffix and LCP Array 377
9.4.6.1 Construction 378
9.4.6.2 Complete Example 386
9.4.6.3 Implementation 389
9.4.6.4 LCP Array Construction 392
9.4.7 Usage 394
9.4.7.1 Finding All Instances of a Pattern 394
9.4.7.2 Longest Repeated Substring 395
9.4.7.3 Counting Unique Substrings 395
9.5 Hashing 396
9.5.1 Hashing Theory 396
9.5.1.1 The String Hash Function 397
9.5.2 Implementation 398
9.5.2.1 Selection of Parameters 398
9.5.2.2 The Birthay Paradox 399
9.5.2.3 Code 399
9.5.2.4 Substring Extraction 399
9.5.2.5 Hashing Hashes 401
9.5.3 Algorithms 401
9.5.3.1 String Comparison 401
9.5.3.2 Pattern Existence 401
9.5.3.3 Finding All Instances of a Pattern 402
9.5.3.4 Longest Repeated Substring 402
9.5.3.5 Longest Common Substring 402
9.5.3.6 Longest Common Extension 402
9.5.3.7 Longest Palindromic Substring 402

9.5.3.8 Minimum Rotation 403
9.6 Finding Minimum Rotations 404
9.6.0.1 The Algorithm 406
9.6.0.2 Implementation 407

10 Other Common Techniques 409
10.1 Interesting Points 409
10.2 non-constructive indexing 409
10.3 String Search 409
10.3.1 Regular Expressions 409
10.4 Caterpillaring 409
10.5 precomputation 409
10.6 scangrid and quick-updates 409

Chapter 1

Introduction

1.1 Why Did I Write this Book?

The compendium of literature written about algorithms is not small. It runs the gamut including white papers, text books, blogs, and Stack Overflow answers. I have read many of these, some which didn't exist when I started competitive programming, and some which were written decades before I was born.

I have always felt, though, that most of these resources left something wanting. Not wanting in the sense that they don't accomplish what they intend to for their chosen audience, but that their message never fully aligned with what I felt I needed for competitive programming. Resources might be very focused on formal proofs, lacking in explanations of implementation details, not providing intuitive explanations of why things work, lacking in discussion of common extensions to the presented algorithms, and generally not providing enough context to take a programmer from a beginner to a place where they would understand and be comfortable using an algorithm or technique in a contest.

Most importantly, the materials are not in one place and thus discoverability is low. You go one place to learn about Dijkstra's algorithm, another to learn about Maxflow. You find that everyone has an opinion on the proper way to set up a DP, but the closest thing to a complete explanation of Ukkonen's algorithm is on Stack Overflow. The lack of a coherent and complete resource is one of the more difficult parts of learning competitive programming. The amount of resource has increased greatly, but the coherency has not.

Ultimately, then, I wrote this book to serve as that source. A resource that enables a reader to go from knowing almost nothing on a topic to being able to not just use it, but understand it. It covers topics ranging from solving ones first problem up through some of the most advanced techniques used in the most challenging problems. It covers intuition behind algorithms as well as the math behind them. It mixes discussion of algorithms with practicalities of their implementation. But most of all, it serves as a resource for competitive programmers of all levels.

1.2 Who is this Book For?

This book covers a wide gamut of topics from intro level to advanced. As such, it is for competitive programmers of all abilities. While there is content for readers of all levels, it does not mean that every topic in the book will be approachable by every reader immediately. Some topics require practice and mastery of less advanced techniques to free the mental load to approach the more difficult. So while each topic is sought to be delivered in as clear an unassuming way as possible, a reader might not be ready just yet, and over time and further experience, may become so.

This book is also for any algorithmically inquisitive coder. A question I often see of advanced undergraduate students is "what next?" after an undergraduate, or even graduate, level algorithms class. Many techniques in this book are covered rarely, or at best inconsistently, in algorithms classes. While this book should not be considered a substitute for a class, it can certainly serve as a primer - an introduction to what else is out there.

This book is also for those who wish to implement algorithms, rather than just be concerned about their usage and runtime behavior. Practical implementation is something that is often given limited treatment in classes and other texts, but as a core requirement for competitive programming, is something that is covered heavily here. While not every algorithm is given in code, hopefully enough context is given that translating the algorithm to code is not a limitation.

This book is also for modern tech industry interview candidates. Securing positions today typically involves coding interviews[1] which require the candidate to quickly and clearly determine efficient algorithms and translate those algorithms into code. These skills align very much with those required for success at competitive programming, and as such, successful competitive programmers will rarely find themselves in difficulty in such interviews.

This book is also for students who may find themselves struggling to grasp algorithms and techniques as presented elsewhere. When learning new and challenging material, as many are when they start their computer science journey, seeing material presented in different ways can often help the approachability of the topic. With the massive growth of interest in the field, instructors may not always have the time to reach each student individually. So this book is for the student who might not have found the resources to understand the material elsewhere.

Lastly, this book is for me. I have taught most, if not all, of the topics in this book. As such, the book serves as a record of what and how I have taught, so that when a topic is taught again, I can ensure that I am delivering the content in a way that I found best resonated with past students. It has also been a

[1]ironically, much to the ire of this author

forcing function to think about techniques I have not yet taught, and further, how I might approach them with students in the future.

1.3 Why Do Competitive Programming?

While every student has different goals for their academic career, the greedy strategy would be to spend available time on activities which have the best chance of furthering those goals. As competitive programming won't get you a job, or into grad school, or even be particularly useful outside of the domain of competitive programming, why would anyone do it at all?

It is undoubtedly true that there are many soft benefits of competitive programming; that is not a particularly convincing justification as it ignores the opportunity cost of time. The time can be spent on any number of other activites, many of which would also provide those soft benefits. So ultimately, the only justification for using competitive programming to realize those benefits is because one wants to. Given that disclaimer,[2] competitive programming is valuable in the following ways:

1. Breaking into industry is not about having the necessary skills to complete the job, and most students should have built those skills over their student career. When there are dozens if not hundreds of applications for open positions, it is initially about standing out among peers. Every single resume will include similar class projects, so after the 50th consecutive resume which lists a "TCP Chat Client" under relevant projects, a resume which lists "ICPC regional medalist" is going to be unique. As noted above, there may be many ways to accomplish this[3], but to have the best shot to advance as a candidate, students should find their own way to go above and beyond the average.

2. As noted earlier, the skills required for competitive programming are very well aligned with the skills required to pass the coding interviews ubiquitous in industry today. Even for smart candidates who are algorithmically aware, it is imperative that getting the code on paper not be a barrier to demonstrating that expertise. The coding interview presents a coding paradigm which is different from typical production coding: small single-function programs which must be written quickly vs large well organized codebases which must be written carefully. Competitive programming matches the former paradigm and puts the a candidate in their comfort zone in an interview room instead of out of it.

3. Competitive programming teaches the value of thinking about hard problems. Solving novel problems in competitive programming requries active thinking. Without the ability to just ask someone the answer, you have

[2]and the fact that as a reader of this text, you likely already are interested in competitve programming

[3]hackathons, research, independent projects, club leadership

to find different mental techniques to break the problem down, to not be scared that a avenue of investigation for solving the problem might prove unfruitful. From a high level, it teaches you to not give up when your first thought is "I don't know how to solve this." It is a way of thinking that is an asset for any who aspire to solve difficult real world problems.

4. There's nothing like seeing the response of "accepted" after working on a challenging problem in a high-pressure contest setting.

1.4 How to Solve Problems

The steps to solving a problem seem obvious if you already know the algorithm, so the topic should be posed as "how to solve problems you don't already know the answer to." This may sound like a silly distinction, but for many, is one of the biggest barriers to continuing to think about a problem instead of "giving up" and accepting defeat. While looking up the answer is a viable strategy in practice,[4] it does not enable the kind of thinking required in a competition setting. So perhaps most correctly, this section could be titled "how to not give up when you don't know how to solve a problem." There is no deterministic way to solve an unknown problem, but there are several strategies one can employ to make progress on a problem you might otherwise give up on.

Active Thinking As insinuated, one of the most common responses when approached with a problem that one can't find a solution to immediately is to give up. Unfortunately, a common response to "What have you tried?" is "I don't know."[5]

In order to improve at problem solving, one can't simply casually think about a problem for a short time, but must actively attack the problem from as many angles as possible. In general, one should really think about a problem for at least a day while making no progress before declaring failure. Not five minutes, and not remembering the problem a day later and realizing no progress has been made, but actively trying new approaches to see if progress can be made. This doesn't mean you can't do anything else in that time, but that thinking about the problem should be on top of mind. As for what to do in that time, consider the other techniques discussed below.

After a day, if no progress has been made, one can then consider asking questions or looking for resources concerning a solution.

Assume the Solution Most problems can be solved by some combination of the techniques and algorithms described in this book. The challenge is knowing which ones those are and how to apply them in a novel situation. In that vein, a common failure mode is dismissing a particular technique, assuming it will not work, before fully investigating it. For instance, a problem is not immediately

[4]and how to do so fruitfully is discussed below

[5]the "I've tried nothing and I'm all out of ideas!" approach

obviously solved by, say, max flow, so the technique is dismissed. However, if more thought were given to how max flow could be applied, a solution arises. The goal, then, is to find ways to avoid the mental bias which arises when techniques are dismissed prejudicially.

In order to avoid this bias, we can assume a solution. In this method, instead of trying to guess which algorithm might apply, we change our mindset slightly and ask:

> If we assume the solution to this problem uses technique X, how would work?

This question then serves as a forcing function for thinking how to apply that technique X, disallowing premature dismissal. If we think about how to apply each technique earnestly, and the correct technique is among the set that we consider, then we have a better chance of finding it.[6]

Therefore, the overall method would be:

1. Develop a list of techniques which you might apply to unknown problems.

2. For each technique in your list, assume that the problem uses that technique, and try to think through all the ways you could apply that technique to the problem.

3. Either the technique will result in a solution or it will prove unfrtuitful.[7]

It takes time to think through why techniques end up failing, but it is a structured way to actively think about a problem. Over time, a solver will gain more experience and be more quickly able to tailor their thinking to techniques which are more likely to prove fruitful.

It should be remembered that thinking through an incorrect solution is not wasted time. It accomplishes multiple things:

- In true Edison fashion, it has limited your solution searh space by one technique.

- It has revealed some truth or structure aboaut the problem itself that makes the given technique incorrect.

- It reveals some properties about the technique that lead to a better understanding of what situations that technique should or should not be applied.

Solve Easier Versions of the Problem Many problems contain several complications which when taken in totality, seem to make a problem unapproachable. In such cases, how can we make progress actively thinking about

[6]From a high level, this is the same technique as "fixing a variable" in order to optimize brute force solutions

[7]hopefully an obvious disjunction

a problem when we try to solve all the complication at once? A common technique is to simply ignore it. If we start by solving an easier, more approachable version of the problem, we often find by the time we're done that we can augment the solution to accommodate the complication. Sometimes we can solve two different easier versions of a given problem and then realize we can combine those solutions to solve the more compex version.

In either case, this is another technique that allows us to actively think about and solve problems that we initialiy might not know how to.

Think of Similar Problems As a solver becomes more experienced, they will have seen hundreds, if not thousands, of problems. Certain patterns emerge such that recalling similar problems from the past may lead to intuition that was used in the previous problem and which may be helpful now. This technique has a secondary benefit that by recalling previously solved problems regularly, it ensures the techniques used in those cases are not forgotten.[8]

To enable this, it makes sense to actively identify problems which used techniques novel to the solver to be mentally "filed away" for future applicability.

Look for Unused Context or Structure As discussed in the next chapter, one of the common techniques is to identify a brute force to every problem. In cases where brute force is not viable, it implies that there must be some additional structure to the data or the problem which is being unused. Often times this structure is given in the problem. Examples:

- The input is sorted
- We are guaranteed to only have a certain number of a particular elements
- Certain factors obey a particular function (one piece of data is always quadratic in another, say)

There are many ways in which a problem can have structure, so a valid question to ask onesself while solving is whether every piece of structure the problem provides is being used. In general, if the problem indicates some peculiar restriction, that restriction is a critical requirement to apply the intended algorithm. If the problem specifies a restriction, we should be sure we are taking advantage of it.

1.4.1 Using Others' Solutions

While learning to think about difficult problems is a critical skill, there will always be situations which were not solved even after an amount of time of active thinking. In such situations, it makes sense to leverage solution descriptions and written solutions. If not done carefully, though, doing so runs the risk of not getting the most forward looking benefit as possible.

[8] effectively forced spaced repetition

Knowing or not knowing, how to solve a problem is not really a binary state. It is a continuum of gradually increased understanding. We can break down the ”stages of problem understanding” as follows.

1. I don’t understand what the problem is asking
2. I udnerstand what the problem is asking, but not how to solve it
3. I have seen code for a solution, but I don’t understand it
4. I understand what solution code is doing, but I don’t understand why it works
5. I udnerstand why a solution works
6. I can apply the solution when presented with a new problem which uses a similar technique

Many solvers stop before they reach the last stage, which means unless the novel problem is substantially similar, the solver will not be any better off from having read the solution. The goal should always be to reach further levels of this continuum. While what is necessary to reach each successive stage varies from person to person and problem to problem,[9] there are a few helpful tips:

- Don’t progress further if you don’t know what a problem is asking. Spend time going through the problem statement to understand the terms used and what they mean. Manually walk through the example test cases to understand why they are correct. If you don’t understand simple cases, you cannot solve complex ones. There is always a chance that this problem is to difficult right now for your ability.[10]
- When presented with a solution description, or especially a coded solution, it may be the case that it is difficult to understand exactly what it is doing. Competition code is, of course, written for speed and not clarity. Some common ways to bridge that gap are to walk through the code, see if you can find a solution with comments indicating what it is doing, look for recognizable function names,[11] and even step through the provided code in a debugger to understand how the data is being transformed over the course of the execution.
- Knowing what a solution does is not enough, one must know why it solves the problem. If I know that *Phil’s Magic Counting Algorithm*[12] solves a given problem, but can’t explain why it solves that particular problem, then I will have difficulty applying it in the future.

[9]unfortuantely, not all resources are available for every problem
[10]And that’s okay! It just means solving it right now is likely not the best use of time to improve at the moment.
[11]a function named `dijkstra()` is a pretty good hint, for instance
[12]not a real thing

- There are several ways to ensuring ability to apply a solution in the future, such as:
  - noting the factors of the problem that enabeld a given technique to be used
  - understanding why the solution was not identified in this case
  - thinking about similar situations where the given technique could be applied
  - thinking about ways the technique could be combined with other techniques

Arguably most importantly, you cannot claim to truly understand a solution until you have written your own and validated it against all test cases. Ultimately the goal is to solve problems in a contest setting, and writing code is essential to further that aim.

1.5 Which Language Should You Use?

Which language one should use depends heavily on goals. If one wants to simply have fun in contests, writing in the language they are most familiar is the way. However, if the goal is to succeed at the top levels, then the answer is more nuanced.

In years past, languages such as Java were widely accepted.[13] As time has evolved, however, Codeforces, and its notoriously tight time limits, has become the seminal contest platform. As Codeforces has a single tight limit for all languages, it is significantly more difficult to succeed with languages other than C++.[14] As a result, most resources available to competitive programmers are now delivered in C++, meaning one has to work to translate to another language they might be using.

To conclude, while it may be possible to reach the highest levels with languages other than C++, the handicap in time limit as well as the lack of available resources, make it significantly more difficult. C++ is the META at the highest levels, and those wishing to succeed at that level should use it in most cases. Despite this, this book seeks to cover both the solutions as they would work in C++ as well as provide enough context for Java users to adapt as necessary.

1.6 What Other Resources Are There?

There are far more resources available than can possibly be covered here, so we focus on a few:

[13]Petr, the best competitive coder for a long time, was a notable Java main

[14]not to mention, the ability to `#define` significant portions of code

- Codeforces is currently the preeminant competitive programming platform. They host nearly weekly contests, provide solution descriptions, have numerous informational blog posts and comments sections about various techniques, and enable you to see code of other submitters. While solution it doesn't necessarily have a highly organized discussion of topics, it is an invaluable informational and practice resource. There is a high correllation of Codeforces rating and success in ICPC.
- Kattis is the de-facto platform for running ICPC contests. Problems used in previous Kattis-run ICPC contests are all available for solving. Kattis ranks problems by approximate difficulty and can roughly suggest problems to solve. It is the largest compendium of ICPC-style problems available today.
- cp-algorithms is a project to translate Russian competitive programming resources to english and contains good descriptions and code examples for some algorithms.
- usaco.guide is step-by-step tool for learning algorithms and techniques for USACO, but is extremely transferrable to ICPC.
- David Van Brackle maintains a site of contest data and judge solutions for most problems which are used in North American ICPC contests. It is located here: serjudging.vanb.org

Chapter 2

Basic Techniques

2.1 Runtime Estimation

Programming contests must have a time limit on the runtime of submitted problems. If this were not the case, any computable problem could be solved with a naive and inefficient solution, removing algorithmic expertise as a requisite for success. As such, we have to learn to estimate whether a given solution is likely to run within a time limit. Therein lies an inherent contractiction, however, as algorithms are typically measured using asymptotic analsyis, which describes how an algorithm theoretically scales with the size of the input rather than its absolute runtime.[1]

Not all is lost, however. Given an input size and and the runtime of an algorithm, we can usually say with pretty high confidence whether it will run fast enough. **Warning: Ignore everything you learned in algorithms class about what a runtime represents**

Consider an algorithm with a runtime of $O(n^2)$, and suppose we are to run it on a dataset where $n <= 1000$. By abusing the big-O notation, we will actually evaluate the n^2 expression against the maximum n, and get an estimated runtime of 1000000.[2] In this estimation model, we generally consider any runtime in the low hundred-millions or lower to be fast enough. In this case, one million is several orders of magnitude lower, and thus, should run in time. Using this as a rule of thumb, we can often deduce a required runtime based on the input size.

- If an input is 10^5 or more, the algorithm must run in $O(n)$ or $O(n*log(n))$. $O(n^2)$ will be too slow.

- If an input is in the high thousands, up to even 10^4, a runtime of $O(n^2)$ is likely acceptable. At the high ends, we run close to the hundred million number, cutting it close.

[1]big-O notation

[2]arbitrary runtime units

- When an input is in the low hundreds, perhaps even up to 500, a runtime of $O(n^3)$ is likely acceptable. Such inputs often hint at DP or MaxFlow algorithms, which often carry higher polynomial exponents.
- When an input is less than 100, it often indicates a brute force solution, especially those which contain an exponential term, such as $O(2^n)$ or $O(n!)$.

While this rule will be essential in guiding which algorithms might work, it is far from perfect. There are some considerations to make.

- Big-O notation famously ignores constant factors and other lesser terms. This means that two algorithms with the same theoretical runtime may have wildly different measured runtimes in practice. While writing algorithms, we should be cognizant of how much work is actually done in each of our arbitrary runtime units. If we estimate a runtime of 100 million array updates, that is far different from 100 million calls to render a frame of Crysis.[3] The higher into the hundreds of millions our estimated runtime gets, the more careful we should expect to be.[4]
- Problems have variable runtime limits. In most modern contests, the time limit for each program is listed with the problem and is generally anywhere from a fraction of a secont to 5 seconds. Clearly, the set of solutions which will run in a fraction of a second is different from those which will pass in several seconds. As such, when we have a longer time limit, we should understand that solutions which push into the hundreds of millions may be acceptable. Conversely, if there is a tight time limit, we should expect that such high runtimes are likely not the intended solution.
- Languages may have wildly different runtimes depending on the nature of the solution. For example, if many data structures and and language features are leveraged, a solution in Java or Python may be significantly slower than one in C++, even at the same asymptotic runtime. As such, it is highly recommended to work in C++, which gives the most leeway with respect to time limits.[5]

Given these considerations, why does this estimation work? The way time limits is typically set is to code intended solutions in multiple languages. The runtime of those solutions is measured, and the ultimate time limit for the problem is set at some fixed multiple of the slowest runtime of a judge-provided solution. If that time limit would allow an optimized but not efficient[6] solution to pass, the input size and corresponding time limit may be increased to better distinguish between efficient and non-efficient solutions.

[3]if it can even run Crysis...

[4]Even at smaller estimated runtimes, solutions may fail if the constant is large enough. Example: iterating through an array backwards rather than forwarthrough an array backwards instead of forward may increase the constant factor by several times.

[5]Coding in C, while possible, is considered prohibitive due to its lack of a standard library

[6]efficient in terms of big-O

Depending on the contest, the time limit may differ from language to language, allowing for the fact that some accepted languages may be intrinsically slower. This is not true in every contest, though, and one should allow that a solution which succeeds in C++ may not in Python.

2.2 Native Types

Before we embark on more complex topics, we'll cover native types. Native types are the building blocks of the language, and it's important to know their usage. Not all types are included here, but for those most useful to contest programming.[7][8][9]

Usage	C++	Java
Integers from -2 billion to 2 billion[10]	`int`	`int`
Integers from -9 quintillion to 9 quintillion[11]	`long long`	`long`
Decimal numbers	`double`	`double`
Boolean	`bool`	`boolean`
Single Character	`char`	`char`
String of Characters	`string`	`String`
Tuple	`pair`	`Point`

In java, there is an additional compilcation. For any native type, there is also a corresponding object type. These are named with capital letters as follows:

`int`	`Integer`
`long`	`Long`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`

These types, while not necessary to use when working with individual variables, are required when declaring the type of a container. For instance `List<int>` is invalid, whwile `List<Integer>` is the required usage to create a list of integers. One should be careful, however, if using these in standalone situations, as the behavior of the objects behaves in subtly differet ways depending on the actual value of the variable.[12]

2.2.1 Converting Between Types

Often times, it is necessary to convert between common types. Here we provide a primer on how to do this.

[7] C++ values are as resolved by a modern conteset compiler.
[8] Java types are given natively, when possible. Encapsulation will be discussed later
[9] Not all of these are technically native types
[10] 32 bits total
[11] 64 bits total
[12] For instance, an Integer may be pass by reference or pass by value depending on the actual value of the Integer.

Conversion	C++	Java
Integer to String	`to_string(i)`	`Integer.toString(i)`

Integerto Hex String	`string s(100);`
	`itoa(i,s.c_str(),16);`
		`Integer.toString(i,16)`

String to Integer	`stoi(s)`	`Integer.parseInt(s)`
Hex String to Integer	`stoi(s)`	`Integer.parseInt(s,16)`
Float to Integer	`(int)f`	`(int)f`

Formatted String to String	`string s(100);`
	`sprintf(s.c_str(),`
	`  "foo %d", my_int)}`
		`String.format(`
		`  "foo %d", my_int)`

String to Split String	`stringstream ss(s);`
	`string tmp;`
	`vector<string>v;`
	`while(`
	`  getline(ss,tmp,"\\s+"))`
	`    v.insert(tmp);`
		`s.split("\\s+");`

2.2.2 Literals

While coding, it is often necessary to directly specify the value of some variable as opposed to assigning it from input or computing it from some other variable. This explicit declaration of a value is known as a *literal*. While most are obvious, some are less so.

Type	C++	Java
Integer	i=3	i=3
Long	i=3ll	i=3L
Double	i=3d	i=3D
Hex Integer	i=0x3	i=0x3
Binary Integer	i=b101	i=0b101
String	i="foo"	i="foo"
Character	i='x'	i='x'
Array	int i[3]=1,2,3	int[] i=1,2,3

The distinctions aroud these literals is often irrelevant, as the languages will typically automatically cast between the various types. In rare cases, such as when performing arithmetic on the literal before assignment, the compiler will not perform the cast until after the arithmetic, leading to unexpected results. Consider the following: `long long l=2000000000*10`. The computed value fits in a long, so we would expect this to work. However, the compiler treats the supplied 2 billion as an integer, so multiplying by 10 causes an overflow before the assignment. In order to tell the compiler to treat the literal as a long, we must instead write `long long l=2000000000ll*10`. This example works as expected. Other instances where the proper literal is critical:

- When performing bit manipulation[13] on 64 bit quantities. `1<<40` produces a value of 0 unintentionally, while `1ll<<40` produces the intended result.
- When performing division, to ensure we are performing integer or decimal division as intended. Note that to perform decimal division, just one of the operands must be a float, so we can also just perform a cast: `(double)1/2`
- When specifying strings, to ensure special characters are escaped appropriately. `string s="this string has a \"quote\" and a slash \\ in it"`

2.3 Basic Structures

Moving on to more advanced constructs than just simple native types, here we'll cover some of the basic structures available in the standard libraries of these languages. This is necessary both as a minimal primer on basic data structures, but mostly since these structures are used without a second thought both in this book and when coding for competition. As such, their workings and the usage of their APIs must be second nature. While we will provide a brief overview of the usage of each structure, the API for each is generally more rich than can be fully covered here, and should be referred to for full functionality.

2.3.1 Static Arrays

The most basic structure is a run of the mill array. As arrays only support direct access by index, they support constant-time lookup and constant-time edit. They do not support "removing" elements in the way further structures do. Arrays will be the primary data structure used in most algorithms, and due to their speed, should be preferred unless there is reason to use a smarter structure. In C++ they must be compiled with a known size, but in Java, their size may be set at runtime. In either case, once set, the size is fixed. In the case of 2-d or higher dimension arrays, a Java array can have the size of its multiple dimensions declared at different times. Further, in Java, each "row" of a 2-d array can have a different dimension.[14] Access methods are identical in the two languages.

Listing 2.1: C++

```cpp
int a[5]; //1d array of length 5
int b[5][6]; //2d array of 5x6
int c[5]={}; //0-initialized array
```

[13]see later in this chapter

[14]From a high level, in a 2-d array, each element is an array in itself, and therefore can have its own unique size. This concept extends to higher dimensions as well. See the below code for an example of this.

```
//setting values
a[0]=1;
b[4][3]=2;
fill_n (c, 5, -1); //set the first 5 values of c to -1
for(auto& i:a)printf("%d\n",i);
for(int i=0;i<5;i++)for(int j=0;j<6;j++)printf("%d\n",b[i][j]);
for(auto& i:b)for(auto& j:i)printf("%d\n",j); //this is exactly the same
    as the above line
array<int,4> d;//another way of declaring an array
```

Listing 2.2: Java

```
int[] a=new int[5]; //1d array of length 5
int[][] b=new int[5][6]; //2d array of 5x6
//setting values
a[0]=1;
b[4][3]=2;
Arrays.fill(a,-1);
//iteration
for(int i:a)System.out.printf("%d\n",i);
for(int i=0;i<5;i++)for(int
    j=0;j<6;j++)System.out.printf("%d\n",b[i][j]);
for(int[] i:b)for(int j:i)
  System.out.printf("%d\n",j); //this is exactly the same as above
//fun with higher dimension arrays
int[][] d=new int[2][]; //each "row" is undefined
d[0]=new int[6];//first row has 6 elements
d[1]=new int[3];//second row has 3 elements. Totally valid!
int x=d.length; //2
x=d[0].length; //6
x=d[1].length; //3
```

2.3.2 Dynamic Arrays

Moving to a slightly more advanced construct, dynamic arrays enable both creating arrays of runtime-defined size as well as auto-growing to fit as many elements as you attempt to put in them. As they are backed by arrays, they enable the same operations and performance as static arrays[15] with slightly higher overhead. In C++, the dynamic array types is a `vector`, and in Java, it is an `ArrayList`. Note that as we move from arrays to actual Java object classes, we must use object types, such as `Integer` instead of native types, such as `int`. This is also the first class where we really see the advantages of C++ in terms of code verbosity relativel to Java. It should be noted that while we may insert arbitrarily many elements, if we try to directly access an index which is larger than the array itself, it will cause a runtime error.

[15]amortized

Listing 2.3: C++

```
vector<int> a; //0-length array.
vector<int> b(5,-1); //length 5 array filled with -1
a.push_back(2); //add an element to a vector
b[0]=3; //set an arbitrary element
for(auto& i:a)printf("%d\n",i);
for(int i=0;i<a.size();i++)printf("%d\n",a[i]);
```

Listing 2.4: Java

```
ArrayList<Integer> a=new ArrayList<>();
a.add(2); //add an element
a.set(0,3); //set an element
for(int i:a)System.out.printf("%d\n",i);
for(int i=0;i<a.size();i++)System.out.printf("%d\n",a.get(i));
```

2.3.3 Lists

While lists, especially linked lists, are heavily taught and used in academic settings, they are generally useless in competition programming,[16] but for two very specific use cases: the *stack* and the *queue*.

2.3.3.1 Queues

Queues enable first-in-first-out ordering. They are commonly used to ensure a given set of items are processed in order, such as with BFS.[17] They are implemented by the `queue` in C++ and the `Queue` in Java. Note that in Java, the `Queue` is actually, an interface implemented by the `LinkedList` class.

Listing 2.5: C++

```
queue<int> q;
q.push(4);
q.push(7);
while(!q.empty()){
   printf("%d\n", q.front()); //4, then 7
   q.pop(); //pop only removes the front of the queue
}
```

Listing 2.6: Java

```
Queue<Integer>q=new LinkedList<>();
q.offer(4);
q.offer(7);
while(!q.isEmpty()){
```

[16]never say never, though

[17]See the *Graph* chapter

```
   q.peek(); //check the front of the queue, but don't remove
   //poll() checks AND removes the front of the queue
   System.out.printf("%d\n", q.poll());
}
```

2.3.3.2 Stacks

Stacks work identically to Queues, but in reverse, in a last-in-first-out ordering. They are implemented by the `stack` in C++ and the `Stack` in Java.

Listing 2.7: C++

```
stack<int> s;
s.push(4);
s.push(7);
while(!s.empty()){
   printf("%d\n", s.top()); //7, then 4
   s.pop(); //pop only removes the top of the stack
}
```

Listing 2.8: Java

```
Stack<Integer>q=new Stack<>();
q.push(4);
q.push(7);
while(!q.isEmpty()){
    q.peek(); //check the front of the queue, but don't remove
    //pop() checks AND removes the top of the stack
    System.out.printf("%d\n", q.pop());
}
```

2.3.4 Hashed Structures

Hashed data structures are a bit of magic. They enable fast look up of particular items, as you would find with an array, but also enable fast insertion and removal of arbitrary items, which is not really supported in an array.[18] The mechanism by which it does this is called hashing, and is not covered in depth here.

2.3.4.1 Hash Sets

The first hashed data structure is the *hash set*. It is our first example of a *set*, which along with being a container of items, guarantees that all items contained therein are unique. If a second item is added which is identical to one already contained, the insertion results in no change to the structure. All insertion, deletion, lookup, and modification operations on a hash set are completed in

[18]unless you manually move items one by one, certainly not fast

constant time except for one caveat. In order to insert an element into a hashed structure, we must know the hash of that object; an operation which scales with the size of the object we are attempting to hash. Therefore, hashing an integer is cheap, but hashing a string might be slow. Worse yet, the computation of the hash gets slower as the string gets longer. While integers are by far the most common element of a hash set, in general care should be taken to ensure the object being stored does not hash so slowly as to be prohibitive. In C++ the hashset is implemented by the `unordered_set` class, and in Java, by the `HashSet` class.

Listing 2.9: C++

```
unordered_set<int> s;
s.insert(4);
s.insert(7);
for(auto& i:s)printf("%d\n",i); //no particular order
if(s.find(4)!=s.end())printf("4 is in the set!");
s.erase(4);
printf("%d\n",s.size());
```

Listing 2.10: Java

```
HashSet<Integer>s=new HashSet<>();
s.add(4);
s.add(7);
for(int i:s)System.out.printf("%d\n",i); //no particular order
if(s.contains(4))System.out.printf("4 is in the set!");
s.remove(4);
System.out.println(s.size());
```

2.3.4.2 Hash Maps

Hash Maps are structures which enable us to lookup one object based on another, such as how one might lookup someone's phone number based on their name in a directory. From a high level, operation and performance are similar to a hash set, however each element of the hash map also carries a *value* that is associated with that hashed *key*. The most common usage of this construct is graph adcjaceny lists, but in general is useful for any context when the data in an array is sparse i.e. most of the elements of the array are empty and unused, but we still want fast iteration.[19] In C++, the hash map is implemented by the `unordered_map` class, and in Java, by the `HashMap` class.

Listing 2.11: C++

```
unordered_map<int,int> m; //maps an integer to another integer
m.insert({7,6}); //preferred
```

[19]and array-like lookup semantics

```
m[3]=5; //be careful doing this (double construction)
//note the result of iteration/find is a pair!
for(auto& i:m)printf("%d %d\n", i.first, i.second); //no particular order
if(m.find(4)!=m.end())
  printf("%d is a key in the map value %d!",
         m.find(4)->first,
         m.find(4)->second);
m.erase(4);
printf("%d\n",m.size());
```

Listing 2.12: Java

```
HashMap<Integer,Integer>m=new HashMap<>();
m.put(3,5);
m.put(7,6);
for(int i:m.keySet())
  System.out.printf("%d %d\n",i,m.get(i)); //no particular order
if(m.containsKey(4))System.out.printf("4 is in the map!");
m.remove(4);
System.out.println(m.size());
```

One critical point about the usage of hashmaps is the behavior when you attempt to fetch a key which does not exists. In C++, if you attempt to access a non-existent element `x` via `m[x]`, C++ will automatically create a default-initialized item and insert it in the array before returning it. C++ users must be exceptionally careful to avoid doing this if unintended by explicitly checking using `find` if existence is unsure. Java operates completely opposite. if `m.get(x)` is executed against an element `x` which does not exist, Java will produce a runtime error. Modern versions of Java, however, have a richer API which allows more flexibility on actions to take in this situation. Regardless of language, if one is not completely sure of behavior, accesses to array elements should be guarded by existence checks.

2.3.5 Ordered Structures

Ordered data structures provide many of the same interfaces as hashed structures, but instead of providing an arbitrary ordering of the items they contain, they provide an absolute ordering. This enables iterating through the elements in order (such as numerical order for integers, or alphabetical order for strings). To provide this added functionality, instead of taking constant time for operations, ordered structures typically take logarithmic time, which is generally not prohibitive for any problems of the sizes used in contests.

2.3.5.1 Tree Sets

As with the hash set for hashed structures, the *tree set* provides the same guarantee that all items are unique. Instead of hashing each entry though, the tree set compares their values to find the location in a search tree[20] where the item should reside. For standard library types, comparators are typically provided, but if necessary, custom comparators may also be supplied to define the required ordering. In C++ the tree set is implemented by the `set` class, and in Java, by the `TreeSet` class. The API for tree sets are some of the richer among these standard containers, and are often used to implement a priority queue with random access.[21]

Listing 2.13: C++

```
set<int> s;
s.insert(7);
s.insert(4);
for(auto& i:s)printf("%d\n",i); //returns 4 then 7
if(s.find(4)!=s.end())printf("4 is in the set!");
s.insert(5);
s.insert(9);
s.lower_bound(6); //iterator to 7
s.upper_bound(6); //iterator to 5
*s.begin(); //4
s.erase(s.begin()); //deletes 4
s.erase(7);
printf("%d\n",s.size());
```

Listing 2.14: Java

```
TreeSet<Integer>s=new TreeSet<>();
s.add(7);
s.add(4);
for(int i:s)System.out.printf("%d\n",i); //returns 4 then 7
if(s.contains(4))System.out.printf("4 is in the set!");
s.add(5);
s.add(9);
int x=s.lower(7); //5
x=s.floor(7); //7
SortedSet<Integer> y=s.headSet(7); //4,5
y=s.headSet(7,true); //4,5,7
x=s.higher(7); //9
x=s.ceiling(7); //7
y=s.tailSet(7); //9
y=s.tailSet(7,true); //7,9
x=s.first(); //4
x=s.last(); //9
s.pollFirst(); //removes 4
s.remove(7);
System.out.println(s.size());
```

[20]typically a balanced binary search tree

[21]as for dijkstras algorithm or minimum spanning trees

2.3.5.2 Tree Maps

As tree sets are to hash sets, *tree maps* are to hash maps. It has all the functionality of a hash map, but with the ordering behavior of a tree set.[22] They are less commonly necessary than the other structures called out in this section, but may be useful in rare scenarios. In C++, these are implemented with a `map`, and in Java, with a `TreeMap`. The interface is more broad than can be covered here,[23] and users should refer to the documentation for a full description of the functionality. Most basic functionality is identical to that of the hash map.

2.3.5.3 Heaps

A *heap* is a structure which provides ordered access to a list of items. It provides the ability to add arbitrary items, but only to access and remove the first. Despite the limitation, there are some advantages over ordered sets:

- Heaps can have multiple items of equal value
- Heaps, being implemented with an array, are more space efficient than the linked structure of the tree set
- Due to the memory efficiency, operations will typically be faster

In C++, heaps are implemented with a `priority_queue`, and in Java, with a `PriorityQueue`.

Listing 2.15: C++

```
priority_queue<int> q;
q.push(4);
q.push(7);
int i=q.top(); //4
q.pop(); //removes 4
```

Listing 2.16: Java

```
PriorityQueue<Integer>q=new PriorityQueue<>();
q.offer(4);
q.offer(7);
q.peek(); //4
q.poll(); //removes 4 and returns it
```

Note that the Java `PriorityQueue` does have a `remove` method, but this is deceptive and runs in linear time, as opposed to equivalent methods on an ordered set.

[22] and the logarithmic cost

[23] given the utility in contest problems

2.3.6 Tuples

The last and perhaps simplest container we will take a look at are tuples, most specifically doubles. These enable you to store two items in a single object. While in most cases this could be replicated by two individual variables, or a array of size 2, it can sometimes be convenient to manage two values as a single unit, such as an x,y coordinate.[24] The C++ implementation is a `pair`, and the Java implementation is a `Point`.[25]

Listing 2.17: C++

```
pair<int,int>p={4,7};
p=make_pair(5,9); //another valid way to make a pair
printf("%d %d",p.first,p.second);
```

Listing 2.18: Java

```
//import java.awt.*;
Point p=new Point(4,7);
System.out.printf("%d %d",p.x,p.y);
```

One should be careful using these objects in Java when storing them in other containers, as they are pass by reference instead of pass by value. Be sure to construct a new `Point` when putting in a structure.[26] Futher, the Java implementation, unlike C++, is not generic and only supports integers. While a class `Point2D.Double` does exist, it is especially complicated to use. Instead, we recommend using a dedicated custom class which has the benefit of also supporting numeric operations.

Listing 2.19: Java

```
class Pp{
   Pp(double xi,double yi){x=xi;y=yi;}
   Pp(Pp p){x=p.x;y=p.y;}
   Pp add(Pp p){return new Pp(x+p.x,y+p.y);}
   Pp sub(Pp p){return new Pp(x-p.x,y-p.y);}
   Pp mul(double p){return new Pp(x*p,y*p);}
   Pp div(double p){return new Pp(x/p,y/p);}
   double cross(Pp p){return x*p.y-y*p.x;}
   double dist2(){return x*x+y*y;}
   double dist2(Pp p){return Math.pow(x-p.x,2)+Math.pow(y-p.y,2);}
   double dist(Pp p){return Math.sqrt(dist2(p));}
   Pp perp(){return new Pp(-1*y,x);}
   double x;
   double y;
}
```

[24]or the index, distance pair we will see in Dijkstra's algorithm

[25]specifically, a `java.awt.Point`, so be sure to import `java.awt.*` if you intend to use it

[26]such as q.offer(new Point(p.x,p.y))

2.4 IO

Input and Output are often the bane of those beginning on a competitive programming journey. It should not be this way, however, and making it such that input and output are second nature is an essential skill to success.

2.4.1 C++

From a high level, there are two methods of I/O in C++:

- C++ style primitives using `cin` and `cout`
- and C style primitives using `scanf` and `printf`

While the choice of primitive is ultimately up to the user, this book recommends using `cin` for input, and `printf` for output for expediency. Note that if you are trying to execute the example code, start from the **Problem Template** section, which contains necessary context for successful compilation.

2.4.1.1 Input

Consider the folloing input description:

> On the first line of input will be 2 integers, a and b, and a decimal, c, where $0 \leq a, b, c < 20$. Following this will be a lines, each consisting of 2 integers, x and y, where $0 \leq x, y < 10^{10}$. Following this will be b lines, each containing a string of alphanumeric characters. Following will be a line of text, s, which may contain whitespace, and lastly will be a line containing a single integer, d where $0 \leq d < b$.

This input may look daunting, but we'll break it up into two large steps.

1. allocating variables to store the data
2. reading the input into those variables

The first step breaks down as follows. These numbers will be labeled in the following code segment.

1. We need two ints, `a` and `b`.
2. We need a double, `c`.
3. We need to store a variable amount of integers, so we will allocate an array. Note that the integers may go up to 10^{10}, so an `int` is not large enough. We will require an array of `long long`. While with some difficulty, we could allocate an array of a variable size in C++, as we know that there will not be more than 20 lines of input, we can simply declare the array

of size 20. Lastly, as there are two integers on each line, we will actually use a 2-d array to capture both values.[27]

4. We need an array of `string`s to hold the next set of lines. As with the above array of integers, we can size this at the maximum size we need for any possible input.

5. We need a single string to hold the one line which might have spaces in it.

6. we need one last integer, which fits in an int, `d`.

We are now ready to see this in code.

Listing 2.20: C++

```cpp
int a,b; //1
double c; //2
long long as[20][2]; //3
string bs[20]; //4
string s; //5
int d; //6
```

While this may seem daunting when reading the input description, if we simply create the necessary variables in order, the code all but writes itself.

The second step involves actually parsing the input. Fortunately, the `cin` construct makes this very straightforward. We again label the code with the numbers oulined just earlier.

Listing 2.21: C++

```cpp
cin>>a>>b>>c; //1 and 2
for(int i=0;i<a;i++)cin>>as[i][0]>>as[i][1]; //3
for(int i=0;i<b;i++)cin>>bs[i]; //4
getline(cin, s); //huh?
getline(cin, s); //now you lost me!
cin>>d; //6
```

Much like the defining of the variables, the reading in of the variables is just as simple with `cin`...except for step 5. To see why this particular step is complicated, we have to know a bit about `cin`. From a high level, `cin` works by reading each character of the input until it reaches a whitespace character. This works great for reading the integers, double, longs, and even strings of characters which contain no whitespace.[28] We run into an issue, however, when we must read an entire string which does contain whitespace. Attempting to use `cin` natively would lead to our variable `s` only containing the first word on the line,[29] not the whole line.

[27]This is far from the only way to do this. We could also use a vector, we could use two arrays, one for the first values on each line, and one for the second, or we could use an array of pairs.

[28]since they are alphanumeric

[29]i.e. up to the next whitespace

In order to alleviate this problem, C++ provites the `getline` function. This function works largely as you would expect, reading an entire line of input. But if it is so simple, then why do we oddly see it doubled, seemingly reading twice to only get one line of input? The reason to this is somewhat involved, but again from a high level, `cin` stops when it gets to whitespace, leaving the next call to deal with it. This means when the last element of the `bs` array is read with `cin`, `cin` will read up to the newline but leave it in place. Therefore, when we make the first call to `getline`, the first thing we see is a newline character. `getline` then thinks it is done and returns an empty line; it has not reached the intended line yet. Instead, the second call to `getline` will read the line we intend it to.

While this may sound complicated, the the details are not all that important. We only need the rule of thumb: if changing from `cin` based input to `getline` based input in the same problem, we need a single extra call to `getline` before proceeding. This rule does not apply for input that only uses `getline`, or when transitioning in the other direction.

In order for `cin` to not impact performance, the following two lines should be included at the start of every solution:[30]

Listing 2.22: C++

```
ios::sync_with_stdio(0);
cin.tie(0);
```

2.4.1.2 Output

Lets continue on from the input description from before, but now add the following output specification:

> Print the output exactly as it was read in, but print c to two decimal places of precision.

We will use the `printf` construct for each element of the output. `printf` allows us to specify an output string while putting placeholders where the variables should go in the string. We cannot completedly reproduce the specification for all those placeholders here but instead provide an overview.

[30]see the *Problem Template* section for examples

Code	Behavior	Output
`printf{"%d", 100}`	`int` placeholder	`100`
`printf{"%lld", 10000000000ll}`	`long long` placeholder	`10000000000`
`printf{"%05d", 100}`	`int` padded to 5 digits	`00100`
`printf{"%f", 100.234908}`	`double` placeholder	`100.234908`
`printf{"%.2f", 100.234908}`	`double` to two decimal places	`100.23`
`printf{"%s", "some string"}`	`string` placeholder	`some string`
`printf{"\n"}`	a newline character	
`printf{"%%"}`	how to print an actual %	`%`

Therefore, the following code would produce the output as described.

Listing 2.23: C++

```
printf("%d %d %.2f\n",a,b,c); //1 and 2
for(int i=0;i<a;i++)printf("%lld %lld\n",as[i][0],as[i][1]); //3
for(int i=0;i<b;i++)printf("%s\n",bs[i].c_str()); //4
printf("%s\n",s.c_str()); //5
printf("%d\n",d); //6
```

Note that **printf** does not automatically insert a newline, so we must manually do so with every output. Also note that all printed strings must be conditioned with `c_str()` when printed with `printf`.[31]

Many people prefer to use `cout`-style output. This is generally not a problem, but sometimes may be more verbose, and takes care to not perform poorly.[32]

2.4.1.3 Problem Template

Cutting straight to the chase, a problem template for solving in C++ should look as follows:

Listing 2.24: C++

```
#include<bits/stdc++.h>
using namespace std;
//any #defines you'd like

//declare input variables

int main(){
   ios::sync_with_stdio(0);
   cin.tie(0);

   //read input

   //do stuff
```

[31]This is not true with `cout`-style printing, one of its advantages.
[32]mostly by using `\n` instead of `endl`

```
   //print output
}
```

We will not prescribe any particular `#define`s here, as which, if any, are used is the preference of the coder. Variables are declared outside of the `main` function to more easily facilitate sharing data across function calls, but again is a matter of preference for the coder. It should be noted, however, that global variables are initialized statically, while local variables are not, and thus must be explicilty initialized.[33]

Multiple Datasets Per Execution The templtate presented makes the assumption of a single test case per execution of the program, which is the input method of the wide majority of programming contests held today. In rare occasions and when solving older problems, multiple test cases are delivered in a single execution of the program. We will look at 3 flavors.

Looking at the first flavor:

> The first line contains a single integer n, the number of test cases.

In this flavor, the input explicitly tells us how much input to expect.

Listing 2.25: C++

```
int main(){
   ios::sync_with_stdio(0);
   cin.tie(0);
   int numc;
   cin>>numc;
   for(int cnum=0;cnum<numc;cnum++){
      //declare variables
      //read input
      //do stuff
      //print output
   }
}
```

The modification is relatively straightforward and involves wrapping test cases in a big loop. We must be careful however, as now that there are multiple test cases in a single execution, global variables will not get reinitialized. It may therefore be better to declare them inside the for loop and explicitly initialize them if necessary.

Looking at the second flavor:

> If a and b and c equal 0, there are no more test cases.

In this flavor, certain signals are used to indicate that input is terminated. These are usually some sort of pattern which would be invalid for an actual test case.

[33] say, if you want an array of zeros

Listing 2.26: C++

```
int main(){
   ios::sync_with_stdio(0);
   cin.tie(0);
   while(true){
      //declare variables
      int a,b,c;
      //read input
      cin>>a>>b>>c;
      if(a==0&&b==0&&c==0)break;
      //do stuff
      //print output
   }
}
```

We use an infinite loop here, and explicitly break out when the input warrants. If there is additional input for each test case, we must be sure to perform the check which breaks out of the loop before attempting to read it, or otherwise risk a runtime error.

The final flavor involves no signal at all, and simply relies on detecting that there is no more input to consume. This is very rare. In these cases, we must depend on the behavior of `cin`[34] to allow us to exit appropriately.

Listing 2.27: C++

```
int main(){
   ios::sync_with_stdio(0);
   cin.tie(0);
   int a;
   while(cin>>a){ //or while{getline(cin, a)}
      //declare other variable
      //read input
      //do stuff
      //print output
   }
}
```

In the case of `cin`, multiple variables may still be read at once inside the `while` conditional.

2.4.2 Java

While Java has slightly different syntax, many of the mechanisms are similar to C++. As such, this section will, while providing the same examples, focus on the features which are distinct from C++. Several references will be made to more full, fundamental explanations given in the C++ section.

[34]or `getline`

2.4.2.1 Input

The most common input method in Java requires the use of the `Scanner` class. As such, each program will have to declare `Scanner in=new Scanner(System.in);` in their program[35] to consume it. We'll use the same input example from C++:

> On the first line of input will be 2 integers, a and b, and a decimal, c, where $0 \leq a, b, c < 20$. Following this will be a lines, each consisting of 2 integers, x and y, where $0 \leq x, y < 10^{10}$. Following this will be b lines, each containing a string of alphanumeric characters. Following will be a line of text, s, which may contain whitespace, and lastly will be a line containing a single integer, d where $0 \leq d < b$.

The variables allocated to store this data:

Listing 2.28: Java

```java
int a,b; //1
double c; //2
long[][] as=new long[20][2]; //3
String[] bs=new String[20]; //4
String s; //5
int d; //6
```

Note that in Java, you must explicitly construct the arrays, unlike C++. This means that if desired, dynamically sized arrays can be created.[36]

Now for the reading:

Listing 2.29: Java

```java
a=in.nextInt(); //1
b=in.nextInt(); //2
for(int i=0;i<a;i++)for(int j=0;j<2;j++)as[i][j]=in.nextLong(); //3
for(int i=0;i<b;i++)bs[i]=in.next(); //4
in.nextLine();
s=in.nextLine(); //5
d=in.nextInt(); //6
```

Several Notes:

- It is common to combine the declaration and input of types, such as `int a=in.nextInt(),b=in.nextInt();`
- Unlike `cin`, scanners cannot read multiple variables in one shot, so for item 3, we create a second `for` loop to enable simple reading of both values.

[35]you can name it whatever you like

[36]The Java interface provides size checking, unlike C++, something Google and Oracle once fought over in the Supreme Court.

- Like C++, going between reading entire lines and individual elements on a line tricky, and there requires an extra `nextLine` when changing to reading lines. See the C++ section for more details.

2.4.2.2 Output

As with C++, we will use the `printf` construct, except here it is conditioned as `System.out.printf`. When convenient, we will also mix this with `System.out.println`. Either is appropriate depending on the situation. The Java `printf` functionality largely follows the syntax described in the C++ section but for a few small details. Long integers may use `%d`, and strings work natively.

Listing 2.30: Java

```java
System.out.printf("%d %d %.2f\n",a,b,c); //1 and 2
for(int i=0;i<a;i++)System.out.printf("%d %d\n",as[i][0],as[i][1]); //3
for(int i=0;i<b;i++)System.out.println(bs[i]); //4
System.out.println(s); //5
System.out.println(d); //6
```

2.4.2.3 Problem Template

A problem template for solving in Java should look as follows:

Listing 2.31: Java

```java
import java.util.*;
class foo {
   static Scanner in=new Scanner(System.in);
   public static void main(String[] args){
      //declare variables/read input
      //do stuff
      //print output
   }
}
```

and the remaining templates:

Listing 2.32: Java

```java
import java.util.*;
class foo {
   static Scanner in=new Scanner(System.in);
   public static void main(String[] args){
      int numc=in.nextInt();
      for(int cnum=0;cnum<numc;cnum++){
         //declare variables
         //read input
         //do stuff
         //print output
      }
   }
}
```

Listing 2.33: Java

```
import java.util.*;
class foo {
   static Scanner in=new Scanner(System.in);
   public static void main(String[] args){
      while(true){
         int a=in.nextInt(),b=in.nextInt(),c=in.nextInt();
         if(a==0&&b==0&&c==0)break;
         //declare other variables
         //read input
         //do stuff
         //print output
      }
   }
}
```

Due to the unreliability of Java detection of the end of input streams, we do not recommend using Java with unbounded input. Fortunately, this format has long been eschewed as an input method.

2.4.2.4 Fast Input

While the `scanner` method is typically fast enough for most competitions and problems, certain input-heavy combinations heavy may cause excessive time to be consumed.[37] In such cases, we can actually write and use an optimized scanner class.

Listing 2.34: Java

```
//import java.io.*;
static InputReader in = new InputReader(System.in);
static PrintWriter out = new PrintWriter(System.out);
public static void main(String args[]){
    //solve problem as usual
    out.printf("the answer");
    out.flush();
}
static class InputReader {
    public BufferedReader reader;
    public StringTokenizer tokenizer;
    String line;
```

[37]especially in Codeforces, with its tight time limits

```
    public InputReader(InputStream stream) {
        reader=new BufferedReader(new InputStreamReader(stream), 32768);
        tokenizer=null;
    }

    public String next(){
        while (tokenizer==null||!tokenizer.hasMoreTokens()) {
            if(line==null){
                try{line=reader.readLine();}
                catch(IOException e){}
            }
            tokenizer=new StringTokenizer((line));
            line=null;
        }
        return tokenizer.nextToken();
    }
    public int nextInt() {return Integer.parseInt(next());}
    public long nextLong() {return Long.parseLong(next());}
}
```

From a high level, we read the input a line at a time into a tokenizer from which we read the variable types we need. The code can easily be adapted/extended to read other data types. Note that we also include a fast output writer in this template which is used in place of the usual `System.out`.

2.5 Language Tools

2.5.1 Custom Comparation

In the course of coding, it often becomes necessary to provide a sorting comparator which differs from standard. In the examples here, we will sort a list of 10 numbers based on a score from a second array. The element of the to_sort array will refer to the index in which the score for that element resides.

Listing 2.35: C++

```
int score[10]={2,6,9,4,6,4,3,7,2,0};
int to_sort[10]={0,1,2,3,4,5,6,7,8,9};

sort(to_sort,to_sort+10); //sorts the array normally
sort(to_sort,to_sort+10,greater<>()); //sorts in reverse
auto cmp=[&](int a,int b){return
    score[a]!=score[b]?score[a]-score[b]:a-b;};
sort(to_sort,to_sort+10,cmp); //sorts by custom comparator
```

Listing 2.36: Java

```
//must be final to be used in a comparator
final int[] score={2,6,9,4,6,4,3,7,2,0};
Integer[] to_sort={0,1,2,3,4,5,6,7,8,9};

Arrays.sort(to_sort);//sorts the array normally
Arrays.sort(to_sort,Collections.reverseOrder()); //reverse
Arrays.sort(to_sort, new Comparator<Integer>(){ //sorts by "score"
    public int compare(Integer a, Integer b){
        if(score[a]!=score[b])return score[a]-score[b];
        return a-b;
}});
```

Critical Rules for Custom Comparators While writing custom comparators, it is critical to obey two rules. Without careful implementation, we can effect undefined behavior, which may result in corruption of the collection we are attempting to sort.

1. The comparator should only return *equal* if the two objects are actually equal. In our case, this means if two indices have equal values in the score array, we should attempt to break the tie. This is typically done by using some arbitrary, but consistent, tiebreaketer, such as the the value in the to_sort array. This rule is more critical for comparisons in tree sets, where improperly indicating two values as equal can lead to undefined behavior.[38]

2. The comparator must adhere to the converse property. Namely, if `compare(a,b)` returns 1, then `compare(b,a)` must return −1. This is most commonly violated in cases such as are covered in rule 1, whereby a tie is broken in an inconsistent way.[39]

2.5.2 C++ Specific Tools

2.5.2.1 Iterators

An iterator is a special construct C++ which provides context about a traversal of a container. They aren't often used directly but for some very specific contexts. There are two primary operations on iterators:

1. dereference it to get the item in the container the iterator points to

2. call `next()` to move the iterator to the next item in the container

There are a few special iterators we have to be concerned about:[40]

[38]maybe it is technically defined by the language, but it should be considered undefined for our purposes

[39]such as `return -1` in case of a tie, instead of a proper tiebreak, which will violate the converse property.

[40]Note that we must use this particular array declaration to leverage iterators

Listing 2.37: C++

```
array<int,4> a={1,2,3,4};
array<int,4>::iterator i;
i=a.begin();
printf("%d\n",*i); //1
i++;
printf("%d\n",*i); //2
i=a.end(); //a special iterator to an item AFTER the last item
reverse_iterator<int*> j=a.rbegin(); //start at the end, go backwards
printf("%d\n",*j); //4
j++;
printf("%d\n",*j); //3
```

Iterators can be used on almost any container, including strings.[41] There are a couple contexts where we most often see them:

Listing 2.38: C++

```
//the find method returns an iterator which
//points to "end" if the item does not exist
unordered_set<int> a;
if(a.find(x)==a.end())printf("%d not in a\n",x);

//to get the first element in a priority queue
//or ordered set
set<int> s;
int on=*s.begin();

//to compute the next permutation. See below
```

This is not an exhaustive example of iterator usage, so its important to understand their high level functionality in order to cope with them when necessary.

2.5.2.2 Next Permutation

Often times in problems, it becomes useful to iterate through all possible permutations of a list of objects. C++ provides a conveninient method to do this. It takes iterators to any two points in some structure, and computes the next permutation of the elements between those two iterators. In most cases, the iteraters are the first and last items of the collection, and the next iteration of the entire collection is returned. The function returns false when the last permutation is reached.

Listing 2.39: C++

```
array<int,4> a={1,2,3,4};
while(next_permutation(a.begin(),a.end()))//do stuff
```

[41] where the iterator points to an individual character

```
1234->1243->1324->1342...4321
```

2.5.2.3 String Access

Strings in C++ are a bit of a lie. While they are a real object, in the background they are in actuality of type `vector<char>`. This means that strings can be indexed to access individual characters. This is convenient for many string problems, especially those on ASCII grids, where character access is essential.

Listing 2.40: C++

```
string s="hello";
printf("%c\n",s[1]); //e
```

2.5.2.4 Multi Maps

When discussing maps earlier, the assumption was the keys of the map formed a set, meaning all keys are unique in the map. *Multi map* is a class which enables having multiple elements per key. This might be useful in gertain circumstances, so is covered here.

Listing 2.41: C++

```
multimap<int,int>mm;
mm.insert({1,4});
mm.insert({1,3});
mm.insert({3,7});
auto r=mm.equal_range(1);
for(auto& i=r.first;i!=r.second;i++)
  printf("%d %d\n",i->first,i->second);//1,4 and 1,3
```

2.5.3 Java Specific Tools

2.5.3.1 BigInteger and BigDecimal

Every once in a while, a problem arises which requires accurate computation on integers up to, or exceeding, 2^{63}. In such cases, `long` is insufficiently large to hold the value, and `double` is not precise enough. In C++, these problems would require implementing a custom library for performing operations on large numbers. In Java, we can simply use the `BigInteger` class. This class contains arbitrarily large numbers and provides many basic arithmetic operations. This seemingly powerful behavior comes at a cost, though: as the size of the integer grows, the operations become slower. This is a direct consequence of basic arithmetic operations actually being $O(log(n))$ and not the $O(1)$ that we often take for granted.

Similarly, the `BigDecimal` class provides decimal numbers to arbitrarily high precision. Like `BigInteger`, the finer the precision, the longer the computation times.

2.5.3.2 Geometry Package

Many computational geometry problems involve computing the relationship between certain geometric constructs such as points, lines, rectangles, and polygons. While the math to do this should be known and is covered in a later chapter, Java provides several constructs that may prove useful in certain circumstances.

Listing 2.42: Java

```java
//import java.awt.*;
//import java.awt.geom.*;
Point p;
Line2D l1,l2;
Rectangle r1,r2;
Point p=new Point(0,0);
Line2D l1=new Line2D.Double(0,1,2,3);
Line2D l2=new Line2D.Double(5,6,7,8);
Rectangle r1=new Rectangle();
Rectangle r2=new Rectangle();
l1.intersectsLine(l2);
l1.ptLineDist(p);
l1.ptSegDist(p);
l1.relativeCCW(p); //whether a point is left of, right of, or on a line
r1.contains(p);
r1.contains(r2);
r1.intersects(r2);
r1.createIntersection(r2);
r1.createUnion(r2);
```

2.5.3.3 Fancy Output Formatting

Consider the following output description:

> Output should be printed to two decimal places of precision. For values greater than 10^3, a comma should be placed between the hundreds and thousands digit, for values greater than 10^6, a comma should be placed after the millions digit, and lastly, for values greater than 10^9, a comma should be placede after the billions digit. All values will be less than 10^{12}.

While we can spend considerable effort to construct the string as required, Java allows solving of this trivially with convenient formatting functions.

Listing 2.43: Java

```
//import java.text.*;
double d=12345.234;
System.out.printf(
  "%s\n",
  new DecimalFormat("#,###,###,###.00").format(d));
System.out.printf("%,.1f\n",d); //even simpler
```

There are several other options in the `java.text` package for clever string formatting.

2.6 Basic Techniques

Lastly in this chapter, we cover some high level basic techniques for problem solving. These may not be specific algorithms, but are techniques that may be used to form an entire solution or as building blocks within a solution.

2.6.1 Brute Force Search

Consider the following problem:

> Given a list of numbers, return the largest one.

A solution to this problem requires checking every number in the list to see if it is largest. This would be considered a *brute force search* of the list. From a high level, a brute force search involves trying every possible solution to find the correct one. It would be akin to looking for your friend's phone number by calling every possible 10-digit combination until you reaach them.[42]

While such approaches are often untenable,[43] this is not always the case. Many problems do, in fact, use brute force as their intended solution. As such, thinking through a brute force solution of a problem should often be a step taken when solving for two reasons:

1. A brute force solution may be correct, and immediately dismissing it may result in a mental blind spot.

2. Thinking through a brute force solution, even if not correct, has two two soft benefits.

   (a) It forces one to ensure they understand the problem enough to come with *some* solution

   (b) Doing so may reveal some structure of the problem which leads to an intuition about an optimization which may prove critical

[42]US numbers without a country code

[43]see earlier section on estimating runtime

In all, thinking through a brute force solution is a forcing function for thinking critically about the problem itself. Consider the following problem which might help the idea click:[44]

> In a classroom, there are 15 students. A teacher is trying to find the largest group of students such that every student in the group likes every other student in the group. The teacher knows for each student which other students that student likes. Return the largest group the teacher can make.

Such a problem may sound daunting at first, and without experience in graph theory, one may struggle. However, let us consider a brute force solution. Can we simply consider every possible grouping of students?

- There are 2^{15} possible groups.[45]
- To check whether a group works, we must check every person in the group against every other person in the group, which is a quadractic operation[46] taking 15^2 comparisons in the largest groups.

Using the estimation from earlier in the chapter, this takes $2^{15} * 15^2 \approx 7000000$ operations, which is plenty performant. So in this case, ensuring we could identify some brute force solution led us to the actual solution.

2.6.1.1 Fixing a Variable

Fixing a variable[47] is a technique where we cannot directly solve for some variable of a problem, so we instead assume that variable to take on a particular value, solve the rest of the problem as if that selection were correct, and finally, try multiple values of the assignment.

Considering again our student grouping problem, the transformation might look like this:

- We know the solution must be some number from 1 to 15, but don't know which.
- Assume the correct solution is some value X.
- Iterate over all groups of size X to see if there is a valid grouping where all students like eachother.
- Search over values of X as apporpriate. In this case we can use a binary search.[48]

[44]or clique
[45]There is 1 group of 1 student, 3 groups of 2 students, 7 groups of 3 students. See the math chapter for more details on combinatorics.
[46]using a double `for` loop
[47]as this author calls it
[48]see the *Binary Search* section of the *Match and Geometry* chapter for more details

In this case, the value X with the largest number of groups to go through is 7, with 6435.[49] It therefore takes about $6435 * 7^2 \approx 300000$ to evaluate whether any group with 7 students works. With a binary search, we have to evaluate at most 4 values of X to determine the largest for which there is a adequate group.

By using the fixing a variable method, we were able to devise a brute-force based solution which used about 6x fewer operations. It should be noted that not all solutions when fixing a variable require binary search. Some involve evaluating all values of X, and some involve other ways to choose the candidate values of the fixed variable to evaluate.

2.6.2 Simulation

Many problem statements boil down to *do exactly what the problem statement tells you.* These problems are known as *simulation* problems.

> You are given a integer-valued piecewise function $x' = f(x)$ and a list ot $x_0, x_1, \ldots, x_n$ which are all unique. At step, each x_i is evaluated against f, producing a new list, $x'_0, x'_1, \ldots, x'_n$. How many steps must be executed before the produced list after that step contains some pair of identical numbers?

Without any information about the nature of f, the only solution is to simply perform the evaluations of f against the list and check for duplicates. This would be a classical example of a simulation problem. In this instance, it's pretty apparent, but in other cases it may be a bit harder to see, being combined with other techniques such as greedy or brute force.

2.6.3 Recursion and Backtracking

One of the classic techniques of algorithmic problem sloving is *recursive backtracking.* Generally this technique asks us to solve a problem in which we must evaluate several states to find one that meets some provided condidition. In order to do this, we will try to find some piece of the state we can validly fix, and then *recurse* to find another piece of state we can fix. If after the recursion, no such state we can fix is found, we try some other way. This is called *backtracking* and is akin to finding a dead end in a maze and going back to try some other path. Consider this classic problem:

> Given a standard 8x8 chessboard and 8 queens, each of whom can move any number of spaces in a single direction - horizontally, vertically, or diagonally - place each of the 8 queens on the board such that none can attack any other in a single move.

To solve this problem, we will write a function which tries to place a queen in an available spot, and once it has found one, recurses, before back tracking if the recursion fails. The pseudocode looks as follows.

[49] 15 choose 7. See the Math chapter for more details

```
func place_queen(queens_remaining, board):
  for each candidate place on board:
    if safe from other queens on board:
      //recurse
      works = place_queen(queens_remaining-1, board with queen at
          candidate place)
      if works: return works
  return !work //backtrack!
```

We can see this in action:

```
.#.#.#.#  .#.#.#.#  .#.#.#.#  .#.#.#.#  .#.#.#.#  .#.#.#.#  .#.#.#.#
#.#.#.#.  #.#.#.#.  #.#.#.#.  #.#.#.#.  #.#.#.#.  #.#.#.#.  #.#.#7#.
.#.#.#.#  .#.#.#.#  .#.#.#.#  .#.#.#.#  .#.#.#.#  .#.6.#.#  .#.6.#.#
#.#.#.#.  #.#.#.#.  #.#.#.#.  #.#.#.#.  #5#.#.#.  #5#.#.#.  #5#.#.#.
.#.#.#.#  .#.#.#.#  .#.#.#.#  .#.#.#4#  .#.#.#4#  .#.#.#4#  .#.#.#4#
#.#.#.#.  #.#.#.#.  #.#.3.#.  #.#.3.#.  #.#.3.#.  #.#.3.#.  #.#.3.#.
.#.#.#.#  .#2#.#.#  .#2#.#.#  .#2#.#.#  .#2#.#.#  .#2#.#.#  .#2#.#.#
1.#.#.#.  1.#.#.#.  1.#.#.#.  1.#.#.#.  1.#.#.#.  1.#.#.#.  1.#.#.#.
```

There is no place to place the eighth queen, so we backtrack and try another location for queen 7. When trying to place it, we find no further locations for it either, so backtrack further to the 6. We again find no places, so backtrack all the way to the 5

```
.#.#.#.#
#.#.#.#.
.#.#.#.#
#.#5#.#.
.#.#.#4#
#.#.3.#.
.#2#.#.#
1.#.#.#.
```

We now see there is not a place for the 6, and continue looking for a place for the 5. We then find no other place for the 5, so backtrack all the way to finding a new place for the 4. This process of trying to find a location, and recursing or backtracking represents the entirety of the process.

In these problems, it is important to consider what information is passed through the recursive function call. For variables passed through, we have to understand whether they are passed by value, or passed by pointer.[50] In the former case, there will be a new copy at each level of the recursion, but in the latter case it will be the same copy each time. Further, if large objects are passed by value, then there is the risk of exceeding available stack space. While competition stacks are typically quite generous, this is still a risk. If unsure, it is often best to be safe by using global variables for shared state, only passing

[50]or reference

through variables which must be unique at each level of the recursion.

2.6.4 Bit manipulation

Integers[51] is stored as collections of bits. While these are typically interpreted as integers and the like, they can also be considered as a collection of individual bits, which allows us to leverage low-level CPU instructions to accomplish certain tasks. In this section, it is important to think of an integer in several ways:

- an integer with a numeric value
- the binary representation of that number
- a set of individual switches, one for each bit of that number

2.6.4.1 Basic Operators

Here we'll cover the basic operators that are the building blocks for all the remaining bit manipulation techniques. In the course of these examples, we will assume `a=3`.

Operator	Example	Explanation
`a<<1`	0011 → 0110	Shift `a` left by 1 bit. If one considers the value the bits represent, this is a multiplication by 2.
`a>>1`	0011 → 0001	Shift `a` right by 1 bit. If one considers the value the bits represent, this is a division by 2.[52]
`a&5`	0011&0101 → 0001	Perform a *bitwise and*. This means a bit will be set in the output only if it is set in both inputs.
`a|5`	0011|0101 → 0111	Perform a *bitwise or*. This means a bit will be set in the output if it is set in either of the inputs.
`a^5`	0011^0101 → 0110	Perform a *bitwise xor*. If a bit is set in the second number, it will flip the bit in the first number, otherwise no change.[53]
`~a`	0011 → 1100	Perform a *bitwise negation*. This will flip all bits of the input

The biggest caveat of these operators is the precedence with respect to others. They often do not behave how people expect. So in general, when performing

[51]and all data...

[52]Java also has the concept of an *arithmetic right shift*, which is not considered here.

[53]Note that the operation is commutative, so which argument you consider flipping is arbitrary.

bitwise operations, the order in which you wish them to be evaluated should be made explicit via parentheses.

2.6.4.2 Mask and Shift

The concept of mask and shift enables us to set and observe individual bits. This is likely the most common usage of bitwise operations. Why would we want to do that? Suppose we had a problem with 64 individual items, on which we perform operations such as adding item number 23, or removing item 17. Suppose, further, that after each operation, we wanted to check whether we reached a state we had previously reached. Using standard types, this would take 64 individual comparisons. However, if these operations are incoded as bits in a `long long`, then we simply need to compare `long long`s with eachother, replacing the 64 comparisons with a single comparison that compares 64 items at once.

Checking if a bit is set The first operation is to check if the n'th bit of a number is set.[54] From a high level, we want to produce a binary value that has a single 1 in the n'th position. This is accomplished using a left shift with `1<<n` and is called the *mask*. This mask is then bitwise-and-ed with the original number. If that bit is unset it in the original number, the result of this and will be 0. If that bit is set in the original number, the output will be a number with a single 1 in the location of the bit.[55] Finally, we check if that output is equal to zero. If the bit is unset, the number will equal zero, and if the bit is set, the number will not equal zero. The code to check if the `n`'th bit of `a` is set is `(a&(1<<n))!=0`.

We can see what happens to check if the third bit is set:

```
ex. 1: 10101              10101      00000      false
mask:  00001->shift left 3->01000->and->01000->!=0->
ex. 2: 11001              11001      01000      true
```

Setting a bit In order to set the n'th bit, the high level process is to produce a binary value that has a single 1 in the n'th position. This is accomplished identically as above. Then, instead of performing a bitwise-and, we perform a bitwise-or operation. This will ensure the bit is set to 1 regardless of what it was originally. The code to set the `n`'th bit of `a` is `a|(1<<n)`.

Again, an example of setting the third bit:

```
ex. 1: 10101              10101     11101
mask:  00001->shift left 3->01000->or->01000
ex. 2: 11001              11001     11001
```

[54]zero-indexed
[55]i.e. identical to the mask

Flipping a bit Flipping a bit is identical to setting a bit, except we use the xor operation instead of the or operation, namely `a^(1<<n)`.

Unsetting a bit Unsetting a bit is the most complicated. We saw that the or and xor operators set and flipped a bit, respectively, so it would seem natural that the and operator would clear it. It is clear that directly adapting the above behavior, `a&(1<<n)` does not accomplish this. In fact, this operation has about the worst impact possible. It will zero-out all the bits we don't want to touch, and have no effect on the only one we do! If only we could reverse that. We need a single 0 in the bit we wish to clear, and 1's everywhere else.[56] We can accomplish this by performing the shift, and then performing the bitwise negation. The ultimate way to clear a bit is `a&~(1<,n)`.

Clearing the third bit:

```
ex. 1: 10101                   10101      10101     10101
mask:  00001->shift left 3->01000->flip->10111->and->
ex. 2: 11001                   11001      11001     10001
```

2.6.4.3 Other Hacks

While mask and shift is the most common usage of bitwise functions, there are several other tricks that are useful and enabled using bitwise operators.

Clear the least significant 1 bit	`a&=a-1`
Set the least significant 0 bit	`a|=a+1`
Clear all but the first group of 1 bits	`a&=a+1`
Find the least significant ones bit	`a&=-1*a`

It should be noted as well that both C++ and Java have builtins which may cover these and even more complicated hacks in a convenient fashion. These are not listed here, but can be found in the *Other Built-ins* section of the GCC specification for C++, or the Integer/Long object specification for Java.

2.6.4.4 Cumulative Bit Count

```
  //cumulative bitcount
ll cum_bc(ll x){
   ll ans=0;
   for(ll i=1;i<=x;i<<=1)ans+=i*(x/(i<<1))+max(x%(i<<1)-i+1,0LL);
   return ans;
}
```

[56]Perhaps more formally, 0 is the identity of a bitwise or, and 1 is the identity of bitwise and. In any operation where we wish to leave bits unaffected, those bits in the mask must be set to the identity of the operation we wish to take.

Chapter 3

Math and Geometry

3.1 Binary Search

foo

3.2 Ternary Search

bar

3.3 Catalan Numbers

baz

3.4 GCD LCM and Euclid

```
int gcd(int a,int b){
  return b==0?a:gcd(b,a%b);
}

int lcm(int a,int b){
   return a*b/gcd(a,b);
}
```

```
//modular fraction
uint64_t euclid(uint64_t a, uint64_t b, int64_t &x, int64_t &y) {
   if(!b)return x=1,y=0,a;
   uint64_t d=euclid(b,a%b,y,x);
   return y-=a/b*x,d;
}
```

```
uint64_t mod_frac(uint64_t n, uint64_t d){
    int64_t a,b;
    euclid(d,MOD,a,b);
    a=(a%MOD+MOD)%MOD;
    return ((n%MOD)*a)%MOD;
}
```

3.5 Probability

asd

3.5.1 combinations

```
int ncr(int n,int r){
    int ans=1;
    for(int i=0;i<n-max(r,n-r);i++){
        ans*=n-i;
        ans/=i+1;
    }
    return ans;
}

ll dp[49][49];
bool done;
ll ncr(int n,int r) {
    if(!done){
        for(int i=0;i<49;i++){
            dp[i][0] = 1;
            dp[i][i] = 1;
        }
        for(int i=2;i<49;i++)for(int j=1;j<i;j++)
             dp[i][j]=(dp[i-1][j]+dp[i-1][j-1])%MOD;
        done=true;
    }
    return dp[n][r];
}
```

3.5.2 permutations

adsf

3.5.3 stars and bars

asdf

3.6 Primality and Factoring

```
vector<int> pf(1000001,1);
for(ll i=2;i<=1000000;i++)if(pf[i]==1)for(ll
    j=i*i;j<=1000000;j+=i)pf[j]=i;
```

```
ll phi(ll n){
    ll ans=n;
    for(ll i=2;i*i<=n;i++)if(n%i==0){
        while(n%i==0)n/=i;
        ans-=ans/i;
    }
    if(n>1)ans-=ans/n;
    return ans;
}
```

3.7 Convex Hull

```
vector<pll> hull(vector<pll> in){
    vector<pll> out;
    sort(in.begin(),in.end());
    auto mi=in[0];
    sort(in.begin(),in.end(),[mi](pll l,pll r){
        ll
            t=(l.first-mi.first)*(r.second-mi.second)-(l.second-mi.second)*(r.first-mi.first);
        return t!=0?t>0:abs(l.first-mi.first)<abs(r.first-mi.first);
    });
    in.push_back(mi);
    for(auto p=in.begin();p!=in.end();){
        if(out.size()<2)out.push_back(*p++);
        else{
            auto& m=out[out.size()-1],n=out[out.size()-2];
            if((m.first-n.first)*(p->second-n.second)-(p->first-n.first)*(m.second-n.second)>0)out.push
            else out.erase(--out.end());
        }
    }
    out.erase(--out.end());
    return out;
}
```

3.8 cross product

```
ll signum(ll a){ return a>0?1:a==0?0:-1;}
ll cross(pll a1,pll a2,pll b2){return
    (a2.first-a1.first)*(b2.second-a1.second)-(a2.second-a1.second)*(b2.first-a1.first);}
ll dot(pll a1,pll a2,pll b2){return
    (a2.first-a1.first)*(b2.first-a1.first)+(a2.second-a1.second)*(b2.second-a1.second);}
bool seg_on(pll a1,pll a2,pll b2){return
    cross(a1,a2,b2)==0&&dot(b2,a1,a2)<=0;}
bool seg_inter(pll a1,pll a2,pll b1,pll b2){
   ll ab1=signum(cross(a1,a2,b1));
   ll ab2=signum(cross(a1,a2,b2));
   ll ba1=signum(cross(b1,b2,a1));
   ll ba2=signum(cross(b1,b2,a2));
   if(ab1*ab2*ba1*ba2!=0)return ab1!=ab2&&ba1!=ba2; //normal case
   return
       seg_on(a1,a2,b1)||seg_on(a1,a2,b2)||seg_on(b1,b2,a1)||seg_on(b1,b2,a2);
       //point on segment case
}
```

3.9 Center of Gravity and Balances

asd

3.10 Convolution and FFT

df

3.11 Coordinate Transformation

asdf

3.12 Matrices

as

3.12.1 Fast Exponentiation

One built in function of math libraries which is often taken for granted is exponentiation. As such a function exists, it may seem that we can be blissfully unaware of the nuances of the implementation. While this is generally true for must numerical types, scalars are not the only thing that can be exponentiated.[1] Therefore, it is iortant to understand the nature of exponentiation so

[1]spoiler alert: Matrices, but also complex numbers, and any other entity over which the power function is defined

that when we must raise a non-scalar type to a power, we can do so efficiently. Many problems require such an implementation in their intended solutions.

Naively, we would compute b^n as follows:

Listing 3.1: C++

```
long ans=1;
for(int i=0;i<n;i++)ans*=b;
```

This computation takes $O(n)$ multiplications, but it turns out, we can do better. Consider b^8. We know by associativity the $b^n = b^{n/2} * b^{n/2}$, and can therefore break up the original computation as follows: $b^8 = (b^4)^2 = ((b^2)^2)^2$. This yields the following bit of code:

Listing 3.2: C++

```
long b_squared=b*b;
long b_fourth=b_squared*b_squared;
long ans=b_fourth*b_fourth;
```

We have computed b^8 using exactly three multiplications instead of the naive eight. Further, it is clear how this extends to any power which is a power of 2. How though, can we generalize to any integral power?

Method 1: Recursion In perhaps the more natural way, we apply one of the following two rules to a number:

- $b^n = b * b^{n-1}$
- $b^n = b^{n/2} * b^{n/2}$

Or more clearly, if the exponent is odd, factor out b, and then compute with the decremented exponent, which is now even. If the exponent is even, compute with the halved exponent, and then multiple that result against itself. We can see how this would work step by step in the case of b^{13}.

- $b^{13} = b * b^{12}$
- $b^{13} = b * (b^6)^2$
- $b^{13} = b * ((b^3)^2)^2$
- $b^{13} = b * ((b * b^2)^2)^2$

By applying these rules, we have reduced this exponent down to 5 multiplications. In code, it is as follows:

Listing 3.3: C++

```
long exp(long b, long n){
  if(n==1)return b;
  if(n%2==0){
    long temp=exp(b,n/2);
    return temp*temp;
  }
  return b*exp(b,n-1);
}
```

Method 2: Binary Instead of performing the recursion, we can look at the exponentiation a different way. Consider the following two facts:

1. The computation is simple for powers of 2
2. Any number can be broken up into a sum of powers of two

Let's examine the second point as it might apply to 13. Consider the binary representation of 13: 1101. This representation tells us that $13 = 8 + 4 + 1$, represented by the first, third, and fourth bits of the binary representation.[2] Knowing this, we can break up b^{13} as follows: $b^{13} = b^8 + b^4 + b$. This leads to the following algorithm:

1. Raise b to each successive power of two (i.e. b^1, b^2, b^4, b^8, ...) up to the intended exponent (13 in our example)
2. Determine from the binary representation of the intended exponent whether or not that computed value should be multiplied into our final answer (i.e. b^4 should be multiplied in since the fours bit is set in the binary representation of 13, but b^2 should not, as the twos bit is not set)

As we can compute b^8 in three multiplications, computing b^2 and b^4 along the way, we have maintained a logarithmic number of multiplications. The code looks as follows:

Listing 3.4: C++

```
long exp(long b, long n){
  long ans=1;
  while(n!=0){
    if(n&1)ans*=b;
    b*=b
    n>>=1;
  }
}
```

Note the use of the bit mask to determine whether the first bit of the number is set.[3]

[2]going from the least significant bit, one indexed
[3]as described in the *Basics* chapter

Application to Matrices The above functions, most especially the second, can be easily adapted to exponentiating matrices. This is far more important than the scalar exponentiation as it is not a function provided by the standard library. The same methods apply with the following caveats:

- Matrix multiplication must be used instead of integer multiplication
- The `ans` variable must be initialized with the identity matrix (ones on the diagonal) instead of the multiplicative identity (1)

This leads to the following complete code for fast matrix exponentiation:[4]

Listing 3.5: C++

```cpp
// assume #define SZ <max array size>
void mul(long long a[SZ][SZ],long long b[SZ][SZ]){
  long long out[SZ][SZ]={};
  for(int i=0;i<SZ;i++)
    for(int j=0;j<SZ;j++)
      for(int k=0;k<SZ;k++)
        out[i][j]+=a[i][k]*b[k][j];
  for(int i=0;i<SZ;i++)
    for(int j=0;j<SZ;j++)
      a[i][j]=out[i][j];
}
void exp(long long in[SZ][SZ],long long n,long long out[SZ][SZ]){
   for(int i=0;i<SZ;i++)out[i][i]=1; //generate identity matrix
    while(n){
        if(n&1)mul(out,in); //if bit is set, include in out
        mul(in,in); //generate next power of 2 matrix
        n>>=1; //move the mask
    }
}
```

This method is especially useful in the case of finite automata, where transition matrices must be raised to high powers in sub-linear time.

3.13 rational math

[4]including the function for the matrix multiplication itself

Chapter 4

Graphs

In terms of algorithms, we are not concerned with pie and line graphs, but on generic objects, and the relationship between them. Such as, traffic intersections, and the roads which connect them. Countries, and the borders between them. Actors, and the movies they've appeared in together.

This chapter provides an overview of graphs, helps us classify them, and breaks down many of the operations we can perform.

4.1 Types and Terminology

The two main components of a graph are *nodes* (sometimes called vertices) and *edges*. Nodes represent the objects in the graph, and the edges are the connections between them.

nodes	edges
intersections	roads connecting them
countries	borderts between them
actors	movies they are in together

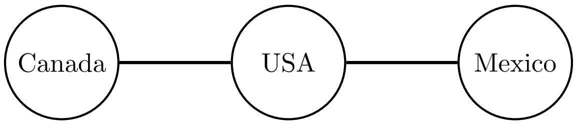


Figure 4.1: A graph of countries and borders in North America

4.1.1 Graph Classification

Within the overall frame of nodes and edges, there are several properties that allow us to classify graphs.

4.1.1.1 Directionality

Fundamentally, there are two big classes of edges.

1. Edges which connect nodes u and v, which imply that you can traverse either direction between u and v. These are equivalent to two-way streets. If edges in a graph are bidirectional, then the graph is known as *undirected*.
2. Edges which connect nodes u and v, which imply that you can traverse only from u to v, but not the other way. These are equivalent to one-way streets. If edges in a graph are directional, then the graph is known as *directed*.

In general, we only consider graphs as having all directed or undirected edges, and if we need two-way edges in a directed graph between, we create two edges, one in each direction between the two nodes in question.

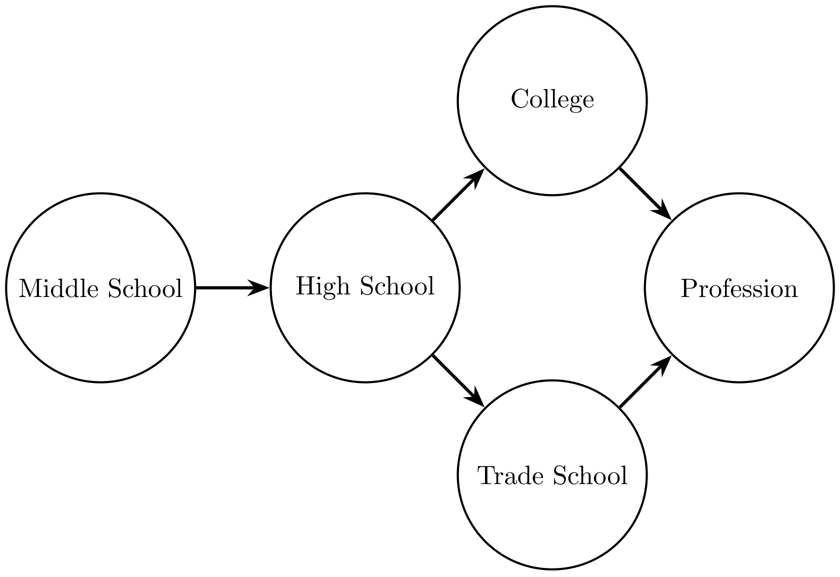


Figure 4.2: An example of a directed graph of career progression. Note the arrows.

4.1.1.2 Weighted-ness

Edges in a graph may have heterogeneous weights attached to them. We might consider this as the length of an edge, using the road analogy. In other graphs, the edges may have no weight (just implying a connection), or all equal weights (often weight 1). Depending on which type a graph falls into, it is known as *weighted* or *unweighted*.

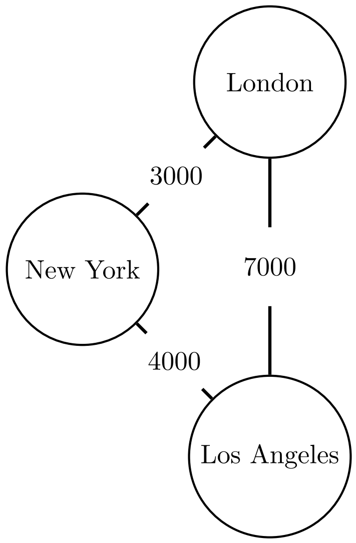


Figure 4.3: An example of a weighted graph indicating the miles between pairs of cities

4.1.1.3 Cycles

A *cycle* in a graph means we can start at one node, traverse through some sequence of edges, and end up back at the same node. Graphs which contain cycles are known as *cyclic* whereas graphs that contain no cycles are known as *acyclic*. Acyclic is an important property that allows us to apply several algorithms that we may not otherwise be able to. There are two tytpes of acyclic graphs:

1. An undirected, acyclic graph is known as a *tree*
2. A directed, acyclic graph is known as a *DAG*

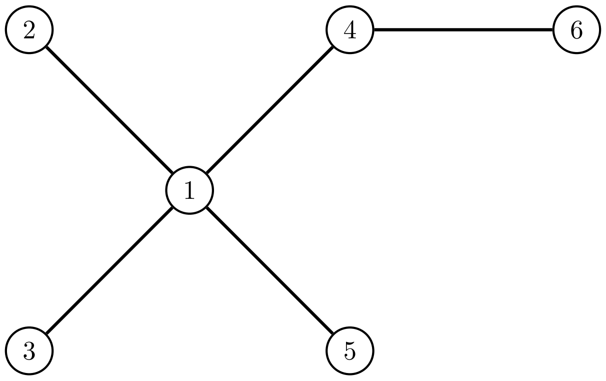


Figure 4.4: A tree

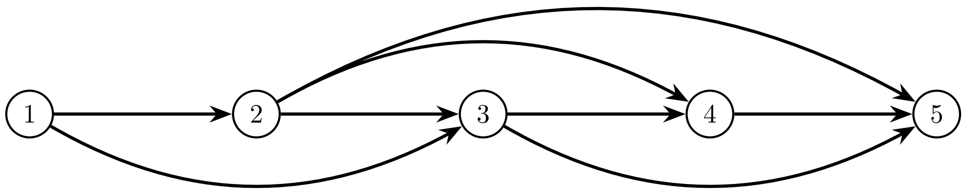


Figure 4.5: A DAG

4.1.1.4 Density

The ratio of the number of edges to the number of nodes often determines how efficiently we can apply algorithms to the graph. In a graph of n nodes, assuming at most one edge from a given node u to v, the maximum number of edges we will have is $O(n^2)$. This is known as a *dense* graph.

Conversely, if the number of edges is much smaller, closer to $O(n)$, the graph is known as *sparse*.

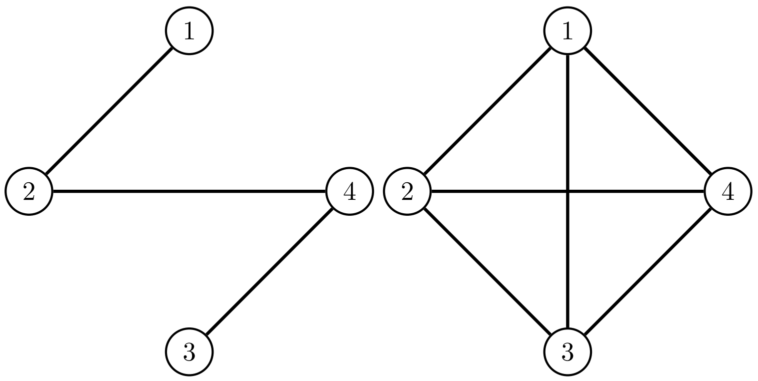


Figure 4.6: A sparse and a desnse graph

4.1.1.5 Connectivity

A graph is known as *connected* if there is some sequence of edges (going in either direction, in the case of a directed graph), between any two nodes in the graph. Otherwise, the graph is known as *disconnected*. There are two special cases of connectivity:

1. If you can reach every node from every other node, when considering edge direction, a graph is *strongly connected*. Note that all connected and undirected graphs are trivially strongly connected.

2. If each node is connected to every other node via two separate paths which do not have any nodes in common (i.e. are *node-disjoint*), the graph is *biconnected*. The implication is that if you remove any single node in the graph, the remaining graph is connected.

It is often useful to look at the connectivity on specific parts of a graph, rather than the entire thing, leading to connected, strongly connected, and biconnected *components*.

4.1.1.6 Bipartite

A graph is known as *bipartite* if every node can be classified into one of two sets, such that there are no edges connecting any two nodes in the same set. Every edge must connect two nodes who are in different sets. An example might be a graph containing nodes indicating people and jobs, with an edge connecting people with jobs they are qualified to perform.

TODO insert picture here TODO add section to BFS on detecting bipartite graphs

4.2 Representation

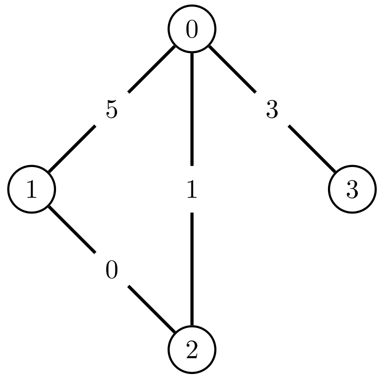
Now that we understand some of the idioms which might describe a graph, we can see how to represent graphs in code. In this section, we will see how to represent graphs which are given via the following description

> The first line of input contains 2 integers, n, the number of nodes, and m, the number of edges. $0 < n < 10^4$. Following this are m lines, each containing a triplet of integers, i, j, and k, indicating there is a directed edge for i to j with weight k. It is guaranteed that $0 <= k < 10^9$ and $0 <= i, j < n$, and that there will be at most one edge from i to j.

4.2.1 Adjacency Matrix

An adjacency matrix represents a graph of N nodes as an NxN matrix M, where M_i, j details some information about an edge originating from node i and ending at node j. In a weighted graph, this will typically be a value indicating the edge weight, and in an unweighted graph, simply a boolean. In an undirected graph, the adjacency matrix will be diagonally symmetric, whereas in a directed graph, the values will be unique. The values on the diagnoal itself will be -1 unless there are edges which connect a node to itself.

We will use the following graph as our example.



For this graph, the described input would be:

```
4 4
0 1 5
0 2 1
0 3 3
1 2 0
```

and the adjacency matrix is:

	0	1	2	3
0	-1	5	1	3
1	5	-1	0	-1
2	1	0	-1	-1
3	3	-1	-1	-1

Let's see input this code:

Listing 4.1: C++

```
// easier to statically allocate
// large-enough array than
// to use vector<vector<int>>
int am[10000][10000];
int n,m;
cin>>n>> m;
for(int i=0;i<10000;i++)
  for(int j=0;j<10000;j++)
    am[i][j]=-1;
for(int c=0;c<m;c++){
  int i,j,k;
  cin>>i>>j>>k;
  am[i][j]=k;
}
```

Listing 4.2: Java

```
int n=in.nextInt(),m=in.nextInt();
int[][] am = new int[n][n];
// Be sure to distinguish between
// no edge and zero-weight edge
for(int[] i:am)Arrays.fill(i,-1);
for(int i=0;i<m;i++)
  am[in.nextInt()][in.nextInt()]=
    in.nextInt();
```

Iterating over each edge from a node is easy.

```
void iterate(int i){
  for(int j=0;j<n;j++)if(am[i][j]!=-1){
    // do something with am[i][j]
  }
}
```

4.2.2 Adjacency List

One may notice that for an adjacency matrix, we used $O(n^2)$ space. This also means if we needed to, iterate over all edges emenating from a node, we would take $O(n)$ time. While these are natural complexities in a desnse graph, it is not very practical in a sparse one. In the latter case, not only is it a waste of memory to store a whole bunch of -1's, but a waste of time to iterate over all those potential edges which don't exist. It would be more efficient to only store edges which actually do exist.

To cope with this problem, we use a more efficient structure, an adjacency list. An adjacency list stores only actual edges, instead of every potential edge, as an adjacency matrix does. It does this by taking each node in the graph, and storing a pair of integers for each edge which originates at that node, indicating destination node and weight. It typically uses a map to perform this lookup, so while the efficiency for each individual edge lookup is slightly raised from traversing the map, it is more than made up for by not having to iterate through non-existent edges.

Let's see how we would input the same data as the previous section. Note that we must store the edge in both directions as the graph is undirected. Also note that in cases where the graph is unweighted, we may be able to use an unordered map or HashSet instead of the map equivalents.

Source Node	(Destination Node:Weight)
0	(1:5), (2:1), (3:3)
1	(0:5), (2:0)
2	(0:1), (1:0)
3	(0:3)

Listing 4.3: C++

```cpp
int n,m;
cin>>n>>m;
unordered_map<int,int> al[m];
for(int c=0;c<m;c++){
   int i,j,k;
   cin>>i>>j>>k;
   al[i].insert({j,k});
   al[j].insert({i,k});
}
```

Listing 4.4: Java

```java
int n=in.nextInt(),m=in.nextInt();
ArrayList<HashMap<Integer,Integer>>
   al=new ArrayList<>();
for(int i=0;i<n;i++)
   al.add(new
      HashMap<Integer,Integer>());
for(int c=0;c<m;c++){
   int i=in.nextInt(),
      j=in.nextInt(),
      k=in.nextInt();
   al.get(i).put(j,k);
   al.get(j).put(i,k);
}
```

Iterating over each edge is still straightforward:

Listing 4.5: C++

```cpp
void iterate(int i){
   for(auto& j:al[i]){
      // do something with j
   }
}
```

Listing 4.6: Java

```java
void iterate(int i){
   for(int j:al.get(i).keySet()){
      // do something with
      // al.get(i).get(j)
   }
}
```

Except in very specific circumstances, we will find that the adjacency list representation is the more convenient to work with.

4.2.2.1 Multiple Edges

In cases where we might have multiple edges between pairs of graphs, we can adjust the above constructs to accomadate. Specifically, instead of storing a single integer distance as the value in the map, we can store a list of integers, one for each edge between pairs of nodes. Alternatively, in C++, we have the option to use a multimap, enabling us to directly store each destination/distance pair.

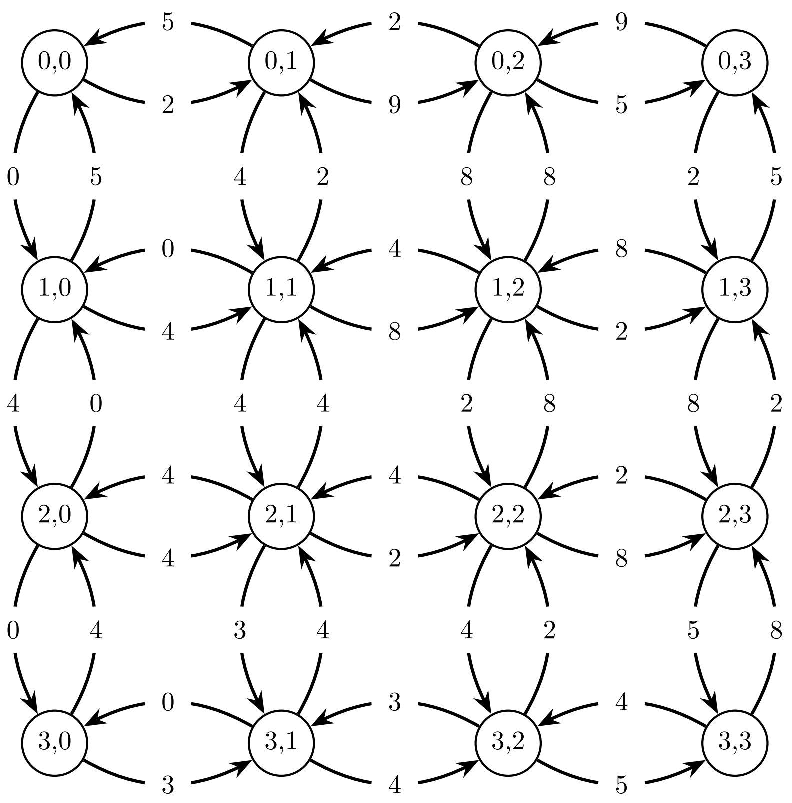
4.2.3 Implied Graphs

While the above are the canonical graph representations, there are comomn cases where it does not make sense to store the graph at all. This is common when the nodes given exist as x, y coordinates. Consider:

> The first line of input contains 2 integers, r and c, the number of colums and rows in the graph. $0 < c, r < 10^3$. Following this are r lines, each containing $c$integers d, where $0 <= d < 100$ and the i-th number in the j-column represents the cost to enter the node at the i-th row and j-th column. One may enter a node d fron any of the 4 cardinal directions.

The input and graph would look as follows:

```
4 4
5 2 9 5
0 4 8 2
4 4 2 8
0 3 4 5
```



While we can feasibly represent the graph, it is a bit cumbersome, and may become impractical in larger and denser graphs. Instead, we notice that the graph is exceptional formulaic, we can take advantage of the fact that the connectivity between two nodes is implied by the description itself. Given a node, say, $2, 3$, we do not need either of the adjacency structures to determine which nodes we have edges going to as it's simply the four cardinal direction. Given that, all we have to do is have a way to look up the weight of that edge, which is easy enough to do with an array.

We would read and iterate over this data structure as follows.

```
void input(){
   int d[1001][1001];
   int r,c;
   cin>>r>>c;
   for(int i=0;i<r;i++)for(int j=0;j<c;j++)cin>>d[i][j];
}

void iterate(int i_0, int j_0){
   // iterate over all cardinal positions which are in bounds
   for(int i=-1;i<2;i++)for(int
       j=-1;j<2;j++)if(i_0+i>=0&&i_0+i<r&&j+0+j>=0&&j+0+j<c&&i*j==0&&i!=j){
```

```
      // do something on d[i_0+i][j_0+j]
   }
}
```

We can easily extend the iterate function to operate on all directions including diagonals:

```
void iterate(int i_0, int j_0){
   for(int i=-1;i<2;i++)for(in
       j=-1;j<2;j++)if(i_0+i>=0&&i_0+i<r&&j_0+j>=0&&j_0+j<c&&xi!=0||j!=0){
      // do something on d[i_0+i][j_0+j]
   }
}
```

4.2.3.1 ASCII Grids

A common use of this type of graph involves inputting an ASCII grid which defines a ”map” of sorts, where the character at a given position in the input defines some property of the node at that position. Consider the following input, which might describe a room bounded by walls (X), with a single door (D), and treasure located at various locations (T). The problem might ask us, say, to find the shortest path from the door that visits all the treasure locations.

```
XXXXXXXXXDXXX
X   T       X
X         T X
X     T     X
X   T       X
XXXXXXXXXXXXX
```

We can input the data as follows, and then use it to process the data as an implied graph, which in this case is usually unweighted.

Listing 4.7: C++

```
string d[rows];
for(int i=0;i<rows;i++)
   cin>>d[i];
// do whatever on d[x][y]
```

Listing 4.8: Java

```
char[][] d=new char[rows][];
for(int i=0;i<rows;i++)
   d[i]=in.nextLine().toCharArray();
// do whatever on d[x][y]
```

4.2.4 Linked Data Structures

Consider a situation where a graph we seek to encode forms a *rooted* tree. This is a tree which has a hierarchy such that one node is the ”entrypoint” to the tree, and all edges and nodes arise therefrom. This is what is often colloquially called a tree, and seen in, say, binary search trees and heaps.

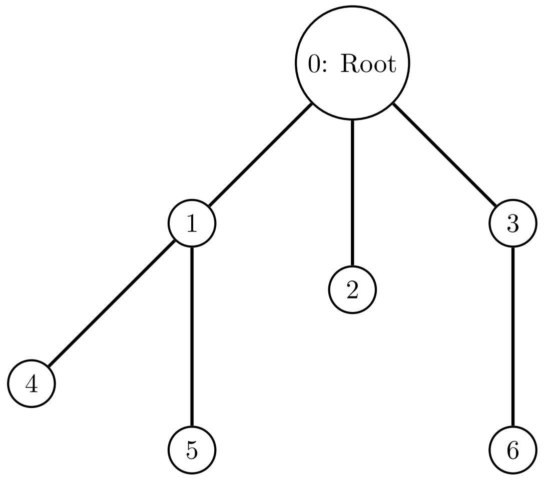


Figure 4.7: A rooted tree

In this graph, we can see that the root has *children*, in this case 1, 2, and 3. And nodes 2 and 3 also have children of their own. Nodes which have no children are known as *leaf* nodes.

This hierarchical nature provides the structure we need to encode this graph in code. We will consider two cases.

4.2.4.1 Binary Trees

In a binary tree, we have generally three pieces we might want to encode for a node, parent, left child, and right child. While in introuctory classes we might create a struct such as this:

```
class node{
   node ancestor;
   node left_child;
   node right_child:
}
```

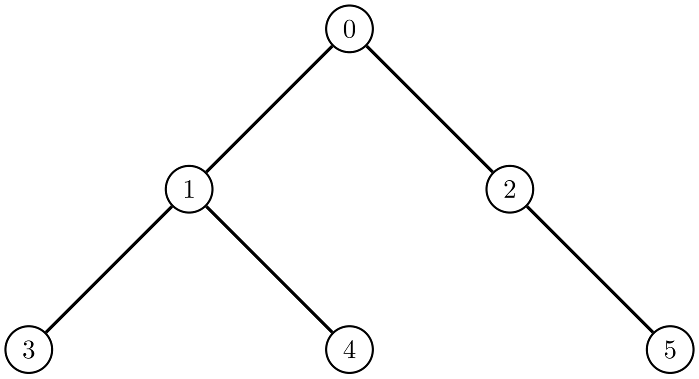
We can instead simply store the data in an array. Consider:

> Input begins with a line containg N, the number of nodes. Following are N lines, each containing 3 integers, the ancestor, left and right child of node i. If an ancestor or child does not exist, -1 is input. It is guaranteed that the input forms a rooted binary tree and $0 < n < 1000$.

The input, tree, and array structure would appear as follows.

```
6
-1 1 2
```

```
0 3 4
0 -1 6
1 -1 -1
1 -1 -1
2 -1 -1
```



index	parent	left child	right child
0	-1	1	2
1	0	3	4
2	0	-1	6
3	2	-1	-1

We can input this data as follows:

```
int d[1000][3]; // parent, left child, right child
int n;
cin>>n;
for(int i=0;i<n;i++)for(int j=0;j<3;j++)cin>>d[i][j];
```

4.2.4.2 General Trees

In general trees, we do not have a fixed number of children, but may have multiple children. Consider:

> Input begins with a line containg N, the number of nodes. Following are N lines, each describing a node. Each line starts with an integer M, indicating the number of descendencts of this node, followed by M integers, indicating a child. It is guaranteed that the input forms a rooted tree and $0 < N < 1000$.

The input is similar to the binary case, however we have to store an arbitrary number of children. We can do this by replacing the fixed array columns which held the two children with a list (or set if we need $O(1)$ lookup).

```
6
3 1 2 3
2 4 5
0
1 6
0
0
0
```

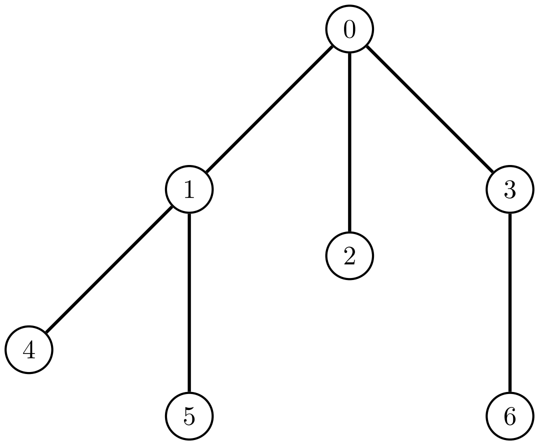


Figure 4.8: A rooted tree

index	parent	children
0	-1	1, 2, 3
1	0	4, 5
2	0	
3	0	6
4	1	
5	1	
6	3	

We can input this data as follows:

```
int n;
cin>>n;
int p[1000];//parents
for(int i=0;i<1000;i++)p[i]=-1;
vector<int> c[1000];//children
for(int i=0;i<n;i++){
   int m;
   cin>>m;
   for(int j=0;j<m;j++){
      int l;
      cin>>l;
      c[i].push_back(l); //set child
      p[l]=i; //set parent
   }
}
```

This method of using two separate arrays/lists is common and can be used to store arbitrary data without the overhead of explicitly creating classes or other custom structures.

4.2.5 Advanced Representations

In some of the more advanced algorithms here, it may be necessary to use structures which provide faster access than an adjacency list, and faster iteration than an adjacency matrix. We present two variants which might be useful.

4.2.5.1 Map-less Adjacency Lists

While the linear iteration over edges and constnat time lookup of the presented adjacency list using a map, in practice, a map structure is slow. The lack of data locality and hashigh require significant[1] time which may impact the viability of certain algorithms, if even it does not impact the runtime complexity. While the use of a map is convenient in that it enables us random lookup of any edge in the graph, in many case we only need to iterate over the edges. We can take advantage of that by using a vector instead of a map. In that vector, we will store a pair representing the destination node as well as the distance.

Listing 4.9: C++

```
vector<int,int> al[SIZE];
```

As noted, we cannot perform an arbitrary lookup into this list, but we can easily iterate over edges touching a node. We also must be careful to not use the index into a given vector outside the context of that vector. For instance, even in an unweighted graph, `al[x][i]` will not necessarily equal `al[i][x]`.[2]

We will use such a structure in the implementation of Dinic's algorithm.

4.2.5.2 Compressed Adjacency Matrices

In cases where adjacency matrices are convenient due to their ability to perform arbitrary lookups, but in which we want to speed up the runtime in sparse graphs, we can compress the matrix. This can be done by creating a second matrix where we only store the indices where non-zero edges appear for a given node. This gives us the ability to perform random lookups at maximum speed

[1]but constant
[2]but `al[al[x][i].first][x] will`

as well as iterate by observing the minimal number of edges. One should note this is effectively an adjacency list, but leveraging the adjacency matrix for faster lookups than any native list would be able to provide. We provide code to generate the compressed matrix as well as to iterate through it.

Listing 4.10: C++

```
//assume adjacency matrix
int amc[SZ][SZ]; //the compressed adjacency matrix
int order[SZ]={}; //the number of edges touching each node
for(int i=0;i<n;i++)for(int
    j=0;j<n;j++)if(am[i][j]|)amc[i][order[i]++]=j; //create the
    compressed matrix
for(int i=0;i<order[K];i++)do_whatever(am[K][amc[K][i]]); //iterate
    through the edges of K
```

We will use such a structure in the implementation of Push-Relabel.

4.3 Algorithms

4.3.1 Search

One of the high level categories of algorithms applied on graphs are search algorithms. These algorithms perform a traversal of a graph in specific ways in order to obtain pieces of information about the graph, such as the distance between nodes. Such seaerches are often used as the building blocks for further algorithms.

4.3.1.1 DFS

Depth First Search is a traversal which visits all nodes by "going as far as possible" before trying other paths. It is the equivalent of how one might typically solve a maze, by going until a dead end is reached, and then backtracking and trying a different path. Walking through a DFS traversal, starting from node 0:

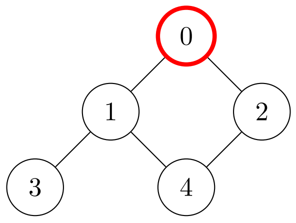


Figure 4.9: Begin at node 0.

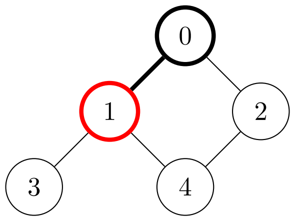


Figure 4.10: Pick any node adjacent to node 0 to traverse first. We arbitrarily choose node 1.

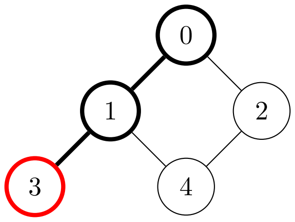


Figure 4.11: Pick any node adjacent to node 1 to traverse. We arbitrarily choose node 3.

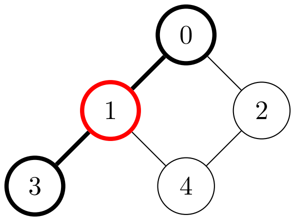


Figure 4.12: We reached a dead end at 3, so we "back up" to node 1 to see if we find any other paths.

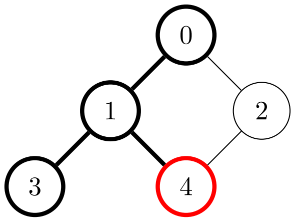


Figure 4.13: From node 1, we find another adjacent node, node 4, so we explore that path.

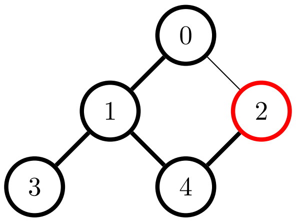


Figure 4.14: From node 4, we find we can continue further, to node 2.

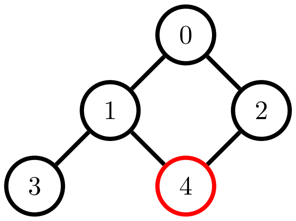


Figure 4.15: From node 2, we have an adjacent node, node 0, but we have already visited it. We are therefore at a dead end, and "back up" to node 4.

At this point, we will find no more outgoing paths from node 4, then node 1, and finally node 0, and the traversal will complete, having visited all nodes. This algorithm is typically presented recursively, with the following pseudocode.

```
void dfs(graph, current){
   mark current visited
   for each next adjacent to current in graph, if next not visited
      dfs(graph, next)
}
```

And finally, an actual implementations on adjacency lists. The java implementation follows similarly, but uses a `HashSet<Integer>` instead of an `unordered_set`.[3]

Listing 4.11: C++

```cpp
//assume adjacency list
unordered_set<int> v;

void dfs(int n){
   v.insert(n);
   for(auto& i:al[n])
      if(v.find(i)==v.end())
         dfs(i);
}
```

[3]The method of iteration over the Java adjacency list is covered earlier in this chapter.

One of the challenges of the above implementation is its recursive nature. Recursive solutions have two major challenges:

1. They can be slower, depending on how well the compiler can optimize them.
2. More importantly, they may exhaust stack space, leading to runtime errors.

While these issues may or may not exhibit in any given scenario, and often recursive solutions are elegant and will be correct, it is sometimes prudent to avoid recursion, or at least be aware of alternate solutions should the situation arise. In this case, the recursion is trivially replaced by a stack data structure. We look at this implementation both to demonstrate how to implement DFS non-recursively, but also as the overall structure of the implementation aligns very nicely with other graph search algorithms.[4]

Listing 4.12: C++

```cpp
//assume adjacency list
unordered_set<int> v;
stack<int> s;

s.push(start);
v.insert(start);
while(!s.empty()){
   int on=s.top();
   s.pop();
   for(auto& i:al[on])if(v.find(i)==v.end()){
      v.insert(i);
      s.push(i);
   }
}
```

As we look at each edge and node exactly once, the runtime is $O(e + v)$. DFS often isn't useful by itself, but is the basis for several other algorithms, including LCA and SCC, both of which are discussed later in this chapter.

4.3.1.2 BFS

Similar to DFS, *Breadth First Search* is a simple traversal of the graph. The big difference is the order in which the nodes are visited. While DFS attempts to traverse until a dead end is reached, in a "last in, first out" order, BFS visits nodes in the order which they are discovered, in a "first in, first out" order.

[4]In Java, the `top()` and `pop()` operations are combined into a single call.

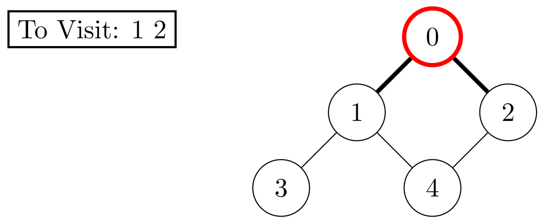


Figure 4.16: Start at node 0, we observe edges to 1 and 2.

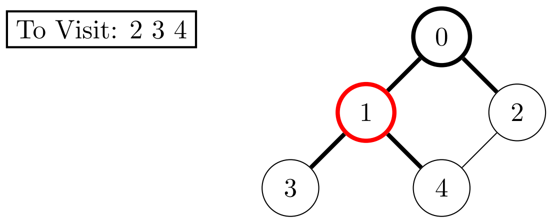


Figure 4.17: Process the next node in our ”to visit” list, 1. We observe edges to 3 and 4.

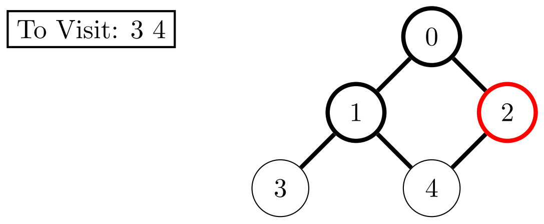


Figure 4.18: Process the next node in our ”to visit” list, 2. We observe edges, but they all are already noted to visit, so nothing to add.

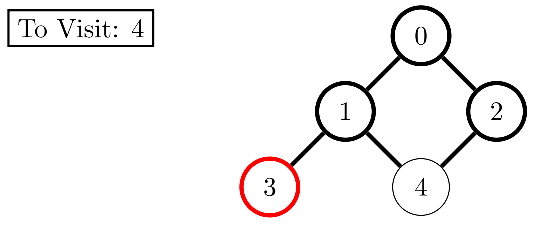


Figure 4.19: Process the next node in our ”to visit” list, 3. We observe edges, but they all are already noted to visit, so nothing to add.

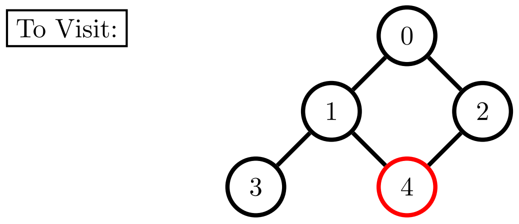


Figure 4.20: Process the next node in our "to visit" list, 4. We observe edges, but they all are already noted to visit, so nothing to add. The "to visit" list is empty, so the algorithm terminates, having seen every node and edge.

The implementation follows very nicely from the DFS implementation, except we use a FIFO queue instead of a LIFO stack:[5]

Listing 4.13: C++

```
//assume adjacency list
unordered_set<int> v;
queue<int> q;

q.push(start);
v.insert(start);
while(!q.empty()){
   int on=q.front();
   q.pop();
   for(auto& i:al[on])if(v.find(i)==v.end()){
      v.insert(i);
      q.push(i);
   }
}
```

Identically to DFS, we look at each edge and node exactly once, so the runtime is $O(e+v)$.

Flood Fill A common application of BFS is called *flood fill.* Flood fill is a process where we access and update every node in a graph which adheres to some property. Consider the following problem:

> Consider an ASCII grid of size N rows x M colums, consisting entirely of '#' and '.' characters. We will treat '#' characters as walls which may not be traversed, and '.' as open spaces. Return the number of contiguous areas of '.' which are fully enclosed by walls. Spaces are considered contiguous if they are connected via only '.' in the 4 cardinal directions.

[5]In Java, a `Queue<Integer> q=new LinkedList<>()` is similarly used, with the `offer()` and `poll()` methods.

```
5 7
..###.#
..#.#.#
###.###
#....##
######.
```

From a high level, there are two major hurdles to solving this problem:

1. Identifying which dots comprise a contiguous area. This can be accomplished by performing a BFS from a given dot, and "marking" visited nodes. In this case, we will "mark" the nodes by converting them to a '0' character, but any character could be used if we, perhaps, wanted to have a unique character for each contiguous space.

2. Identifying which contiguous areas are fully enclosed. We can see in the example input that there are at least three areas of dots which border the edge of the input, and thus not fully enclosed. We can resolve this problem by checking for dots along the edge of the input, and flood filling all dots to ensure we do not count them as contiguous areas.

We can now see a complete implementation of this problem. Be sure to review the earlier notes on operations on implied graphs. Aside from that, there are two major notes on the implementation of the `fill()` method:

1. As a node is identified by ah `r,c` pair instead of a single int, we use a queue of pairs instead of ints, to represent the nodes we need to visit.

2. Instead of explicitly maintaining a list of nodes we visited, we modify the graph itself (changing the character from '.' to '0') to imply that the node has been visited.

Listing 4.14: C++

```cpp
string d[100];
int n,m;

// fills all old chars with new chars
// This is just fancy BFS!
void fill(int r,int c, char oldc, char newc){
    queue<pair<int,int>> q;
   q.push({r,c});
    d[r][c]=newc;
    while(!q.empty()){
        pair on=q.front();
        q.pop();
        //iterates through all 4 cardinal directions,
        //checks if the new location is within bounds of the input,
        //and finally, if the character is correct for us to iterate to
        for(int i=-1;i<2;i++)
            for(int j=-1;j<2;j++)
                if(i*j==0&&i!=j&&
                   on.first+i>=0&&on.first+i<n&&on.second+j>=0&&on.second+j<m&&
                   d[on.first+i][on.second+j]==oldc){
            q.push({on.first+i,on.second+j});
            d[on.first+i][on.second+j]=newc;
        }
    }
}

int main(){
    cin>>n>>m;
    for(int i=0;i<n;i++)cin>>d[i];
    // fill the spaces along the four edges
    for(int i=0;i<m;i++){
        if(d[0][i]=='.')fill(0,i,'.','0');
        if(d[n-1][i]=='.')fill(n-1,i,'.','0');
    }
    for(int i=0;i<n;i++){
        if(d[i][0]=='.')fill(i,0,'.','0');
        if(d[i][m-1]=='.')fill(i,m-1,'.','0');
    }
    // fill and count any remaining spaces, as they must be fully
        enclosed
    int ans=0;
    for(int i=0;i<n;i++)for(int j=0;j<m;j++)if(d[i][j]=='.'){
        ans++;
        fill(i,j,'.','0');
    }
    cout<<ans;
}
```

For completeness, we also provide a Java implementation. Note the usage of `java.awt.Point` instead of `std::pair`.

Listing 4.15: Java

```
char[][] d;
int n,m;
Scanner in=new Scanner(System.in);

// fills all old chars with new chars
// This is just fancy BFS!
void fill(int r,int c, char oldc, char newc){
   Queue<Point> q=new LinkedList<>();
   q.offer(new Point(r,c));
   d[r][c]=newc;
   while(!q.isEmpty()){
      Point on=q.poll();
      //iterates through all 4 cardinal directions,
      //checks if the new location is within bounds of the input,
      //and finally, if the character is correct for us to iterate to
      for(int i=-1;i<2;i++)
         for(int j=-1;j<2;j++)
            if(i*j==0&&i!=j&&
               on.x+i>=0&&on.x+i<n&&on.y+j>=0&&on.y+j<m&&
               d[on.x+i][on.y+j]==oldc){
         q.offer(new Point(on.x+i,on.y+j));
         d[on.x+i][on.y+j]=newc;
      }
   }
}

void solve(){
   n=in.nextInt();
   m=in.nextInt();
   d=new char[n][];
   for(int i=0;i<n;i++)d[i]=in.next().toCharArray();
   // fill the spaces along the four edges
   for(int i=0;i<m;i++){
      if(d[0][i]=='.')fill(0,i,'.','0');
      if(d[n-1][i]=='.')fill(n-1,i,'.','0');
   }
   for(int i=0;i<n;i++){
      if(d[i][0]=='.')fill(i,0,'.','0');
      if(d[i][m-1]=='.')fill(i,m-1,'.','0');
   }
   // fill and count any remaining spaces, as they must be fully enclosed
   int ans=0;
   for(int i=0;i<n;i++)for(int j=0;j<m;j++)if(d[i][j]=='.'){
      ans++;
      fill(i,j,'.','0');
   }
   System.out.println(ans);
}
```

Distance in Unweighted Graphs Perhaps the most important application of BFS is its ability to compute distance in an unweighted graph. From an intuitive perspective, when we execute a BFS, we visit nodes in the order of closer to the start node to further. Looking at the earlier diagrams, we can see how the nodes and edges expand out from the start node.

Rough Proof It is straightforward to prove that nodes will always be visited in distance order. First, nodes with distance 1 are placed in the to-visit queue after processing the start node, and therefore all distance 1 nodes will be processed before any further node. During processing of distance 1 node, we will be able to observe all distance 2 nodes, and after processing all such distance 1 nodes, all such distance 2 nodes will be on the to-visit queue. This argument follows for any distance d and d+1.

Implementation In order to implement the distance algorithm, instead of simply encoding nodes we've visited as a set, we have an array of distances, which we initialize to some large number, representing infinity. We could check if a node was visited by seeing if it's distance is non-infinite, but an equivalent way is to check if the distance to that node via the current node is less than the previously recorded distance to that node. While we can prove that the only possible recorded distances to a node are infinite, or the minimum distance, coding the check as such enables other behavior, such as taking distinct actions if we find two paths which are of equal distance (which would be impossible if we were simply comparing with infinity).

Listing 4.16: C++

```cpp
//assume adjacency list
vector<int> d(n, INT32_MAX); // distance to each of n nodes.
queue<int> q;

q.push(start);
d[start]=0;
while(!q.empty()){
   int on=q.front();
   q.pop();
   for(auto& i:al[on])if(d[on]+1<d[i]){
      d[i]=d[on]+1;
      q.push(i);
   }
}
```

When execution of the above is completed, the distance vector, d, will contain the distances to all nodes, or INT32_MAX for disconnected nodes. The algorithm is trivially modified for java by using an array instead of a vector, and Integer.MaxValue instead of INT32_MAX. Note that this implementation works equally well in directed and undirected graphs.

Path Recovery A common extension to distance problems is to not only ascertain the distance to a given node, but the node-by-node construction of the actual path, or the "directions" to get from the start node to another node along the shortest path. This problem is known as *path recovery*. This is accomplished with a straightforward modification to our algorithm Instead of just noting the distance to a node, we also note the node along which we found the path. We can then construct the path by walking from the destination node back to the start node.

Listing 4.17: C++

```
//assume adjacency list
vector<int> d(n, INT32_MAX);
vector<int> p(n); // previous node on the path to this node
queue<int> q;

q.push(start);
d[start]=0;
p[start]=-1; // start node has no previous node
while(!q.empty()){
   int on=q.front();
   q.pop();
   for(auto& i:al[on])if(d[on]+1<d[i]){
      d[i]=d[on]+1;
      p[i]=on;
      q.push(i);
   }
}

list<int> path;
int on=destination;
while(on!=-1){
   path.push_back(on);
   on=p[on];
}

// path is built backwards, so reverse it to get the actual path
path.reverse();
```

Count of Shortest Paths Another common extension to the shortest path problem is the count of shortest paths to a node. In other words, if there are multiple paths to a node which all have a length equal to the shortest distance to that node, how many are there?

As with the path recovery, we note an additional piece of information for each node. In this case, it is the count of shortest paths. In order to set this value, when we find a shortest path to a node, we increment the count of shortest paths by the number of paths we found to our current node. If we later find a different path to some node whose distance is equal to the shortest path we had previously found to that node, we also perform this increment.[6] In that case, however, we do not need to add that node to the to-visit queue, as it will already have been placed there by the first path we found. Note that the nature of BFS, and as a corollary to the above rough proof, guarantees that all shortest paths to a node will have been found before a node is popped off the queue.

[6]this is where it is useful to perform an explicit distance check, instead of comparing against INT32_MAX

Listing 4.18: C++

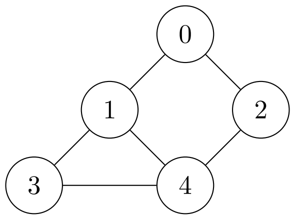
```
//assume adjacency list
vector<int> d(n, INT32_MAX);
vector<int> c(n, 0); //count of shorters paths
queue<int> q;

q.push(start);
d[start]=0;
c[start]=1; // Single degenerate way to reach the start node
while(!q.empty()){
   int on=q.front();
   q.pop();
   for(auto& i:al[on])if(d[on]+1<=d[i]){ // shorter OR equal length path
      c[i]+=c[on]; // add to our current count of found paths
      if(d[on]+1<d[i]){ // perform the usual work if it was previously
          unvisited
         d[i]=d[on]+1;
         q.push(i);
      }
   }
}
```

Distance to a node BFS will return us the distance from a start node to all other nodes. But what if we must find the distance from every node to a single node? This is the same question if a graph is undirected, but what if it is directed? Executing BFS from every node, or running Floyd-Warshall may be prohibitively slow. Instead, we can reverse the direction of all the edges in the graph before executing BFS on the modified graph.

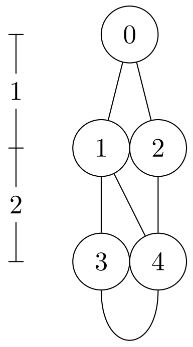
The most straightforward way to do this is to create a second adjacency list, walk all edges in the original list, and add the reversed edge into the new list, before executing BFS from the target node against the reversed list.

A Physical Explanation While the term *shortest path* is rellatively straightforward in terms of what it defines, we can also consider this value as the exact distance between two nodes; not the shortest distance, but the exact distance. For instance, one would say that 2400 miles is the shortest distance from New York to Los Angeles, one would just say it's the distance. While this distinction is sensible for cities, how does it apply to graphs?

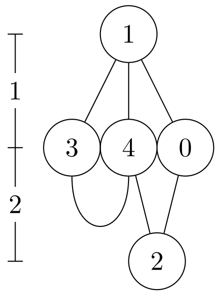
Lets take the following graph, a slightly modified version of the example from earlier:



If we imagine the edges are strings[7] of length 1, and dangle the graph via the start node, 0, the physical distance between the start node[8] and each other node will be the distance between those nodes in the graph, and therefore there will be some path to that node whose length is that distance. Furthermore, any string which is taught in this dangled graph will be on some shortest path, and any string which is loose will not. The distance is labeled via a scale on the left hand side.



Or, perhaps, if we want the distances from node 1:

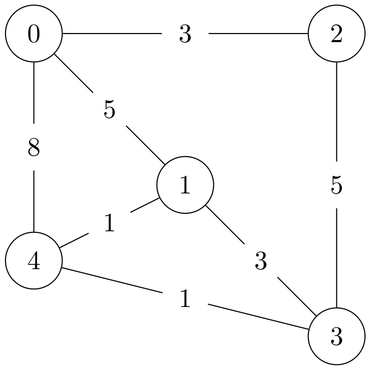


Understanding this physical manifestation is useful in further contexts, such as with weighted graphs and negative edges, and can sometimes help when trying to map problems in the physical world to algorithms.

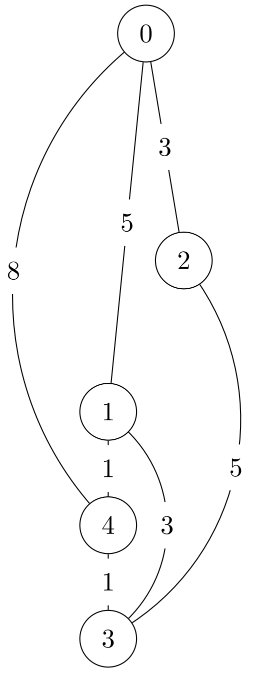
[7]physical strings, not a string of characters. Actually yarn, as you would, say, knit with
[8]assuming the size of each node is infintesimally small

4.3.1.3 Dijkstra

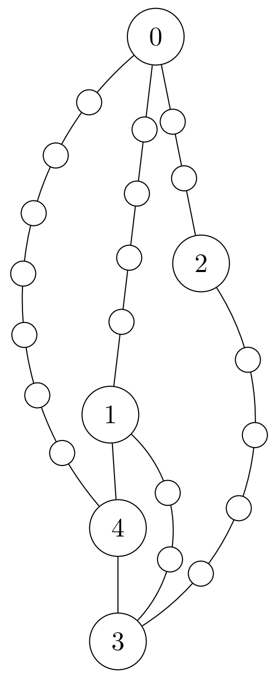
Having addressed distance in an unweighted graph, the next natural question is how we address distance in a weighted graph. One way we can trivially accomplish this is by converting the weighted graph into an unweighted one. This is done by adding nodes along each edge ($n-1$ nodes for an edge of length n) to split each edge into length 1 chunks. We can then apply BFS and extract the distance to each of the original nodes. Consider the following weighted graph:



First, we'll arrange the graph like before, as if "dangled" from node 0.



And finally, we convert to an unweighted graph. Note that with the addition of the new nodes, the length of every edge is 1.



We can now BFS through the transformed graph.

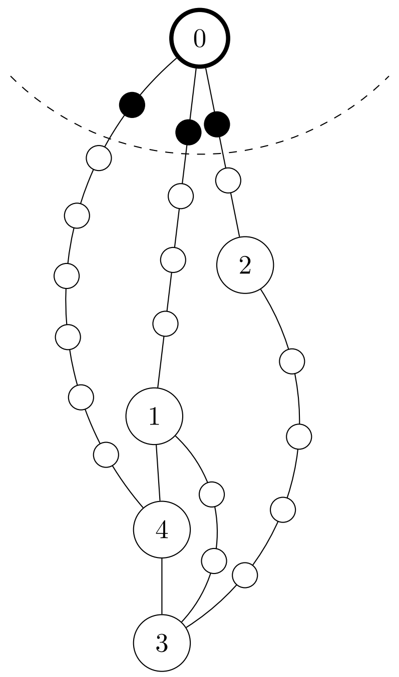


Figure 4.21: After processing all 3 distance nodes of distance 1. We'll draw a dashed line to indicate how far along we've progressed.

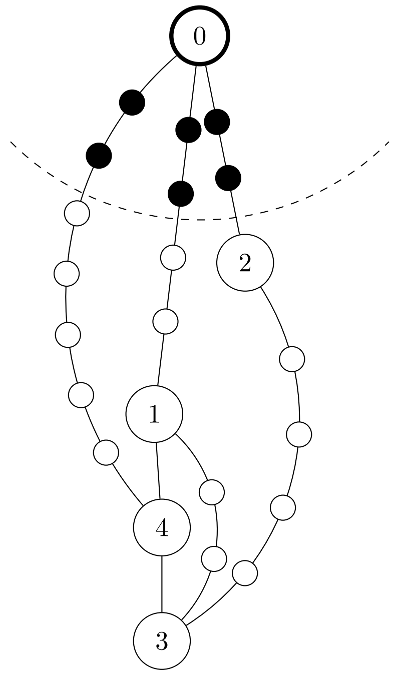


Figure 4.22: After processing a further 3 nodes of distance 2

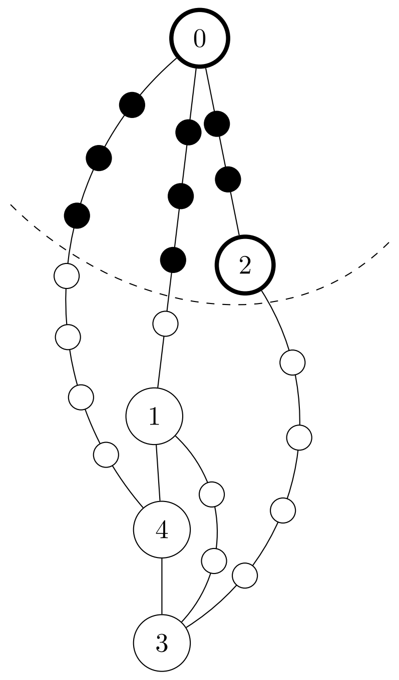


Figure 4.23: After processing nodes of distance 3, we have finally found a node in the original graph, and confirmed its distance as 3.

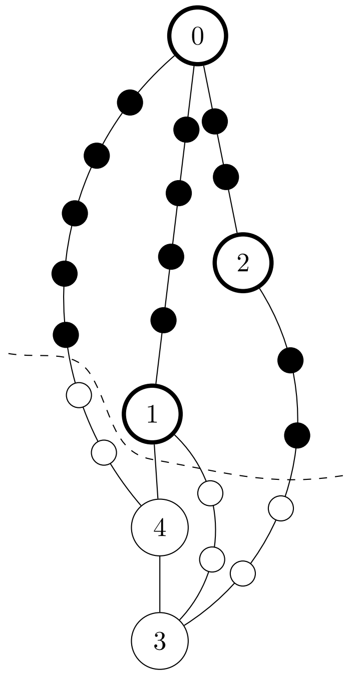


Figure 4.24: After processing a further 6 nodes of distance 4 and 5, we find another node from the original graph (1), and confirm its distance as 5.

The rest is the same, and we'll be able to find the distances of the other nodes. We note, though, some critical things:

- We did a lot of work before we even reached our first "real" node, visiting 9 nodes before finding node 2 which was just the first node.
- While the performance of BFS is quite good in general, $O(e + v)$, we have created a lot of extra nodes in this algorithm, namely $O(v + e * l)$, where l is the maximum length of an edge. While BFS in the original graph only had to visit 5 nodes, BFS in the transformed graph had to visit 25.
- As a consequence of BFS, the real nodes are all visited in the order of their distance from the start.

We're doing a lot of extra work, and an algorithm which depends on edge weights often does not perform well. The goal will be to accelerate the BFS in such a way that we aren't forced to wade through a huge number of fake nodes in order to reach the real nodes.

To accomplish that, we take a closer look at the third bullet point above: *nodes are visited in the order of their distince from the start.* Can we leverage this to determine which node we will visit next?

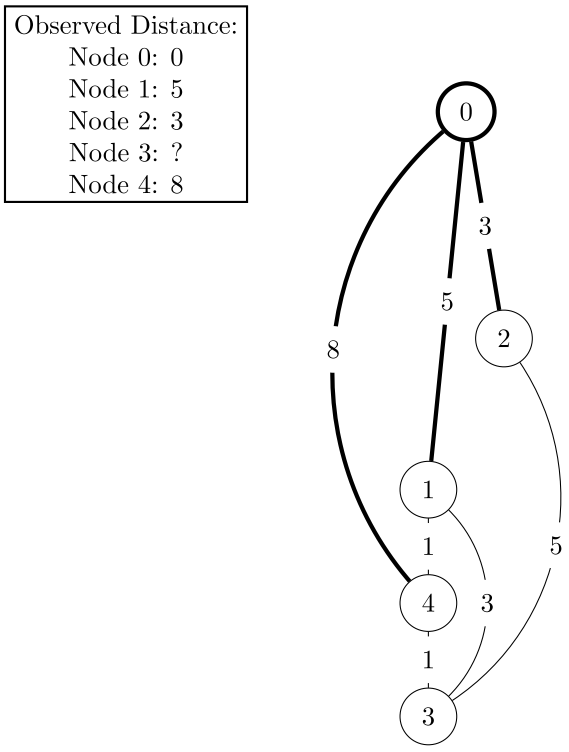
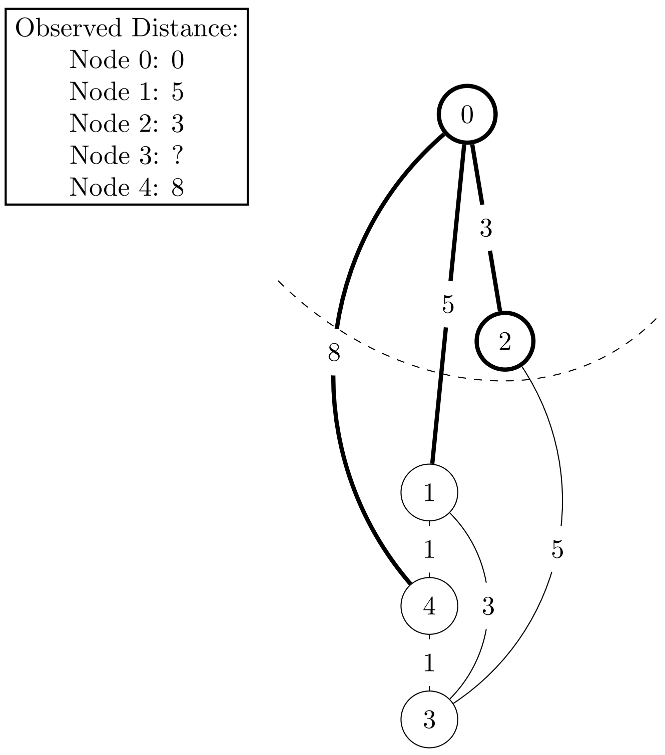


Figure 4.25: From the start node, we can observe 3 other nodes, and we note these in our observed distance table.



Observed Distance:
Node 0: 0
Node 1: 5
Node 2: 3
Node 3: ?
Node 4: 8

Figure 4.26: Based on the earlier assertion that nodes are visited in order, we can know necessarily that node 2, with distance 3, will be the first that the BFS would visit. We can therefore skip straight ahead to visiting node 2, without having to BFS through 9 nodes first. We can imagine we accelerated the dotted line straight to the first real node in a single step.

The question then becomes, how can we figure out which node will next be visited by the BFS, and skip straight to there? Our earlier assertion is we visit the nodes in order, and based on the distances we originally observed from the start node, this would be node 1, with distance 5. But what if some other node incident to 2 would be closer to the start than is our current observation of node 1? In order to prevent this, we first check all nodes adjacent to 2 and update our observed distance table. In this case, we find node 3.

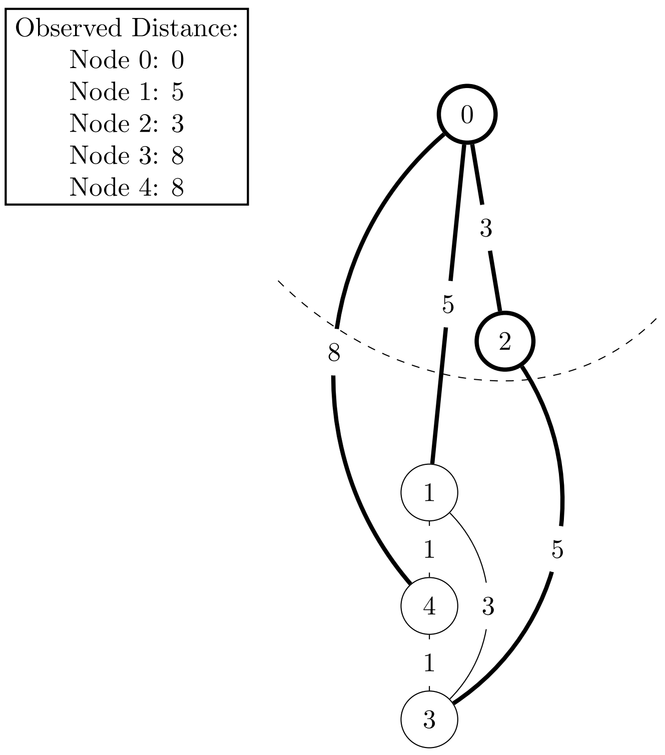


Figure 4.27: We find node 3 after searching the edges incident to 2, and update the observed distance table.

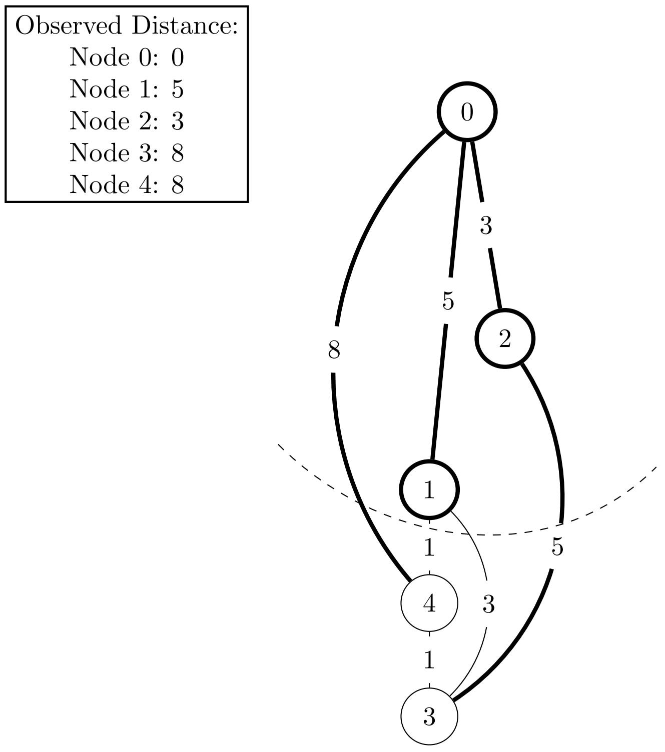


Figure 4.28: Now that we have all our affairs in order, we can say definitively that the next node that the BFS would visit after node 2 *is* in fact node 1 with distance 5. We have skipped BFSing through several extra nodes and advanced our dashed line.

Once again, to determine which node will be next in the BFS, we must check all edges incident to 1. In this case, we find we have a path to 4, via node 1, which is shorter (6) than any previusly found path (8). This means that in the BFS, we would reach node 4 along the path via node 1 earlier than we would have reached it via the longer, previoulsy found path. As such we update its distance. Note that the path to node 3 we find is identical to the previous distance we found (8), so no update is necessary.

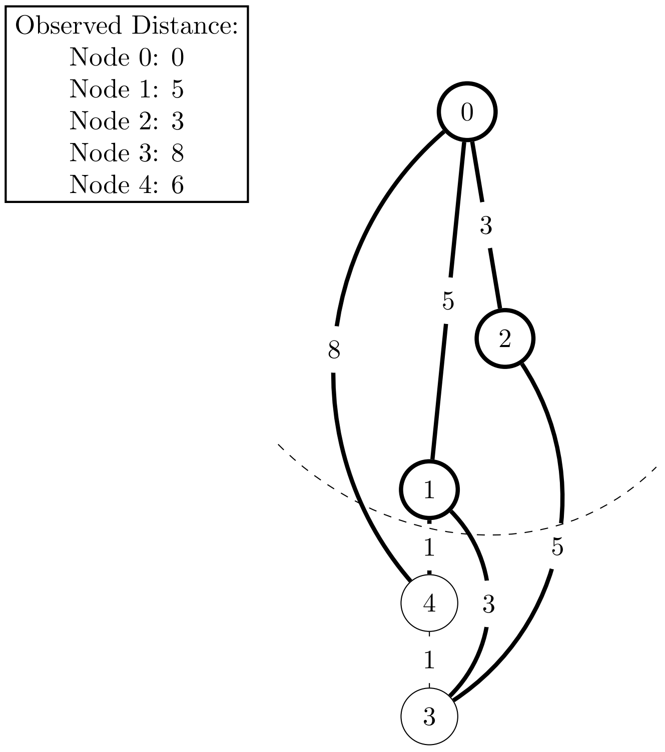


Figure 4.29: Node 4 is updated to the new shorted found path, and even though we found a path to node 3, it is the same as the one we already have, so it remains at distance 8.

With updates complete, we can now advance to the nearest node.

Observed Distance:
Node 0: 0
Node 1: 5
Node 2: 3
Node 3: 8
Node 4: 6

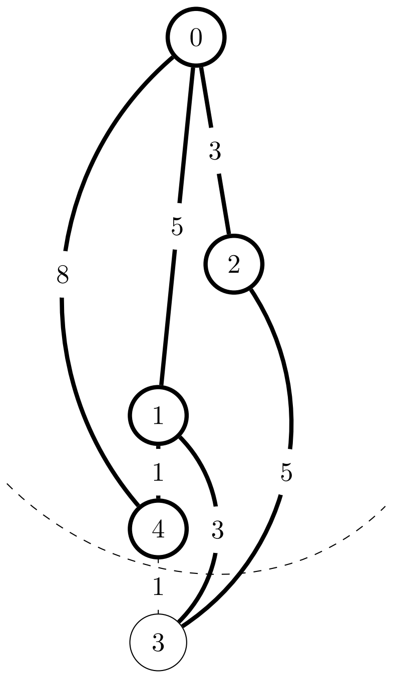


Figure 4.30: We find that node 4 is the next closest node, and therefore where the BFS would reach next, so we advance to node 4.

Once again, we will check if we must update our observed distances.

Observed Distance:
Node 0: 0
Node 1: 5
Node 2: 3
Node 3: 7
Node 4: 6

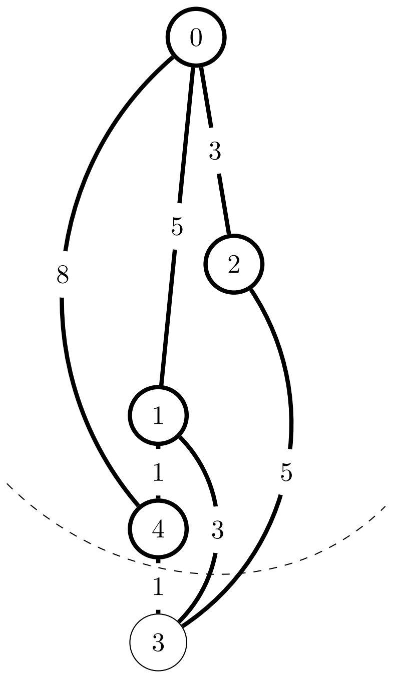


Figure 4.31: We have a shorter distance to node 3, so we update the observed distance table.

And one last time, we advance to the next closest node.

Observed Distance:
Node 0: 0
Node 1: 5
Node 2: 3
Node 3: 7
Node 4: 6

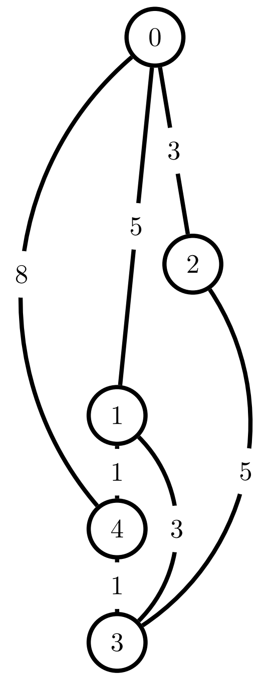


Figure 4.32: Node 3 is the last remaining, and thus closest node. We advance to it, and have completed our traversal. All node distances are now known correct.

Having visited all the nodes, we have effectively performed an accelerated BFS traversal which does not require the extra nodes, but still is able to maintain the shortest-path guarantees provided by BFS. This method of iteratively choosing the next closest node, then updating the best-known distances to all remaining nodes to ensure we properly can identify the next node is known as *Dijkstra's Algorithm*.

Rough Proof of Correctness While the proof follows nicely from the BFS proof, we can independely prove it correct. Dijkstra's algorithm chooses the closest known node to visit at each step. Suppose this does not result in the shortest distance to the node. This would mean there is some node closer to the root with a path to this node which is shorter than the one we traversed. As we know the previous node must have been visited, we also have observed the path from that node to this node, which means the shorter path would have been selected, leading to a contradiction. Therefore the path along which we first visit a node must be shortest.

Runtime In the course of the algorithm, we visit each node once, and observe each edge once. During the phase where we select the next closest node (which happens v times), we must observe $O(v)$ nodes in order to choose the best possible one. This leads to an overall runtime of $O(v^2 + e)$.

Optimizing in Sparse Graphs In dense graphs, the above runtime, quadratic in the number of nodes, is acceptable, as there are $O(v^2)$ edges anyway (and we must necessarily examine all edges in the graph). But in sparse graphs, where the number of edges is much less than this, the v^2 component dominates, so reducing it is of utmost importance. This can be done by using a priority queue to return the node which is next closest to the start. Instead of walking all nodes to find the next closest, in $O(v)$ time, we instead simply remove the first element from the priority queue, in $O(log(v))$ time. Adding to the queue, in case a new node is found, or updating a value in the queue, if we have found a better path, all take the same logarithmic time. We remove exactly v nodes from the queue over the course of the algorithm, and further, we only add or update at most once per edge in the graph. As such, the total runtime is $O(v + e + v * log(v) + e * log(v))$, or more succinctly, $O((v + e) * log(v))$, a significant savings over the quadratic solution. Note that in cases where the graph is desnse, this performance is $O(v^2 * log(v))$,[9] which is actually slower by a factor of $log(v)$.

Implementation The high level implementation of Dijkstra follows identically to the template defined by DFS and BFS, except instead of using a stack or queue, we use a priority queue to choose the next node to process. This priority queue can be implemented in one of two general ways:

1. In a true priority queue (which is generally implemented as a heap), there is no random access. As such, there is no ability to update or remove an existing element. To deal with this, we simply ignore the problem. If we must update the observed distance to a node in the course of the algorithm, we simply add a second entry for that node into the priority queue. The nature of the priority queue ensures that we still find the one with the smallest distance first, at which point we can ignore any future elements which reference that node. While this means there may be e elements on the queue instead of v, this does not impact the runtime.[10]

2. Alternatively, we can use an tree set, ordered by the distances to each node in the set. This structure does not enable us to have duplicate entries for a node,[11] but it does enable random access. Therefore, we can update the observed distances as we find them. we must complete this opteration in three steps:

    (a) Remove the node from the tree set

[9] because $e = O(v^2)$

[10] Remember, e is at worst $O(v^2)$ and $log(v^2) = O(log(v)$

[11] because it's a set

(b) update the observed distance

(c) Re-add the node to the tree set

Performing these steps out of order will lead to corruption of the set, and generally a broken algorithm. Performing twice as many logarithmic operations (by explicitly removing and then re-adding the node) does not change the overall runtime.

We will generally choose to implement the second of thsee two options in each case. Furthermore, implementations are trivially adaptable to use an adjacency matrix rather than an adjacency list if convenient, especially in cases with dense graphs.

C++ In C++, the tree set is implemented via simply the `set` container. The data we store in the `set` is a `pair` containing the distance to the node, and the node number *in that order*. This is critical. The ordering of `pair` by default is determined by the first element, and then the second as a tiebreak. As such, by placing the distance first, the `set` will intrinsically sort by distance, which is the intended and required behavior.

Listing 4.19: C++

```
//assume adjacency list
vector<int> d(n, INT32_MAX); // distance to each of n nodes.
set<pair<int,int>> pq;

d[start]=0;
pq.insert({d[start],start});
while(!pq.empty()){
   int on=pq.begin()->second;
   pq.erase(pq.begin());
   for(auto& i:al[on])if(d[on]+i.second<d[i.first]){
      pq.erase({d[i.first],i.first});
      d[i.first]=d[on]+i.second;
      pq.insert({d[i.first],i.first});
   }
}
```

The implementation is nearly identical to the BFS implementation, but with the priority queue instead of the queue. Similarly, implementations for path recovery, path count, and distances to a point all follow directly from the BFS equivalents.

Java The Java implementation is slightly more involved than the C++ one. This is due to the fact that Java will not automatically sort pairs in the same way. To account for this, we will be required to write a custom comparator for our set. This comparator will take a pair of integers, which represent node numbers under comparison, and return the one that has the lowest value in the distance array.

Critical Rules As noted in the basics chapter, there are critical rules which must be adhered to when using custom comparators which are especially critical when applying them to sorted sets. Here we recount those rules and indicate how they are applied when implementing Dijkstra.

1. The comparator should only return *equal* if the two objects are actually equal. In our case, this means if two nodes have equal distance from the start node, we *must* break the tie. This is typically done by using some arbitrary, but consistent, tiebreaketer, such as the node id. Without doing this, the set may identify the wrong element when attempting to remove[12] or add, causing the algorithm to be incorrect.

2. The comparator must adhere to the converse property. Namely, if `compare(a,b)` returns 1, then `compare(b,a)` must return -1. This is most commonly violated in cases such as are covered in rule 1, whereby a tie is broken in a way which is inconsistent.[13]

Listing 4.20: Java

```java
//assume adjacency list
final int[] d=new int[n];
Arrays.fill(d,Integer.MAX_VALUE);
TreeSet<Integer> pq=new TreeSet<>(new Comparator<Integer>(){
   public int compare(Integer a, Integer b){
      if(d[a]!=d[b])return Integer.compare(d[a],d[b]); //compare
          distance first
      return a-b; //otherwise, tiebreak!
   }
});

d[start]=0;
pq.add(start);
while(!pq.isEmpty()){
   int on=pq.first();
   pq.remove(pq.first());
   for(int i:al.get(on).keySet())if(d[on]+al.get(on).get(i)<d[i]){
      pq.remove(i);
      d[i]=d[on]+al.get(on).get(i);
      pq.add(i);
   }
}
```

[12] any node with the same distance, rather than the specific node we're looking for may be found without a tiebreak

[13] such as `return -1` in case of a tie, instead of a proper tiebreak, which will violate the converse property.

Dense Graph Solution As noted above, the solution in dense graphs does not use a priority queue, but instead uses the distance array directly. One caveat of this implementation is that we must be careful to not visit a node multiple times, so must also maintain context on which nodes have been visited. It should be noted that this implementation is not typically necessary, but may be useful in specific scenarios, such as online solutions where the topology of the graph changes during the course of the algorithm.

Listing 4.21: C++

```cpp
//assume adjacency list
vector<int> d(n, INT32_MAX); // distance to each of n nodes.
unordered_set<int> v;

d[start]=0;
while(v.size()!=n){
    int min=INT32_MAX,on=-1;
    for(int i=0;i<n;i++)if(v.find(i)==v.end()&&d[i]<min){
        on=i;
        min=d[i];
    }
    v.insert(on);
    for(auto& i:al[on])if(d[on]+i.second<d[i.first]){
        d[i.first]=d[on]+i.second;
    }
}
```

Warning: Dijkstra in Graphs with Negative Edges Dijkstra should generally never be used in graphs with negative edges. There are two reasons for this.

1. Dijkstra's algorithm has no mechanism to detect negative cycles. In a graph with negative cycles, the distance to any node reachable from the cycle is undefined. We could attempt to modify the algorithm to do detect such cycles, but there are better suited tools.[14]

2. Even without negative cycles, Dijsktra's algorithm is not guaranteed to find the correct distances. Recall one of the guarantees around which we based our proof of correctness: nodes are visited in the order of distance from the start node. In a graph with negative cycles, we may visit a node and then later find an incomiong edge with a large enough negative weight to lower the distance to this node. In order to be correct, we would have to then re-process the entirety of the sub-graph reached by edges outgoing from this node. While this would be handled by a priority queue based solution, we have removed all guarantees of performance. Again, there are

[14]Spoiler Alert: Bellman-Ford

better suited tools which are both correct and performant in the face of negative edges.[15]

4.3.1.4 MST

The *minimum spanning tree* problem asks what is the most efficient way to connect all nodes in a weighted graph. The solution for a given tree is a set of edges which satisfy the following properties:[16]

- The edges form a tree
- The graph is spanned, or more clearly, every node in the graph is connected
- The total weight of the selected edges is minimum over all sets of edges which satisfy the other properties

The minimum spanning tree of the following graph is highlighted with bold edges:

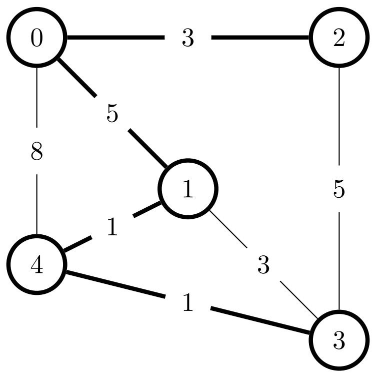


Figure 4.33: The sum of the edges in the MST is 10. There is no smaller way to connect all the nodes.

Note that while the paths which comprise the shortest distances from Dijkstra's algorithm also form a tree, that tree optimizes for the distance to a particular node, whereas the MST optimizes for the total distance required to connect all nodes. This problem arises often in the field, such as determining the minimum amount of wire to connect a set of locations,[17] and commonly arises in programming contests.

To develop an algorithm consider the following: if we have a single node of the graph, we know at least one edge from that node must be in the MST, but can we determine which one? The natural guess would be that the shortest edge. We can, in fact, prove such an edge must be in the minimum spanning tree.

[15]Spoiler Alert: Bellman-Ford
[16]which seem obvious in hindsight, when considering the name of the problem
[17]assuming fixed routes

Rough Proof Suppose the minimum weight edge (u, v) incident to node u is not in some MST. That means there must be some other path in the tree from u to v via other nodes.[18] If we examine the first edge on that other path, the one leaving u, one of the following must be true:

- The other edge has a weight greater than to (u, v), in which case, swapping in (u, v) for that other edge leaves us with a spanning tree of lesser weight, violating the assertion that the tree not containing (u, v) is minimum.
- The other edge has a weight less than the weight of (u, v), violating our assertion that (u, v) is the minimum weight edge leaving u.
- The weight of the two edges are equal, and swapping them yields distinct MSTs, violating our assertion othat (u, v) is not in some MST.

As all of these cases lead to a contradiction, edge (u, v) must be an edge in some MST.

Once this is established, we can treat the two connected nodes as a single, combined node, considering all the edges incident to either as incident to the combined node. Via the same proof, we can demonstrate that the minimum edge incident to the combined node (which would connect a new node) must also be in the MST. This process is repeated until we have found $n - 1$ edges, at which point, we have formed the MST. This algorithm is known as *Prim's algorithm.*

Implementation The implementation of the algorithm, whereby at each step, we select the minimum weight edge, should sound similar to the implementation of Dijkstra. In fact, it is almost identical! But for our priority queue, isntead of using the distance to the previous node plus the weight of the edge, we just use the weight of the edge itself. We also must track visited nodes to ensure we do not select the same node a second time after it has already been connected.

Listing 4.22: C++

```
//assume adjacency list
vector<int> d(n, INT32_MAX); //length of edge which connects node n
vector<bool> v(n, false);
set<pair<int,int>> pq;

d[start]=0; //Choice of start is arbitrary
pq.insert({d[start],start});
while(!pq.empty()){
   int on=pq.begin()->second;
   pq.erase(pq.begin());
   v[on]=true;
   for(auto& i:al[on])if(!v[i.first]&&i.second<d[i.first]){
      pq.erase({d[i.first],i.first});
```

[18] since the tree must be spanning

```
      d[i.first]=i.second;
      pq.insert({d[i.first],i.first});
   }
}
```

Returning the total weight of the tree simply involves summing up the elements of the distance array, and if the exact edges are required, the path recovery scheme used for Dijsktra and BST are equally applicable. Identically to Dijkstra's algorithm, Prim's runs in $O((v+e)*log(v))$. Furthermore, again like Dijkstras, in dense graphs, a flat distnace array may also be used in place of the priority queue.[19]

Edge-Based Implementation An alternate way of implementing Prim's algorithm involves directly placing edges directly on the queue instead of nodes. This has the benefit of producing the edges in the MST directly, instead of requiring path recovery, while the cost is a slightly more involved setup for the priority queue. Both methods are presented and are equally performant.

Listing 4.23: C++

```
//assume adjacency list
vector<bool> v(n, false);
set<pair<int,pair<int,int>>> pq; //now contains edges rather than nodes
list<pair<int,int>> e; //output

v[start]=true;
for(auto& i:al[start])pq.insert({i.second,{start,i.first}});
while(!pq.empty()){
    auto on=pq.begin()->second;
    pq.erase(pq.begin());
    if(v[on.second])continue; //might find this node multiple times
    v[on.second]=true;
    e.push_back(on);
    for(auto& i:al[on.second])if(!v[i.first])
      pq.insert({i.second,{on.second,i.first}});
}
```

Java While the node-based implementation is a straightforward translation to Java, if one has read the Dijkstra section, the implementation of the edge-based implementation is slightly trickier due to the complexities of the comparator. An implementation is provided here which operates on a graph of real, rather than integral, edge weights. Note that the tiebreakers in the comparator go two levels deep to ensure that we can properly distinguish the edges. Also note that the comparator depends directly on the adjacency list, which therefore must be declared `final`.

[19] A fibonacci heap reduces this slightly to $O(e+v*log(v))$.

Listing 4.24: Java

```
// assume "final" adjacency list
HashSet<Integer> v=new HashSet<Integer>();
ArrayList<Point> e=new ArrayList<Point>();
PriorityQueue<Point> pq=
  new PriorityQueue<Point>(al.size(),new Comparator<Point>(){
   public int compare(Point o1, Point o2) {
      double d=al.get(o1.x).get(o1.y)-al.get(o2.x).get(o2.y);
      if(d!=0)return (int) Math.signum(d);
      if(o1.x!=o2.x)return o1.x-o2.x;
      return o1.y-o2.y;
   }
});

v.add(start);
for(int i:in.get(start).keySet())pq.offer(new Point(0,i));
while(v.size()<al.size()){
   Point t=pq.poll();
   if(v.contains(t.y))continue;
   v.add(t.y);
   e.add(t);
   for(int i:in.get(t.y).keySet())if(!v.contains(i))
      pq.offer((new Point(t.y,i)));
}
```

Kruskal's Algorithm *Kruskal's algorithm* is a second algorithm for computing the MST. It is also greedy, has the same runtime, and has similar code complexity to Prim. It is included here for two reasons:

1. It is common enough in usage, as opposed to Prim, to warrant discussion
2. At least one truth of MSTs arises cleanly from the correctness of the algorithm

The algorithm itself, instead of selecting the shortest edge which extends a growing tree by one node, selects the shortest edge in the entire tree which joins two disconnected trees.[20] Pseudocode for the algorithm is as follows:

```
for all edges u,v in order of length:
   u0: find(tree containing u)
   v0: find(tree containing v)
   if u0 and v0 are different:
      add u,v to MST
      join(u0, v0)
```

[20]equivalently, does not form a cycle

The crux of this algorithm are the two functions, `join`[21] and `find`, which are provided in logarithmic time by the aptly named union-find algorithm, discussed later in this chapter.

One key fact arising from this algorithm is as follows:

> For any two spanning trees whose weight is minimum in some graph, then for any edge weight w, if the first tree has k edges of weight w, then the second tree must also have k such edges.

Perhaps more simply, the weights of the set of edges in different MSTs of the same graph are the same, even if the edges themselves differ. If there are four weight 7 edges in one MST, there must also be in every MST of that graph.

To understand why this must be true, we can observe Kruskal's algorithm. As it processes edges in order by weight, it will process those four weight 7 edges consecutively. This means during the processing of weight 7 edges, kruskal was able to find exactly four disconnected trees which can be connected[22] by weight 7 edges. Therefore, regardless of selection of edges, we know Kruskal's algorithm must join those four disconnected trees to some other tree, so there is no execution path which results in using fewer than four weight 7 edges. Applying this arugment over all edge weights, as well as acknowledging the fact that the sum of all edges must be the same in all MSTs, means that we cannot use more than that count either.

4.3.1.5 Other Graph Search Topics

Negative Weights As hinted in the discussion of Dijkstra's algorithm, better solutions exist than attemping to modify Dijkstra's algorithm. The algorithm which is that better solution is *Bellman-Ford.* From a high level, Dijkstra's algorithm works by examining only edges which it believes can update the distance array at each of v steps. This is provably only the edges which are incident to the last vertex we processed. Bellman-Ford, on the other hand, examines every edge at each of the v steps to see if it would update the distance array. This eliminates the necessity of the priority queue, or knowledge of which nodes have been visited.

Implementation Much of the implementation is straightforward, and works like Dijkstra, if we were to check every edge every step instead of having a specific node we are processing. One critical difference, however, is the instantiation of the distance array. In prior implementations, we used `INT32_MAX`. In this case, however, as we will be processing nodes before they may have a non-infinite distince,[23] we may attempt to add that infinite value to an edge weight, leading to integer overflow. To resolve this, we should instead initialize to something smaller, such as `INT32_MAX/2`.[24]

[21] also known as union

[22] not necessarily to eachother

[23] because we process every edge every step, not just ones we've already visited

[24] so long as it is larger than the max possible path length in the graph

Listing 4.25: C++

```
//assume adjacency list
vector<int> d(n,INT32_MAX/2);

d[start]=0;
for(int i=0;i<n;i++) //loop v times
  for(int j=0;j<n;j++)for(auto& i:al[j]) //check all edges
    if(d[j]+i.second<d[i.first]) //check and update distance
      d[i.first]=d[j]+i.second;
```

As the algorithm runs v loops over e edges, the runtime is very apparently $O(v+e)$. Note that the check inside the triple `for` loop, as well as the method in which the distance array is updated, is identical to the check and update performed in Dijkstra's algorithm, but without the priority queue to maintain. As with the other search algorithms, the algorithm is trivially extended to recover actual paths by storing a "previous" node along with the distance when the distance is updated.

Rough Proof of Correctness Consider all the edges which comprise the shortest paths to all nodes. Those edges form a tree, rooted at the node we started from. Within in that tree, the maximum number of edges between the root and any node is at most $n-1$.[25] Consider the inner loop of Bellman-Ford where we loop over all edges. After the first time through that entire loop, we are guaranteed to have found the shortest path to all nodes whose shortest path contain exactly one edge. After the second time, we are guaranteed to have found the shortest path to all nodes whose shortest path contains exactly two edges. Therefore, after $n-1$ loops, we are guaranteed to have found all shortest paths of any length, regardless if they contain negative weight edges.

Negative Cycle Detection As proven above, if a node has a defined distance from the start, it is guaranteed to be found in at most $n-1$ iterations. Therefore, if the distance array is modified in the n-th cycle, there must be a negative loop. In order to identify all nodes which have undefined distances due to a negative cycle, we can execute another n loops of the algorithm, and any node whose distance changes during any of those loops must have an undefined distance.

As large negative edge weights could add up quite quickly,[26] one must be sure to use a data type which is large enough to store the the largest, and smallest, values which might be seen during the entire $2*n$ loop algorithm.

A Physical Explanation In the BFS and Dijkstra discussions, we used a physical model of a graph, using strings, to help grasp the concept of distance. The question here is whether we can extend that to also represent negative

[25] otherwise there would be a loop in one of the paths
[26] given we initialize nodes to `INT32_MAX/2`

distances. A positive edge weight indicated, in a physical sense, that a node could hang below another node *no further* than the weight of that edge. A string perfectly encapsulates that behavior. In the case of a negative edge, a node must be above another node *at least* the weight of that edge. Or put another way, if node u has an edge of -1 to node v, then u must be at least 1 further from the start node than node v. This makes sense, as if u were to move closer to v, then the -1 edge would ensure that v could move up equivalently. The physical representation of this phenomenon is a spring. The spring may stretch infinitely, allowing v to be much closer to the start node than u, but it guarantees that node v is always at least some distance above u.

All Pairs Shortest Path While Dijkstra and Bellman-Ford enable us to get the distance from one node to all other nodes in a variety of situations, it is often the case that we neede the distance between arbitrary pairs of ndoes in a graph. While executing those algorithms v times, once for each start node, works, as those algorithms are typically dominated by the number of edges, this can become prohibitive. This is especially the case in dense graphs, or if the graph has negative weight edges. This problem is known as the all-pairs shortest path problem, and is solved using the *Floyd-Warshall* algorithm.

From a high level, the algorithm iterates through each node k once, and for each other pair of nodes, i, j, checks if the path between i and j can be shortened by going through k instead. Note that as this algorithm makes the most sense on dense graphs, it assumes an adjacency matrix. It further assumes that we can freely update that matrix to use it as the output.

Listing 4.26: C++

```cpp
//assume modifiable adjacency matrix
for(int k=0;k<n;k++)
  for(int i=0;i<n;i++)for(int j=0;j<n;j++)
    am[i][j]=min(am[i][j],am[i][k]+am[k][j]);
```

As the algorithm is simply a triple `for` loop over all nodes, the runtime is exactly $O(v^3)$.

Path Recovery Path recovery is slightly different from the other algorithms we have seen. While previously we had to construct the paths in reverse order, with Floyd-Warshall, we can do it directly. When we find a new shortest path, we update the `next` array to point to the same node as the sub-path along which we found the new shorter path.

Listing 4.27: C++

```cpp
//assume modifiable adjacency matrix
int next[n][n];
for(int k=0;k<n;k++)
  for(int i=0;i<n;i++)for(int j=0;j<n;j++)
    if(am[i][k]+am[k][j]<am[i][j]){
        am[i][j]=am[i][k]+am[k][j];
        next[i][j]=next[i][k];
    }
```

Rough Theory of Operation Consider any shortest path which has been found at the completion of the algorithm. Further, consider the nodes on that shortest path in reverse index order. Taking the highest such node on that path as k, we can see the values of i, j, and k when the overall distance for this path was set during the execution the algorithm. At that time, the two pieces of information we needed to set `am[i][j]` were `am[i][k]` and `am[k][j]`. We also know that the maximum node index on the shortest path either from i to k or from k to j must be less than k. Therefore, as k is the outermost loop variable in the algorithm, those values would have been computed previously. This paradigm applies recursively to generate all paths simultaneously.

Ultimately, the truth at the foundation of this algorithm is that for any path with n nodes, we will need at least n iterations of the outer loop to have built that path, and that path will finally be constructed when we reach the highest node index, m, on that path. Fortunately, on a path of n nodes, the highest index m must be at least n. Therefore, when we reach the m-th iteration of the outer loop and construct the final path, we have guaranteed to have executed the n iterations necessary to have built the requisite subpaths. In the end, all paths will be able to be constructed in one cubic pass.

4.3.2 Flow and Match

So far we have considered weights on a graph as the length of string connecting them. There is another useful physical representation though, in which we consider the edge weights as the amount of flow we can push through a pipe. In this case, we refer to the weight as *capacity* instead of distance, and it most directly can be considered the size of the pipe. This representation opens up a class of algorithms known as *network flow* algorithms, which solve a wide variety of problems on graphs.

4.3.2.1 Maximum Flow

Consider the following problem:

> Given a specification of a graph, and capacities for each edge, what is the maximum capacity which can flow from some node u to another node v?

This is known as the $maximumflow$, or maxflow. The node from which the flow originates is known as the *source*, and the node where the flow terminates is known as the *sink*. Let's look at a an example.

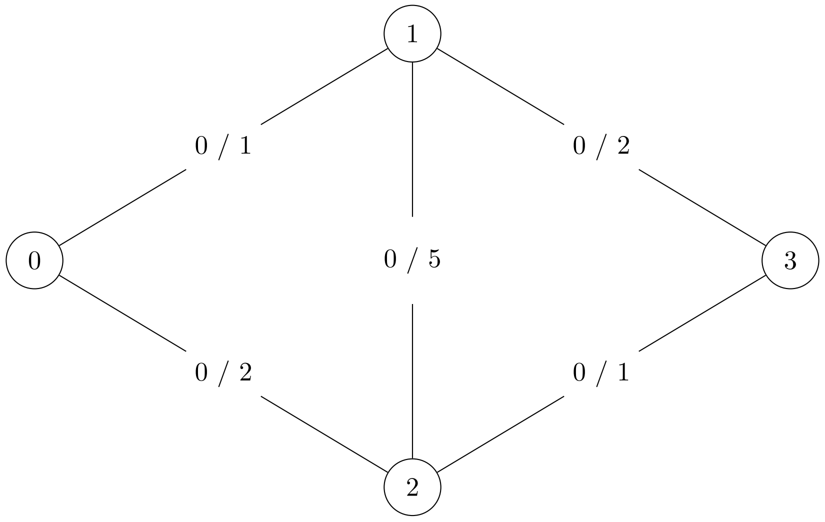


Figure 4.34: The lower number on each edge will represent capacity. The upper number represents the current flow along that edge. By convention, flow will go from left to right, so node 0 is the source, and node 3 is the sink.

It may take a second, but the maximum flow from $0 \to 3$ is 3. We can see why this is the case by looking at the edges which each unit of flow traverse.

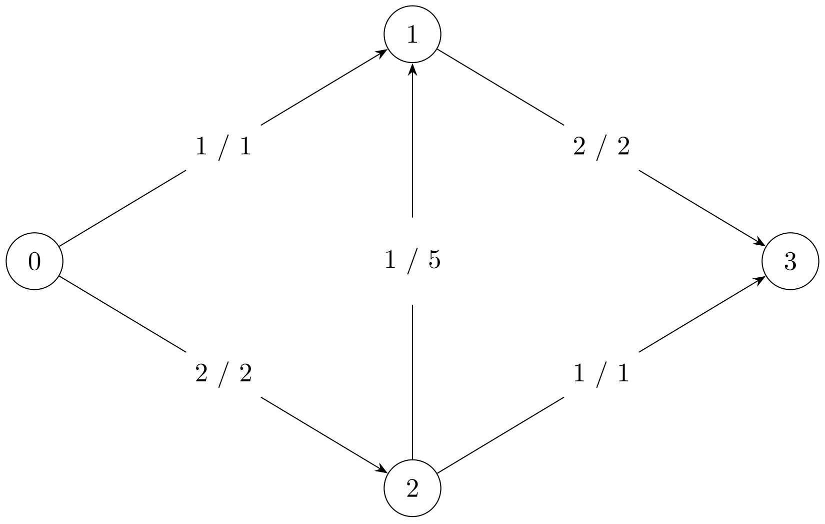


Figure 4.35: A flow of 1 is going from 0→1→3. A flow of 1 is going from 0→2→3. Lastly, a flow of 1 is going 0→2→1→3. Arrows are added to the indicate the direction of the flow along that edge.

High Level Idea To compute the solution to this problem, we take the following high level steps:

1. Find any path from the source to the sink on which we can push at least a single extra unit of flow.
2. Increase the flow along that found path as much as possible.
3. Repeat until no path is found.

This algortihm is known as the *Ford-Fulkerson* algorithm, and notably does not prescribe any particular method for finding such paths. Let's see it in practice, selecting paths using DFS. Each found path is known as a *augmenting path*, and therefore Ford-Fulkerson is known as an *augmenting path algorithm*.

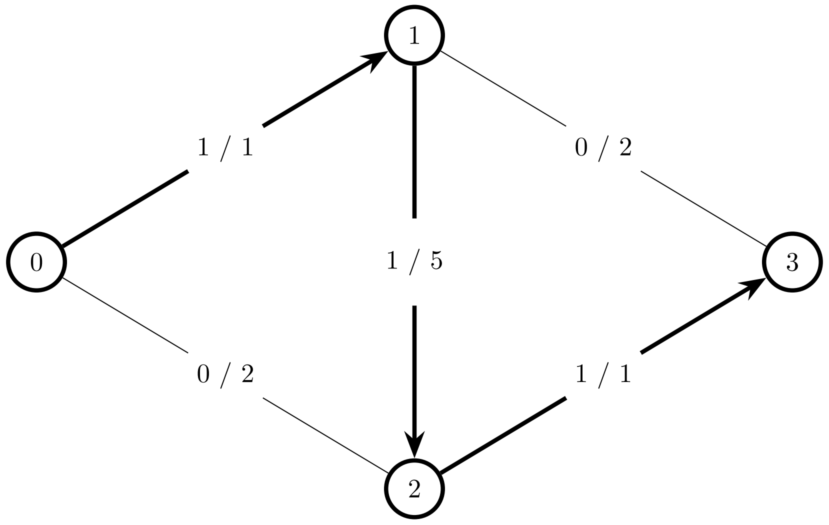


Figure 4.36: In the first iteration of the algorithm, we find path 0→1→2→3, along which we can push a total of 1 flow. We are limited by both the edges from 0 → 1 and from 2 → 3.

We have no reached an impasse. We know that the maximum flow is 3, but we have found ourselves stuck as there appear to be no more available paths on which we can push flow. The problem is that we pushed flow downward through the edge from 1 → 2, but we know in the optimal solution, there is actually a flow upward in that edge. In order to address this, we must have a mechanism to undo that "mistake". We do this by allowing our path finding algorithm to "cancel" flow which we had previously pushed along an edge. In our example, this means if we were to push a flow of 1 from 2 → 1, we would cancel out the incorrect downward flow of 1, setting it back to 0.

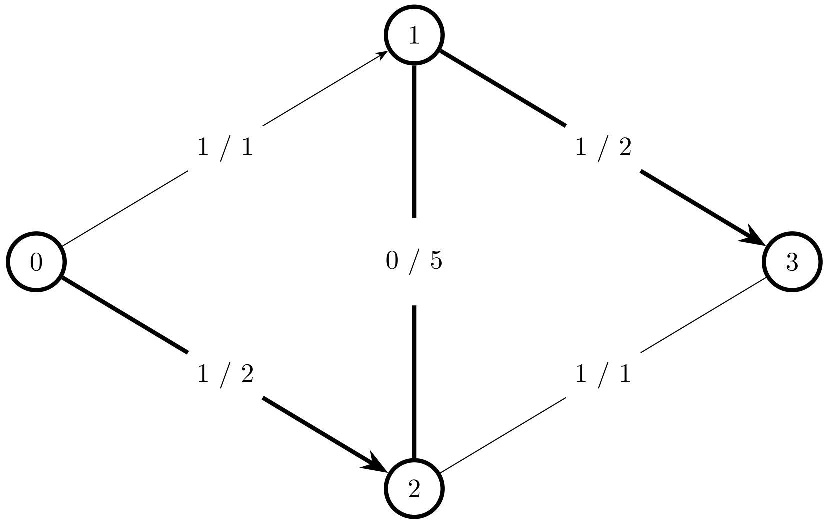


Figure 4.37: By allowing canceling of flow, we now have another path, from 0→2→1→3. We will 1 extra flow to cancel out the downward flow, returning that edge back to 0.

Lastly, with the middle edge neutralized, we realize we can push an additional flow along that same path, arriving at the optimal flow of 3.

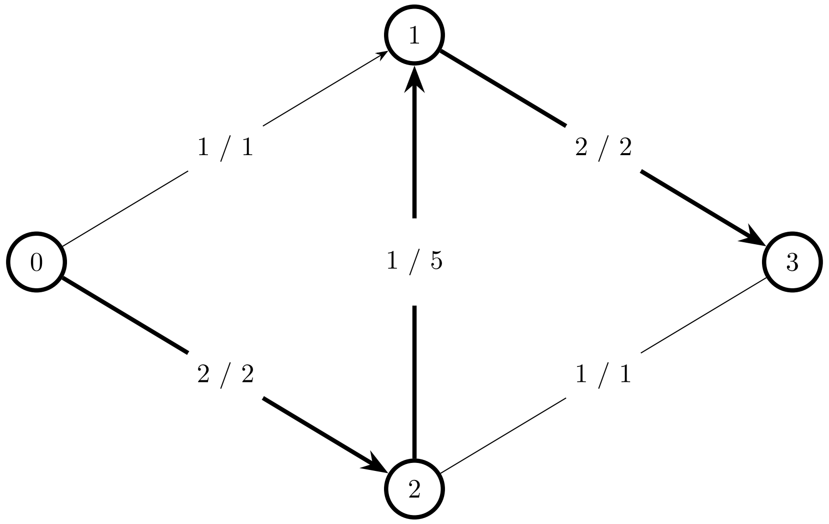


Figure 4.38: We find our final path and push the flow of one, limited by the edges from $0 \to 2$ and $2 \to 3$.

An astute observer could note that we can optimize and combine the last two steps, canceling out and reversing the flow in a single step. More clearly, we can determine the amount of extra flow we can push along an edge via the following two rules:

- If there is no flow on the edge, or the flow is in the same direction we wish to add more add flow, the amount of flow we can add is the total capacity of the edge minus the current flow on the edge.
- If the flow on the edge is in the opposite direction we wish to add flow, the amount of flow we can add is the total capacity of the edge plus the current flow on the edge.

This means for a given edge, the amount of extra flow we might be able to push on a given edge depends on the direction which we intend to traverse it. Consider the example after the first step when we had a flow of 1 going down the middle edge. If we intended to push more flow down along that edge, the total amount we would be able to push is $5 - 1 = 4$. However if we intended to push flow up the edge, we would be able to cancel out the current downward flow of 1 and reverse the direction up to an additional 5. The total amount we would be able to push is $5 + 1 = 6$. The amount of additional flow we are able to push from one node to another along an edge is known as the *residual*, and the maximum amount we are able to push along a given path is the minimum residual of any edge along that path.[27]

If we again reexamine our example after the first step, we find a single candidate path. The residual from $0 \to 2$ is 2, as there is no flow. As discussed, the residual from $2 \to 1$ is 6, and the residual from $1 \to 3$ is also 2. As the minimum residual is 2, we can push a maximum of 2 flow along this path.

Implementation When implementing the described algorithm, a common way to path find is BFS. When BFS is used, the algorithm is known as *Edmonds-Karp*. While DFS path finding algorithms may meander, examining many edges even if the source is close to the sink, BFS enables us to more tightly bound the time spent finding augmenting paths.

One of the trickiest parts of implementing maxflow in general is the proper handling of the bi-directional edges and cancellations. Encoding the current direction of an edge is not something that was concerning in the search algorithms described earlier in this chapter. We will present an intuitive way to handle this, followed by an optimization which greatly eases implemetation.

Splitting the edge A clean way to handle this is to split each undirected edge of capacity c into two directional edges in opposite directions, each with capacity c. Further, instead of being able to reverse the direction of a edge,[28] we are only able to cancel out the flow, and return the edge capacity back to 0. The maximum amount of capacity we can push between two nodes is thus the

[27] The set of edges with residual capacity is known as the *residual graph*.
[28] which is impossible, since the edge is directed now

sum of the remaining capacity in the edge directed in the same way we would like to traverse, and the magnitude of the current flow in the edge directed in the opposite direction. When we first push flow, we will first cancel out flow in the opposite direction before adding flow in the forward directed edge.

We can see how this works in our example, again assuming we start after step 1.

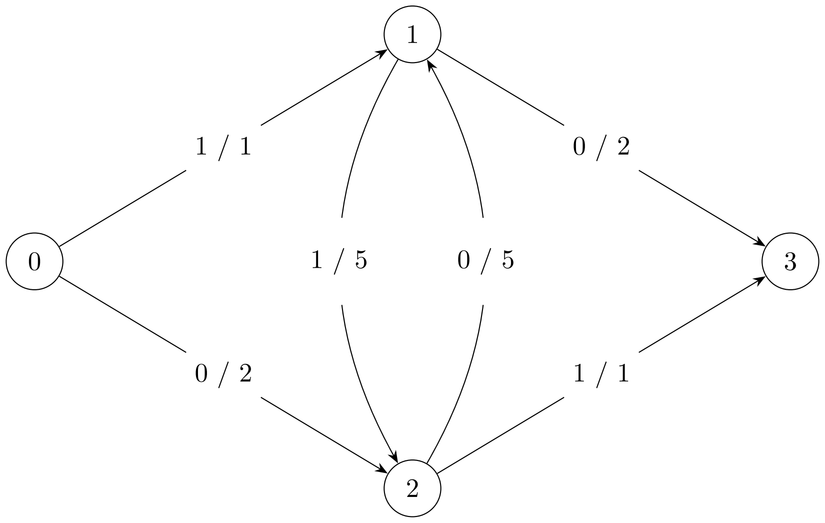


Figure 4.39: The state of things after step 1. We have converted the undirected middle edge to two directed edges. In a rigorous image, all 5 edges would be so converted, but we modified the graph slightly for clarity.

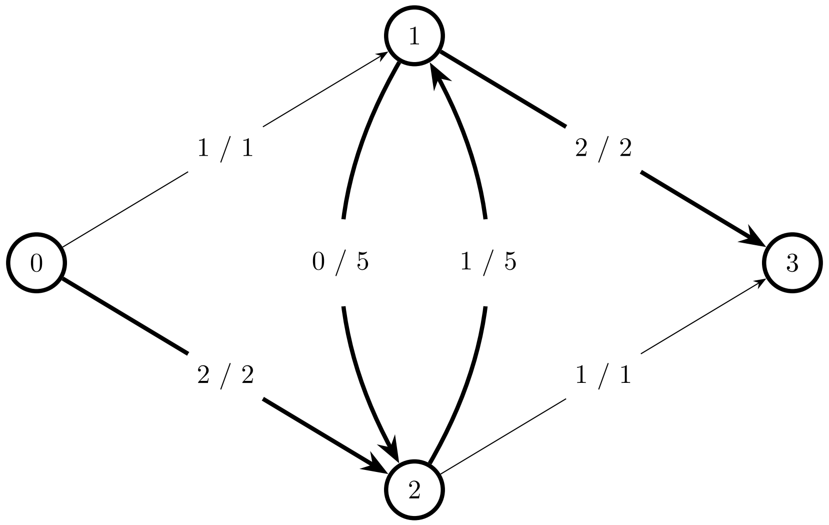


Figure 4.40: Once again, we find the path pushing flow back from $2 \rightarrow 1$. In this case, we can push 1 flow back to cancel out the downward going edge, and potentially up to 5 more on the upward going edge. We are limited to 2 though, by the other edges on the path. We therefore cancel out the flow first with a flow of 1 before pushing the remaining 1 in the upwards edge.

Symmetric Flow Trick Instead of keeping track of both edges, we can instead use what we'll call the *symmetric flow trick*. In this scheme, when we push flow along an edge, we both add the flow to the forward edge and subtract it from the backward edge. This may lead to a negative flow along an edge,[29] but greatly simplifies the check of how much flow we can push along an edge and the update logic.

We can see how this works with the same example as before:

[29]which we can ignore at the end of the algorithm

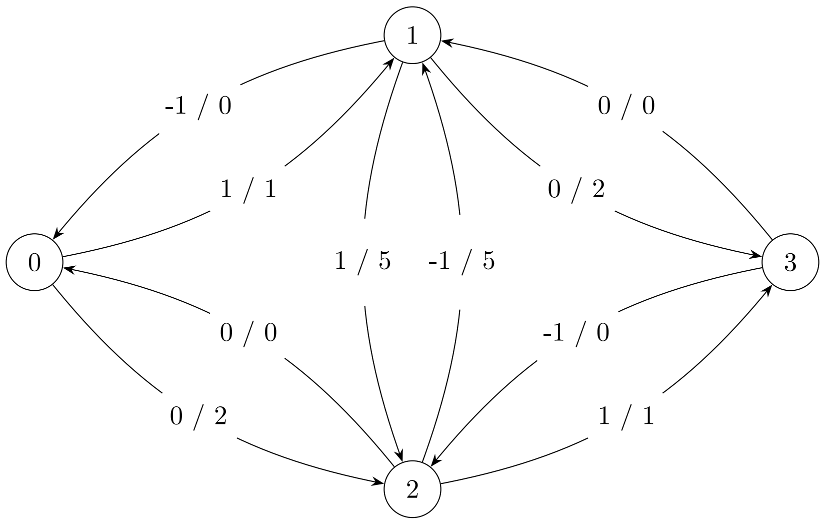


Figure 4.41: The state of things after step 1. Along each edge we have pushed 1 flow, we have accounted -1 flow along the opposite edge. Note that if an edge is directed in the original graph, its reverse edge will have capacity 0, as opposed to bidirectional edges such as between 1 and 2, which will both have the full capacity.

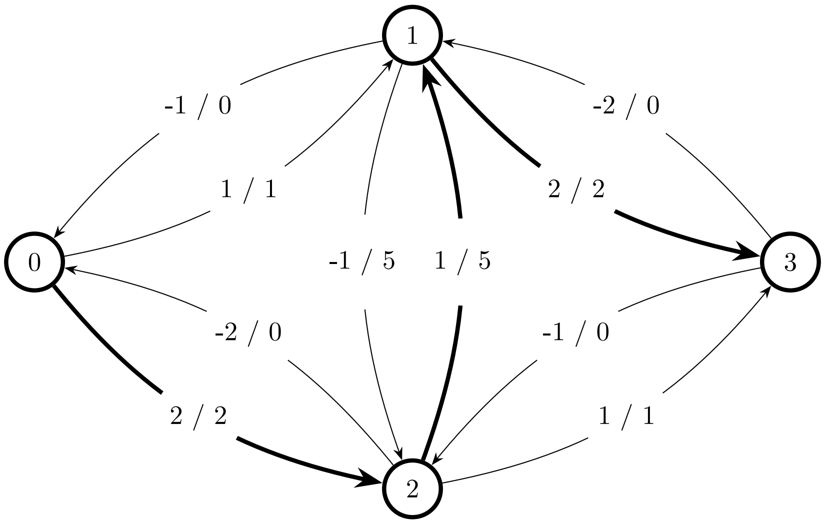


Figure 4.42: Once again, we find the path pushing flow back from 2 to 1. We add the 2 flow to the forward edge, and subtract it from the backwards, leading to the correct values on all edges.

Once the algorithm is terminated, we can ignore the edges with negative capacity and get our accurate flow along each edge.

Finding Incoming Edges BFS, and specifically BFS using an adjacency list, is intended to work with forward edges only. The adjacency list directly returns the outgoing edges from our current node but has no provision to return incoming edges. It is necessary to find such edges in order to cancel flow. To alleviate this situation, we add a weight 0 edge to the adjacency list in the direction opposite of every other edge existing in the graph.

In cases where no such edge existed before, the addition of the 0 weight edge does not impact the outcome, but BFS can now find this edge to enable flow cancelling. In cases where such a reverse edge already existed, we do not need to add the edge, but doing so does not impact the algorithm.[30]

We are now ready to see an implementation. An adjacency list is used, and as such, we will assume edges are already directed.

Listing 4.28: C++

```cpp
//assume adjacency list

//Add backwards edge if it doesn't exist
for(int i=0;i<n;i++)for(auto& j:al[i])al[j.first][i]+=0;
unordered_map<int,int> flow[n]; //current flow between two nodes
int tot=0; //the final answer
while(true){
    //BFS phase
    queue<int> q;
    int res[n]; //how much we can push to this node
    vector<int> prev(n,-2);
    q.push(u);
    prev[u]=-1;
    res[u]=INT32_MAX; //infinite flow into first node
    while(true){
        if(q.empty())goto OUT; //no paths
        int on=q.front();
        q.pop();
        for(auto& i:al[on])if(prev[i.first]==-2){
            if(al[on][i.first]-flow[on][i.first]>0){
                prev[i.first]=on;
                res[i.first]=
                    min(res[on],al[on][i.first]-flow[on][i.first]);
                if(i.first==v)goto IN; //found the sink
                q.push(i.first);
            }
        }
    }
//update phase
```

[30]It will simply be an extra edge that will be iterated through, but so long as the actual edge still exists, and only a single directed flow value is maintained for this pair of nodes, the algorithm will be correct.

```
IN:
    tot+=res[v];
    int on=v;
    while(prev[on]!=-1){
        flow[on][prev[on]]-=res[v];
        flow[prev[on]][on]+=res[v];
        on=prev[on];
    }
}
OUT:
//do whatever with tot
```

Runtime The algorithm takes $O(e)$ time to run each iteration of the BFS, so the question becomes how many times does the BFS execute? It turns out there are two limits:

1. As the observed flow increases by at least 1 for each intance of the BFS, we are limited to the total value of the ultimate maxflow. If we call the maxflow f, then the runtime is $O(ef)$
2. Through some hand waving, the BFS will never run more than $O(ev)$ times, making the overall runtime $O(ve^2)$.[31]

The runtime will be the lesser of these two. Even in cases where they seem prohibitive, most problems are not sized with large enough edge counts to prevent usage. This is especially true in matching problems, where the graph is typically sparse. The structure of graphs often will result in the overall runtime being less than prescribed in nuanced and difficult to prove ways. Regardless, problems sized to be solved by maxflow typically have **low hundreds of nodes and edges**. This sizing is common enough that if a graph has such a limit on size, it is a strong hint that maxflow is the intended solution.

There are, however, insteances where this performance is insufficient. We will look at other algorithms further.

4.3.2.2 Faster Max-Flow Implementations

While the described Edmonds-Karp method is fast enough for most maxflow problems,[32] there can be situations where the graph is dense and it is prohibitive. Therefore we present first a further improvement on the Ford-Fulkerson method, and finally, a second class of algorithm which is more complex in nature, but also significantly faster in the general case: the *Push-Relabel* family.[33]

[31] In practice, the algorithm doesn't exhibit this worst casea, and if understanding the exact runtime is necessary, a faster algorithm should likely be used.

[32] and the most intuitive

[33] There are further algorithms such as *MPM*, but they are not useful enough to warrant inclusion.

Dinic's Algorithm One of the main problems with the Ford-Fulkerson method is the expense of finding new paths. In the BFS method prescribed by Edmonds-Karp, we must traverse every edge near to the source before finding paths to the further away sink. When using the DFS method of finding paths, we could search a large number of long paths only to find a relatively shorter path to the sink. *Dinic's Algorithm* improves on both of these schemes combining the two path finding algorithms to get the best of both worlds. From a high level, the intuition is as follows:

- The problem with DFS is that it can meander around before finding a path to the sink.
- If we could "guide" the DFS to more quickly find the sink, it would be much more performant.
- Execute DFS on a graph where we know what distance each node is from the source. Only select edges which move us further away. This will help prevent DFS from meandering.
- BFS enables us to label nodes based on their distance away from the source. We can run BFS first and then use the result to guide the DFS.

Taking this intuition, we create an algorithm where the following two steps which are repeated.

1. Perform a BFS traversal from the source using only edges on which we could push additional flow, but instead of terminating when the sink is reached and adding flow along the discovered path, simply label each node with the distance from the source.
2. Perform a DFS traversal from the source, but only select edges[34] which increase the distance from the source.[35] When the sink is reached along some path, push the amount of flow as we would have done in Ford-Fulkerson.
    - Repeat this step until there are no more DFS paths along which we can push flow

Each iteration of DFS will leave different edges along which we can push more flow,[36] so the distances found in the BFS will change as well.

Lets see an example, using the same graph we used above.

[34]which have available capacity

[35]If the current node has a distance of 2, then we should only consider edges which go to a distance of 3.

[36]including edges where we could cancel flow

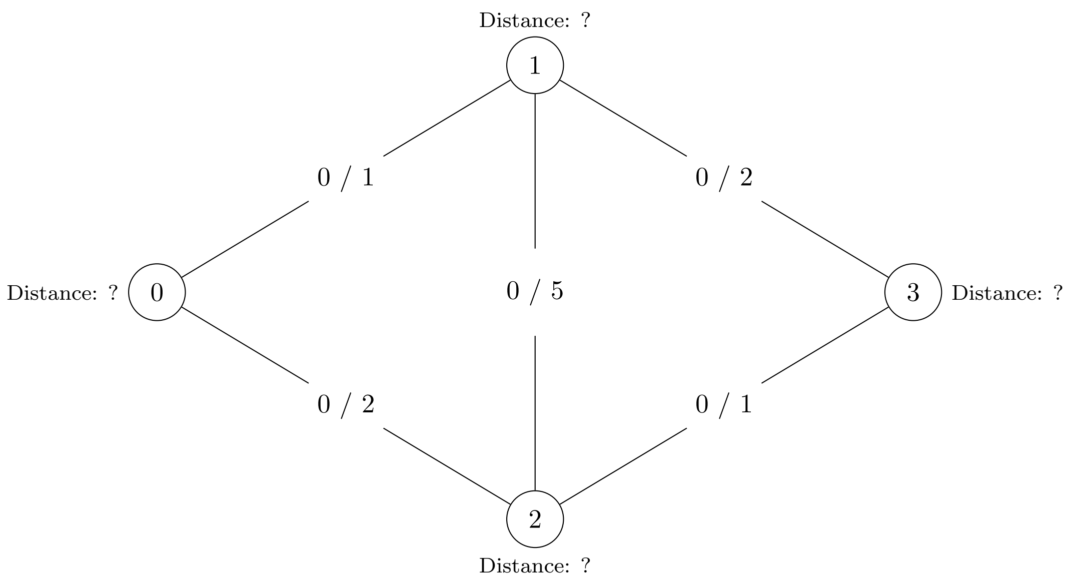


Figure 4.43: The initial state

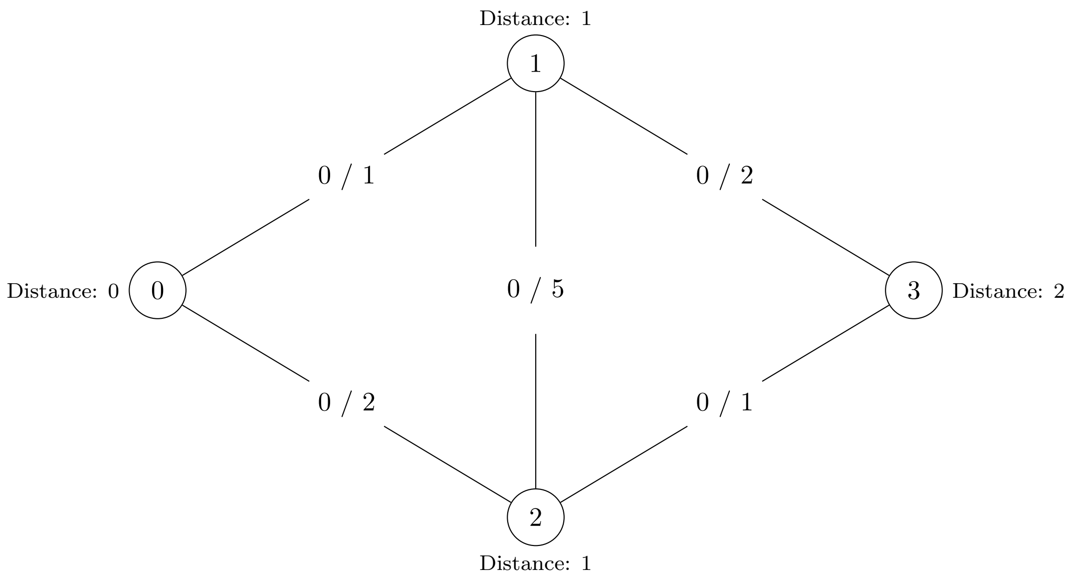


Figure 4.44: After the BFS, we have detected the node distances. As every edge has available capacity, all the edges can be used in the BFS.

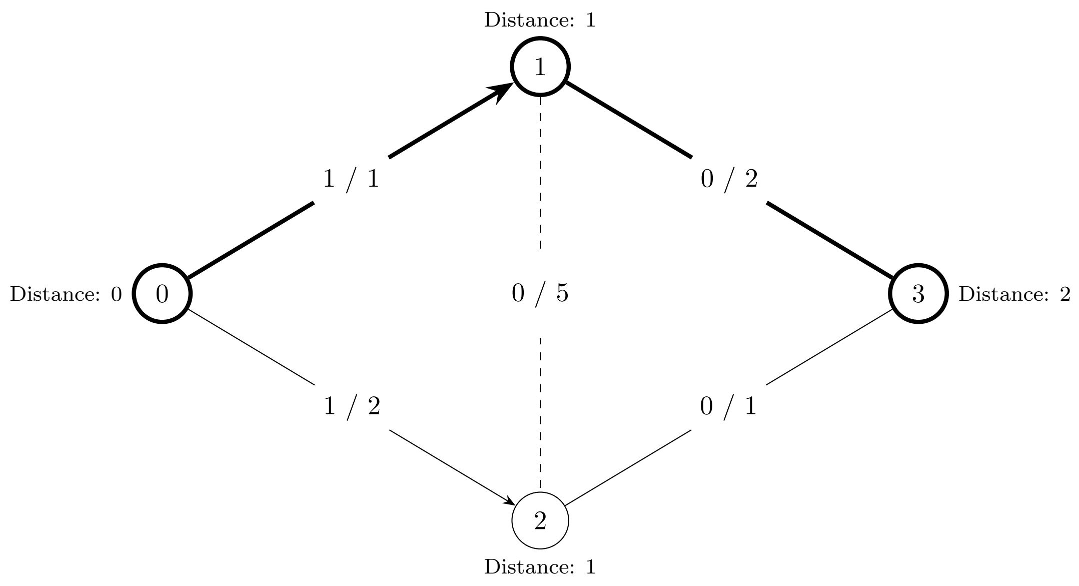


Figure 4.45: We find a path during our first DFS. The dashed line goes between two nodes of equal distance so does not increase the distance from the source. It cannot be used at this time.

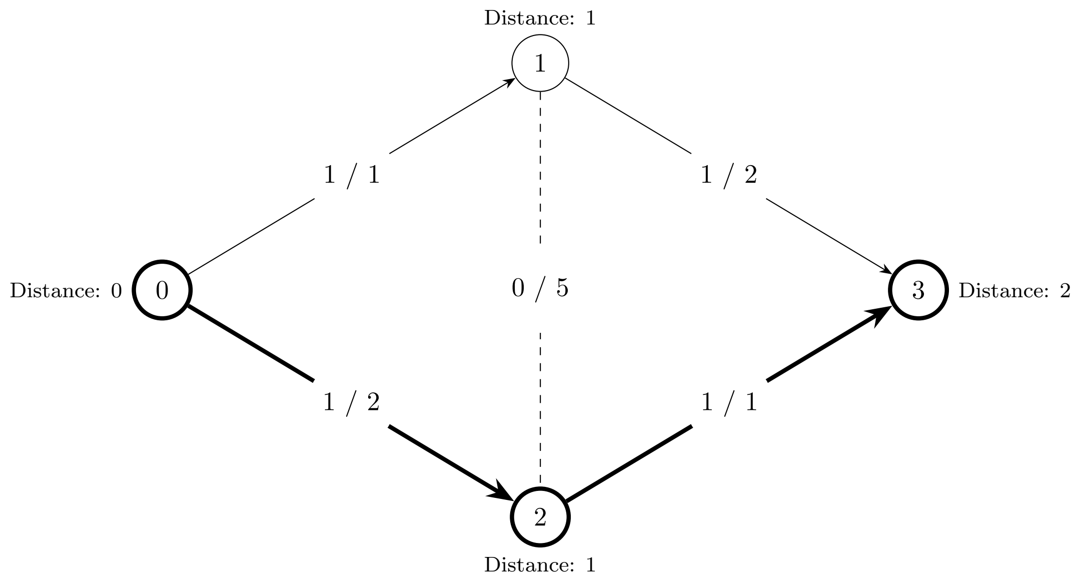


Figure 4.46: We find a second path using DFS. There are no more paths we can find during this iteration as the dashed edge is still unusable.

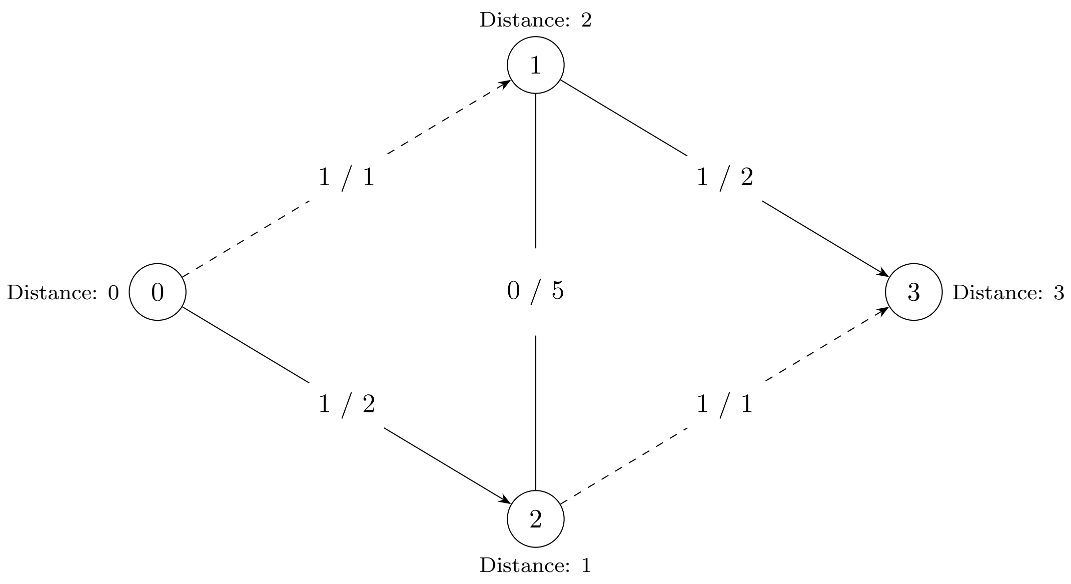


Figure 4.47: We return now to perform another BFS phase. Note that the dahsed edges cannot be used during the BFS as they do not have any available capacity. We could potentially cancel flow along them, however, if such a situation arose.

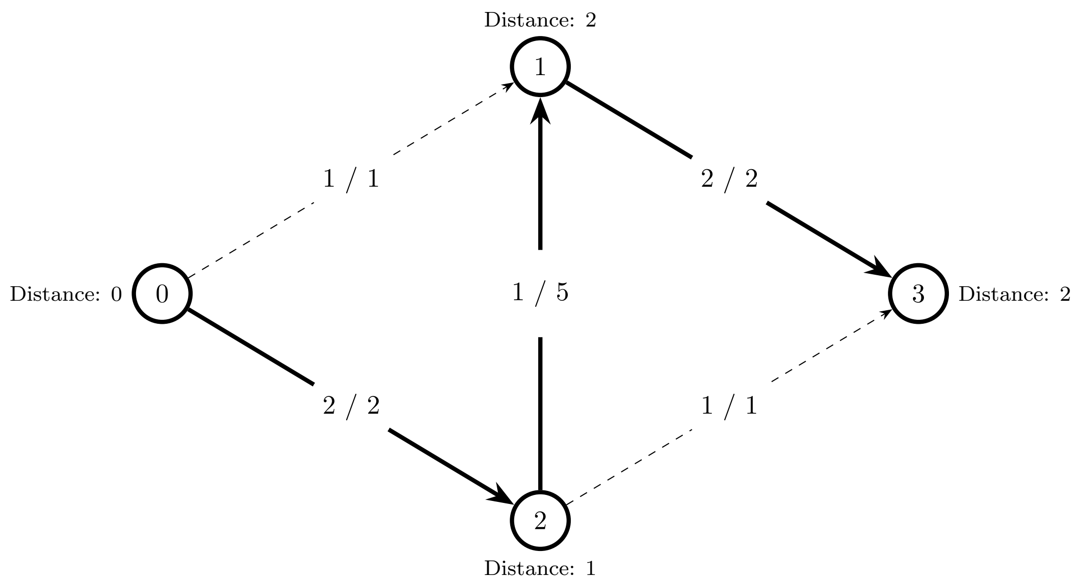


Figure 4.48: As BFS revealed a difference in distance between nodes 1 and 2, that edge is available for use in the DFS. We find a path using that edge and push flow.

At this point, there are no more paths which can be found with DFS. Further, there are no more paths which can be found with BFS, so the algorithm terminates having found the maximum flow.

Rough Proof of Correctness The flow is valid at any step of the algorithm, and if we cannot find a path from the source to the sink on which we can push more flow, then the flow must be maximum. When BFS cannot find such a path, we know we have found the max flow.

Rough proof of Runtime The distance that BFS finds from the source to the sink will increase each iteration, limiting the number of BFS iterations to v.[37] Between iterations of BFS each of the edges can become saturated at most once, limiting the number of runs of DFS between every BFS to $O(e)$. If we cache the edges taken in a given DFS, allowing us to "skip" trying edges along which we know there is no more flow, the total number of edges examined during all the DFS instances in this phase is also $O(e)$. As the path length for any DFS is $O(v)$, the total time for each phase is $O(ve + e)$ or $O(ve)$. With the v phases, the overall runtime is $O(v^2e)$

Graphs with Unit Capacities In graphs where all capacities are 1 and each node has either a single incoming edge or a single outgoing edge, Dinic's

[37]handwave

algorithm is especially efficient, operating in $O(e\sqrt{v})$. In general graphs with capacities of 1, bounds are only slightly worse: $O(ev)$ in all cases.[38]

Implementation The implementation of Dinic contains several aspects that we saw in Edmonds-Karp. These include:

- adding reverse edges to the adjacency list
- updating the residual on each edge after finding an augmenting path,
- utilizing the symmetric flow trick

That said, there are a few details of the implementation which differ. Further, as this is an algorithm we will use when we need high performance, the requisite level of optimization is higher.

- We eschew the map-based adjacency list for the vector-based one. This provides significantly faster lookups, and is possible since we do not need random lookup of edge weights.
- In order to achieve optimal runtime, we have to avoid exploring DFS paths which we already know contain no route to the sink. We do this by maintaining a `next` vector indicating which edge to investigate next for any node. This value is incremented each time we push an edge from the given node to the DFS stack.[39] If we reach this node again in a subsequent DFS, we will use this vector to avoid reexploring previously seen paths. A caveat is if we find some path to the sink, we must ensure the values of the vector indicate to reexplore all the edges of that path. This is necessary as there may be a second path which shares some number of those edges with the current augmenting path. We accomplish this by decrementing the `next` value for all nodes along the agumenting path, ensuring the first path we explore is exactly the previous augmenting path. One could say we restart right where we left off.
- To avoid recursion, the DFS is maintained with a stack. The various loop invariants and checks ensure the value of `on` is set appropriately and that the stack is popped when necessary. This is perhaps an unnecessary optimization in many cases, as using recursion greatly simplifies the implementation at a small cost of speed.

The algorithm contains a myriad of loops, which are described here:

- The main algorithm loop contains both the BFS and DFS phases. It exits when a run of BFS and DFS does not increase the overall flow.
- The BFS loop exits when either the sink is reached, or no path to the sink is found.

[38]Tighter bounds can be proven, but are not covered here.
[39]indicating we "explored" that edge

- Within the BFS loop, we iterate over all edges which have available capacity.
- The outer DFS loop executes multiple iterations of DFS, exiting when no augmenting path can be found.
- The inner DFS loop actually performs an iteration of DFS. It exits when the sink is found or no path can be found. It maintains the `next` values from the previous DFS iteration such that we can *start where we left off* and not reexamine dead-end edges.
- The DFS edge loop checks all edges from a node until either we have pushed a new node on the stack, have exhausted all edges for this node,[40] or have reached the sink.

Listing 4.29: C++

```
int flow[SZ][SZ];
//assume no duplicate edges or edges looping on a single node
int max_flow(vector<pair<int,int>>* al,int n,int u,int v){
    //generate back-edges
    for(int i=0;i<n;i++)for(auto& j:al[i])al[j.first].push_back({i,0});
    int ans=0,prev_ans;
    do{ //main algo loop
        prev_ans=ans; //use this to figure when we found no new flows
        queue<int>q;
        q.push(u);
        vector<int> d(n,INT32_MAX); //store distances here
        d[u]=0;
        while(!q.empty()&&d[v]==INT32_MAX){ //bfs loop
            int on=q.front();
            q.pop();
            for(auto& i:al[on])
              if(d[i.first]==INT32_MAX&&i.second-flow[on][i.first]>0){
                d[i.first]=d[on]+1;
                q.push(i.first);
            }
        }
        //res = how much flow we can push to a node
        //next = the next edge to DFS for a node
        vector<int> res,next(n,0);
        do{ //blocking flow loop
            stack<int>q;
            q.push(u);
            res.assign(n,0);
            res[u]=INT32_MAX;
            while(!q.empty()&&res[v]==0){ //dfs loop
                int on=q.top();
```

[40]and should thus pop it from the stack

```
                //check edges, but break out if a new node is on the stack
                for(;q.top()==on&&next[on]!=al[on].size()&&res[v]==0;
                  next[on]++){
                    int j=al[on][next[on]].first;
                    //further away and available flow
                    if(d[j]>d[on]&&al[on][next[on]].second-flow[on][j]>0){
                        res[j]=min(res[on],
                                   al[on][next[on]].second-flow[on][j]);
                        q.push(j);
                    }
                }
                if(q.top()==on)q.pop(); //didn't find an edge. backtrack
            }
            if(res[v]!=0){ //saturate the path
                ans+=res[v];
                int on=u;
                while(on!=v){
                    //this gets incremented *before* we bail out of the
                    //inner DFS loop, so reset it back to the edge that
                    //formed the augmenting path
                    next[on]--;
                    int j=al[on][next[on]].first;
                    flow[on][j]+=res[v];
                    flow[j][on]-=res[v];
                    on=j;
                }
            }
        }while(res[v]!=0); //DFS when we can't push anymore flow
    }while(prev_ans!=ans); //the entire phase didn't push any flow
    return ans;
}
```

Push-Relabel In the Ford-Fulkerson family of algorithms, we use graph traversals to find paths along which we can push more flow. This means in each iteration of the algorith, we must examine $O(e)$ edges to find a candidate path. More importantly, any work done examining edges which are not in our augmenting path[41] is thrown away before starting a new iteration of the algorithm. Consider the following graph:

[41]or edges which are on our path, but have remaining capacity

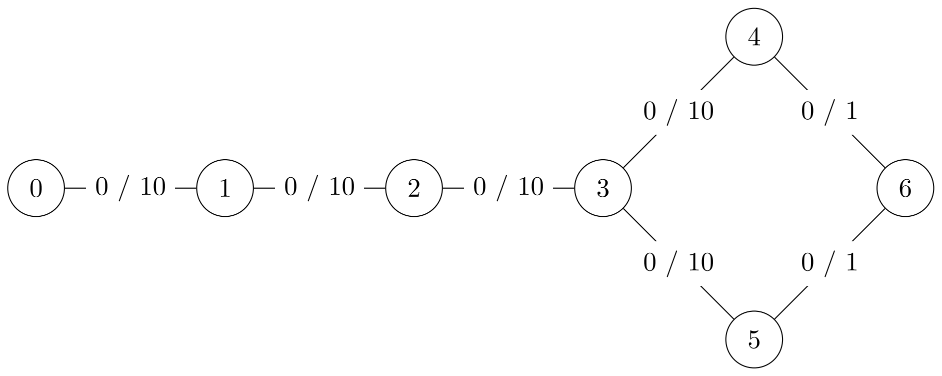


Figure 4.49: A long chain of edges with two limiting routes near the sink

In the first iteration of Edmonds-Karp, we will iterate through 6 edges of BFS before finding the Augmenting path:

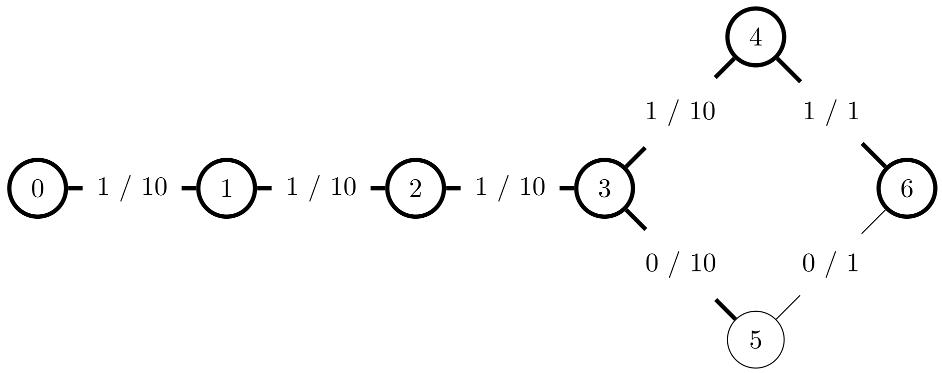


Figure 4.50: All edges and nodes (with available capacity) closer to the source than the length of the found path are examined!

In the second iteration, the first 4 nodes and 5 edges we visit are identical to those from the previous iteration:

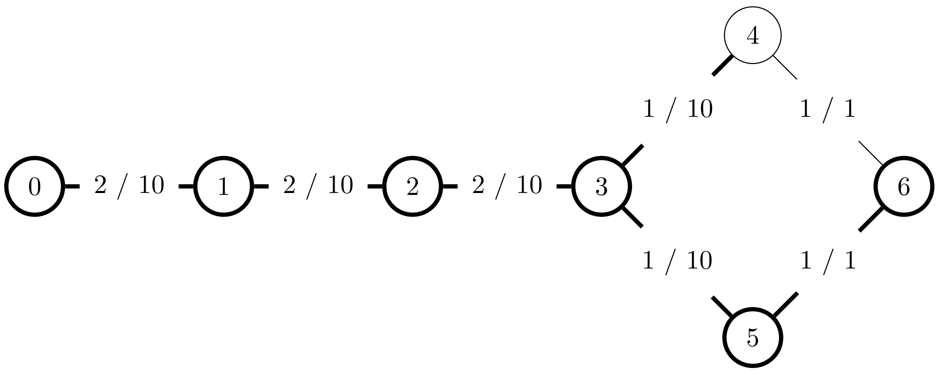


Figure 4.51: Everything is the same, but for the slightly different branch at the end.

This inefficiency is necessary of Ford-Fulkerson-based algorithms, which require that after each path is added to the solution, the flow into any node equals the flow out of that node.[42] The *Push-Relabel* method allows us to dispense of this invariant, performing subsequent iterations of the algorithm without redoing work.

From a high level, the algorithm allows us to "remember" nodes which have extra flow not being currently used, later attempting to move that excess towards the sink. The algorithm operates node by node instead of on entire paths, allowing us to find multiple routes to the sink at the same time. Let's see an example of the idea.

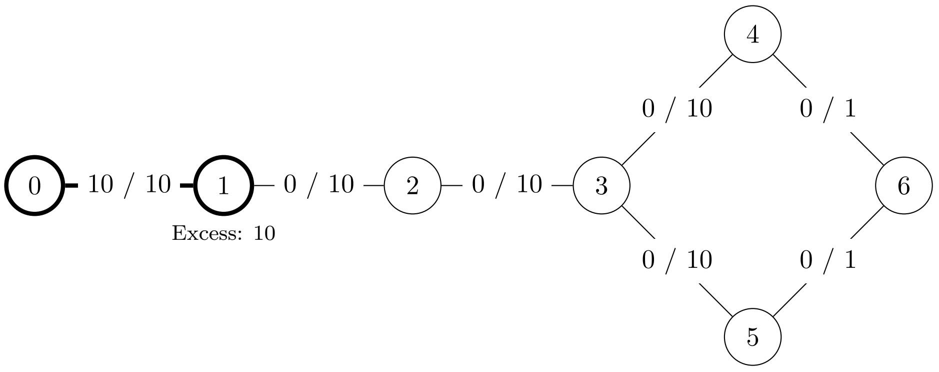


Figure 4.52: We saturate the first edge, leaving 10 *excess* at node 1.

[42]also known as a *valid* flow

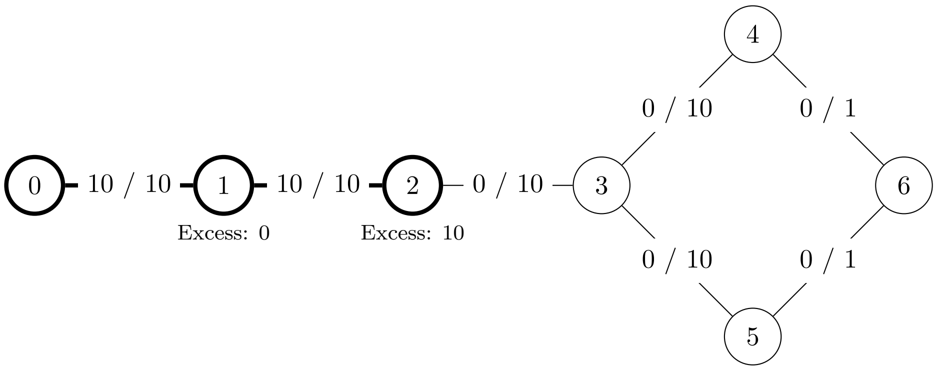


Figure 4.53: We saturate the second edge, consuming all the excess at node 1, which then ends up at node 2.

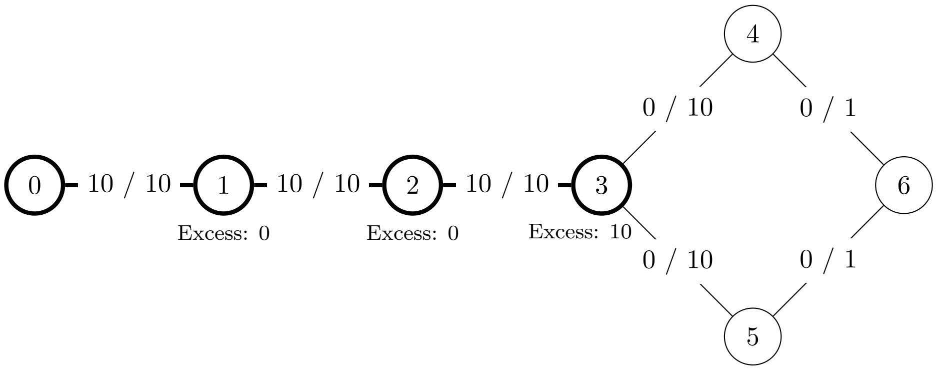


Figure 4.54: We repeat the same process, moving the excess to node 3.

Challege 1: More Capacity than Excess When we reach the current state with 10 excess at node 3, we have a problem. We have 10 excess, but 20 available outgoing capacity.[43] In order to cope with this problem, we will arbitrarily choose an edge on which to push the excess. If we later find that decision to be incorrect, we will resolve the problem by pushing the excess back from the chosen node.[44]

[43] 10 each to nodes 4 and 5

[44] similarly to flow cancellation in Ford-Fulkerson

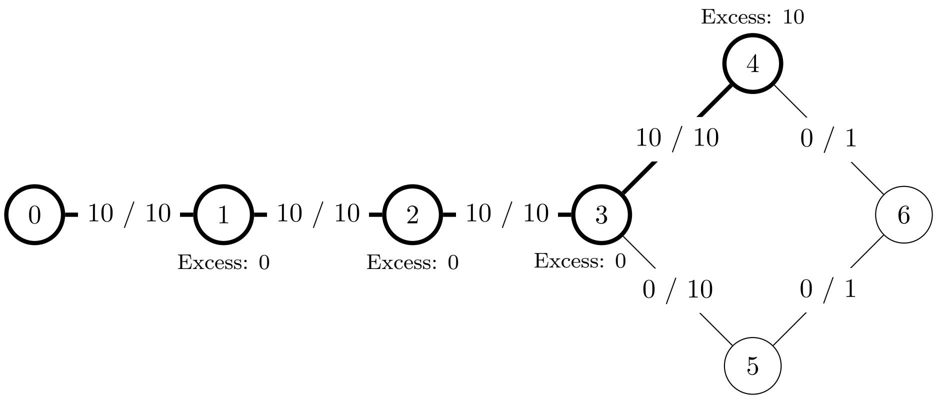


Figure 4.55: We arbitrarily choose the path from $3 \to 4$ to push our flow.

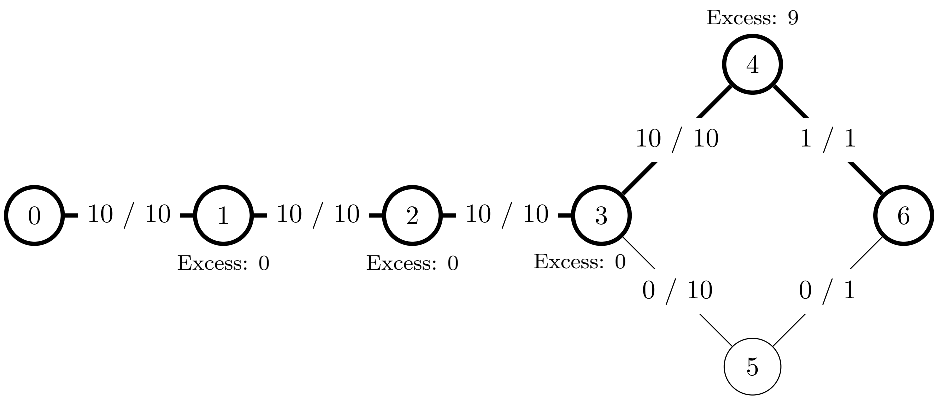


Figure 4.56: When we push from $4 \to 6$, we can only consume 1 of the excess, leaving 9.

Now that we have exhausted the ability to push flow from 4, we push it back to node 3.

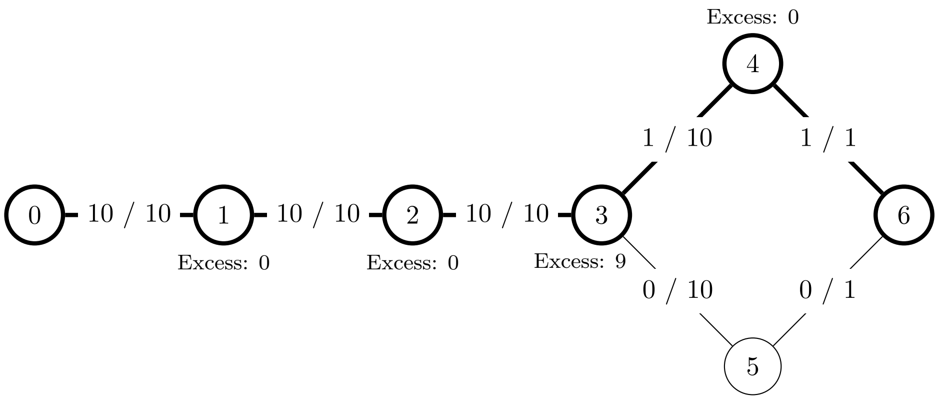


Figure 4.57: We can push the remaining 9 flow back from 4 → 3, cancelling the previous flow. This leaves 9 excess at 3.

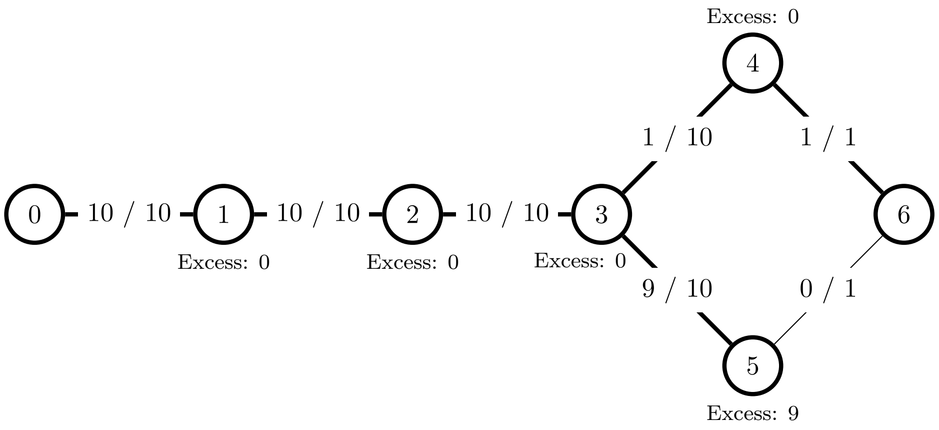


Figure 4.58: We can now try pushing the excess flow to node 5.

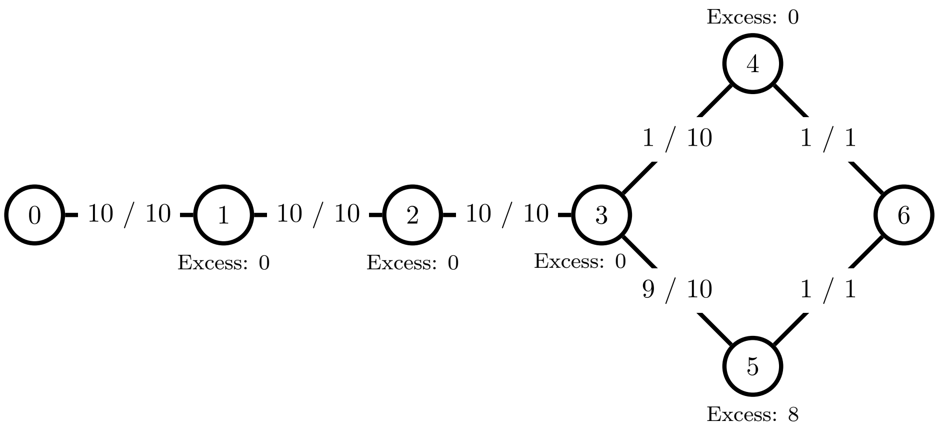


Figure 4.59: We push the flow from $5 \rightarrow 6$, getting our 2 flow to the sink.

Finally, after we have reached the optimal flow, and similarly to how we pushed flow back from 4 to 3, we push all the excess flow back through the graph, leaving the same flow network as we would have gotten from Ford-Fulkerson.

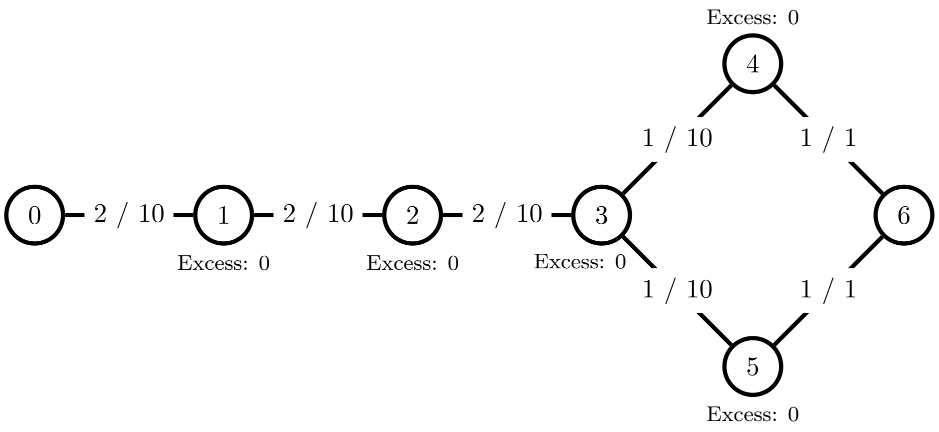


Figure 4.60: All excess is 0, so this is a "valid" flow.

Astute readers will realize we glossed over some serious challenges with this algorithm. We'll address them now that we have the intuition of how such a node-based algorithm might work.

Challenge 2: Node Ordering In Challenge 1, we introduced the concept of pushing flow backwards, enabling us to deal with branches. What we ignored, however, is how we prevent sending flow backwards *prematurely* when there might be other on which to send it. Consider after we first pushed flow to node

3. While we intuitively know we must attempt to send that excess to nodes 4 and 5, what prevents us from instead immediately sending it back to node 2? Further, after we have sent flow back from a node (such as after we sent it back from 4 to 3), what stops us from sending it from 3 to 4 again?

To solve this problem, we will introduce a value for each node indicating a preference of where to send flow. In this scheme, we will conventionally prefer nodes with lower preference values, placing the source at an infinite preference and the sink at 0. When we push flow to a node, we will increment its preference by 1 to prevent prematurely pushing flow back to this node.

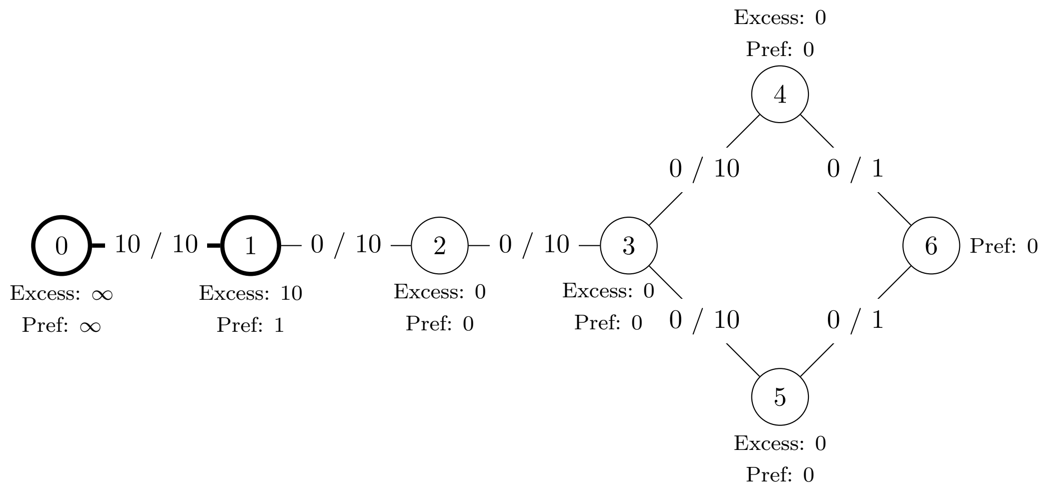


Figure 4.61: We pushed the flow to 1 and incremented its preference. We can see how if we only allow pushing flow to a lower preference value, this prevents us from pushing flow back to the source prematurely, as the preference of node 2 is less than that of node 0.

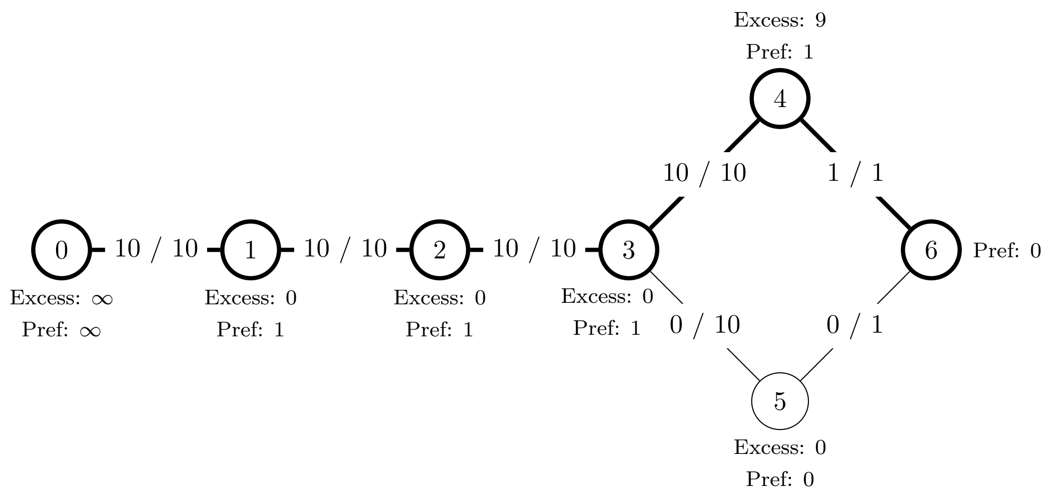


Figure 4.62: The graph after pushing flow all the way to the sink via node 4. Note that we have increased the preference values along the way, ensuring the flow always moved toward the sink. We do not increment the preference value of the sink, as flow never reverses back once reaching it.

We have now reached an impasse at node 4. We have exhausted our paths to nodes with lower preferences values, but still have excess capacity. In order to cope with this situation, we will now allow allow node 4 to push to nodes with preference 1, enabling pushing the flow back to node 3.

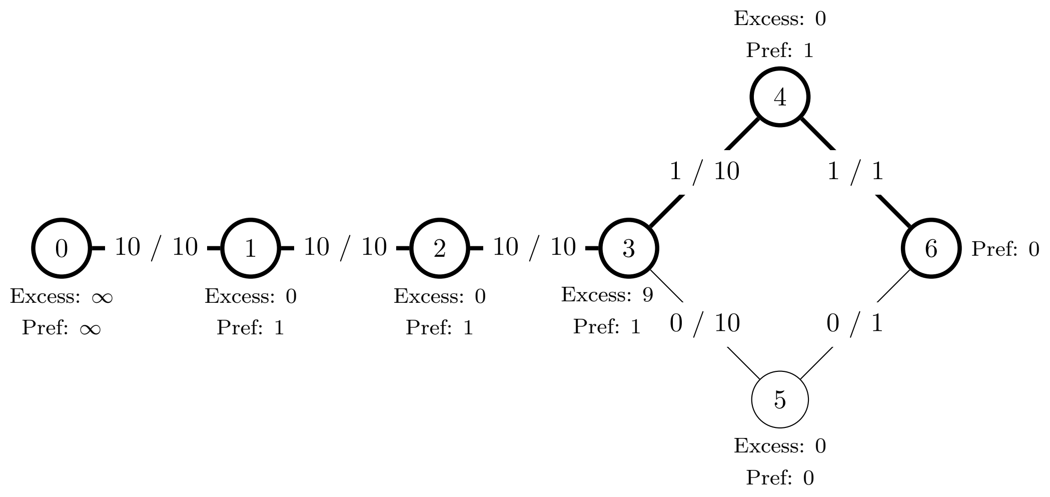


Figure 4.63: Node 4 exhausted pushing flow to all nodes with preference 0, so pushed flow back to node 3.

With our excess flow back at 3, we now find a node with preference 0, node 5, which we can push flow to. We push flow along that path similarly to how we did with the node 4 branch, and end up in the following situation:

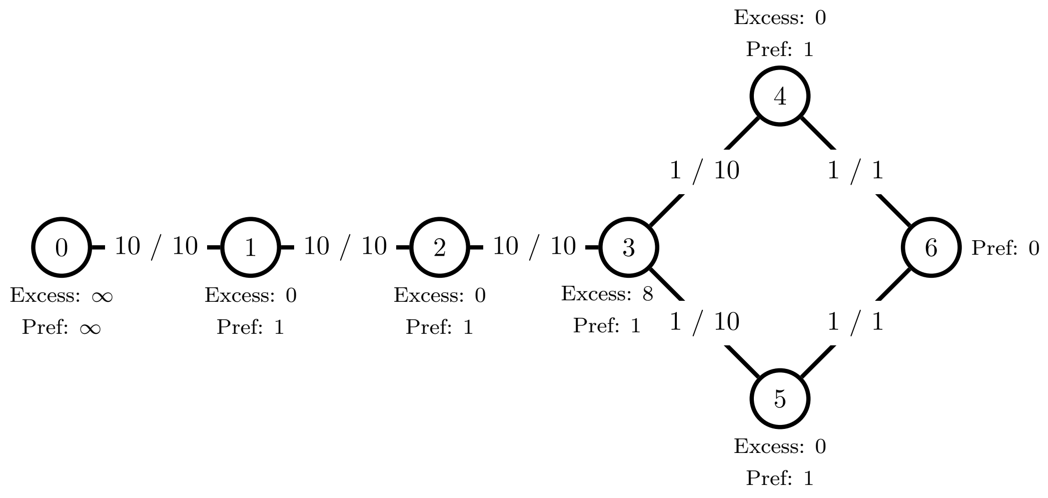


Figure 4.64: We pushed 9 flow to 5, then 1 flow from 5 $\to$ 6, and finally pushed the remaining 8 back to from 5 $\to$ 3.

As can be seen, the situation is now that we have excess at node 3, and three nodes with preference 1 to which we can send it. This gives us no obvious way to ensure we don't get in an infinite loop, pushing flow band and forth between nodes 3, 4, and 5 indefinitely.

Challenge 3: Eliminating Loops The high-level goal of labeling the node preferences was to avoid pushing flow to a node unless there was no better option. However after we processed nodes 4 and 5 and saturated the edges to node 6, we didn't make a change to its preference to prevent trying that way again.[45] To avoid this, we can introduce the following rule:

> When pushing flow to another node, increment our preference value to greater than that of the node we're pushing to. This encourages the node to push flow along other directions instead of back and forth.

Put another way, once we find that we have excess flow at node 4 that cannot go anywhere, we can treat this as a second source. Identically to the real source,[46] we want to try not to push the same flow back unless we have no other option. By the same argument that originally introduced node ordering, we increment

[45]leaving them all at preferencee 1 and creating the loop
[46]node 0

the preference value to encourage flowing away from this node instead of towards it. We should note however, that despite starting this "BFS" at preference 2, we can still push flow to nodes which we hadn't visited in the previous BFS[47] and visit not only preference 1 nodes, but also preference 0 nodes.[48]

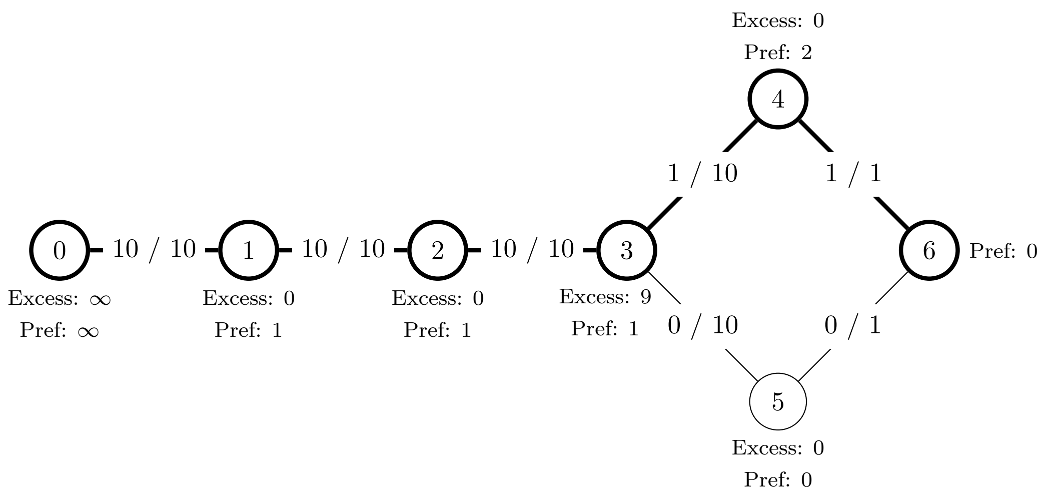


Figure 4.65: When pushing the flow back from 4 → 3, we increment node 4's preference to 2. The excess at node 3 will now prefer to go to node 5 with preference 0.

[47] The one which started at node 1 with preference 1

[48] provably, we will never go directly from 2→0, but we can go 2 → 1 → 0

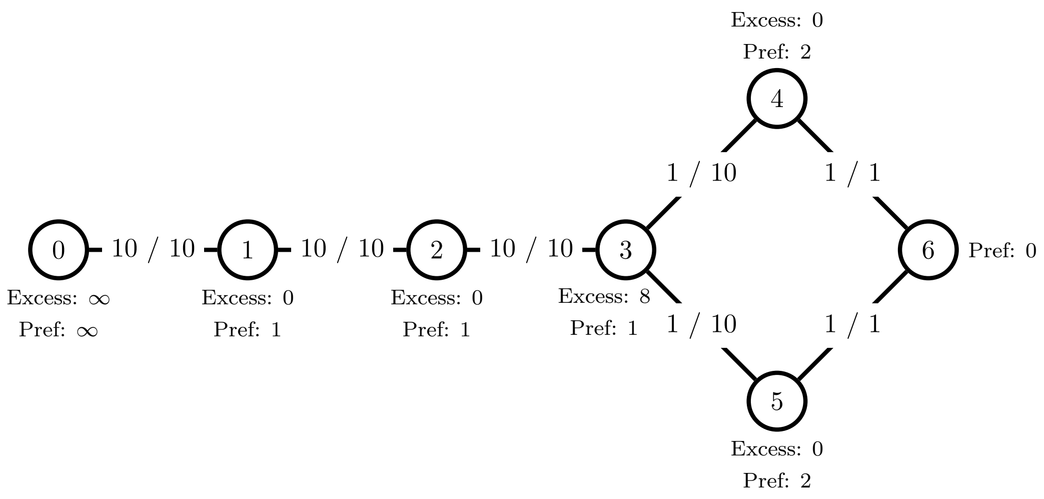


Figure 4.66: After processing the node 5 branch and pushing the excess back to node 3, its preference is similarly set to 2.

By incrementing the preference value before pushing flow back to a previously equal preference value, we ensure that that destination node does not try to push that same flow back to us. Node 3 will now be forced to push flow back towards the source instead of forming a loop.

The selection is made to push flow back towards node 2, and finally node 1, incrementing the preference of those nodes to 2 along the way. We then find another issue.

Challenge 4: Infinite Source Preference Once all the flow has been pushed to node 1, we have a graph in the following state:

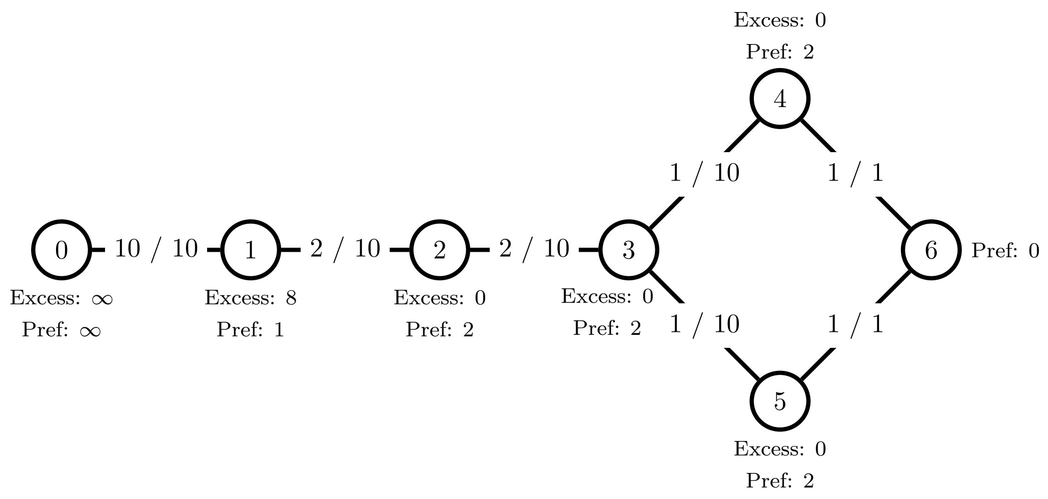


Figure 4.67: All the excess flow is in node 1 and has nowhere to go.

If we followed our earlier logic, we would want to push the flow to the lowest preference. This would result in pushing to node 2 at preference 2 and setting node 1's preference to 3. If we continued along this way, we would eventually reach nodes 4 and 5 and again find no path to the sink. Finally, we would return all the flow back to node 1 again. As the source preference is infinite, this bouncing of flow back and forth continues forever, never providing an opportunity to push the last bit of excess back to the source.

To eliminate this problem, we must set the source preference to some non-infinite value which is large enough to ensure we don't prematurely push flow back when there are viable paths to the sink. As the longest path between the source and sink is at most v nodes, we can safely set the preference value of the source node to v. This will guarantee that the algorithm will eventually terminate, but by the time it has, all better paths to the sink will have been explored. This selection of source preference does not ensure that the flow won't "bounce" back and forth between the the nodes closest to the source and the nodes closest to the sink,[49] but it does enable the the algorithm to terminate. This is sufficient to examine the algorithm a bit more formally.

Description of the Canonical Push-Relabel Algorithm The *Push-Relabel* algorithm formalizes the steps we have investigated above. It computes the maximum flow of a graph using the following rules:

- Each node is assigned a preference value called a *height*.[50]
  - Flow can be pushed only to nodes with lower height

[49]or more preciseley, the nodes closest to the minimum cut
[50]equivalent to the above "preference"

  - The source has height fixed at v, the number of nodes
  - The sink has height fixed at 0

- If a node has excess flow and has a height greater than some surrounding node, it can push flow to that node, limited only by its own excess and the remaining capacity of the edge connecting those two nodes.[51] This is called a *push* operation.

- If a node has excess flow but its height is less than or equal to all its neighbors, we increase its height to be one greater than the minimum height of all its neighbors, guaranteeing we have some to which we can push flow. This is called a *relabel* operation.

These steps formalize our actions in the earlier example to the degree that we will not walk through the entirety of the algorithm again. While in that example, we simulated the steps in the order that they would be naturally taken via a DFS of the graph,[52] the the two operations[53] may be taken in an arbitrary order on any node which the rule applies.

To avoid issues where we attempt to push flow from the source to some node x after that node x has pushed flow back to the source, we initialize the algorithm by saturating the edges from the source to all its neighbors. We never again push flow from the source.

For comparison to Ford-Fulkerson, we present the earlier example in its entirety.

[51] or cancelling flow along a previously pushed edge
[52] pushing flow as far as possible before pushing flow back and trying another path
[53] push and relabel

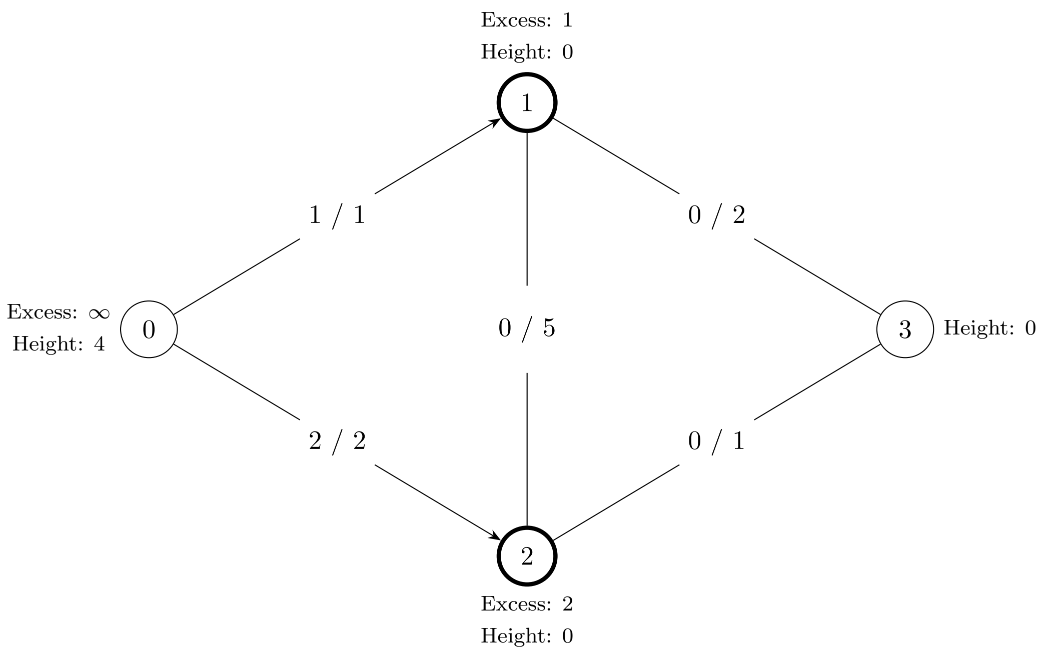


Figure 4.68: The source is initialized with height equal to the number of nodes. All other nodes are initialized at height 0. We saturate the edges adjacent to the source, and then no longer push any flow from the source. Nodes adjacent to the source have their excess set accordingly. By definition, the sink does not have excess flow, and has a height fixed at 0. Nodes with excess flow are highlighted.

We may select any node with excess flow to perform either a push or relabel operation. For simplicity, we will select node 1.

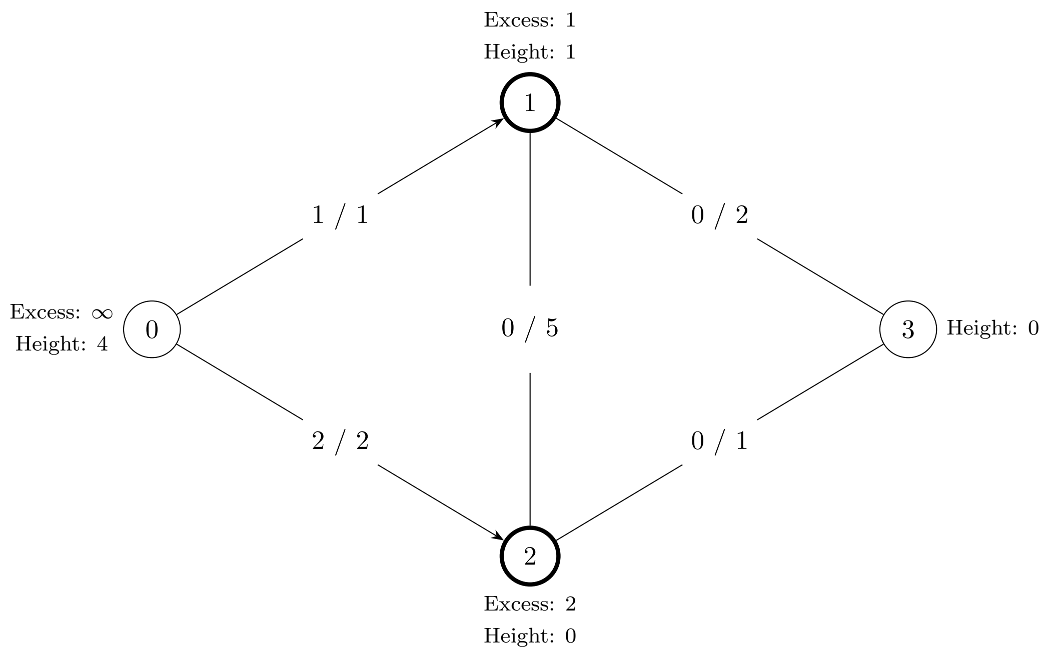


Figure 4.69: We select an arbitrary node of excess flow, node 1. As it has no neighbors with lower height, we increase its height to be one greater than the lowest-height neighbor (1). This is a relabel operation.

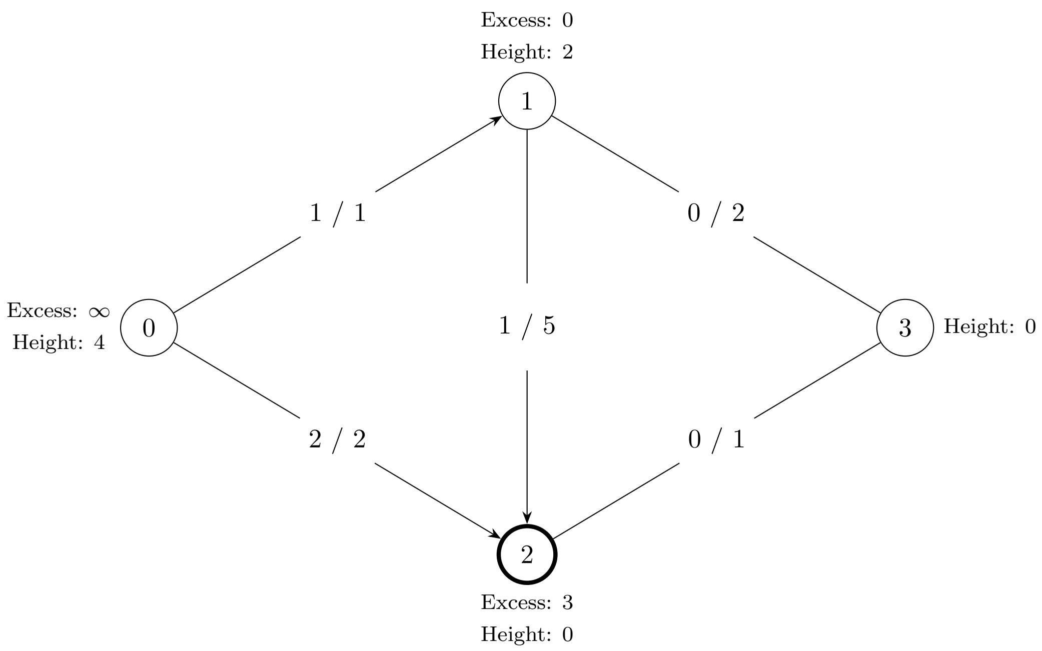


Figure 4.70: We select an arbitrary node of excess flow, node 1. It has height greater than either of its neighbors, so we arabitrarily select node 2 to push flow towards. The amount we can push is limited by the excess amount currently in node 1 (1 excess). The amount of excess at node 2 now increases due to the additional incoming flow. This is a push operation

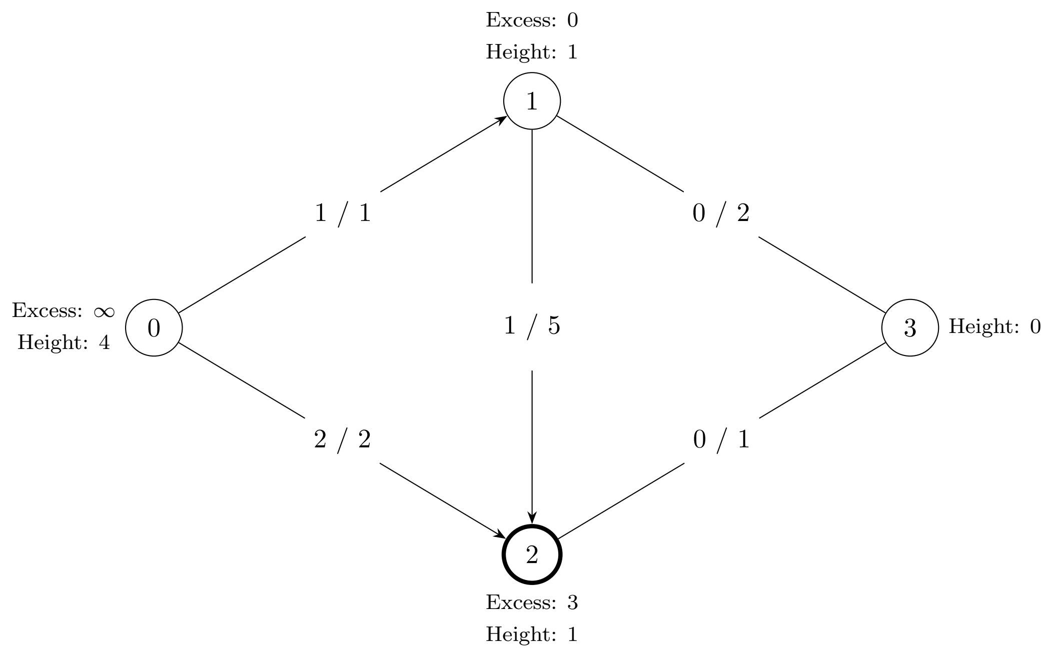


Figure 4.71: Node 2 is the only node with excess. It has no neighbors lower than it, so we increase its height to be just higher (1) than its lowest neighbor (node 3 with height 0). This is another relabel operation.

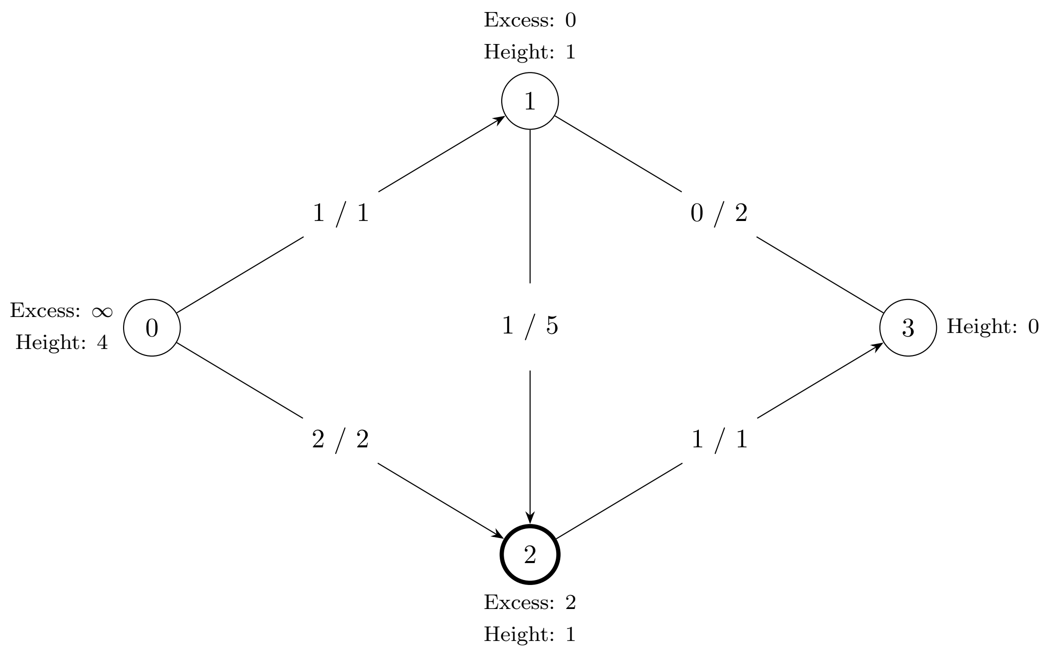


Figure 4.72: Again node 2 is the only node with excess flow, and we can now perform a push operation to node 3. Unlike the push from node 1 which was limited by the amount of excess in the node, this push is limited by the capacity of the edge along which we pushed.

We pause for a second to highlight the the two types of pushes we have made:

1. When we pushed from $1 \to 2$, we were limited by the amount of excess at node 1 rather than the edge on which we were pushing. This is known as a non-saturating push. It means that we can no longer select node 1 for a push or relabel operation until it has gained some additional excess.
2. When we pushed from $2 \to 3$, we were limited by the residual of the edge rather than the excess of the node. This is known as a saturating push. As the node has further excess, we may immediately perform another push or relabel operation as appropriate.

As the previous push from node 2 was non-saturating, we will resume our action there.

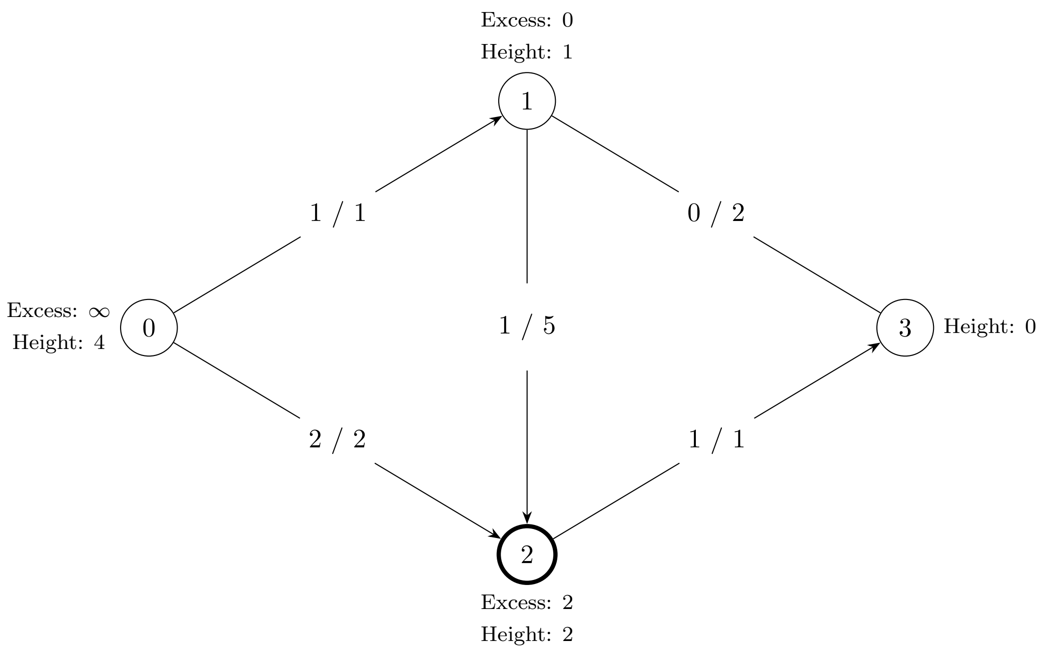


Figure 4.73: Still node two is the only node with excess flow, but with no neighbors to whom we can push flow. Nodes 0 and 1 have higher heights, and the edge to node 3 is full. We therefore perform a relabel.

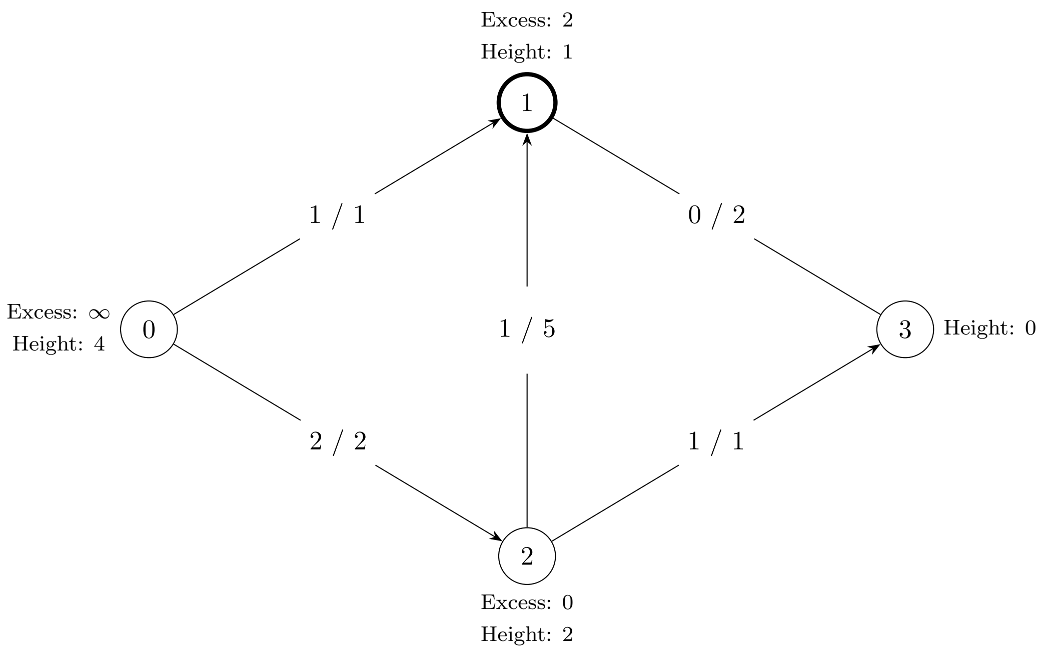


Figure 4.74: We can perform a push operation to node 1. As our excess is currently 2, we can cencel the 1 incoming flow from node 1, and push an additional 1 flow back. Note this reverses the edge between nodes 1 and 2.

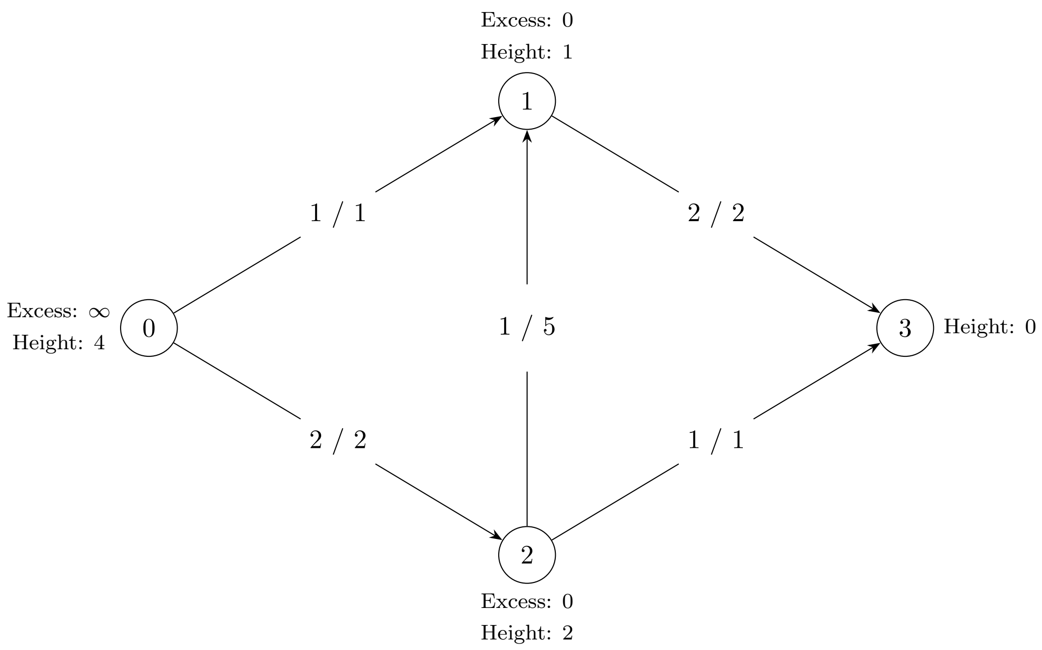


Figure 4.75: Node 1 is the only node with excess flow. It has a greater height than node 3, and can thus push its remaining flow there.

At this point, there are no edges with excess flow, so the algorithm terminates.

Rough Proof of Correctness We combine three high level facts to prove that the algorithm produces a maximum flow:

1. When the algorithm terminates, there cannot be nodes with excess flow. If there were, we could always increase height and push the flow back towards the source on the path it arrived from.

2. Along every edge on which we could push more flow, the start of that edge can have height at most one more than the destination of that edge. This must be the case, as the relabel algorithm chooses the minimum adjacent node when bumping the height, and could not have reached a higher height unless we saturated this edge first.[54]

3. Based on (1) we would have to find a path from the source to the sink in order to push new flow.[55] Based on rule (2) and the fact that the

[54]Once that edge is saturated, that node is not considered in the relabel calculation, allowing us to select the next lowest node as the minimum neighbord during the relabel.

[55]Which, because we know Ford-Fulkerson is correct, must be possible if we want to increase a valid flow!

heights of the source and sink are fixed exactly v apart, the path must contain exactly v edges. Unfortuantely, any non-looping path in the graph can contain only $v - 1$ edges. This means that such an augmenting path cannot exist and therefore the flow is maximum.[56]

To put it even more intuitively, if the flow is not maximum, there must be a path to the sink. By raising the height by the minimum necessary, we would have pushed any excess flow on that path towards the sink before we pushed it back to the source.

Rough Proof of Runtime We demonstrate the runtime based on the total times we perform a given operation.

1. We know that no height can raise above $O(2v)$,[57] and therefore, the maximum number of relabel operations is $v * (2v - 1) = O(v^2)$.

2. For any given edge, we can only completely fill it up (or empty it) once for any pair of height values of its two endpoint nodes.[58] This means for a given edge, we can only completely fill it up or empty it $O(v)$ times, for $O(ve)$ total *saturating pushes*.

3. If we consider the sum of the heights of all nodes with excess, a non-saturating push decreases this value by 1.[59] Relabel operation can only add $O(v^2)$ to such a sum,[60] and each saturating push can only add $O(v^2e)$ to the total.[61] As the sum must both start and end at 0, and each non-saturating push decreases it by 1, the total number of such pushes is $O(v^2 + v^2e) = O(v^2e)$.

We are limited to $O(v^2)$ relabels, $O(ve)$ saturating pushes, and $O(v^2e)$ non-saturating pushes, so the total runtime is $O(v^2e)$.

Implementation Push-Relable does not prescribe an ordering of push or relabel operations to achieve correctness or performance. We make the following choices for practicality:

- Any node with excess is stored in a queue, and processed in FIFO order.

[56]This justifies why v is the minimum possible value for the height of the source node.

[57]In the worst case, nodes 1 away from the source can push all flow back to the source when they reach height $v + 1$, and thus will never exceed that value. Nodes 2 away from the source, when they reach height $v + 2$, and in general, nodes the maximum of $v - 1$ away, when they reach $v + v - 1$ height.

[58]If we fill up an edge from $1 \to 2$ with heights $height(1) = x$ and $height(2) = x - 1$, then node 2 must increase to height x+1 before another push can be made on this edge.

[59]By definition, a non-saturating push drains all excess from the start node of that edge. While the destination may enter the sum, it has a height 1 lower than the start node, so the overall sum must decrease by 1.

[60]v nodes, each can only increase their height by $O(v)$ over the course of the entire algorithm

[61]$O(ve)$ pushes, each which add at most $O(v)$ to the total

- When we process a node, we will continue to push and relabel until all its excess is consumed.

The second point is problematic depending on how we determine which edge to push flow on. Imprecise implementations can accidentally introduce an additional factor of v into the runtime.

Suppose that we perform an exhaustive search of neighbor nodes to see if there are any we can push to. This extra work is acceptable for relabel and saturating push operations[62] but would be prohibitive for our $O(v^2e)$ non-saturating pushes, making the total runtime $O(v^3e)$. In order to avoid this behavior, for each node, we remember the last edge we pushed flow on. When the originating node again has excess, we know the remembered edge must have capacity[63] and we can immediately push more flow on it. The cost of operating in this way breaks down as follows, enabling allocation of the edge search to a saturating push:

1. We search $O(v)$ edges in order to find the edge along which we will make the initial non-saturating push. We cache this edge.
2. When we have excess at this node again, it takes $O(1)$ work to find an edge to push to, since we have remembered we have not yet saturated it.
3. Eventually, we will make a saturating push on this edge, again finding the edge in $O(1)$.

Viewing these steps in totality, we are guaranteed to have done only $O(v)$ work to find the edge, and we have performed a saturating push. As the number of saturating pushes is limited, this method ensures we are within the appropriate runtime.[64]

Even when implemented properly to achieve the desired runtime, seemingly minor implementation choices can greatly impact the runtime. We detail those choices here.

1. Use an array instead of a map for all structures.
2. For a node with excess, we only need saturate each edge once before performing a relabel. This can be shown as for an edge to be saturated twice before a relabel would mean the node on the other end of that edge would have to have a higher height enabling cancelling of some amount of flow on that edge. If that node does have a higher height, we cannot push to it now without performing our own relabel. This avoids an extra iteration through all edges before performing a relabel.

[62] Recall we perform $O(v^2)$ relabels and $O(ve)$ saturating pushes, therefore incurring an extra factor of v to perform one of those operations does not exceed the overall $O(v^2e)$ runtime.

[63] since the last push was non-saturating

[64] we are attributing the $O(v)$ time towards the single saturating push rather than to the non-saturating ones.

3. The inner while loop, where we check whether we can push flow on a given edge, is the most frequently executed part of the algorithm. We should strive to do as little work as possible in this part of the code, including extra conditional statements, executions of `min` and `max`, modulus operators, and arithmetic. Note that in the provided implementation, even computing the minimum of the excess and the capacity of the edge is pushed inside the `if` block.

4. Similarly, using the symmetric flow trick, as described in the Edmonds-Karp implementation, reduces the required arithmetic and branches.

5. While in theory, the allowance of $O(v)$ extra work for certain operations would enable the algorithm to execute on adjacency lists or matrices without impacting the runtime, in practice, not iterating over unused edges can result in a significant speedup. To accomplish this without resorting to using a map, we use the compressed adjacency matrix structure.[65]

Listing 4.30: C++

```
//Assume adjacency matrix
int flow[SZ][SZ];
int amc[SZ][SZ]; //the compressed adjacency matrix
int max_flow(int n,int u,int v){
    int height[SZ]={},excess[SZ]={},next[SZ]={},order[SZ]={};
    for(int i=0;i<n;i++)for(int
        j=0;j<n;j++)if(am[i][j]||am[j][i])amc[i][order[i]++]=j;
    queue<int> q;
    height[u]=n;
    for(int i=0;i<n;i++){ //seed initial edges
        flow[u][i]+=am[u][i];
        flow[i][u]-=am[u][i];
        excess[i]+=am[u][i];
        if(excess[i]>0&i!=v)q.push(i);
    }
    while(!q.empty()){
        int on=q.front();
        q.pop();
        while(excess[on]>0){ //drain this node
            if(next[on]==order[on]){ //seen all edges. do a relabel
                int mh=INT32_MAX;
                for(int i=0;i<order[on];i++)
                    if(am[on][amc[on][i]]-flow[on][amc[on][i]]>0)
                        mh=min(mh,height[amc[on][i]]);
                height[on]=mh+1; //the actual relabel!
                next[on]=0; //reset the pointer back to the first edge
            }
            int j=amc[on][next[on]];
```

[65]The vector-based adjacency list might also suffice, however we choose the matrix-based version here simply to demostrate its use.

```
            if(am[on][j]-flow[on][j]>0&&height[on]>height[j]){ //push!
                //only add to queue if we're the first to push to this
                    node
                if(excess[j]==0&&j!=u&&j!=v)q.push(j);
                int r=min(excess[on],am[on][j]-flow[on][j]);
                flow[on][j]+=r; //the actual push!
                flow[j][on]-=r;
                excess[on]-=r;
                excess[j]+=r;
            }
            else next[on]++;
        }
    }
    return excess[v];
}
```

Optimizing to $O(v^3)$ It turns out that through a quirk, the above presented algorithm accidentally runs in $O(v^3)$ time, better than the advertised $O(v^2e)$. The choice of a FIFO queue for selecting a node from which to drain the excess enables us to prove a tighter bound.

In order to see why this is true, consider the execution of the algorithm in phases, where each phase consists of draining the nodes on the queue when the phase starts. As each node can only be on the queue once when the phase starts, and each drain results in at most one non-saturating push, the number of non-saturating pushes in any phase is $O(v)$. We will ultimately show why the total number of phases is $O(v^2)$.

Now consider those phases by whether they have any relabel operations.

- Recall that regardless of the magnitude of the change in height during relabel operations, the the total change in height over the course of the algorithm is $O(v^2)$.[66] Consider also the maximum height of any node with excess. As the height gained via relabel is limited, so must increase in this maximum height.[67]. As the total number of relabels is limited, The number of phases in which we perform a relabel is $O(v^2)$.

- If a phase does not have a relabel operation, it means that all nodes in that phase were drained of their excess. Consider the node drained during this phase which had the highest height. It must have drained to some node with height at least one lower than it. As such, the maximum height node with excess must have decreased by 1. As this maximum height value must be non-negative, can only increase by a total of $O(v^2)$,[68] and

[66]since the maximum height of all nodes is v nodes is $2v$

[67]For that maximum value to increase during a relabel, the relabeled node must now have the highest label. The amount this highest label increased must be less than or equal to the change in the height of the relabeled node.

[68]See justification in the first bullet point

must decrease by 1 for each phase considered here, the total number of these non-relabel phases must also be $O(v^2)$.

As the total number of phases of the algorithm can be bound by $O(v^2)$, and each phase has $O(v)$ non-saturating pushes, the total number of non-saturating pushes, and therefore overall algorithm runtime, can be bound to $O(v^3)$.

A similar cubic bound can be achieved by choosing to drain the node which was most recently relabeled. This can be implemented with a linked list of the nodes, where when a node is relabeled, it is removed from its current location and inserted at the head of the list.[69]

Optimizing to $O(v^2\sqrt{e})$ Performing even better than FIFO we select the node with the highest height to drain. If so selected, we achieve a bound strictly better than the above $O(v^3)$, and which is potentially significantly better in sparse graphs.

As with the proof of the FIFO runtime, consider the algorithm in phases, where phase boundaries are changes in the max height of any node with excess. This change occurs when a node is relabeled or when all nodes with excess at a given height have been drained. We recap that the total increase in height of all nodes during relabel operations is $O(v^2)$,[70] and therefore decreasing the max height when we've drained nodes can simillarly only happen $O(v^2)$ times while still remaining positive. This limits us to $O(v^2)$ total phases.

To help us, we consider the concept of an ”ancestor” of a node. Within each phase, flow moves from higher height nodes to lower ones. The draining of each node ends with a non-saturating push, and we consider the graph of the edges which received the most previous non-saturating push from its source node. These edges form a set of trees,[71] and we count the number of ancestors of any given node in this tree. In this scheme, ancestor both aligns with the standard definition in a tree,[72] and intuitively represents any node who made a non-saturating push with some unit of flow which might have ended up at this node.

From a high level, we generate a set of constructs that allow us to split non-saturating pushes up into separate categories based on a parameter X. We solve for the optimal value of X.

The Potential Function In order to help break up nodes later, we use a function which has differing behavior depending on whether the number of ancestors of a node is greater or less than X. Consider the sum over all nodes with excess in the graph of $max(0, X - A(i))$ where A is the number of ancestors of a given node. Note that this function is only valid on the range $[0, X]$. We can track changes to this function as follows:

[69]See CLRS for a complete analysis of this method, called *relabel-to-front*.

[70]The max height is $O(v)$ and there are v total nodes. This is a slightly different thing than the total number of relabel operations, which is also $O(v^2)$.

[71]a forest

[72]the count of all nodes who have a directed path to this node

1. After a node is relabeled, its ancestors are "cleared". At worst, we increase the overall sum by X due to $A(i)$ becoming 0.

2. After a saturating push, at worst the receiving node now has excess and increases the overall sum by X.

3. A non-saturating push from a node with fewer than X ancestors removes source node and adds the destination node. As the destination must have more ancestors than the source,[73] the value of the function decreases by at least 1. We will call this an early push.

4. A non-saturating push from a node with at least X ancestors does not change the function, as in such a case, the value will remain at 0 both before and after.[74] We will call this a late push.

Note that in some cases the destination node already has excess when we attempt to push to it. This may result in an additional decrease in the sum than what is described. This does not impact the following analysis.

Class 1: Late Pushes As all nodes drained during a phase have the same height,[75] the set of ancestors for each must be node-disjoint.[76] In any node-disjoint graph decomposition, for any X there are at most v/X components with X nodes or more.[77] As each component only has a single node which will be drained in this phase, it limits the late pushes to the maximum number of components, which is v/X per phase and therefore $O(v^3/X)$ overall.[78]

Class 2: Early Pushes Consider the potential function. The amount of increase by relabels is a total of $O(v^2X)$, and the amount of increase by saturating pushes is $O(veX)$.[79] The total increase in the function is thus $O(v^2X + veX) = O(veX)$. As early pushes decrease the value of this function by at least one, we have at most $O(veX)$ early pushes.

Computing X In order to come up with the tightest bounds, we will balance the two distinct complexities we came up with: $O(v^3/X)$ for class 1 and $O(veX)$ for class 2.

- $v^3/X = veX$
- $v^2 = eX^2$

[73] because we defined ancestors based on where non-saturating flow went

[74] Note that if the pushing node has at least X ancestors, the receiving node must as well.

[75] by definition

[76] i.e. not share any nodes. This must be true as each node has only one outgoing edge by definition, and none of the drained nodes in this phase can be ancestors of one another.

[77] This is basic division. For given sized groupings of any fixed number of objects, the total number of groups is limited.

[78] because we only have $O(v^2)$ phases

[79] Both values are arrived at due to the number of that type of operation and the worst case increase of X for each.

- $\sqrt{v^2/e} = X$
- $v/\sqrt{e} = X$

Substituting X back in to the runtime of any class yields the intended runtime of $O(v^2\sqrt{e})$.

Intuition The proof is relatively difficult to grasp initially, even though it builds on concepts that are used in the earlier push-relabel based proofs. The important intuition is at the core.

- non-saturating pushes form a tree.
- In a tree (or any other graph of node-disjoint paths), the number of edges close to any given node is limited by the edges in the graph, and the number of edges far away from a node is limited by the number of paths in the graph. The number of paths in the graph is limited by the number of nodes.
- The most optimal way to balance "close" and "far" is based on the square root of the number of edges.

This is similar to the intuition used in optimizing array range-queries by precomputing ranges sized at the square root of the array size. More generally, the square root arises from the fact that the minimum value of the sum of two factors of a value X is $2\sqrt{X}$.

Implementation The implementation largely follows closely to the earlier FIFO node selection algorithm with a more complicated selection scheme to identify the excess node of highest height. Some sources recommend rescanning the nodes after each relabel to identify ones of highest height. This does not yield the intended performance, as we have $O(v^2)$ relabels, and doing v work for each immediately puts us over budget at $O(v^3)$. Instead, we must carefully track nodes which have excess at each height. We accomplish this with a queue of nodes at each height.[80]

As a relabeled node immediately becomes an active node of greatest height, we can find the highest-height queue after a relabel in $O(1)$ time. In order to find the next lowest height, we simply decrement the value of the max height until we find a height with a non-empty queue. While it may seem this is prohibitively slow, it is easy to see it is not by multiple arguments:

- All pushes happen to nodes at height one lower than the pushing node. This guarantees there is always an active node at height one less, except in the case where we have pushed to the source. In such cases, we may have to search through $O(v)$ heights to find the next highest height. Fortunately,

[80]This can be implemented even faster using a vector operating as a stack.

there are at most $O(v)$ phases in which we drain to the source, limiting the number of such searches to a total time of $O(v^2)$.[81]

- The total increase in max height over the course of the algorithm is $O(v^2)$,[82] so the amount of we decrease this value must be similarly bound. Therefore, with $O(v^2)$ relabels, the time to find the next lowest height amortizes to $O(1)$ per relabel.

Listing 4.31: C++

```
//assume adjacency matrix
int flow[SZ][SZ];
int amc[SZ][SZ];

int max_flow(int n,int u,int v){
    int height[SZ]={},excess[SZ]={},next[SZ]={},order[SZ]={},h=0;
    queue<int> q[2*SZ];
    for(int i=0;i<n;i++)for(int j=0;j<n;j++)
      if(am[i][j]||am[j][i])amc[i][order[i]++]=j;
    height[u]=n;
    for(int i=0;i<n;i++){
        flow[u][i]+=am[u][i];
        flow[i][u]-=am[u][i];
        excess[i]+=am[u][i];
        if(am[u][i]>0&&i!=v)q[0].push(i);
    }
    do{ //This loop exits when we reach height 0
        while(q[h].empty()&&h>0)h--; //find next lower height
        int nh=h; //use this to determine if there was a relabel
        while(nh==h&&!q[h].empty()){ //relabel or no more nodes
            int on=q[h].front();
            q[h].pop();
            while(excess[on]>0){ //drain the node.
                if(next[on]==order[on]){
                    int mh=INT32_MAX;
                    for(int i=0;i<order[on];i++)
                  if(am[on][amc[on][i]]-flow[on][amc[on][i]]>0)
                     mh=min(mh,height[amc[on][i]]);
                    height[on]=mh+1;
                    nh=height[on]; //set "next height"
                    next[on]=0;
                }
                int j=amc[on][next[on]];
                if(am[on][j]-flow[on][j]>0&&height[on]>height[j]){
                    if(excess[j]==0&&j!=u&&j!=v)q[height[j]].push(j);
```

[81] We could also solve this problem by "caching" the highest height node of height less than v, and "jump" down to that value in $O(1)$ time after a series of pushes to the source.

[82] The amount a given vertex height increases over the course of the function is bound by $O(v)$, and there are v nodes.

```
                    int r=min(excess[on],am[on][j]-flow[on][j]);
                    flow[on][j]+=r;
                    flow[j][on]-=r;
                    excess[on]-=r;
                    excess[j]+=r;
                }
                else next[on]++;
            }
        }
        h=nh;
    }while(h>0);
    return excess[v];
}
```

4.3.2.3 Min-Cut

Consider the following problem:

> Given a weighted graph, and nodes u and v, what edges whould we delete from the graph such that there is no longer any path from $u \to v$ and the sum of weights the deleted edges is minimized?

This is known as the *Minimum Cut* problem and exactly equals the maximum flow from $u \to v$.

To understand why this is the case, consider what happens as we approach the end of a run of any Ford-Fulkerson based algorithm. As we try to find paths, we run into a "wall" where any edge we try is filled to capacity, preventing us from further reaching the source. This wall represents a bottleneck of sorts, and also represents the minimum cut.

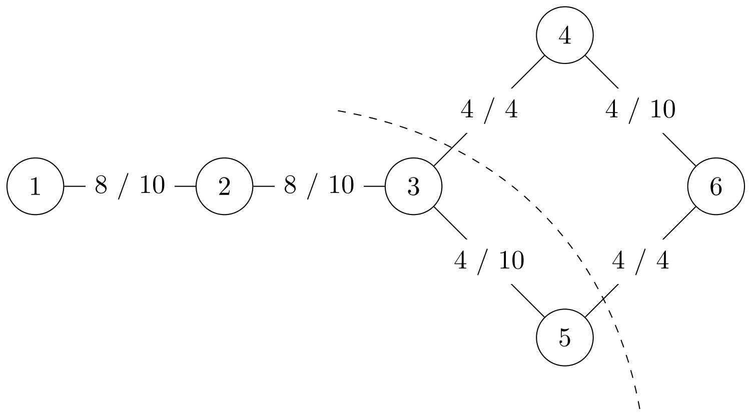


Figure 4.76: A graph with its minimum cut indicated. Note that this cut crosses edges which are filled to the brim, and are the proverbial "wall" preventing additional flow from being pushed to the sink. The cut of 8 equals the maxflow.

Intuitively, the min-cut must be across edges which are filled to capacity. This means we can find the cut iteratively.

1. Start by creating a cut of just the edges emenating from the source.
2. If any edge along this cut is not saturated, push the cut beyond the node at the other end of this edge.
3. Repeat the process until all edges along the cut are saturated.

While this method is correct, it is also difficult to implement. Instead, we can realize that as the minimum cut represents the bottleneck of the graph, then all nodes on the source side of the cut must be reachable via a BFS of edges with remaining capacity. Similarly, no nodes on the sink edge of the cut are so reachable. Therefore, we can simply BFS from the source along such edges,[83] and the list of visited nodes exactly equals the nodes on the source side of the cut.

After this point, any edge leaving one of the nodes in this set towards a node which is not part of the set is necessarily filled to capacity[84] and part of the minimum cut.[85]

4.3.2.4 Bipartite Matching

The bipartite matching problem asks us to take a bipartite graph and find a way to match as many nodes from the two sides as possible, such as if we were matching people with job openings.

[83]as we have been doing all along through the execution of Edmonds-Karp

[84]otherwise we would have been able to BFS to it

[85]If we are using one of the later described Push-Relabel family of algorithms, we can also simply identify nodes whose height is greater than or equal to the source height. These are nodes who must have drained excess back to the source, and therefore are on the source side of the cut.

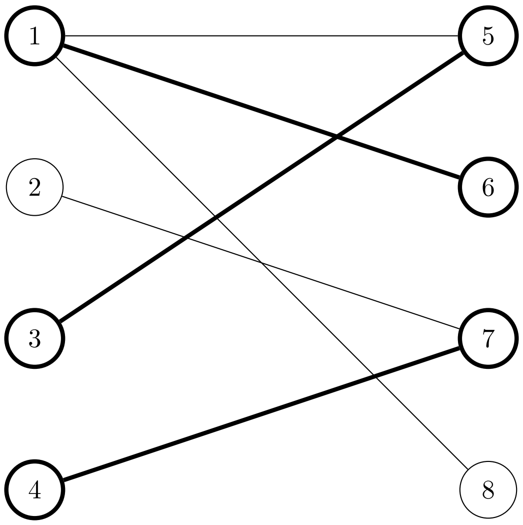


Figure 4.77: A bipartite graph with a maximum matching of 3. One possible matching is highlighted.

Consider the following transformation of the above graph, noting the "middle" of the graph is identical as above:

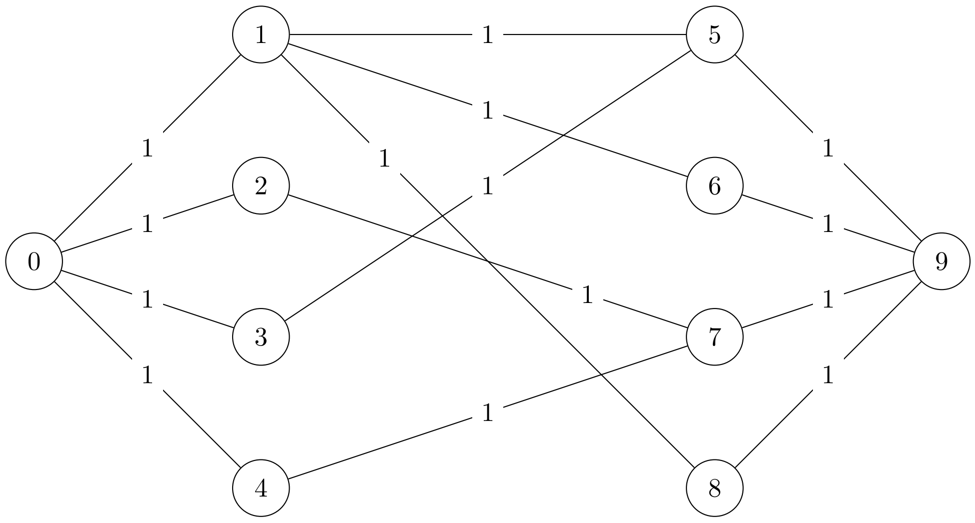


Figure 4.78: Create a node on the left hand side with an edge weight 1 to all nodes on that half, and a corresponding node on the right hand side.

Running maxflow on the fransformed graph will result in the maximum matching of the graph. The addition of the edges of weight 1 between the source and sink ensure that each node is only matched once. Further, in a maximum matching of size n, there are, by definition, exactly n edges which bridge the two sides of the graph which use a given vertex only once. This intuitively guarantees the max flow will equala the size of the matching.[86]

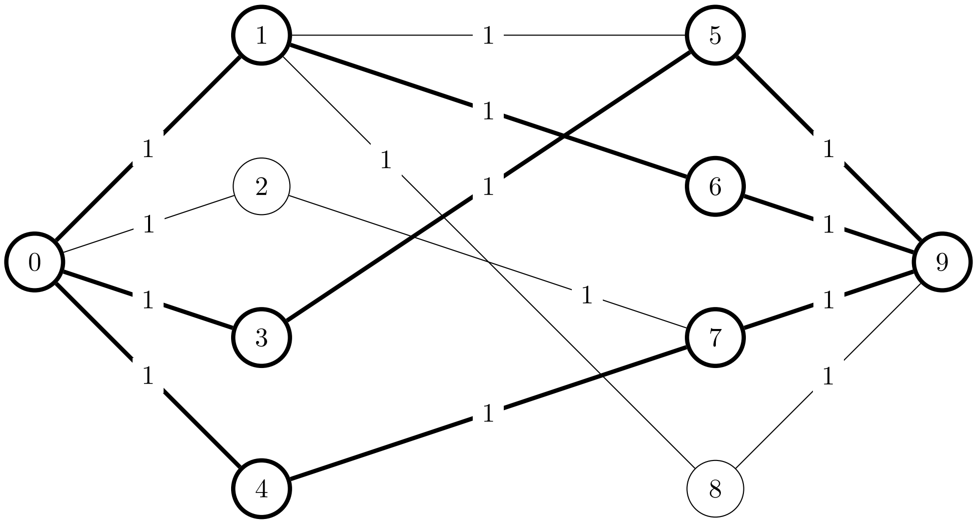


Figure 4.79: The maximum flow in the transformed graph is the maximum bipartite matching.

Minimum Vertex Cover The minimum vertex cover problem asks us to identify the minimal number of nodes to select such that every edge in a graph is touching at least one node in the set. It turns out this is exactly the maximum matching. We can see why this is the case:

- The matching cannot exceed the cover. As the matching does not use any node twice, it means that we need at least as many nodes in the cover as the matching, one each to cover the edges in the matching.
- The cover cannot exceed the matching. Considering a matching of M, if the cover exceeds this value, it would imply there is some edge which is not covered if we are limited to M nodes. If such an edge existed, we would be able to include it in the matching itself,[87] violating the assertion that the matching was maximum.

This duality between the minimum vertex cover and the maximum matching is known as *Kőnig's Theorem.* It only applies in bipartite graphs, and computing the minimum cover in other graphs is NP-complete.

[86]A formal proof would show why the max-flow cannot exceed the maximum matching and vice versa.

[87]since neither node is currently in the matching, and there is an edge between them

In order to construct the vertices in the cover itself, we use the following intuition:

1. Each node in the cover is incident to exactly one matched edge, and vice versa.[88]

2. No unmatched nodes are in the cover, as a corrolary of the first point.

3. Matched nodes with an edge to the unmatched node must be in the cover.

4. Two nodes which are matched cannot both be in the cover, again as a corollary to the first point.

These points lead to the following algorithm:

1. Start with all unmatched nodes on one side of the graph. We'll call it side A. We know these nodes cannot be in the cover, per rule 2 above.

2. Any edge from these nodes to side B must reach a matched node. We know all those nodes must be in the cover, per rule 3 above.

3. Following the matched edges back to side A, we know all these nodes must not be in the cover, per rule 4 above.

4. Following all remaining edges (necessarily numatched) from those nodes back to side B, we again know the nodes must be in the cover, by rule 2.

We can repeat steps 3 and 4 to collect a set of nodes we know must or must not be in the cover. Any remaining unvisited nodes on side A must be matched, and we can guarantee the corresponding matched nodes on side B must not be in the cover.[89] We can therefore conclude these remaining matched nodes on side A are also in the cover. Our constructed cover has now reached the proper number of nodes[90] and must therefore be complete.

[88]this follows from the the theorem itself, that the cover equals the matching

[89]or we would have found them in the above BFS

[90]exactly the size of the matching

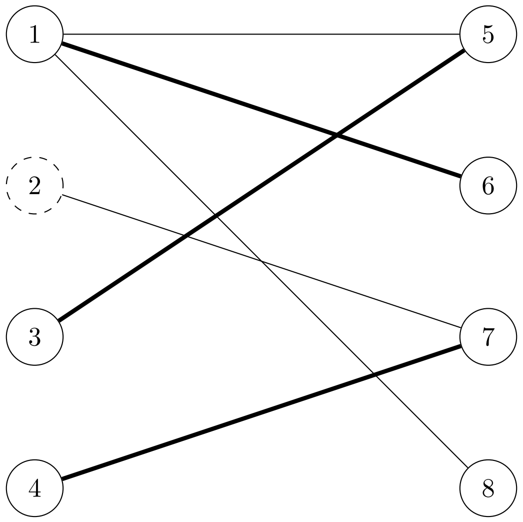


Figure 4.80: Starting the algorithm, node 2, as an unmatched node on side A, must not be in the cover.

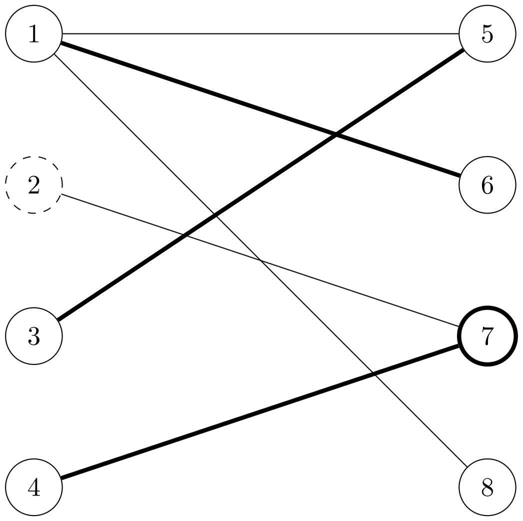


Figure 4.81: Following the unmatched edges to side B, node 7 must be in the cover.

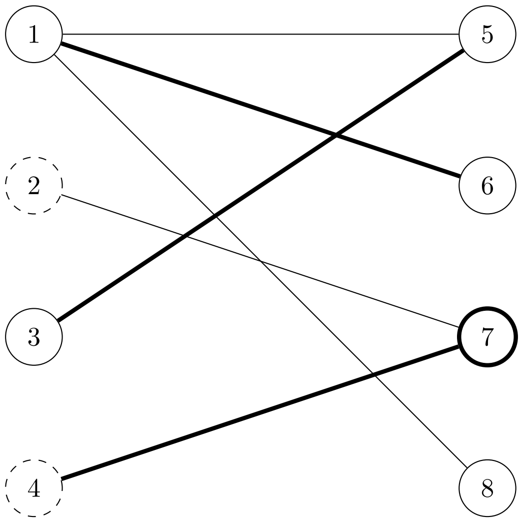


Figure 4.82: Following the matched edges back to side A, node 4 must not be in the cover.

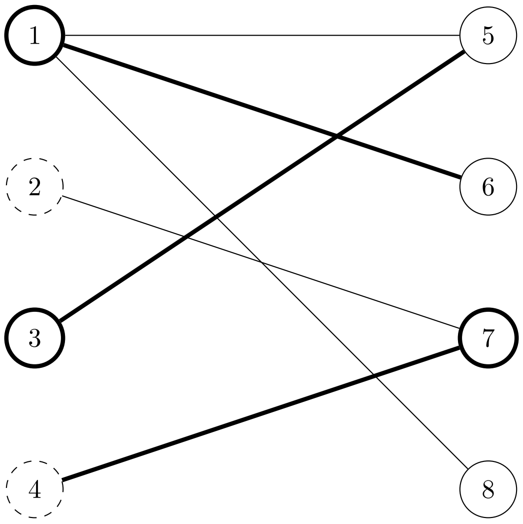


Figure 4.83: Having completed the BFS, the only remaining unvisited nodes on side A must be matched nodes, and we can include them in the cover. The cover is now complete.

4.3.2.5 Extended Matching: Using Flows to Solve Problems

Matching is one of the most straightforward problems, enabled by leveraging flows on transformed graphs. Sucht transformation can be used to solve a host of similar problems which require associating nodes of different "classes" and limiting how many nodes in each class may associate with one another. Consider the following problem:

> A pizza shop would like to serve pizza to 100 children at a birthday party. The pizza shop will make pizzas to serve to those children. Children are hungry so each consume a whole pizza. Children are also picky and will only eat pizzas with certain toppings. The pizzas are made by 200 special machines which each can only make a single pizza over the course of the party, and worse yet, each machine can only make a specific kind of pizza, defined by the crust type and topping (e.g. thin crust pepperoni, or thick crust pineapple). The restaurant further is limited in its quantity of any given crust type, such that it cannot make more pizzas of that crust type than that limit course of the party.
>
> Given these constraints, how many children can get their own pizza at this party?

This problem lends itself to be constructed as a flow problem for several reasons:

- The problem indicates some amount of matching items of various classes: Children to ingredients, pizza machines to ingredients and crust types, and ultimately children to pizza machines.
- The problem provides specific constraints on certain aspects of the classes. Children consume exactly one pizza, each machine makes one pizza, each machine is limited to one crust type, and each crust can only be used a certain number of times.
- The problem has a limit in the low-hundreds.

We can therefore construct a graph as follows:

1. Create a source node.
2. Create a node for each child. As we know each child is limited to exactly one pizza, an edge of capacity 1 limits any child to a single path through the eventual graph.
3. Create a node for each pizza machine. As we know which children would eat the pizza from any given pizza machine based on the toppings, we can draw an edge between each child and the machines they would eat pizza from. This ensures we match children to pizza machines. While the weight ultimately will not matter,[91] we use weight 1 for convenience.

[91] since we have already restricted each child to 1 pizza in the previous step

4. Create a node for each crust type. As each pizza machine is used only once, draw an edge of weight 1 from each pizza machine to its matching crust type. This encodes a matching of crust type to pizza machine, and limits the machine to a single usage.
5. Create a sink node. Draw an edge to each crust type with a weight equal to the number of times that crust type may be used.

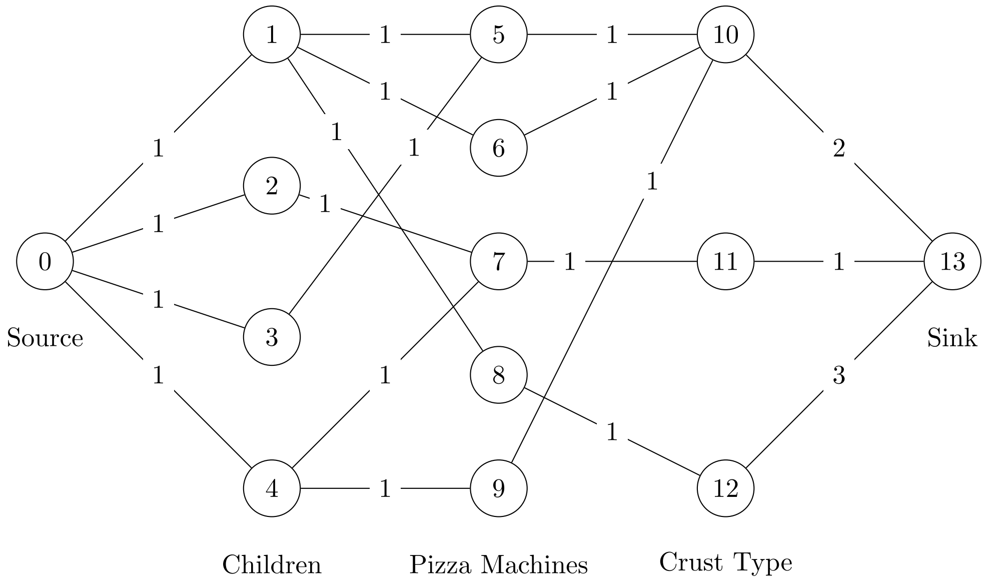


Figure 4.84: A path from the source, to a child, to a pizza machine, to a crust type, to the source indicates that we will serve that child a pizza from that machine, and it will use that crust type. The first set of edges limits each child to one pizza. The second set of edges indicates which machines a child can eat a pizza from. Third set of edges connects a pizza machine to a crust type and limits us to one pizza per machine. The last set of edges limits the amount of a given crust type.

As with the basic bipartite matching, the maximum flow through this graph indicates the maximum number of children which can be "matched" with some machine, given the other constraints.

Such constructions are powerful and can solve many problems depending on the exact graph layout. As indicated above, there are several strong hints that a graph flow may be an avenue of investigation, and once that is decided, the two most difficult parts of solving are determining the overall structure of the graph and keeping track of the indices of all the nodes as the code is being written.

4.3.2.6 In-Out Nodes

Consider again the above pizza problem with a small modification:

> Each pizza machine makes a single pizza, but there are multiple crust types that machine can choose from, as opposed to just a single type.

A reasonable adjustment to thye graph would be to add multiple edges to each pizza machine, one to each crust type that machine can make. The graph might look like this:

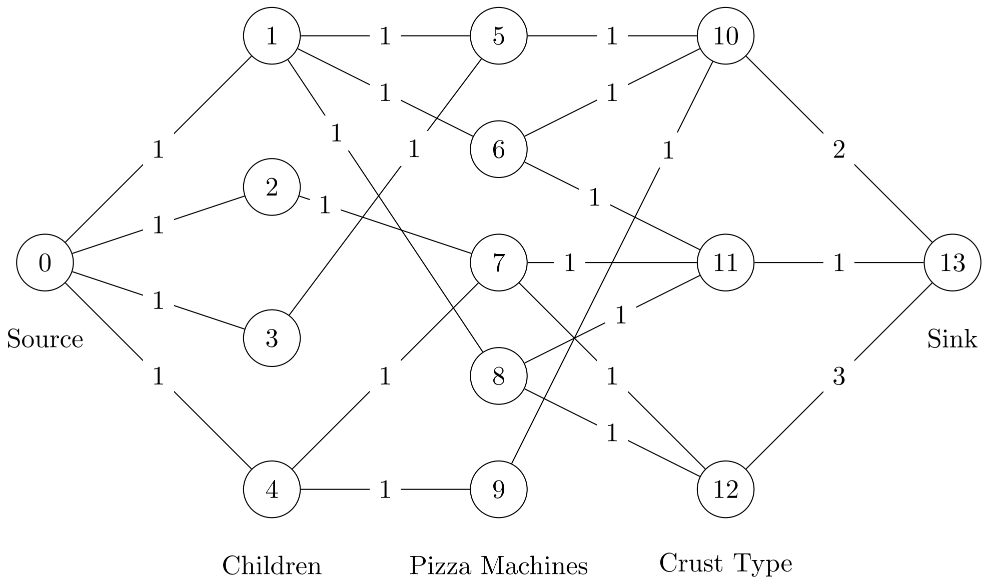


Figure 4.85: Now each pizza machine does not have just one edge going to the group of crust nodes, but potentially multiple.

While this adjustment seems sensible, a critical impact is that pizza machines, which used to be guaranteed to only have a single unit-weight edge leaving them, now may have multiple edges entering *and* multiple edges leaving. This means we might have two valid paths flowing through a single pizza machine, which violates the constraint that a pizza machine is only used once. Here is an example of such a flow:

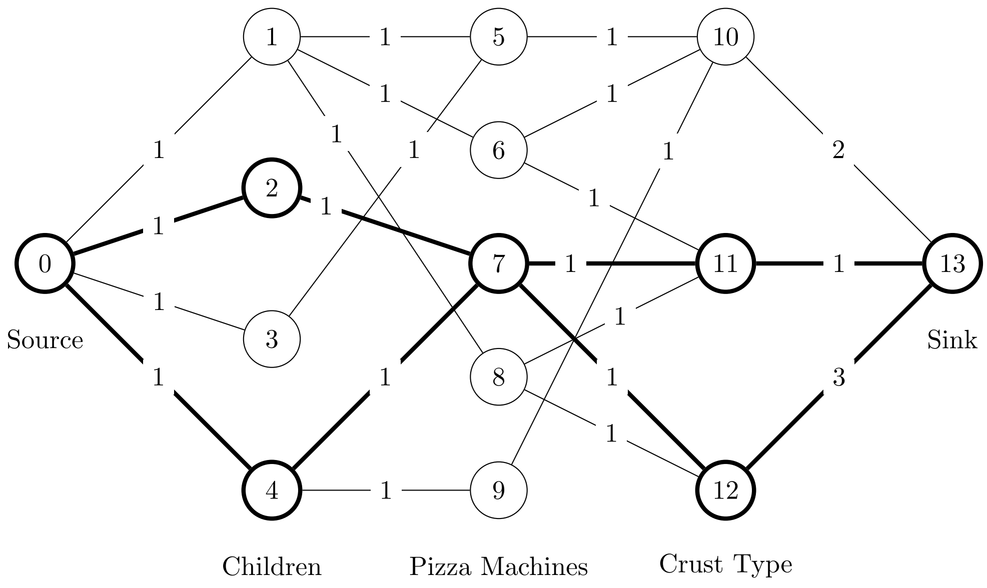


Figure 4.86: As node 7 has two edges entering to account for the two children who could eat pizzas from this machine, and two edges leaving to account for the two crust types this machine can make, we have accidentally created a flow which can incorrectly use this machine twice.

In order to resolve this discrepency, we use a concept called *in-out nodes*. In this scheme, in order to limit the flow which crosses a given node, rather than the flow which enters it or exits it, we split that node into two sub-nodes. All incoming edges arrive at the "in" node, and all outgoing edges leave from the "out" node. The two nodes are connected by a single edge with a capacity limiting the number of times the original node should be crossed. The graph with the machines converted to in-out nodes looks as follows:

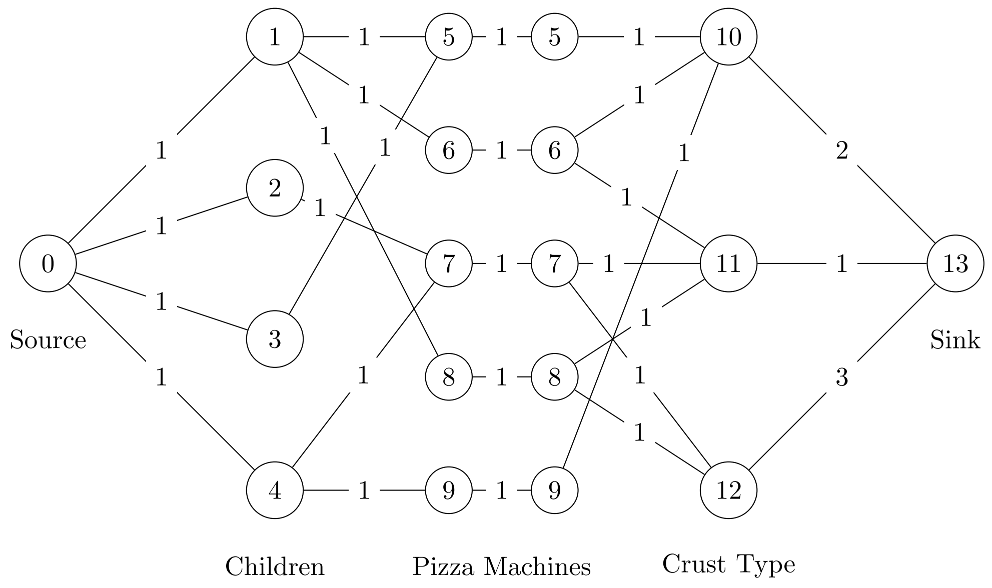


Figure 4.87: With this construction, each child is limited to 1 pizza as it only has a single incoming edge. Each pizza machine is limited to a 1 pizza by the edge joining its in and out nodes. Each crust type is limited by the outgoing edges to the source. In theory, the other edges may be infinitely large and not impact the outcome.

With the in-out nodes, the previously invalid flow which double-used machine 7 is prevented by the unit edge connecting the in and out nodes representing node 7. Again, the largest challenge is to keep track of the index that represents any node when constructing the graph representation.

4.3.2.7 Node Disjoint Path Cover

A *path cover* is a set of paths which visit every node in a graph. The paths are said to be *node-disjoint* if none of them share a vertex. The *node-disjoint path cover*[92] probelm asks us what is the minimal number of paths we need to visit every node exactly once.

[92] or vertex-disjoint path cover

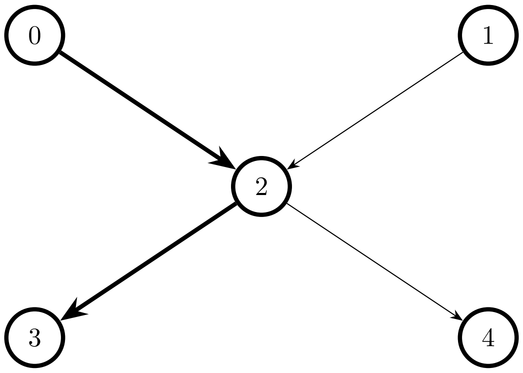


Figure 4.88: This graph can be covered with 3 paths. $0 \to 2 \to 3$, and a path each containing a single node 1 and 4.

In order to help solve this problem, we transform the graph. We create two sets of n nodes, a source set, and a destination set. If there is an edge from node u to v in the original graph, we draw an edge from the equivalent node u in the source set to the equivalent node v in the destination set. The transformed graph looks as follows:

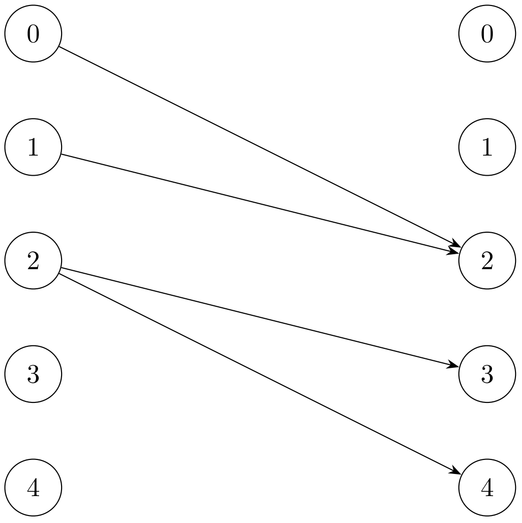


Figure 4.89: This bipartite graph encodes the same information as the original graph, just in a different way.

Lets consider if we compute the maximum matching in this graph, using the above described max-flow techniques.

1. We know that only one edge will leave each node on the left, and only enter each node on the right. If we consider the equivalent in the original

graph, this means only one path entering this node is used, and one path leaving it is used. More specifically, in a matching, only one path traverses this node.

2. If a node on the left is left unmatched, it means there is no way to increase the matching by departing to some other node. This node is the end of some path in the original graph.

3. If a node on the right is left unmatched, it means there is no way to increase the matching by entering from some other node. This node is the start of some path in the original graph.

By computing the maximum matching, we have found the maximum arrangement such as many nodes as possible are matched and therefore as few nodes as possible start or end a path in the original graph. Further, as each node is only used at most once in the matching, we have guaranteed the paths represented by this matching are node-disjoint.

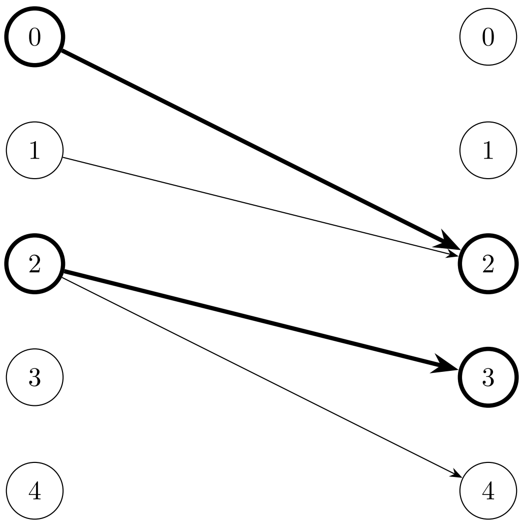


Figure 4.90: The matching which is equivalent to the earlier un-transformed graph. The matched edges from $0 \to 2$ and $2 \to 3$ are equivalent to the $0 \to 2 \to 3$ path in the original graph. No edge leaves 1, 3, and 4 in the matching, and thus a path ends at those nodes. No edge enters 0, 1, and 4 in the matching, and thus a path starts at those nodes.

The value of the minimum path cover is the number of unmatched nodes on either side of the graph as that is the number of times a path in such a cover starts or ends. **This transformation and algorithm is only valid in directed, acyclic graphs.**

Path Cover in non-DAGs As noted immediately above, this transformation and algorithm only is a solution in DAGs, but why is that? Consdier the following graph and its transformation.

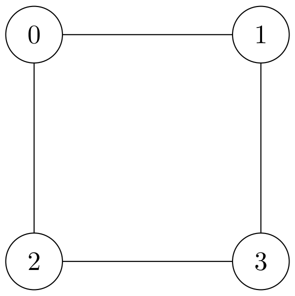


Figure 4.91: A simple loop with a clear minimum path cover of 1. Note that the edges are undirected.

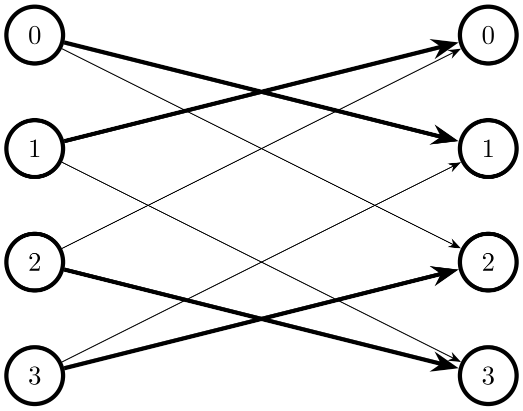


Figure 4.92: The found matching makes a "cycle" between 0 and 1, and another cycle between 2 and 3.

The algorithm which works on DAGs not only produces the wrong answer (it would say 0 since there are no unmatched nodes), but also in an intuitive sense, fails to generate the proper path (a single cycle of all 4 nodes) as it might find smaller intermediate cycles. In order for the transformation to work, we must ensure that the edges in the transformed graph never arrive back at a node which might have already been on our current path, i.e. there must be a topological ordering the node.[93] This implies that the graph must be a DAG.

The solution in non-DAGs is NP-complete.[94] This is easy to see as answering the question *is there a single node-disjoint path which covers the entire graph* is

[93]See the section on DAGs and topological sort.

[94]See NP-Completeness chapter

equivalent to looking for a hamiltonian path.

Non-Solution with Topological Sort A common suggestion to solve this problem suggests to perform a traversal of the graph in topological order, only counting a "new" path if no parent of this node has been previously used to create a path to some other node. Such a solution ultimately fails. Consider the following case:

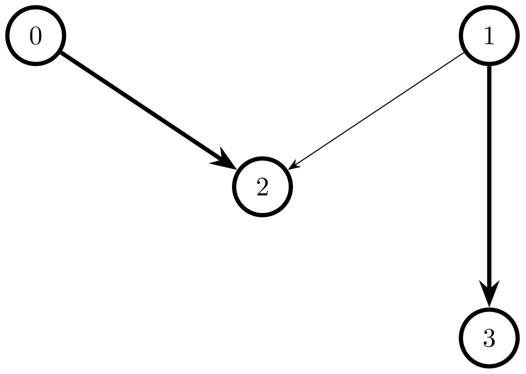


Figure 4.93: A curious graph, and it's minimum path cover

In this graph, with this algorithm, the following non-opimal case could occur:

1. Process node 0, note that a path must start there.
2. process node 1, note that a path must start there.
3. Process node 2. As there are two nodes which could be it's "parent", we arbitrarily select 1.
4. Process node 3. Node 1 has already been used, so we must start a new path here.

The above flow produced a cover with 3 paths instead of the minimal 2. THe root of the problem is that there exist cases where we may choose either parent but have no insight as to which to choose. Flow algorithms allow us to undo earlier earlier bad decisions by cancelling flow,[95] but a topological sort has no such affordance.

4.3.2.8 Min-Cost-Max-Flow

4.3.2.9 Hungarian Matching

[95]and in this case, would allow us to undo the matching of 2 and 1, and instead match 2 and 0

4.3.3 Componentization

Componentization is the process of breaking a graph into components. In the transformed graph, each component can generally be treated as a node in itself. Edges between nodes in different components are treated as edges between the components themselves.

4.3.3.1 SCC

Strongly-Connected Components in a graph are the groups of nodes in a directed graph such that each node in the group has some path to every other node in the graph. Each such component is considered *strongly connected*.[96] The size of each group is maximal, meaning that there is no node outside an SCC for which there is a path both to and from the nodes within the SCC.[97]

[96]This differs from a clique, which must have an edge between all pairs of nodes in the group as opposed to simply a path. Computing cliques in a graph is NP-hard.

[97]If such a node existed, it would be reachable from every node in the SCC, and it would be able to reach every node in the SCC and thus be part of the SCC itself.

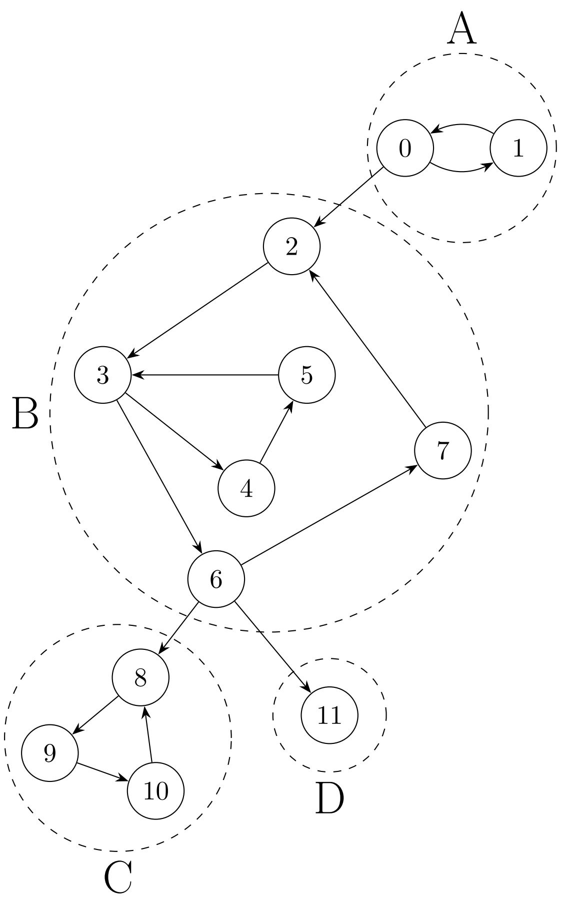


Figure 4.94: A directed graph with strongly connected components circled. Within a circle, each node has a path to every other node, and there is no node outside that circle that both has a path from and to some node in the circle.

As each SCC may only have *either* an edge to or from another SCC, the graph formed by SCCs is necessarily a DAG. Enabling algorithms which are typically only applicable on a DAG is one of the main use cases of computing SCCs.

$$\begin{tikzcd}
 & & A \arrow[dl] \\
 & B \arrow[dl] \arrow[dr] & \\
C & & D
\end{tikzcd}$$

Within this DAG, as all DAGs, nodes with no incoming edges are known as *sources*, and nodes with no outgoing edges are known as *sinks*.

Computing SCCs While identifying all the SCCs in a graph may seem difficult, it can be computed in $O(n)$ time. To see how, lets look at some behaviors of DFS with respect to known components.

Pop Order DFS is implemented with a stack. As such, we have a guarantee that all descendents of any given node are visited before that node is removed from the stack. Lets look at the order in which nodes are popped from the stack in an execution of DFS. Note we've already labeled the nodes in the order they would be visited with DFS.

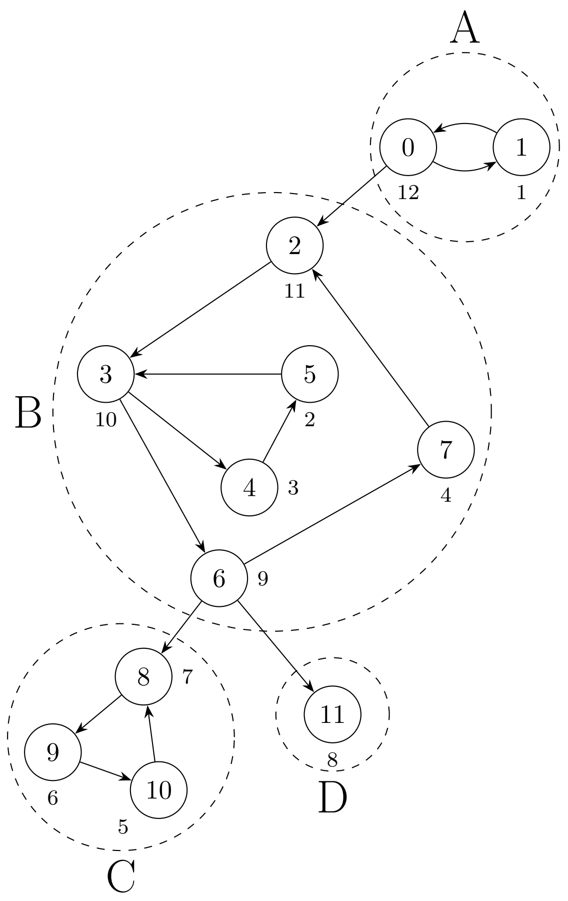


Figure 4.95: Node labels indicate the order in which nodes are visited (preorder), and the small annotations indicate the order in which the nodes are considered fully processed and popped from the DFS queue (postorder).

The small numbers are generated by walking the graph in the DFS order defined by the large node labels themselves. The small label shows the order in which we last leave the node, or when that node is popped from the DFS stack. [98]

Consider the DAG of SCCs in this graph, and a topological ordering of those SCCs: ABDC.[99] Further consider the order in which **last** node in an SCC is fully processed and popped from the stack. C is completed first, with node 8 popped seventh. D is next, with node 11 popped eighth. Then B (node 2, eleventh) and finally A (node 0, twelfth). The final ordering in which the last node in an SCC is popped is thus CDBA.

In general, order in which DFS completes processing SCCs is in reverse topological order of the SCC graph itself (ABDC vs CDBA). This makes intuitive sense as DFS reaches as far into the graph as possible before retreating, processing sinks first, then their parents, and so on.

The key takeaway is that all nodes in a given SCC are popped before the last node of any ancestor SCC is popped. All nodes of C and D are popped before the last node of B is popped, and all nodes of B are popped before the last node of A is popped.

SCC Root In the course of the DFS, some node must be the first node visited in each SCC. This node will be referred to as the *root* of the SCC and the goal will be to determine which nodes are roots.[100]

If a node is the root of a SCC, we must have the following two properties:

1. There must be some path from this node arriving back at itself. This is a necessary property of belonging to an SCC.[101]

2. Once this node is popped, we are guaranteed to have visited (and popped) all nodes in this SCC.

While the first point is straightforward, the second is slightly harder to see. We know that all nodes in the SCC must have been visited, as by definition we have a path from this node to those nodes. If one of those nodes has not yet been popped, it must mean that node was visited before this candidate root in the DFS order. This violates the assertion that the root was the first visited node in the SCC, meaning property 2 must be true.

Putting it Together Combining the above two sections, when we pop a root, we are guaranteed both to have poped every node in our SCC, as well as the root of any descendent SCC. If we could determine definitively that a node was a root, upon popping, we can declare any node we visited - which isn't

[98] Note the two node labels are the same as the what would be produced by a Eulerian walk, as used in LCA computation.

[99] or ABCD

[100] Note that the concept of a root is valid for a particular DFS, and not a general property of a strongly connected component.

[101] nodes in this SCC must be able to reach this node

already part of some other SCC - as part of the SCC rooted at this node.[102] The only remaining step is to know for sure if a node is a root.

The SCC root must have some path back to itself. Assuming we can track whether a given node has a path back to itself, how do we know that such a path implies the node is a root? Consider the example graph. Node 3 has a path to itself, but is not the root of the SCC... node 2 is. Our goal is to distinguish when a node having a path to itself is a root and when it is not.

During a DFS, any node on the stack must be an ancestor of the current node. If there is a path from a the current node to a node on the stack, it means there must be a loop, and therefore those nodes must be in the same SCC. In the example case, 2 is on the stack while processing 3. In the course of processing 3, we find a path to node 2,[103] indicating a loop including both 2 and 3. As node 2 is on the stack and an ancestor of 3, it must occur earlier in the DFS order and therefore node 3 is not a root.

The overall conclusion is a node is a root if it has a path to itself and that path has no nodes which are higher than that node in the stack.

The algorithm to identify SCCs can therefore be summarized as follows:

- Perform a DFS of the graph, tracking the earliest node in DFS order which is seen while processing all the children.
- If we are to pop a node where the earliest seen node of any of the children is the node itself, we know this node must be the root of an SCC.
- When the root of an SCC is identified, any descendent nodes which have been visited, but which have not been identified as part of some other SCC must be part of the SCC rooted at this node.

This is known as *Tarjan's Algorithm* and is one of the most common linear-time algorithms for computing SCCs.[104]

Walkthrough We will perform a walkthrough of the algorithm using our example graph. As noted before, the nodes are labeled in the order that DFS will visit them. Along with this, we track the following pieces of data:

- Whether a node is currently on the DFS stack. This will be indicated via a thick border.
- Whether a node has been visited by the DFS. This will be indicated by a small '?' aside the node.
- Whether a node has been positively identified as part of an SCC. This will be indicated by the SCC label replacing the '?'.

[102] Recall that SCC roots will be popped in reverse topological order. Therefore any descendent SCC of the one rooted at this node (and by induction, all nodes it contains) must have been positively identified

[103] via nodes 6 and 7

[104] *Kosaraju's Algorithm* is the other common choice. It defers to Tarjan here as there is no large benefit and Tarjan more easily extends to solve other problems such as BCC.

- The earliest node seen while processing any children of this node. This will be represented by a number aside the aforementioned '?'

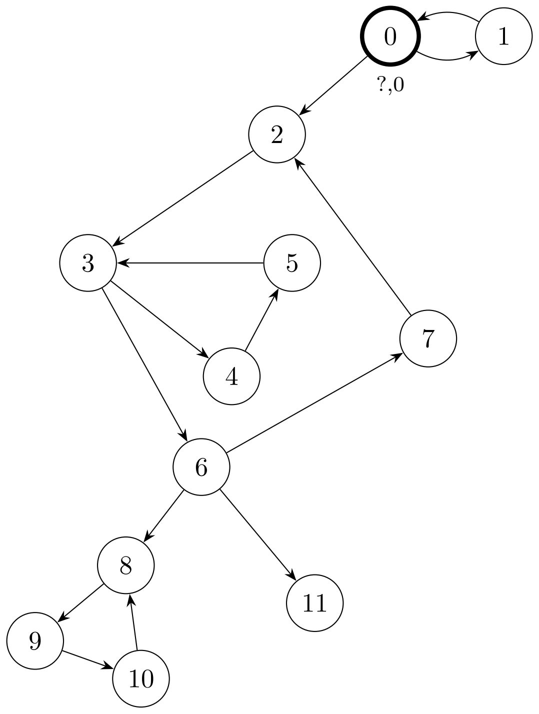


Figure 4.96: At the start, we indicate that node 0 will be on the stack, its SCC is unknown, and the earliest reachable node is itself, 0.

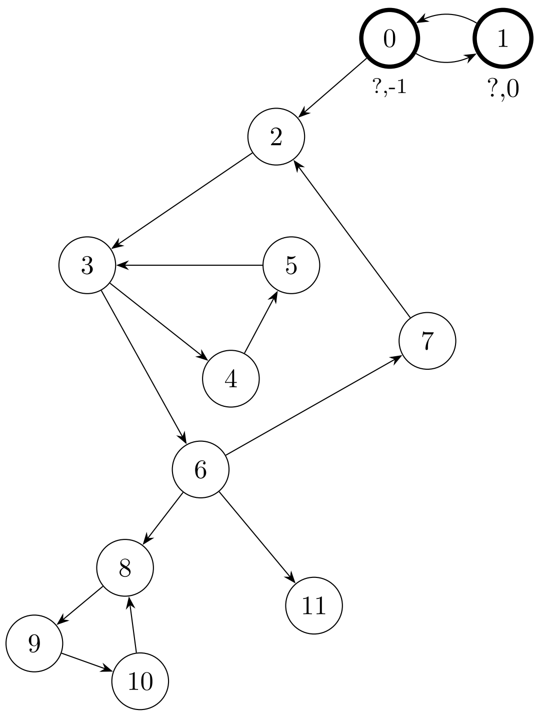


Figure 4.97: We process node 1. We do not know its SCC, and we see that the neightbor, node 0, is on the stack and cache this as the lowest seen node.

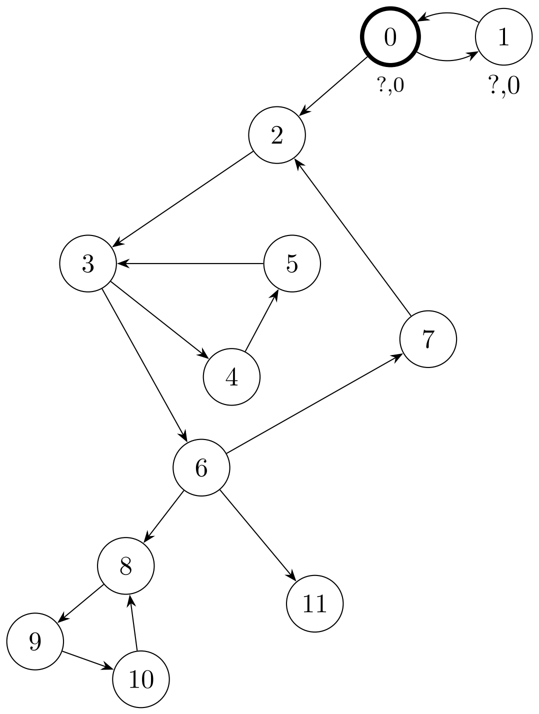


Figure 4.98: As node 1 has no more children, it is popped from the stack. It observed a path to an earlier node (0) than the node itself (1) and therefore is not the root of an SCC. The earliest seen node of 0 remains (0) as node 1 has not seen an earlier node than that. While this value equals the value of the node itself, we have not completed processing of all the children. As we are not yet popping the node, we cannot guarantee there is not some other path which reaches a higher node, nor that if this is a root, that we have seen all nodes in the SCC. We'll continue to process the other children.

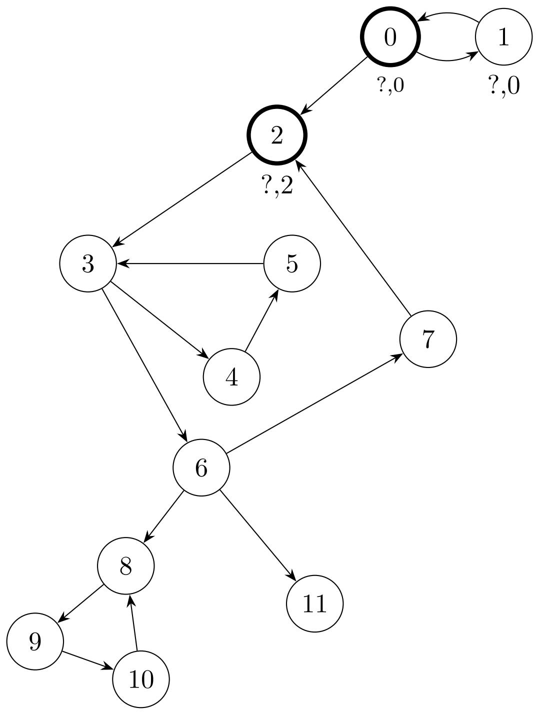
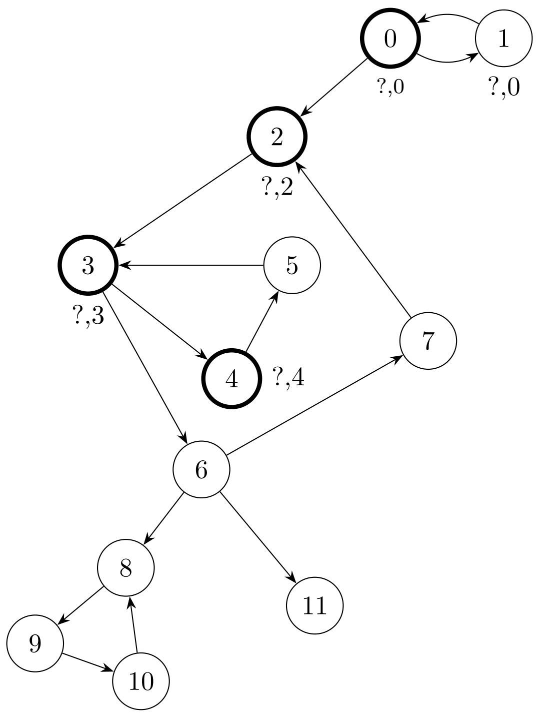


Figure 4.99: We add node 2 to the stack, and recurse.

The DFS will similarly process nodes 3 and 4, all not having a known SCC, nor having seen any earlier nodes.


0
1
?,0
?,0
2
?,2
3
?,3
5
4
?,4
7
6
8
11
9
10

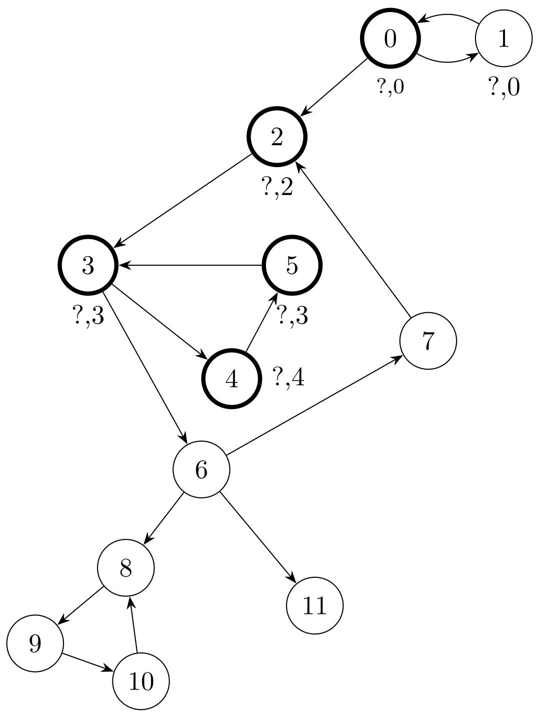


Figure 4.100: When node 5 is processed, we note that we have seen node 3, which is lower than the node itself (5).

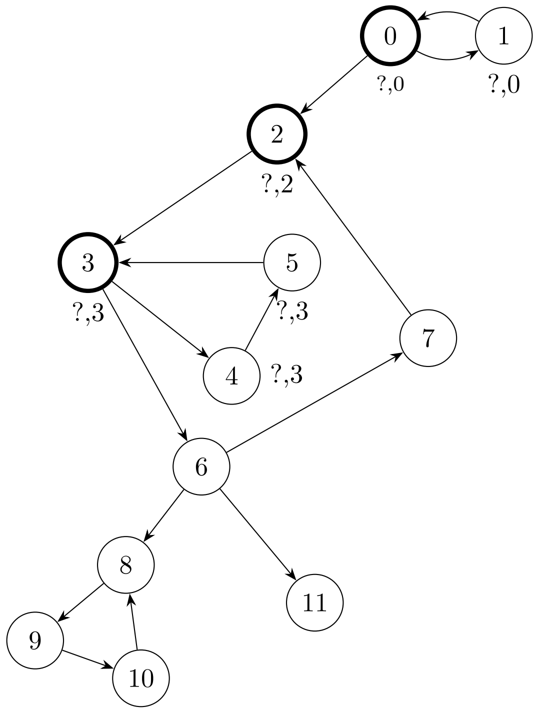


Figure 4.101: Popping 4 and 5 off the stack, we note the lowest seen node as 3. While node 3 has now seen a path to itself, as with node 0, we have not completely processed all children, so cannot know whether this node is a root.

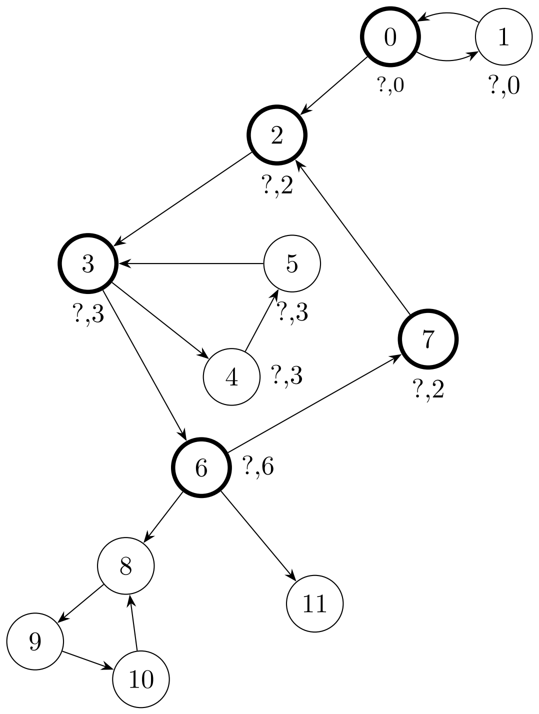


Figure 4.102: Skipping ahead to the processing of node 7, where we see a lowest seen node of 2. Similar to above, we will end up popping this off and updating the lowest seen node at 6 before following the other children of node 6.

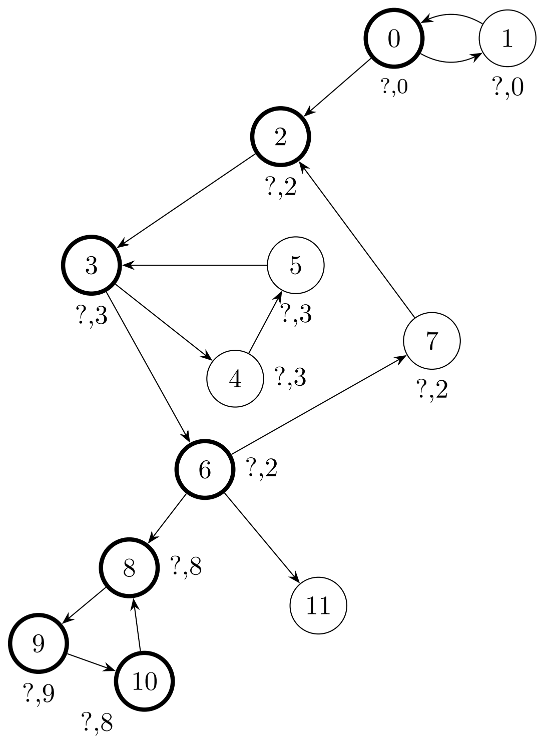


Figure 4.103: After popping node 7, we explore other children of 6, namely 8, 9, and 10. Note hte lowest seen node from 10 is 8.

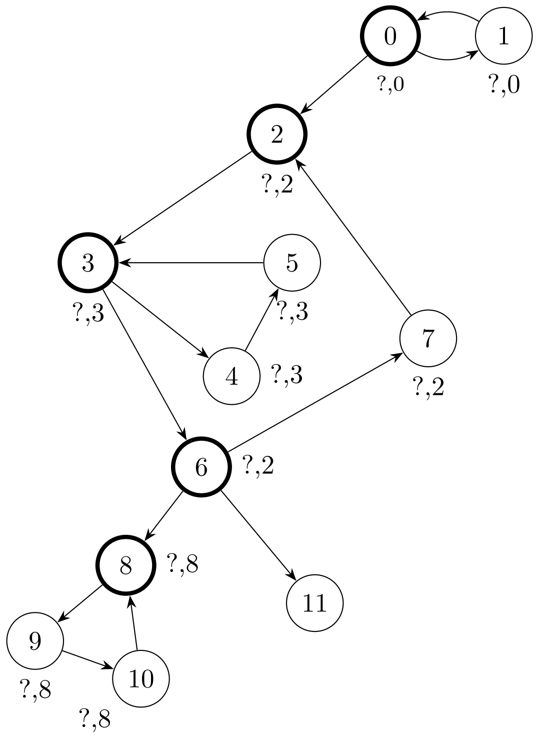


Figure 4.104: Upon popping 9 and 10, we arrive back at 8 and note that the lowest seen node is node 8 itself. Unlike the earlier cases, node 8 has no more children to explore, has a path back to itself, and is guaranteed to not have a path back to some earlier node. Therefore we can conclude it is part of an SCC, and must be the earliest node which was visited in this SCC. It must be the root of the SCC.

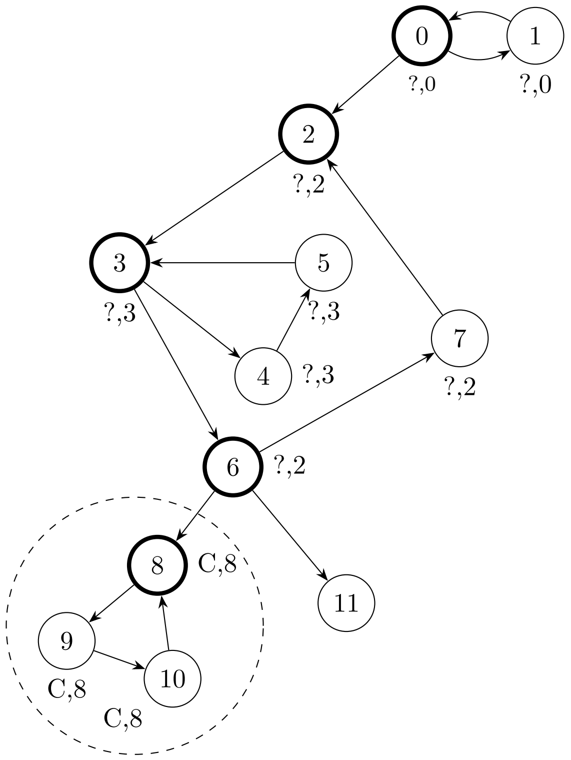


Figure 4.105: Nodes 9 and 10 are the only descendent nodes which which have an unknown SCC. They, along with node 8, will conclusively form this SCC. We will name this SCC "C", though the name is arbitrary.

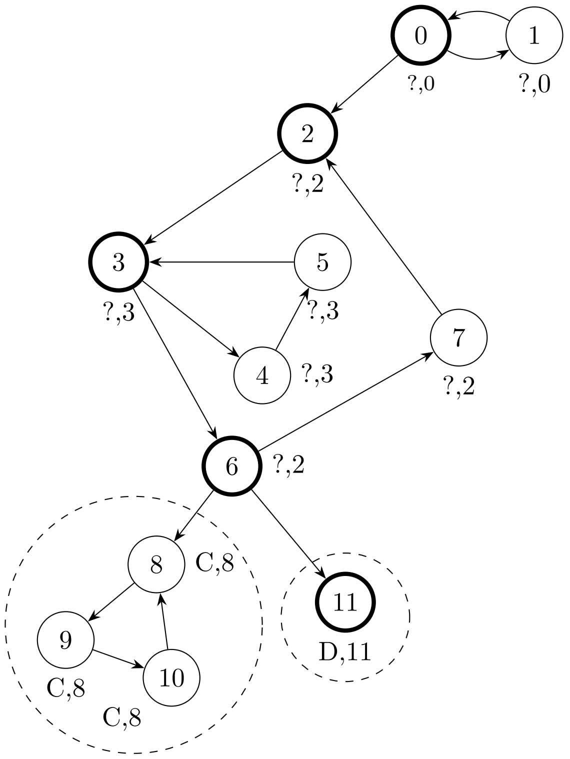


Figure 4.106: We continue the DFS to node 11. Note that the ”lowest seen” value at node 6 remains at 2, as the earliest seen value at node 8 (8) is not earlier than value node 6 currently has (2). At node 11, there are no outgoing edges, so node 11 must form a degenerate SCC of only itself. Its lowest seen node is itself.

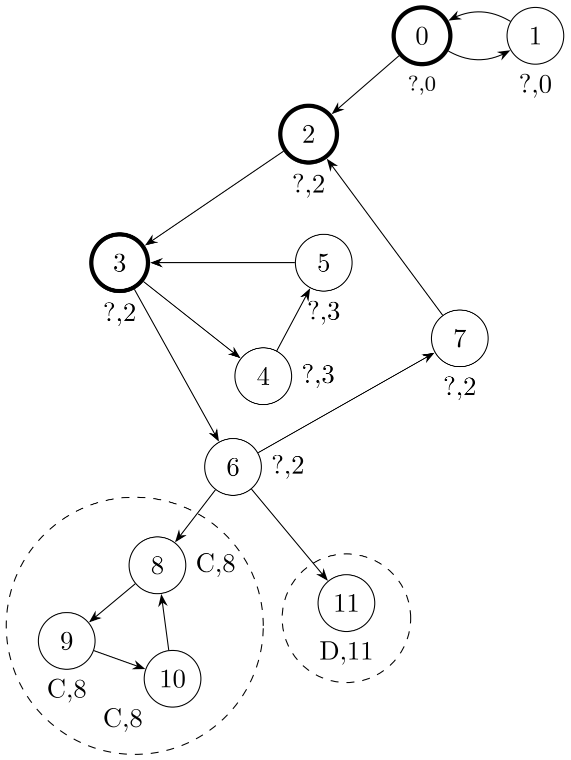


Figure 4.107: We pop nodes off the stack, and pause at node 3 to note that as its child, node 6, has seen a lower value (2), than it had previously seen (3), we must update.

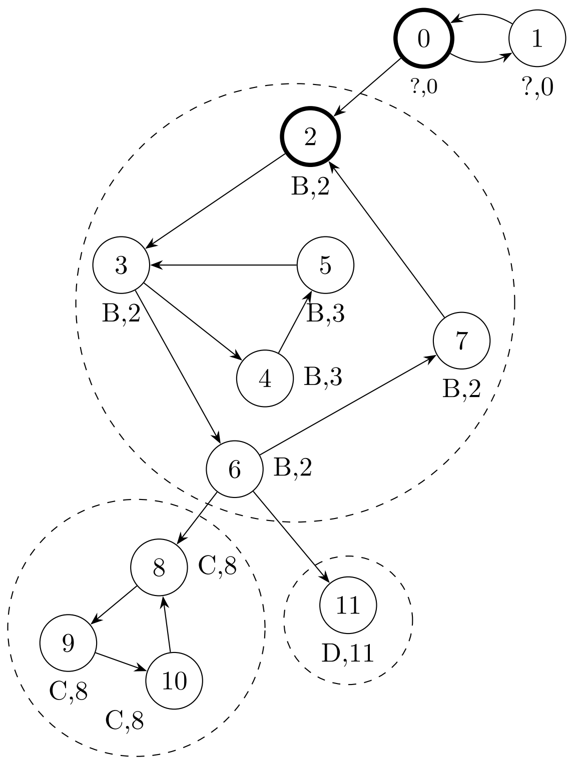


Figure 4.108: When we are ready to pop node 2, we have updated that the earliest node seen by any of its children is itself (2, as seen by node 3). As there are no more children and the earliest seen node is itself, node 2 must be the root of a strongly connected component. Any visited node with a higher label than 2 (i.e. descendent) which does not currently have an identified SCC must be in this new SCC.

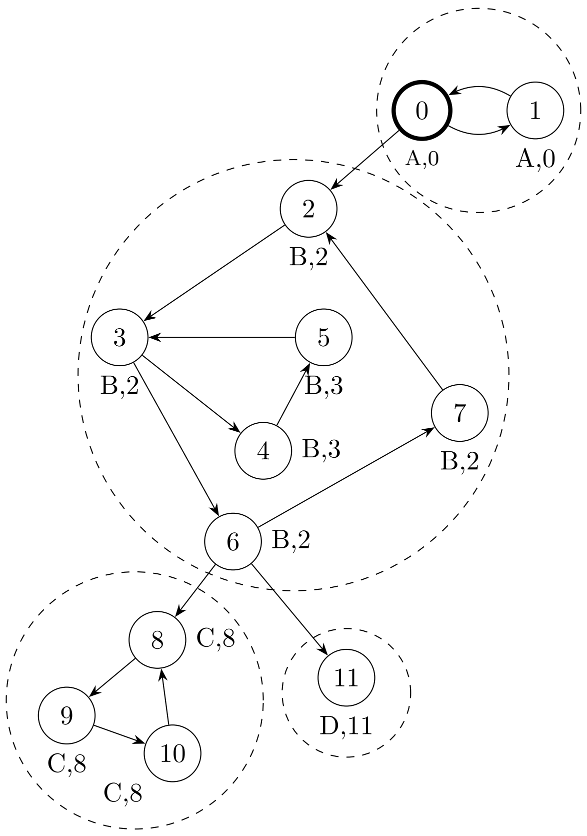


Figure 4.109: We finally try to pop node 0, and like node 2, see that the earliest seen value equals the node label. Node 1 both has an unknown SCC and is a descendent, so must be in this last SCC.

The algorithm is now complete, and we have identified the strongly connected components of the graph in a single DFS pass.

Implementation While the algorithm is somewhat intuitive, the implementation can be very finicky. There are several points to consider:

1. DFS using recursion can exceed stack limits in some contests. To avoid this, we'll recurse without making a function call, preserving the necessary state of the node and the next edge to examine in a vector.

2. One of the missing elements of the above algorithm description is how we identify the descendent nodes we've reached which aren't yet part of an SCC after we've found a root. If we simply maintained an "order" array, we'd have to search all elements with greater order than the SCC root which are not already part of an SCC, incurring additional $O(n)$ time

per SCC. One way to alleviate this would be to use a tree set, enabling identifying elements with an order greater than the root, as well as removing items when the SCC is set. This unfortunately incurs an additional $O(log(n))$ time per SCC.

In order to avoid all additional time, we maintain a second stack of nodes we visit over the course of the algorithm. Nodes are added to this stack at the same time they are added to the DFS stack, but are not removed when we pop from the DFS stack. When we identify an SCC, any node between the root and the top of this second stack is in the SCC. As we add the node to the SCC, it is popped from this secondary stack, ensuring the guarantee remains for the next SCC.[105]

3. When identifying the lowest link for a node after processing all its children, we have to be sure not to include any children which are already known SCCs themselves. Such nodes may have a lower link in certain scenarios.[106]

4. In the above description, we ignore the issue of selecting the initial node. We cannot know ahead of time that a given node will be in a source SCC, and even more so, there may be multiple source SCCs, such that no single node can reach every other. To solve this problem, we wrap the DFS in a for loop which initiates a DFS from any node which is unvisited in previous runs. We maintain the same metadata on visit order and SCCs across runs such that we do not visit SCCs which have already been traversed.[107]

Listing 4.32: C++

```
//assume adjacency list of vector<int> al[]

//output scc, link, order, second stack
vector<int> scc(n,0),l(n,INT32_MAX),o(n,0),s2(n);
//DFS stack, stores node and next edge to examine
vector<pair<int,int>> s;
//Next index to use for order and scc arrays
int oi=1,scci=0;

for(int i=0;i<n;i++)if(!o[i]){
   s.push_back({i,0});
   s2.push_back(i);
```

[105]This can be implemented with only the single stack that is used for the DFS itself, utilizing a bit of pointer juggling to "skip back" to the correct node when we have finished processing a node which is not the root of an SCC, but we choose to maintain two for simplicity.

[106]Consider a graph $A \to B$, $A \to C$, $B \to C$. If C is processed before B, when B is evaluated, it would find a link (via C) with order 2, vs node B, whose order is 3. Then B will not properly be recognized as an SCC.

[107]This method is tantamount to creating a "virtual" source SCC which is a source to every other SCC in the graph. It does not impact any other SCC, as it has no incoming edges, but guarantees every other SCC in the graph is visited.

```
   //main DFS loop
   while(!s.empty()){
      RECURSE:auto& on=s.back();
      //first time we see a node, set its link and order
      if(!o[on.first])l[on.first]=o[on.first]=oi++;
      for(;on.second<al[on.first].size();on.second++)
        if(!o[al[on.first][on.second]]){//unvisited
         s.push_back({al[on.first][on.second++],0});
         s2.push_back(s.back().first);
         goto RECURSE;
      }
      //we've seen all the nodes, find minimum link
      for(int j:al[on.first])
         if(!scc[j])l[on.first]=min(l[on.first],l[j]);
      if(l[on.first]==o[on.first]){//found SCC
         scci++;
         while(!scc[on.first]){
            scc[s2.back()]=scci;
            s2.pop_back();
         }
      }
      s.pop_back();
   }
}

// build SCC graph, if necessary. Sources have no incoming edges,
//sinks have no outgoing edges
vector<int> al2[scci+1];
for(int i=0;i<n;i++)for(int j:al[i])
   if(scc[i]!=scc[j])al2[scc[i]].push_back(scc[j]);
```

Making a graph strongly connected A common extension to simply finding all strongly connected components is identifying edges to add to a graph to make it strongly connected. We need at least `max(num_sources,num_sinks)`, as each SCC will need at least an incoming and outgoing edge. The question is whether that number of edges is sufficient to make a graph strongly connected. To demonstrate this, we'll come up with a way to construct the edges pairing sources and sinks to demonstrate that this amount is sufficient.

Create a bipartite graph of sources and sinks with edges from sources to each sink it can reach. In practice, this transformation can be done explicitly, with a BFS from each source node,[108] SCCs which are both a source and a sink should appear on both sides of the bipartite graph with an edge connecting them. We perform the following on the transformed graph:

[108]or to be more efficient if there are more sources than sinks, from each sink node along reversed edges

1. Compute the maximum matching of this graph.
2. Take the matched edges in an arbitrary order and draw an edge from the sink of one matching to the source of the next, and from the last sink to the first source. The combination of the matched and newly drawn edges forms a cycle; a cycle which will form the backbone of the final SCC.
3. Arbitrarily connect unmatched sinks to unmatched sources 1:1 until either sources or sinks are exhausted. By Kőnig's Theorem[109] any unmatched node must have a path to some matched node. Therefore adding such an edge between unmatched sinks and sources couples them to the cycle formed in the previous step.
4. Arbitrarily connect the remaining sinks to any source, or any sink to remaining sources.

In the above algorithm, whichever of sources or sinks have the highest cardinality[110] will have exactly one edge added per node. Further, by using the above algorithm, we have guaranteed that all nodes are strongly connected, meaning the above assertion of `max(num_sources,num_sinks)` is exactly the number of edges which must be added. The construction of such a set of edges to add runs in $O(ve)$ time.[111]

4.3.3.2 BCC

While SCCs deal in directed graphs, *Bi-Connected Components* answers a similar question in undirected graphs, namely, sets of nodes in a graph such that each node in the group has at least two node-disjoint paths to every other node in the group.[112] Lets look at our example graph from earlier, now undirected, and its BCCs.[113]

[109]See *bipartite matching*

[110]count

[111]This amount of time is either needed to preprocess the edges to generate a true bipartite graph. Alternatively, Edmonds-Karp can be run directly on an untransformed graph with all edges capacity 1. The matching will be equivalent, but the runtime of the matching itself will increase due to the intermediate nodes, achieving the same runtime.

[112]i.e. the two paths between a given pair of nodes in a BCC cannot share any node

[113]Non-degenerate BCCs

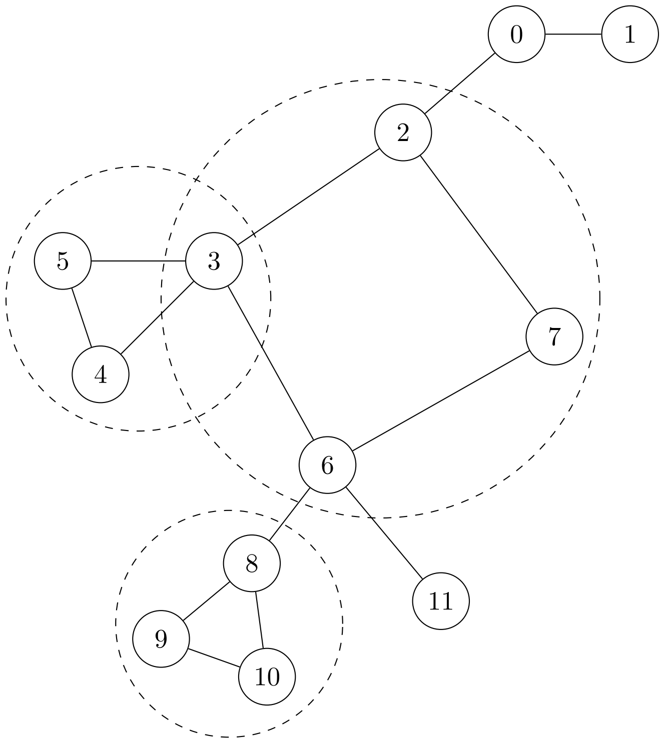


Figure 4.110: The BCCs look largely the same as the SCCs. Each node in each circle has at least 2 paths to every other node in the circle, and those two paths do not share nodes. Note the difference from the SCC graph however, in that 4 and 5 are not in the same BCC as 2, 6, and 7. This is as any path between those two groups of nodes must go through node 3, which violates the constraint that the two paths not share nodes.

Not all BCCs must be a simple cycle. Here is an example:

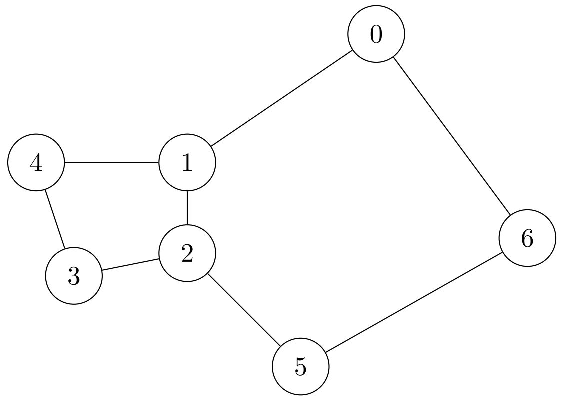


Figure 4.111: All the nodes in this graph form a single BCC which is more complex than a single cycle. For any pair of nodes, there are at least two paths between those nodes.

There are several properties of BCCs which must be articulated:[114]

- A given node may exist in multiple BCCs. Such nodes are special in that if they were removed from the graph, they would disconnect the graph. These nodes are known as *articulation points*.
- Each edge in a graph belongs to at most 1 BCC.
- Edges which are not otherwise in a in a BCC are called *bridges*, and are treated as special BCCs. Bridges comprise exactly the two nodes that define that edge and the edge itself. With the addition of these degenerate cases, each edge in the graph belongs to exactly 1 BCC, and each node is part of at least 1 BCC.
- Similar to SCCs, we can create a tree of BCCs where there is a node for each BCC and each articulation point. Edges are drawn between the two if the given articulation point is contained in the given BCC. This graph is a tree[115] and is known as a *block-cut tree*.

The earlier example is redrawn here with the above in mind. Bridges are drawn dashed, and articulation points are bolded.

[114]pun intended
[115]as there cannot be a cycle between or among distinct BCCs

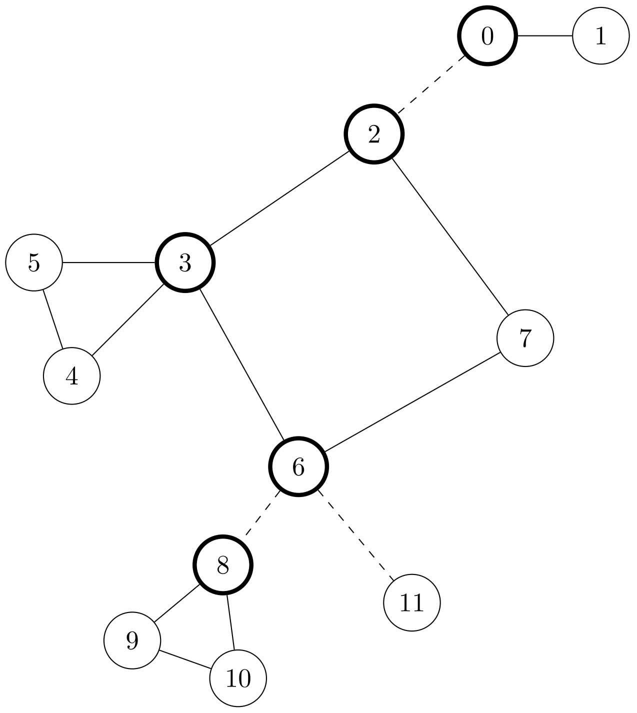


Figure 4.112: This graph contains 7 total BCCs, 4 with multiple edges (0-1, 2-3-6-7, 3-4-5, and 8-9-10), and 3 degenerate ones (0-2, 6-8, 6-11). If any of the bolded articulation points are removed, the graph becomes disconnected. Conversely, if any of the non-bolded nodes are removed, the graph does not become disconnected.

Lastly, we can draw the block-cut tree[116] of this graph:

[116]sometimes known as *BC tree*

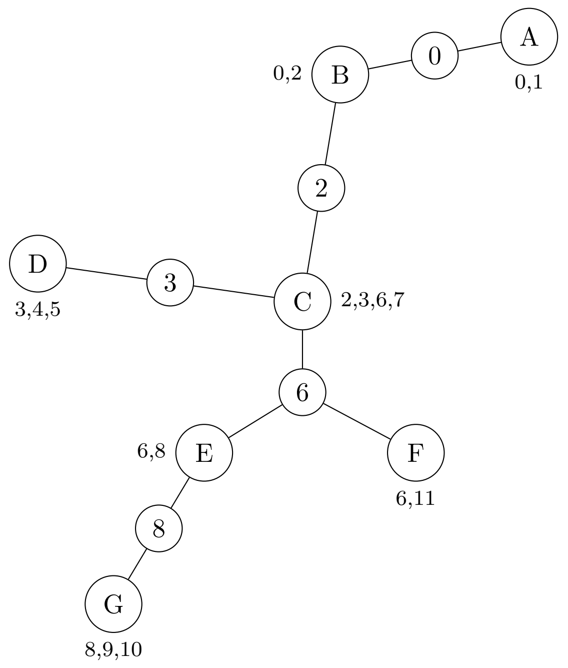


Figure 4.113: The tree of BCCs. Articulation points are labeled with the number of the node that represents that point. BCCs are labeled with letters, and the nodes contained therein in small font. It is clear when looking at this BCC tree why articulation points disconnect the graph. The BCCs which contain exactly two nodes are also bridges.

Computing BCCs The computation of BCCs follows similarly to SCCs, however with the added complication of producing sets of edges rather than nodes. The logic is as follows:

- Given a DFS starting from some node, each BCC has an edge which defines its root. This is the first edge which is traversed in this BCC.
- All edges in the BCC will be traversed before the root edge is popped from the stack.
- Assuming the lowest-order node reached from a given node is tracked as with SCC, then an edge is the root of a BCC if the link of the destination node of that edge is the same as the order of the source of that edge. If an edge goes from $0 \to 1$, and the minimum node reachable from 1 is 0, then $0, 1$ is the root of a BCC.[117]

[117] This necessarily implies there is a second route back to the source node from the destination node which does not include this edge.

- Assuming we track the edges in a stack, as with SCC, when we have found the root of a BCC, we pop the delta between this stack and the DFS stack.[118]

The implementation follows from the SCC implementation of Tarjan, with the above accommodations for dealing with edges rather than nodes.

Some techninal notes which may not be obvious for first time implementers:

1. When we examine an edge for the first time, whether we add the edge to our secondary "edge stack", and whether we recurse on the node at the other end may be different. Every edge must end up on the stack when it is first found, but nodes should be recursed to only once.

2. Due to the first point, we must be careful not to traverse edges a second time in the reverse direction. This is accomplished by tracking the parent edge/node of each node we visit.

3. In order to handle degenerate BCCs, our check if an edge is a root of a BCC looks whether that edge leads to an equal or higher link, rather than just equal. Normal BCC's will have a link equal to the index of the source node of the root edge, and bridges will have a link to some higher value than the source index. All other nodes will have a lower link value.

Listing 4.33: C++

```cpp
int n=al.size();
vector<int> l(n,INT32_MAX),o(n,0),p(n),v(n,0);
vector<pair<int,int>> s,s2;
vector<vector<pair<int,int>>>bcc;
int oi=1;
for(int i=0;i<n;i++)if(!o[i]){
   s.push_back({i,0});
   p[i]=-1;
   while(!s.empty()){
      RECURSE:auto& on=s.back();
      if(!o[on.first])l[on.first]=o[on.first]=oi++;
      for(;on.second<al[on.first].size();on.second++){
         //push this edge if we haven't used it yet, whether we've
         //visited the node or not
         if(al[on.first][on.second]!=p[on.first]&& //don't go to parent
            o[al[on.first][on.second]]<o[on.first]) //already traversed
             s2.push_back({on.first,al[on.first][on.second]});
         if(!o[al[on.first][on.second]]){
            p[al[on.first][on.second]]=on.first;
            s.push_back({al[on.first][on.second++],0});
            goto RECURSE;
         }
      }
```

[118]This is the same as SCC, but with edges instead of nodes.

```
      //find backlink
      for(int j:al[on.first])
         if(j!=p[on.first])l[on.first]=min(l[on.first],l[j]);
      if(p[on.first]!=-1&&l[on.first]>=o[p[on.first]]){ //found BCC
         bcc.push_back({});
         do{
            bcc.back().push_back({s2.back().first,s2.back().second});
            s2.pop_back();
         }while(bcc.back().back()!=pair(p[on.first],on.first));
      }
      s.pop_back();
   }
}
```

Building the Block-Cut Graph The BCC algorithm produces a mapping of BCC to the edges it contains. This is not particularly conducive to solving most real problems, where we typically need more organized information. The following snippet computes the following:

- Mapping from node to the BCCs it is in
- Mapping from BCC to the nodes it contains
- List indicating whether a node is an articulation point, and if so, an index representing that articulation point
- The adjacency list representing the BC tree

Listing 4.34: C++

```cpp
unordered_set<int> bccm[n],bccn[bcc.size()];//node->bcc and bcc->node
for(int i=0;i<bcc.size();i++)for(auto& j:bcc[i]){
   bccm[j.first].insert(i);
   bccm[j.second].insert(i);
   bccn[i].insert(j.first);
   bccn[i].insert(j.second);
}
int aps=0;
int ap[n]={};//map of node to AP
for(int i=0;i<n;i++)if(bccm[i].size()>1)ap[i]=++aps;
vector<int> al2[aps+bcc.size()];//adj. list of BCC tree. APs then BCCs
for(int i=0;i<n;i++)if(ap[i])for(int j:bccm[i]){
   al2[ap[i]-1].push_back(j+aps);
   al2[j+aps].push_back(ap[i]-1);
}
```

Computing Disconnection Consider the following problem:

> Given a connected graph and intended start and end vertices A and B, return whether there is a path from A to B which does not traverse a specified node C.

Computing this in $O(v)$ time is straightforward with BFS, but we can leverage BCCs to expedite this lookup if there are many such queries. We can determine this using a few rules:

1. If A and B are in the same BCC, then there is guaranteed to be a path which does not traverse C.[119]

2. If C is not an articulation point, then C can be similarly avoided. The removal of non-articulation points does not disconnect the graph.

3. If the path from the BCC containing A to the BCC containing B does not traverse the BCC containing C, then there is a path from A to B without vising C.

The first two rules are computed directly from the BCCs and BC Tree. The last is slightly more difficult, as we cannot perform a traversal without incurring linear cost. In order to efficiently perform this query, we can leverage LCA.[120] Arbitrarily select a root for the tree, and perform an Euler walk as described in the LCA section, and compute the LCA of A and B. C is only on the path from A to B if it is an ancestor of A or an ancestor of B, and the LCA is an ancestor of C. If this is the case, C must lie either on the path from A to the LCA, or from the LCA to B, and therefore its removal disconnects A and B.[121]

In this way, the BC Tree and Euler walk are completed in linear time, but each query only takes $O(log(n))$ time for the binary lift to compute the LCA.

Similar logic can be applied to identify whether an individual edge disconnects A and B by checking whether that edge is a bridge,[122] where both endpoints are on the path form A to B.

4.3.3.3 2SAT

One unexpected application of SCCs is solving *2-Satisfiability* or *2SAT*.

Backgroud The satisfiability problem asks us, given several boolean variables and a boolean expression, is there an assignment of true/false to those boolean variables to make the whole expression true? Consider the example:

```
(A||B)&& !A
```

The above expression is true if `A==false` and `B==true`.

[119]by definition of BCC, there must be two node disjoint paths between any pair of nodes, and therefore any single node can be avoided

[120]See section on Lowest Common Ancestor

[121]assuming we have applied the earlier rules 1 and 2

[122]a degenerate BCC only containing a single edge

Notation and Terminology There are many ways of representing boolean logic. This text will try to stick to representations that are seen in code, but for completeness, here are other common representations:

Code	`(A||B)&& !A`
Logic	$(A \vee B) \wedge \neg A$
Digital Logic	$(A + B) * \overline{A}$

Further, some logic terminology, and its mapping to programming concepts:

Disjunction	Logical-or
Conjunction	Logical-and
Literal	A boolean variable. Example: `X` or `!X`
Complement	The opposite of a given literal. `X` is the complement of `!X`
Conjunctive Normal Form	A conjunction where each term is a disjunction of one or more literals. Also known as *CNF*

Based on these definitions, *2SAT* can be described as finding the assignment of values to variables[123] of an expression in conjunctive normal form,[124] where each disjunction[125] has at most 2 literals.[126] Example:

```
(A||B)&& (!A||C)&& (!A||!D)&& (C||!D)
```

Implications and Implication Graphs While the standard boolean operators of and, or, not, and xor are commonly known, there is another operation called the *implication*. If A implies B ($A \implies B$), then if A is true, B must also be true. Here is the truth table for the various operators, including implies.

	A and B	A or B	A xor B	A implies B
`A=false, B=false`	false	false	true	true
`A=false, B=true`	false	true	false	true
`A=true, B=false`	false	true	false	false
`A=true, B=true`	true	true	true	true

The implies column is trivially met when A is false, since definitionally the operation only applies when A is true. When A is true, the expression is only met when B is also true. When A is true and B is false, it violates the defition of the operator, and thus the expression is false.

There are three important points to consider about implications:

[123]technically called *atoms*
[124]and of ors
[125]or
[126]variables

1. `A->B` is equivalent to `!A || B`. This is intuitive, as if A is false, the implication does not apply and is trivilally met, and if B is true, then the implication will be met regardless of the status of A. These two expressions can be used interchangeably.

2. Given the logical-or operator is commutative, `!A || B` is equivalent to `B || !A`. Applying rule 1 to the second representation allows us to show these two expressions are also equivalent to `!B->!A`.[127]

3. A series of implications can be represented as a graph, where if `A->B`, then there is an edge from a node representing A to a node representing B. If we are trying to "choose" nodes to set in a graph, and choose node A in such a graph, than the outgoing edge to node B mean that node B must also be chosen to ensure the implication is met.

We can see this in action for 2SAT:

1. Start with a boolean expression: `(A||B)&& (!A||C)&& (!A||!D)&& (C||!D)`

2. Convert each expression to an implication using rule 1 above: `!A->B && A->C && A->!D && !C->!D`

3. Add in the equivalent implications using rule 2 above: `!A->B && A->C && A->!D && !C->!D && !B->A && C->!A && D->!A && D->C`

4. Create a graph for each node for each literal and its complement: `A, !A, B, !B, C, !C, D, !D`

5. Draw edges for each implication, creating the implication graph.

[127]Rules one and two make intuitive sense together. For a given logical-or of two variables, if one is not met, it implies the other must be met for the expression to be true. There are two situations in which we have to "force" the other variable, one for each of the inputs to the logical-org, and these lead to the two equivalent implications.

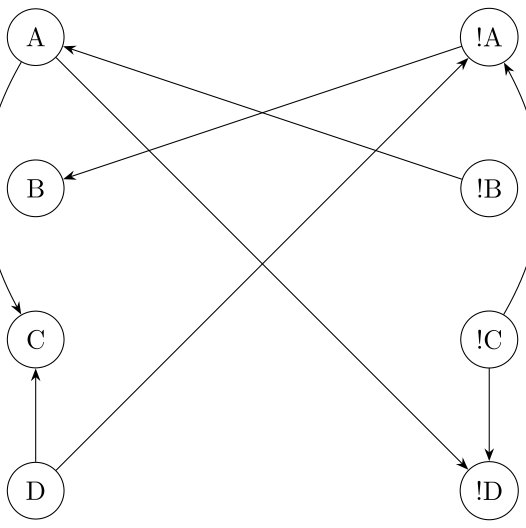


Figure 4.114: While the graph is not bipartite, as edge go between nodes on both sides, the symmetry is still readily visible.

The most important intuition of this graph is its nearly symmetrical nature. This nature will become important in understanding why the following algorithms are correct.

Solving 2SAT using the Implication Graph In order to solve the 2SAT problem, we have to choose nodes such that each variable is represented by exactly one chosen node. We cannot choose both A and !A, for instance. Further, given the graph represents implications, for any node we choose, we must also choose any node on a path from that chosen node. This means if we have a path from `A->B`, and we choose A, we must also choose B. Not doing this would violate the implication. Here is a potential solution:

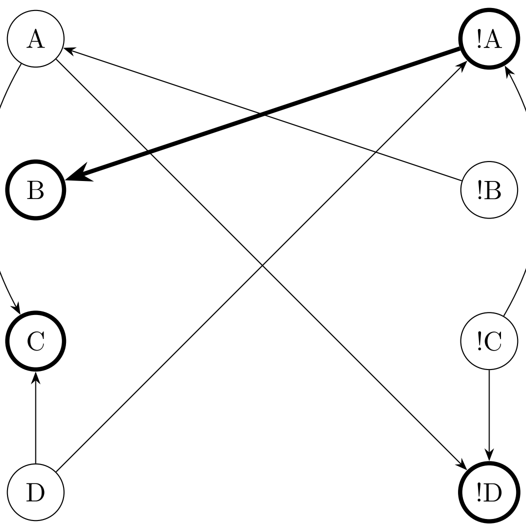


Figure 4.115: Exactly one node representing each variable is chosen, and for any node chosen, any descendent nodes are also chosen, as we see between !A and B.

One important intuition from these rules defining a valid assignment is that if a node has a path to its complement, we must choose the second node. Further, if that second node has a path back to the original node, there is no valid way to select either of those nodes without violating some implication. This leaves some variable unassigned, meaning the original expression has no valid assignment.

> If any node is in the same SCC its complement, there is no valid assignment.

The above assertion follows directly from the definition of a strongly connected component, as a cycle formed by a node and its complement must be part of the same SCC. While this allows us to know in some cases that there is no possible assignment, we must still show that if no SCC contains both a node and its complement, that there must be a valid assignment. We do this by demonstrating that in all such cases[128] we can always construct the assignment itself.

Constructing the 2SAT Assignment Let's first look at the graph in a slightly different arrangement. We will use this graph to see how we can choose nodes pursuant to the above rules.

[128]where no SCC contains both a node and its complement

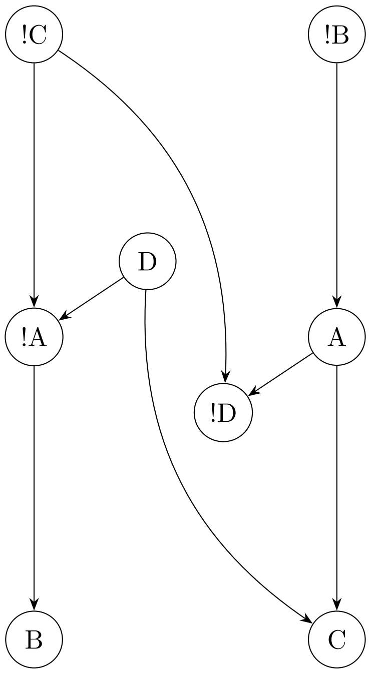


Figure 4.116: The graph is roughly displayed in topological order. If higher nodes are chosen, then the lower nodes with paths from that node must also be chosen. The symmetry inherent in the graph is also still apparent.

Recall the rule that for any node and its complement, if a node has a path to its complement, the latter must be chosen and not the former. This is equivalent to stating that for any individual node and its complement, the chosen node must occur later in topological sort order. We can thus suppose a way to choose all nodes:

> Choose nodes in reverse-topological order, but never choose nodes for which we have already selected the complement (or any of their ancestors).

Let's perform this process on our graph, assuming the topolical ordering of:

```
!B A D !C !D !A B C
```

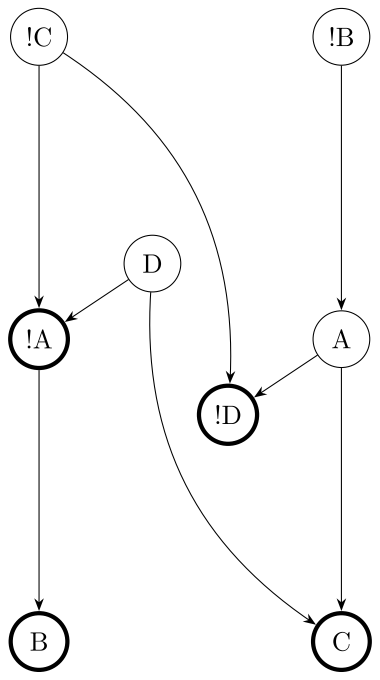


Figure 4.117: In reverse order, C, B, !A, and !D are the first 4 nodes processed. !C, D, A, and !B are not processed as we have already selected their complements.

It is important to note that even if we had never processed the complement of !B, we could not process it as we cannot process node A. We will refer to such nodes which cannot be chosen due to one of their descendents already having their complement chosen as *blocked*.

There are many valid topological sorts, and they all seem to lead to a valid assignment:

Topological Sort	Valid Assignment
!B !C D !A A B C !D	A B C !D
!B A !C D !A B C !D	!A B C !D
!B A !C !D D !A C B	!A B C D
!B A D C !C !A B !D	!A B !C !D
!C D !A B !B A C !D	A !B C !D

It turns out the nature of the graph guarantees this to be the case. As all implications are encoded in the graph, so long as we chose in reverse-topological order, we will never choose two nodes which conflict. We thus only need to show that this greedy selection never reaches a state where we cannot choose a node representing each variable.

Suppose some variable A has both of its nodes blocked.[129] If A is blocked by a descendent B, then it means !B must have been already chosen. The symmetric nature of the graph guarantees that if B is a descendent of A, then !A is a descendent of !B. As !B is chosen, it means !A must also have already been chosen.[130] !A having been chosen contradicts the assumption that A has both of its nodes blocked, proving that this assignment method will always be able to choose at least one node for each variable.

Disconnected Components The above works fine when all ndoes are in a single connected component, but there are two more cases to consider:

1. If a literal is in one component, and its complement is in another component, then due to ymmery, the complements of all literals in one of the components must be in the other component. As neither component contains both a literal and its complement, we can choose every node in one of the components, and require no nodes from the other.
2. If two components share no variable in common,[131] Then the two components are independent and the selection of nodes in one will have no impact on the other.

Practically, adhering to the reverse-topological-order rule will also solve both of these cases as well.

Wrapping Up

- A boolean expression where each disjunction has only two literals can be converted to a graph of implications.
- If no SCC in this graph contains both a literal and its complement, then there is a valid assignment of true or false to each of the variables to make the expression true.
- If some SCC contains both a literal and its complement, there is no solution.
- The assignment itself can be constructed by iterating through nodes in reverse topological order, excepting nodes which are blocked if their complement was already chosen, or one of their descendents is blocked.

All of these processes execute in linear time.

[129]have some descendent which cannot be chosen due to that descendent's complement being chosen

[130]since !A must come later in topological order

[131]One component only contains A, B, and C or their negation, and the other only contains D, E, and F, for instance

4.4 State Explosion

4.5 FSMs

4.6 DAGs

4.6.1 Topological Sort

4.7 Trees

4.7.1 HLD

4.7.2 Binary Lifting

TODO update this with the alternate method Suppose we were posed the problem:

> We are given a tree. Each node is assigned an arbitrary value, such that the value of every node is guaranteed to be strictly less than the value of its direct ancestor. We wish to respond to a set of queries where each query is a node, and we are to respond with the highest ancestor node whose value is at most 10x our own, or -1 if none such exists.

If the tree were balanced, we could walk from the node representing each query to the root, until we found the first admissable node. The tree, however, is not guaranteed to be balanced, and therefore, for N nodes andd Q queries, our runtime might be $O(N * Q)$.

Binary lifting is a technique which allows us to solve such queries in $O(log(N))$ time after $O(N * log(N))$ preprocessing, or more generally, if we have a function which is monotonic on the path from a given node to the root, it allows us to binary search the specific ancestor which fulfills some criteria.

From a high level, the idea is to preprocess the tree, such that each node maintains a pointer to it's immediate ancestor, its ancestor 2 above, its ancestor 4 above, etc. With these links, we can both traverse to the n-th ancestor of any node in $log(N)$ time, but it also provides us the structure to perform the binary search we need.

The links in a tree might look as follows:

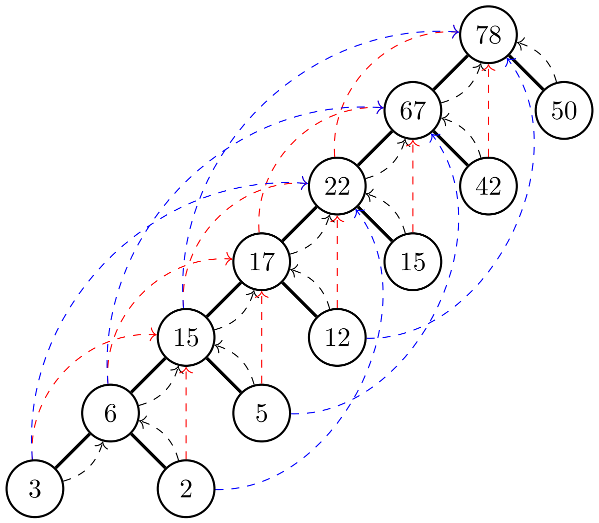


Figure 4.118: A preprocessed tree with uplinks. The 1-links are in black, the 2-links are in red, and the 4-links are in blue.

In order to perform the requested lookup, at say node 3, we would simply perform a binary search. We start with a pointer at the 3 node, and check the largest link, the 4 link, and see that the value 22 is fewer than the necessary 30, so we move our pointer to that node. We now check the 2 link at that node, see that it is too big at 78, so we do not move our pointer. We check the 1 link, and see that it is 67, too large. So 22 is our answer, correctly, and was queried in $O(log(N))$ time.

To perform the preprocessing, we create an array for each node which stores the 1, 2, 4, etc. links. This array is sized at most the heigh of the tree. The 1 link is simply our parent. We can find the other links easily as well. the 1 link or our 1 link is our 2 link. The 2 link of our 2 link is our 4 link, etc.

Listing 4.35: C++

```
void preprocess(int node){
  // assume links initialized to -1
  links[node][0] = parent[node];
  // we look up the i-1'th link of our i-1'th link to get our i'th link
  for(int i=1;i<=max_link;i++)
    links[node][i]=links[links[node][i-1]][i-1];

  preprocess(child[node][0], child[node][1]);
}

int search(int node){
  for(int i=max_link;i>=0;i--)
    if(links[node][i]!=-1&&my_condition(links[node][i]))
```

```
        node=links[node][i];
    return node;
}
```

4.7.3 Lowest Common Ancestor

The lowest common ancestor of two nodes (LCA) is the lowest node in a tree which is an ancestor of two specified nodes. Or thought about differently, if one started at the root and was attempting to traverse to a pair of nodes, the LCA is the point where those two paths diverge.

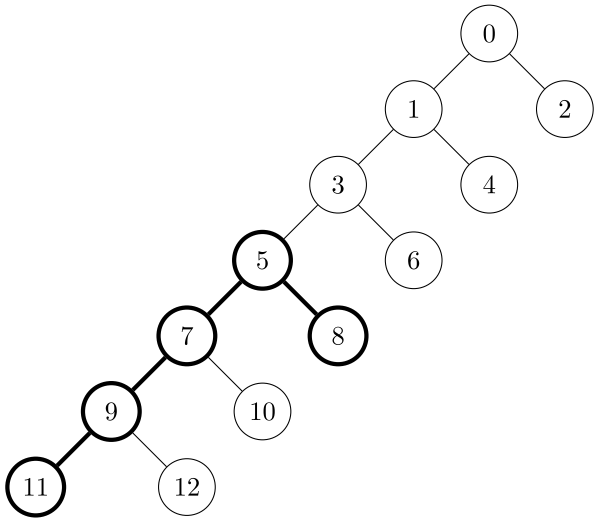


Figure 4.119: The LCA of nodes 11 and 8 is 5.

4.7.3.1 Standard Algorithm

In the standard algorithm, we in turn, start at each node and traverse back to the root. We push the nodes visited on a stack. Then, we successively compare the nodes on the head of each stack, The last pair of nodes which are euqal is the LCA.

Path from 8 to root	8,5,3,1,0
Path from 11 to root	11,9,7,5,3,1,0

If we look at successive nodes in the two paths from the end (as we would were they on a stack), we would see 0, 1, 3, and 5 all match. As 5 is the last match, it is the LCA.

This algorithm wowrks great in balanced trees ($O(log(N))$) or if we only have a single query ($O(N)$), however it does not do well if we have many queries in an unbalanced tree.

4.7.3.2 Unbalanced Trees

With a bit of preprocessing, we can get efficient LCA queries even in unbalanced trees using the binary lifting technique. The overall technique is as follows:

1. Assume we have a constant-time function that allows us to check whether some node is a direct ancestor of another.
2. If one node is a direct ancestor of the other, we are finished, as that node is the LCA.
3. Choose one node A, and binary lift from that node with a condition of "is the candidate an ancestor of B". We assumed we can evaluate this condition in constant time, and the condition is monotonic as we go from node A towards the root (going from "false" to "true"), so binary lifitng applies.

The only missing piece is the ability to determine in constant-time whether some node is a direct ancestor of another. It turns out this is straightforward. If we perform a DFS traversal, labeling nodes in the order that we visit them (pre-order, though it turns out not to matter), and save the value of the first descendent we visit, and the last, then we can check if a node is our descendent by simply checking whether the node's value is between these first and last values. Here is our tree labeled after the DFS:

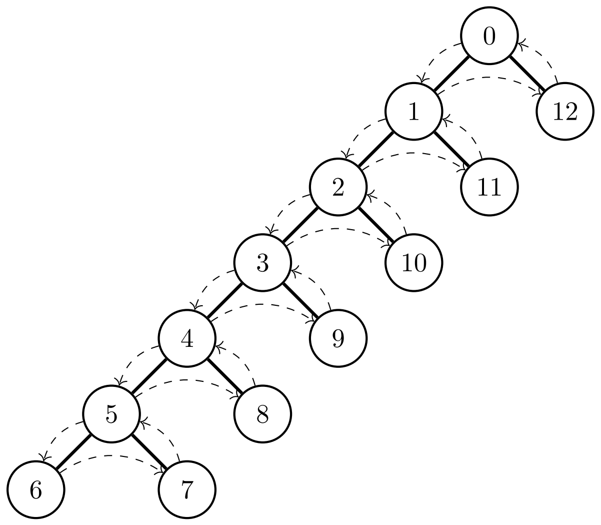


Figure 4.120: Example: Node 2 would cache 3 and 10, as 3 is the first node in 2's subtree, and 10 is the last.

Listing 4.36: C++

```
int index=0;
void dfs(int node){
  order[node]=index++;
  smallest_descendent[node]=index+1;
  dfs(child[node][1]);
  dfs(child[node][2]);
  largest_descendent[node]=index;
}

bool is_ancestor(int child, int node){
  return
      smallest_descendent[node]<=order[child]&&largest_descendent[node]>=order[child];
}
```

Now that we have established a constant-time query for is ancestor, we simply perform the binary lift from one node to find the lowest node which is an ancestor of some other node. This algorithm takes linear preprocessing time for the DFS, but queries only take logarithmic time, making this technique applicable for even multiple queries in unbalanced trees.

4.7.3.3 Constant Time

Number nodes in order as if it were a complete binary tree. xor two nodes, take highest 1-bit. That's the number of bits to mask out to get the LCA.

Map it to non-binary trees if necessary.

4.8 counting loops

4.9 Union-Find

Many common operations on graphs involve the following two operations on some defined grouping of nodes within the graph:

1. Identifying the group a node is in
2. Merging two groups of nodes into one

These two opereations are known as *finding* the group, and computing a *union*. The goal is to perform both operations in sub-linear time.

Most naive solutions perform one of the two operations in linear time.

- Creating an explicit mapping of nodes to groups incurs linear time during a union operation if we attempt to update the map for each node in the group.
- A redirection scheme, such as indicating a group is "part" of a larger group and then finding a "top level" group which is not part of some larger group may incur linear time during the find phase.

To perform these operations effectively, we perform the following:[132]

- For each node, store a parent. If a node has no parent, it is considered the root of a group.
- When performing a find operation on node X, traverse through the parents of X until a root R is found. This represents the group X belongs to. After doing so, perform the traversal again, replacing the parent of each node with R directly. This flattening of the structure is known as *path compression*.
- When performing a union operation on groups X and Y, set the parent of the root of the smaller group to the root of the larger group. This avoids producing long chains of parents. This optimization is known as *union by size*.

While difficult to prove, these two optimizations are sufficient to provide exceptionally fast performance.[133]

Listing 4.37: C++

```cpp
vector<int> p(n,-1),s(n,1);
int ufind(int x){
    int ans=x;
    while(p[ans]!=-1)ans=p[ans];
    int on=x;
    while(p[on]!=-1){
        p[on]=ans;
        x=on=p[x];
    }
    return ans;
}
void uunion(int x,int y){
    int xg=ufind(x),yg=ufind(y);
   if(xg==yg)return;
    if(s[xg]<s[yg])swap(xg,yg);
    p[yg]=xg;
    s[xg]+=s[yg];
}
```

[132]known as *Disjoint-Set Union* or *DSU*
[133]Bounded in the average case by the inverse Ackerman function

Chapter 5

Greedy Algorithms

Chapter 6

Dynamic Programming

Like the greedy algorithms we saw in the last chapter, *dynamic programming*, or *DP*, represents a class of algorithm more-so than a description of an algorithm itself. While we can can define dynamic programming with some mumbo-jumbo such as *a method of computing a solution based on breaking it up into consistent subproblems and then solving those subproblems iteratively to arrive at the ultimate answer*, but that is largegly meaningless unless you already understand the technique. That being the case, we'll jump in with some standard problems and show how the technique arises naturally and understandably.

6.1 Fibonacci Numbers

The fibonacci sequence is the well known sequence: $1, 1, 2, 3, 5, 8 \dots$. While trivially calculable by hand, it is more formally defined as

$$f_n = f_{n-1} + f_{n-1}$$

where

$$f_0 = 1 \text{ and } f_1 = 1$$

6.1.1 Recursive Computation

The above recursive relation extends naturally to code.

Listing 6.1: C++

```
int fib(int n) {
    if(n==0||n==1)return 1;
    return fib(n-1)+fib(n-2);
}
```

While the above code is correct, we find it is also incredibly slow! Even attempting to compute `fib(50)` takes a significant amount of time. One can see why this is intuitively, as the algorithm will take the following steps:

- Evalute `fib(50)`
    - Evalute `fib(49)`
        - Evalute `fib(48)`
            - ...
    - Evaluate `fib(48)`
        - Evalute `fib(47)`
            - ...

Even in this small breakdown that we are computing the same thing multiple times! The reality is quite ugly when we look at the total number of times the function is called recursively, even for such small situations as computing `fib(10)`.

n	0	1	2	3	4	5	6	7	8	9	10
calls to `fib(n)`	34	55	34	21	13	8	5	3	2	1	1

We've made 177 total recursive calls just to compute the 10th fibonacci number! This number grows exponentially with our input, which explains why the relatively small input of 50 takes a significant time to execute. We have to ask why, despite making 21 separate calls to `fib(3)`, do we have to actually compute `fib(3)` 21 separate times? Do we expect the 21st computation to be different from the 20th or 19th?

6.1.2 Saving work with Memoization

As the previous sentence so subtly hints, the recursive function has an important property:

For a given n, every call to `fib(n)` will yield the same result

This means that once we have computed a given value of `fib(n)`, we can save that value, and directly return it the next time it is requsted instead of recomputing it recursively again. The function looks like this:

Listing 6.2: C++

```cpp
int memo[N]; //N is the maximum value of n we could see
int fib(int n) {
  if(memo[n])return memo[n]; //If we've already computed, don't
      recompute
  if(n==0||n==1)memo[n]=1;
  else memo[n]=fib(n-1)+fib(n-2);
  return memo[n]; //we've saved the computed value for easy access next
      time
}
```

This small optimization, simply saving the result and returning, causes an immense speedup. We likely can run for `n` into the hundreds of millions. The call counts reflect this speedup; the 19 calls with the optimization is a far cry from the earlier 177.

`n`	0	1	2	3	4	5	6	7	8	9	10
call to `fib(n)`	1	2	2	2	2	2	2	2	2	1	1

6.1.2.1 Runtime

The runtime analysis of this optimization is quite simple, and consists of two parts:

1. We execute the main body of the function exactly once per entry of the `memo` array.
   - An execution of the main body of the function can only occur if the entry in the array equals 0, and that execution causes the entry to not equal 0, so the execution only happens once per value of `n`.
2. Each execution of the main body of the function makes at most 2 function calls

Since we are limited to the size of the array ($O(n)$), and the work done for each entry of the array is constant time, the total execution time is also $O(n)$.

6.1.3 Eliminating the Stack

Though we have addressed the runtime nicely, stack limitations may prevent us from evaluating particularly large values of `n`. To solve this, let's look at the state of the array as the algorithm progresses.

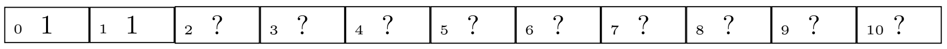


Figure 6.1: The initial state of the array (with base cases populated)

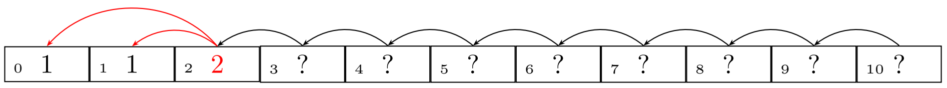


Figure 6.2: The state of the stack when we first recurse to `fib(2)`. The two dependent values are known, so we can compute the value.

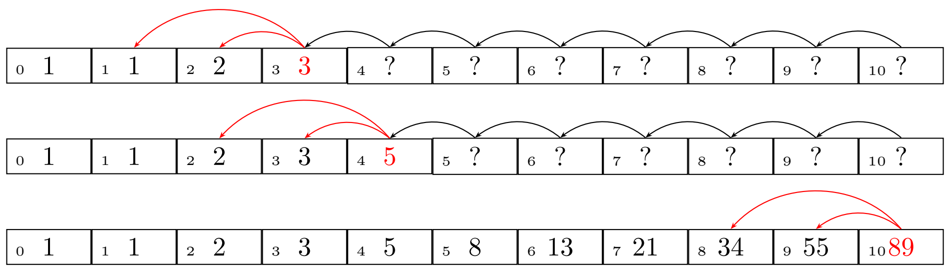


Figure 6.3: Values are filled into the array as the dependencies become fulfilled.

It comes as no surprise that since the arrows indicating the recursive calls always go towards the left, the array ends up being populated starting from the left and progressing towards the right. Given that we know the exact order the array can be populated from the recursive calls, we can explicitly fill them in that order rather than depending on recursion.

Listing 6.3: C++

```cpp
int memo[N]; //N is the maximum value of n we could see
int fib(int n) {
   memo[0]=memo[1]=1;
   for(int i=2;i<n;i++)memo[i]=memo[i-1]+memo[i-2];
   return memo[n];
}
```

This move from a recursive solution where we cache the result of each call to an iterative one where we directly compute the values in a logical order is dynamic programming. It is a powerful tool which applies in a wide number of problems which will be explored in the remainder of this chapter.

6.2 Building Blocks

Now that we've seen dynamic programming used to solve a rather trivial problem, we can break out some of the building blocks which apply more generally. Formulating a DP solution requires 5 common building blocks:

1. What the index into the DP array represents
   - For fibonacci, this was i, the index of a fibonacci number
2. What value is stored at each index in the DP array
   - The actual value of the i-th fibonacci number
3. What is the relation between entries

- `memo[i] = memo[i - 1] + memo[i - 2];`

4. In what order are the entries computed

   - From the relation, we can see they are computed from small to large i

5. What are the base cases

   - Any entry for which the relation would refer to a nonexistent entry, in this case, where $i = 0$ and $i = 1$
   - `memo[0] = memo[1] = 1;`

Once we've answered these five questions, we can translate our candidate solution to code. Here we break out the prevoiusly presented DP solution to fibonacci annotated with the above steps

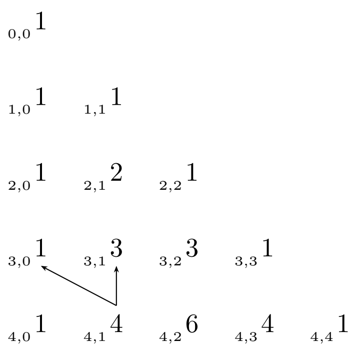
Listing 6.4: C++

```cpp
int memo[N]; //Our DP array, with an index i referring to the i-th
    fibonacci number
int fib(int n) {
   memo[0]=memo[1]=1; //base cases
   for(int i=2;i<n;i++) //iteration order
      memo[i]=memo[i-1]+memo[i-2]; //relation between entries
   return memo[n];
}
```

6.3 Multiple Dimensions

The fibonacci example above only had a single index into its flat DP array. This is considered a 1-dimentional DP. DPs we run into in the wild will generally have multiple indices into their array, which can be 2-D, 3-D, or any higher amount. Let's take a look at how that works using binomial coefficients (N-choose-R).

If you recall, the binomial coefficients can be computed using Euler's triangle, where each element is the sum of the two elements just above it. We'll left justify the grid to make it easier to index.



The two arrows visually demonstrate the computation: `ncr(i,j)= ncr(i-1, j)+ ncr(i-1, j-1)`. This looks an awful lot like a recursive relation. Can we define the other components required for DP?

1. Indices: the N and R of N-choose-R
2. Value: the actual value of N-choose-R for the particular indices
3. Relation: `ncr(i,j)= ncr(i-1, j)+ ncr(i-1, j-1)`
4. Iteration Order: As shown in the earlier diagram of the relation, the arrows go up, and up and to the left. Therefore, if we compute the triangle from top to bottom, all dependent values will be ready when we need them. Most naturally, we will iterate `i` from $2 \rightarrow n$, and for each `i`, `j` from $1 \rightarrow i$.
5. Base Cases: As the arrows go up and to the left, we will need base cases on the left side and top side of the grid.[1] Our base case along the left is `ncr[X,0]=1`. Our base case along the top occurs when `i==j`, and therefore means `ncr[X,X]=1`.

With these five things defined, we can proceed to write our code in a very similar manner to fibonacci.

```
int dp[N][R]; //use 2-D array since 2 dimensions
int ncr(int n,int r) {
    //initialize the base cases
    for(int i=0;i<=n;i++) {
        dp[i][0] = 1;
        dp[i][i] = 1;
    }
    //The iteration order
    for(int i=2;i<=n;i++)for(int j=1;j<i;j++) {
        dp[i][j]=dp[i-1][j]+dp[i-1][j-1]; //the recursive relation
    }
    return dp[n][r];
}
```

[1]where the arrows might point "off" the grid"

6.3.0.1 Runtime

We can analyze the runtime very similarly to how we did for fibonacci, namely:

1. Determining how many array elements we have to populate
2. Determining how much work we have to do to compute each one

The array is sized at $O(n^2)$.[2] The relation involves a constant number of lookups, meaning the overall runtime is quadratic.

6.3.1 Benefits of Recursion

So far, we've extolled the benefits of DP over recursion with memoization, but there are times when using a recursive solution may be beneficial.

- When the depth of the recursion isn't particularly deep, and the recursive solution is easier to wrap your head around
- When the number of reachable states is sparse. So far, we've seen DP examples which depend on almost the entire array being populated. This is not always the case. There are some DP problems which require large tables where many states are not useful or reachable. With a DP solution, we still iterate over and solve those cases. A recursive solution, however, is on-demand. Namely, it only makes a recursive call (and subsequently populating an element in the array) when that value is actually known to be needed. If the array is sparse enough, one might even consider a hashmap to store the memoized solutions, instead of an array.
- When it's hard to determine an iteration order. With the examples we've seen, it's been rather straightforward to figure out where the dependency arrows point, and thus what order to iterate through the array. This is not always the case. Sometimes it is very difficult to come up with a simple ordering. Recursion solves this problem by computing values as they're needed instead of in some global order.

In short, while DP solutions are great and often work more effectively, sometimes it is useful or even necessary to revert to recursive ones.

[2]The fact that we only populate half of it does not impact the complexity.

6.4 Standard DP Problems

6.4.1 Knapsack

Consider the following problem:[3]

> Sam is at a buffet with many different types of food. Each kind of food on the buffet has an associated happiness, such that Sam's overall happiness will increase by h_i after eating a serving of food i. Sam doesn't like to eat very much, however, and so has a maximum number of servings (S). What is the maximum amonut of happiness Sam can achieve while only eating S servings?

We'll consider three variations of this problem which result in different solutions.

6.4.1.1 As Stated: Greedy

The problem as stated lends itself to a greedy solution. At each step, we can simply select the remaining item which has the highest value of happiness. This is intuitively true, and a simple proof by contradiction demonstrates it to be mathematically true.[4]

Though it may not be apparent, the key factor that makes this solution valid is that regardless of selection of items, we can always use exactly S total servings regardless of our selection of items. Therefore, the amount of servings a gien food takes does not impact our ability to optimize for happiness.[5]

6.4.2 Inequal Serving Amounts

Let's add the following to the problem to ensure it doesn't trivially reduce to the greedy solution.

> Each food i has an associated quantity q_i. If food i is eaten, it must be consumed in multiples of q_i servings.

While it may seem innocuous, this small additional restriction means that there is no longer a guarantee we can hit S exactly. Some choices put us in a position where if we select sub-optimally, we might not be able to consume our full allotment of servings. Here's an example:

food	happiness	quantity
0	100	10
1	95	4

[3]Astute readers will note the similarity, and slight deviation, from canonical knapsack. Hold your horses.

[4]Suppose the ultimate solution did not contain the foods with the highest happiness. We can swap 1-for-1 a serving of food with higher happiness in, removing one with lower happiness. This increases overall happiness for the same number of servings, invalidating the supposition.

[5]The greedy proof fails if two foods cannot be swapped 1-for-1 with respect to servings.

If $S = 10$, then greedily eating food 0 will garner us the optimal 1000 happiness. However, if $S = 12$, then we will still only be able to get the same 1000 happiness. After greedily eating food 0 and its 10 servings, we are left with 2 remaining servings and unable to eat any other food. If we instead select the slightly less happy item 1, we can then consume 4 servings of it multiple times, hitting 12 servings exactly. This produces a much higher happiness of 1152. Given the greedy solution fails, we need something more clever to ensure that choosing high-happiness foods does not leave us with too many uneaten servings.

Often when greedy solutions fail, we look to brute force. In this case, a brute force might involve evaluating every possible order of eating food up to the serving limit, and taking the one which results in the highest happiness. Let's take a look at the code to do this:

```
// returns the maximum happiness for S servings
int happy(int s) {
    if(s==0) return 0; //base case...no servings, can't eat!

    int ans=0;
    // just try everything and then backtrack.
    // the recursion would yield an actual result
   for (int i=0;i<food_types;i++)if(q[i]<=s){
     //value we get if we eat this food, so we have s-q[i] servings left
       int recursion=happy(s-q[i]);
       //if eating q[i] of this food gives us a better answer, do it!
       ans = Math.max(ans,h[i]*q[i]+recursion);
    }
    return ans;
}
```

While this solution would be correct, it is undoubtedly too slow for any reasonable input size. With fibonacci, we were able to solve the problem more quickly by realizing that calls to `fib(n)` would always yield the same results. In this case, will `happy(s)` ever return a different value for a given value of `s`? Intuitively, if I know how many servings I have left, it doesn't matter what foods i've eaten previously or in what order. Just like evaluating fibonacci will always return the same value fo `fib(n)` every time it is called, `happy(s)` will return the same value every time it is called. If we cache this value, we can greatly speed up the execution.

```
int cache[S];
// returns the maximum happiness for S servings
int happy(int s) {
    if(s==0) return 0; //base case...no servinces, can't eat!
   if(cache[s]!=0)return cache[s];

    int ans=0;
    // just try everything and then backtrack. Only recurse if we have
```

```
        enough servings left and
    // the recursion would yield an actual result
   for (int i=next_food;i<food_types;i++)if(q[i]<=s){
      //value we get if we eat this food, so we have s-q[i] servings left
        int recursion=happy(s-q[i],i+1);
        //if eating q[i] of this food gives us a better answer, do it!
        ans = Math.max(ans,h[i]*q[i]+recursion);
    }
   cache[s]=ans;
    return ans;
}
```

We can articulate our five components in DP as follows:

1. Indices: The amount of servings we can consume
2. Value: The maximum amount of happiness we can get in that many servings
3. Relation: memo$(s) = max_{i\in\text{foods}}(h_i * q_i + \text{memo}(s - q_i))$, the food which gives us the maximum happiness if eaten and added to the value after recursing on $s - q_i$
4. Iteration Direction: Every relation refers to lower values of s, therefore we should iterate from $s = 0$ upward.
5. Base Cases: We don't recurse if a food would give us negative servings, and defalt to 0.

With the above, we can easily construct the DP code.

```
// dp[x] returns the maxium happiness for x servings
int dp[S];
for(int i=0;i<=S;i++){ //iteration order
   dp[i]=0;//Base case
   for(int j=0;j<food_types;j++)if(q[j]<=i)//relation
      dp[i]=max(dp[i],h[j]*q[j]+dp[i-q[j]]);
}
```

Example Lets work through one example to visualize how this ultimatly works. We'll use the following two foods types.

food	happiness	quantity
0	20	5
1	15	3

We'll evaluate up to $s = 8$, and for the purposes of demonstration, we'll use red squares to indicate un-processed nodes and red arrows to indicate evaluatioas which are skipped as there are not enough available servings.

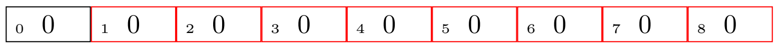


Figure 6.4: The initial state of the array

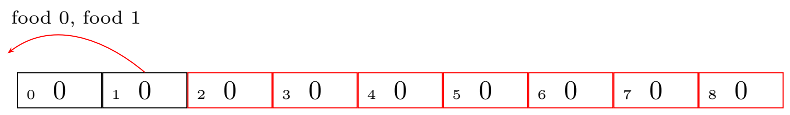


Figure 6.5: Evaluation of entry 1. Both food types go off the end of the array, so there is no food which can be eaten.

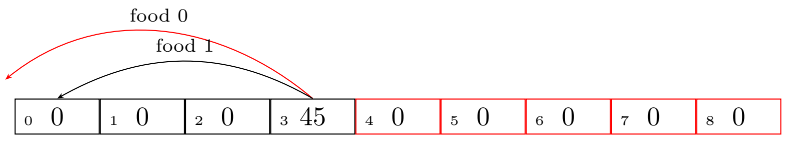


Figure 6.6: Once we get to $S = 3$, we have enough servings to consume food 1. We can visually see our arrow points to a real element, and thus the value in box 3 will be $15 * 3$ because we eat the 3 required servings of food 1.

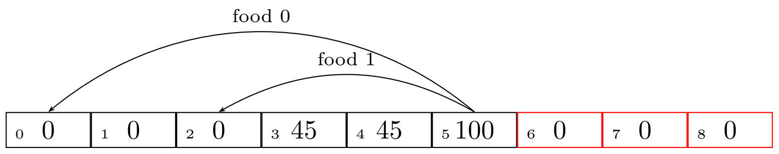


Figure 6.7: At nodes 4 and 5, we are able to eat 3 servings of food 1, as we did at node 3, but we are also able to consume 5 servings of food 0. Since $3 * 15 < 5 * 20$, we will eat food 0.

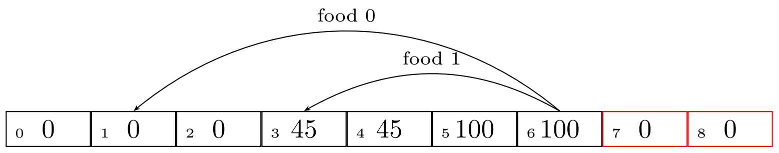


Figure 6.8: At node 6, we can either eat food 1 or food 0. If we eat food 1, we have 3 servings remaining, and we know the maximum happiness we can get with 3 servings is 45, leaving us with 90 total for 6 servings. Alternatively, we can eat food 0, getting us to 100 happiness. The latter is greater, so we choose that option.

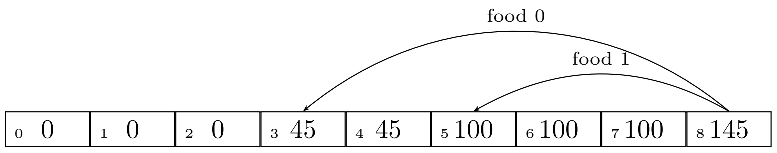


Figure 6.9: At node 8, if we choose food 0, we add the 100 to the best from the remaining 3 servings (45). This total is the same as if we choose food 1, and add the 45 to the 100 remaining from 5 servings. Either leads to a total happiness of 145, which is optimal.

Runtime The runtime analysis follows similarly to how it did before. The array size is $O(s)$, but the work done to populate each entry in the array is not constant. For each value of s, we have to examine each type of food (f).This makes the overall runtime $O(sf)$.

6.4.2.1 Item Limitations

We'll consider one final variant of this problem, perhaps the most common:

> Each food i has an associated quantity q_i. If food i is eaten, it must be consumed once for exactly q_i servings.

While this may seem like a small change, it has major ramifications on the algorithm. If we recall and adapt our recursive solution above, we can simply include in the recursion which foods we've already used so as to not use them twice.

```
// returns the maximum happiness for S servings
int happy(int s,set<int>& used_foods) {
    if(s==0) return 0; //base case...no servings, can't eat!

    int ans=0;
    // just try everything and then backtrack.
    // the recursion would yield an actual result
   for (int
        i=0;i<food_types;i++)if(used_foods.find(i)==used_foods.end()&&q[i]<=s){
      //value we get if we eat this food, so we have s-q[i] servings left
      used_foods.insert(i);
        int recursion=happy(s-q[i]);
      used_foods.delete(i);
        //if eating q[i] of this food gives us a better answer, do it!
        ans = Math.max(ans,h[i]*q[i]+recursion);
    }
    return ans;
}
```

While this solution is correct, we find that once we add this additional value to our function call, we can no longer use the same caching solution we used before. Calls to this function are now no longer solely dependent on the value of s, but also on the exact contents of `used_foods`. We can modify the dimensions of our cache to include both s and `used_foods`, which will be correct.[6] The five DP steps would appear as follows:

1. Indices: The amount of servings we can consume, and the foods we are allowed to eat
2. Value: The maximum amount of happiness we can get in that many servings using only the indicated foods
3. Relation: $\text{memo}(s) = max_{i \in \text{available foods}}(h_i * q_i + \text{memo}(s - q_i))$, the food which gives us the maximum happiness if eaten and added to the value after recursing on $s - q_i$
4. Iteration Direction: Every relation refers to lower values of s, therefore we should iterate from $s = 0$ upward. The iteration value through the available foods doesn't matter since every evaluation refers to a lower value of s.
5. Base Cases: We don't recurse if a food would give us negative servings, and defalt to 0.

We could write code to implement this DP, however we should look at the runtime first. The amount of work we do per state has not changed and is still $O(f)$. The number of states is much larger now, however, at $O(s2^f)$.[7] This exponential runtime of $O(sf2^f)$ will be prohibitively slow for any non-trivial f.

If we think about how a solution is calculated, we see where duplicate work occurs. Consider a solution which requires us to eat foods 0, 1, 2, and 3. For a given s, we will have to iterate through some of the following states:[8]

- if 0 is eaten, make recursive calls indicating $0, 1, 0, 2, and 0, 3$ are eaten.
- if 1 is eaten, make recursive calls indicating $0, 1, 1, 2, and 1, 3$ are eaten.
- if 2 is eaten, make recursive calls indicating $0, 2, 1, 2, and 2, 3$ are eaten.
- if 3 is eaten, make recursive calls indicating $0, 3, 1, 3, and 2, 3$ are eaten.

We make only 6 unique recursive calls, but each one is made twice. This problem is exacerbated at deeper recursion levels, where we'll find that a significant amount of time is spent iterating over recursive calls that we've already

[6]In order to use `used_foods` as a dimension, we would need to either use a hashmap with the `used_foods` itself as a key, or use a bitmask of length `food_types` instead of a set to indicate which foods have been consumed.

[7]Each of the f food types is either available or not available, hence the exponential possible states of `used_foods`.

[8]This is not an exhaustive list

computed.[9] Even with the caching, the excess work comes from iterating over states which we have already seen in some other way.

The key intuition, and ultimately where the excess work comes from, is that if we eat food x and then food y, this is the same as eating food y and then food x. Given this, we can impose an arbitrary ordering to the food. If our `used_foods` list includes some food x, we can safely only examine foods f where $f > x$.

The next intuition is that if we are only examining foods greater than some x, then our recursion doesn't depend on the foods we've already eaten. We know that we have not eaten any food $> x$, and we know that any food $<= x$ will not be examined in this function call, and thus whether such a food has been eaten will not impact our execution. This means our function call now only inludes s and x. As we have only the two values, we can further trivially cache these values.

```
// returns the maximum happiness for S servings, only using foods >x
int cache[S][food_types];
int happy(int s,int x) {
    if(s==0) return 0; //base case...no servings, can't eat!
   if(caches[s][x]!=0)return cache[s][x];

    int ans=0;
    // just try everything and then backtrack.
    // the recursion would yield an actual result
   for (int i=x;i<food_types;i++)if(q[i]<=s){
      //value we get if we eat this food, so we have s-q[i] servings left
        int recursion=happy(s-q[i],i);
        //if eating q[i] of this food gives us a better answer, do it!
        ans = Math.max(ans,h[i]*q[i]+recursion);
    }
   cache[s][x]=ans;
    return ans;
}
```

1. Indices: The amount of servings we can consume, and an index indicating the minimum food which we can eat

2. Value: The maximum amount of happiness we can get in that many servings using only foods greater than the indicated food

3. Relation: memo$(s) = max_{i>x}(h_i * q_i + \text{memo}(s - q_i))$, the food which gives us the maximum happiness if eaten and added to the value after recursing on $s - q_i$

4. Iteration Direction: Every relation refers to lower values of s, therefore we should iterate from $s = 0$ upward. The iteration value through the

[9]Calls with 2 foods eaten will be called at least twice. Calls with 3 foods eaten will be called at least 3 times, calls with n foods eaten will be called n times.

available foods doesn't matter since every evaluation refers to a lower value of s, however going from $i = x$ upward is most practical.

5. Base Cases: We don't recurse if a food would give us negative servings, and defalt to 0.

From a runtime perspective, we still do $O(s)$ work for each entry in our array, but the number of entries in the array is now only $O(sf)$, leading to a significantly faster runtime of Osf^2.

This is still slow, though, and we can do better. To understand how, lets take a step back. Consider the final DP state after we've completed an instance of the algorithm with repeats allowed:

$_0$ 0	$_1$ 0	$_2$ 0	$_3$ 45	$_4$ 45	$_5$ 100	$_6$ 100	$_7$ 100	$_8$ 145

Figure 6.10: Recall we had a food which allowed 3 servings at 15 happiness, and 5 servings at 20 happiness.

Now, suppose we discover a third class of food, one with say 4 servings and 18 happiness that we can only eat once. How might we augment the found solution to accomadate the new food? One way is to simply recompute the entire array, but we can do better than that with the following, where dp_x is the entry in the existing solution, and dp'_x represent the augmented solution.

$$dp'_x = max(dp_x, dp_{x-4} + 4 * 18)$$

For each serving value in our agumented array, we have a choice. We can either accept the previous solution for this serving cout, without using this new food, or we can choose to use the new food, then looking up the optimal value in the existing array for the remaining servings. By referring to the existing array instead of the augmented one in this latter case, we ensure that any solution only chooses to consume this new food once.[10]

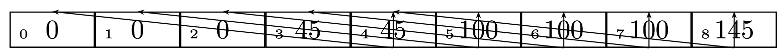


Figure 6.11: The augmented array is shown above the original array. We see two arrows for each serving amount: one indicating we consume the new food to reach the optimal happiness for this serving amount, and one indicating we don't. Note that no arrow goes from one box in the augmented array to another in the augmented array, guaranteeing this new food is only consumed once.

TODO update the values in this figure to the corect things

[10]This distinguishes it from the general repeated case, where we continuously refer to the same array, thus providing an opportunity to use each food an arbitrary number of times.

Putting It Together We can now see the full DP implementation:

```
// dp[x] returns the maxium happiness for x servings
int dp[S][food_types];
for(int i=0;i<=S;i++)for(int j=x;j<food_types;j++){ //iteration order
   dp[i][j]=0;//Base case
   for(int j=0;j<food_types;j++)if(q[j]<=i)//relation
      dp[i]=max(dp[i],h[j]*q[j]+dp[i-q[j]]);
}
```

As with the non-limited case, we can visiually see how the array gets populated. We'll use the following food categories:

food	happiness	quantity
0	1	1
1	4	1
2	4	2

We'll simulate up to a maximum of 3 servings.

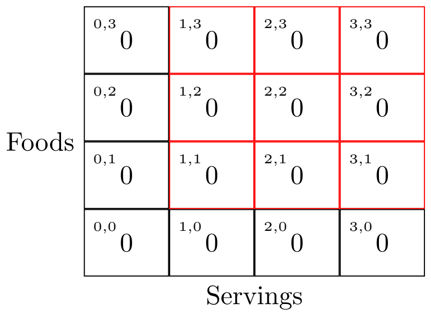


Figure 6.12: The initial state of the array

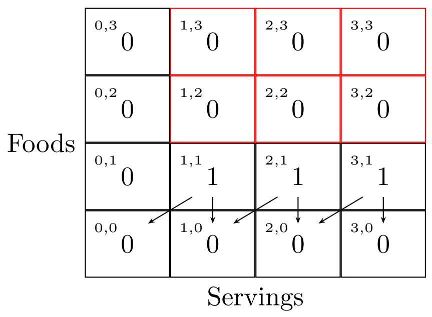


Figure 6.13: When we evaluate each entry for food 0 (index 1), we simulate eating and not eating it and take the maximum. The diagonal arrows represent eating the food (since we have to subtract the servings), and the vertical arrows represent skipping this food, since we have the same amount of servings available. In both cases, we go to the row below, since that represents the previous food item. In this case, eating the food is always optimal.

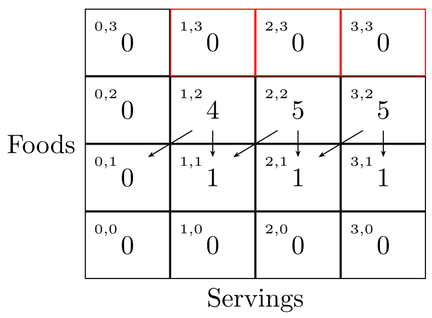


Figure 6.14: We evaluate food 1. We find eating it is always optimal. Each value will end up being the sum of the happiness of food 1 plus the values pointed to by the diagonal arrows, simulating eating this food.

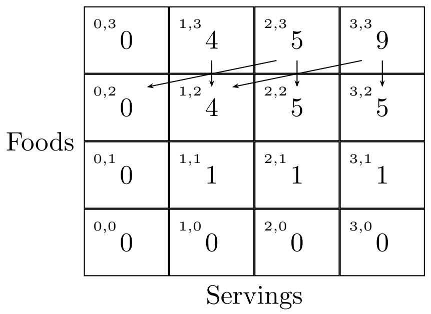


Figure 6.15: We evaluate food 2. When we have fewer than 2 servings available, we skip this food and just use the two previous foods (vertical arrow). When we have 2 servings, we find it is still optimal to skip this food (5 vs 4). When we have 3 servings, we eat this food and come out better (9 vs 5). Note the diagonal arrows go back 2 boxes since food 2 takes 2 servings.

food	happiness	quantity
0	100	10
1	95	4
2	96	4
3	97	4

If $S = 10$, then greedily eating food 0 will garner us the optimal 1000 happiness. However, if $S = 12$, then we will still only be able to get the same 1000 happiness. After greedily eating food 0 and it's 10 servings, we are left with 2 remaining servings and unable to eat any of the other foods. If we instead select the slightly less happy item 1, we can then consume foods 2 and 3. This produces a much higher happiness of 1152. Given the greedy solution fails, we need something more clever to ensure that choosing high-happiness foods does not leave us with too many uneaten servings.

Often when greedy solutions fail, we look to brute force. In this case, a brute force might involve evaluating every possible order of eating food up to the serving limit, and taking the one which results in the highest happiness. Let's take a look at the code to do this:

```
// returns the maximum happiness for S servings
int happy(int s, int next_food) {
    if(s==0) return 0; //base case...no servinces, can't eat!

    int ans=0;
    // just try everything and then backtrack. Only recurse if we have
        enough servings left and
    // the recursion would yield an actual result
   for (int i=next_food;i<food_types;i++)if(q[i]<=s){
      //value we get if we eat this food, so we have s-q[i] servings left
        int recursion=happy(s-q[i],i+1);
        //if eating q[i] of this food gives us a better answer, do it!
        ans = Math.max(ans,h[i]*q[i]+recursion);
    }
    return ans;
}
```

While this solution would be correct, it is undoubtedly too slow for any reasonable input size. With fibonacci, we were able to solve the problem more quickly by realizing that calls to `fib(n)` would always yield the same results. In this case, will `happy(s,next_food)` ever return a different value for a given value of `s` and `next_food`? Intuitively, if I know how many servings I have left and foods to examine, does it matter which of the previous foods I've already eaten? Just like evaluating fibonacci for some lower number does not depend on the evaluation of higher values, evaluating this happy function does not depend on what food we have already eaten so long as we know how many servings and what foods are left. If we can cache this result, we can greatly speed up the execution.

```
// returns the maximum happiness for S servings
int happy(int s, int next_food) {
    if(s==0) return 0; //base case...no servinces, can't eat!

    int ans=0;
    // just try everything and then backtrack. Only recurse if we have
         enough servings left and
    // the recursion would yield an actual result
   for (int i=next_food;i<food_types;i++)if(q[i]<=s){
      //value we get if we eat this food, so we have s-q[i] servings left
        int recursion=happy(s-q[i],i+1);
        //if eating q[i] of this food gives us a better answer, do it!
        ans = Math.max(ans,h[i]*q[i]+recursion);
    }
    return ans;
}
```

6.4.2.2 All or Nothing

Let's add the following to the problem to ensure it doesn't trivially reduce to the greedy solution.

> Each food i has an associated quantity q_i. If food i is eaten, exactly q_i servings of it must be consumed, no more no less.

In fractional knapsack, we are allowed to take fractions of items, instead of whole items. In this case, we might eat half a serving, or a tenth, or any arbitrary amount. In these cases, it is readily seen that we should select the available food with the highest happiness (or value-to-weight ratio in classical knapsack) which is still available.

Whether there is a limited amount of each item or not, the key insight is that regardless of what we have selected so far, we can always consume up to exactly the serving limit, and generally, if we have a guarantee that we can consume exactly the limit (either servings or weight, in the canonical knapsack), then greedy selection will be optimal.

There are three distinct characteristics of this problem that we note:

1. There are two variables, of which one must be maximized, and the other minimized. Here, we are trying to maximize happniess while minimizing (or bounding) the number of servings.

2. There are multiple classes of items, each contributing varying amounts to the two variables. Here, each food has a different happines value.

3. We can choose to take or not take some of each item.

Problems of this sort arise in many situations, and are known as *Knapsack* problems. In the canonical knapsack problem, we are trying to fill a knapsack with items that maximize the total value while minimizing weight, where each of the items has a different value and weight. Here we are maximizing happiness while minimizing servings.

6.4.2.3 Another Example

Since it is so common, lets consider another example.

> Barbara likes to go skiing, and while doing so, she likes to have as much fun as possible. Barbara isn't very good, though, and if she goes down too long of steep slope, she will crash and be severely injured. The slope is divided up into sections, section 0 at the top and n at the bottom. Each section has an associated fun value, and danger value. As Barbara skis down the slope, she accumulates the fun and danger from each section. In order to prevent the accumulated danger from exceeding the amount which would cause barbara to crash, Barbara can choose to walk any number of sections. If she chooses to walk a section, her danger decreases by 10, and Barbara accumulates no fun for the section. Further, her accumulated fun halves. What is the maximum amount of fun barbara can have without her danger exceeding the amount that would cause her to crash?

Looking back at our earlier list of factors that hint hint at knapsack:

1. There are two variables, of which one must be maximized, and the other minimized: We are trying to maximize fun while restricting danger

2. There are multiple classes of items, each contributing varying amounts to the two variables: each section may increase or decrease the fun/danger

3. We can choose to take or not take some of each item: we can walk or not walk

Since we only pass each section once, this is most similar to the limited knapsack. Lets construct the four elements of the DP:

1. Indices: We have to know our current danger so we don't exceed it. We have to know which segment we're on (like knowing which food before). These two variables compose our index.
2. Value: The thing we're trying to maximize: our fun!
3. Relation: Like before, we can choose to either walk or ski any section. The relation is very similar before with:

   - s: the segment we're on
   - d: the amount of accumulated danger
   - f_s: the fun for a given segment
   - a_s: the danger for a given segment

   memo$(d, s) = max($memo$(d + 10, s - 1) * .5, f_s +$ memo$(d - a_s, s - 1))$

   The first term of the max represents walking the section. It may seem counterintuitive that we are ADDING 10 to d. Consider, though, that to achieve d as our current danger after walking, it must have been 10 MORE than d on the previous segment, before decreasing during the walking. The same argument is made for subtracting dangersegment in the second term representing skiing this segment.
4. Base Cases: The danger values have to be bound by 0 and the maximum danger.

The runtime matches that of the previous problem ($O(sd)$), and the code is also quite similar.

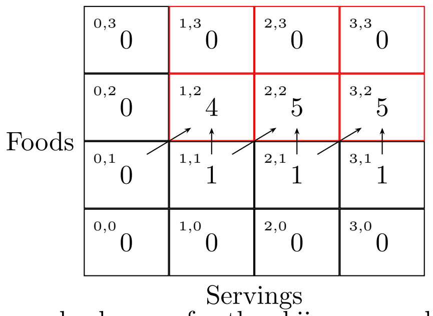
```java
// returns the maxium fun for a given amount of max danger
double[][] memo;
double fun(int max_d) {
    memo = new int[max_d + 1][segments + 1];
    for(int i=0;i<=max_d;i++)Arrays.fill(memo[i], 0);

    for (int i=1; i <= segments; i++) for (int j=0; j <= max_d; j++) {
        if (j + 10 <= max_d) memo[j][i] = Math.max(memo[j][i],
            memo[j+10][i-1] * .5); // get here by walking
        if (j - a[i] >= 0) memo[j][i] = Math.max(memo[j][i], f[i] +
            memo[j-a[i]][i-1]); // get here by skiing
    }

    return memo[max_d][segmets];
}
```

6.4.3 Forward-Looking Iteration

In the examples so far, we've built our recurrence relation by only looking at values we've previously calculated. As we saw with the skiing problem in the previous section, this sometimes leads to awkward relations where we have to think backwards. We can make things slightly easier to reason about by changing the calculation ever so slightly. Instead of arriving at a cell with all dependent values calculated, when we arrive at a cell, we will "lock in" the value that is stored in that cell as the optimal value, and update cells which might depend on this cell. One might note that this is very similar to how many shortest path algorithms work. When drawing how the diagram of how the array is populated, the following changes take place:

- The arrows point upwards and to the right, instead of downwards and to the left
- Boxes still go from red-¿black (simulating holding a known-optimal value) when we reach them in the iteration, however, no computation takes place for that box at that time. Computation for that box has already taken place.
- We update values in boxes that could be affected by our current box

Here is the example of the computation that occurs while processing food 1 from the earlier exercise. Note that row 1 has become black, while row 2 is still red, despite storing some values already.



Here is how the code changes for the skiing example above:

```
// returns the maxium fun for a given amount of max danger
double[][] memo;
double fun(int max_d) {
    memo = new int[max_d + 1][segments + 1];
    for(int i=0;i<=max_d;i++)Arrays.fill(memo[i], 0);

    for (int i=0; i < segments; i++) for (int j=0; j <= max_d; j++) {
        int danger_walking = Math.max(0, j-10); // if we walk with less
            than 10 danger, it goes to 0, elsewise, j-10
        memo[danger_walking][i+1] = Math.max(memo[danger_walking][i+1],
            memo[j][i] * .5); // walking
        if (j + a[i] <= max_d) memo[j+a[i][i+1] =
            Math.max(memo[j+a[i][i+1]], f[i] + memo[j][i]); // skiing
    }

    return memo[max_d][segmets];
}
```

We can see that the code follows much more closely with how we might think about the problem. Both methods are equally valid in this case, though there may be some instances where one makes more sense than the other.

6.4.4 Travelling Salesman

Consider the following problem:

> Given a weighted graph, determine the shortest path from $v \to u$ which visits all nodes.

This is a classical statement of the NP-Complete travelling salesman problem, for which no known efficient algorithm is known. Some algorithms are more efficient than others, though!

The typical brute force recursion-with-backtracking solution to this problem involves recursing on all nodes which haven't yet been visited, and taking the one that yields the shortest path. The runtime is $O(n!)$. In each recursive call, we need to know which node we are at and which nodes have yet to be visited. Ask yourself this, though: If we know we are at a given node, and we know which nodes have been visited, does it matter which path we took to get there? Does the solution for the recursion change? If not, why do we have to compute it multiple times, we should cache it!

1. Indices: the node we're at and which nodes have already been visited
2. Value: the minimum distance to visit those nodes, ending at the current node
3. Relation: We'll use a forward looking relation here. Iterate over all possible mext nodes to see if it yields a better path langth ending at that node
4. Base Cases: it takes 0 distance to be at the starting node with only that node visited

We could attempt to write the DP code at this point, but we'd find difficulty when attempting to figure which order to iterate in. Unlike the previous problems, the recursive relation does not provide an obvious ordering of dependencies. Lets examine the graph with 4 nodes to get an intuitive grasp of where the dependeencies lie.

Diagram

We see that the guarantee we need to enforce is that we need to visit entries in the array which have 1 visited node before those that have 2 visited nodes before those that have 3 visited nodes, etc. That gives us a few ways to perform the iteration.

1. Simply resort to the recursive solution with memoization. Then we don't have to worry about ordering as the recursion stack takes care of it implicitly.

2. Iterate over all permutations of 1 node visited, 2 nodes visited, 3 nodes visited, etc. This is correct, but somewhat unwieldy, since we have to compute next permutation for each count of nodes visited as well as iterating over all the nodes we could be at for that iteration.

3. BFS. We know that the ordering we need requires visiting nodes in order by distance from the start node, where distance is simply the number of ndoes on the path. This is equivalent to the ordering in which BFS visits the array entries. This is far more straightforward to code.

Fortunately, we can reduce this even further. Our above stated requisite guarantee is a bit too strong. Instead of visiting ALL entries where have fewer nodes visited, we only have to guarantee we've visited each entry whose composite nodes represent a strict subset of the current entry. To simplify, considering 4 nodes represented a bitmask, the original guarantee meant we had to visit nodes in the following order (given 0001 as the start node):

1. 0001

2. 0011

3. 0101

4. 1001

5. 0111

6. 1011

7. 1101

8. 1111

Intuitively, there is no reason to have to visit 1001 before 0111, since other than the start node, they share no common visited nodes. This represents the relaxation of the guarantee. Given the relaxation, the following ordering is also valid:

1. 0001

2. 0011

3. 0101
4. 0111
5. 1001
6. 1011
7. 1101
8. 1111

It's easy to see that each node only depends on ones that have come previously, and more importantly, the entries now go in numerical order. It's easy to prove that we can always just iterate through the nodes in numerical order and still be correct. For a given entry, each state we could possibly visit has exactly one more visited node, and thus exactly one more 1 bit in its representation. Changing a bit from 0 to 1 in a binary number necessarily makes it bigger. QED. We can therefore just iterate over all valid states of nodes visited in numerical order, and then which node we are at for that state.

```
// returns the TSP solution starting at 0 and ending at adj_mat.size
int[][] memo;
int tsp(int[][] adj_mat) {
    memo = new int[1 << (adj_mat.length - 1)][adj_mat.length]; //
         2^nodes since we need all combinations, and then which node
         we're at
    for(int i=0;i<=max_d;i++)Arrays.fill(memo[i], Integer.MAX_VALUE);
    memo[1][0] = 0; // distance to start node with only start node
         visited is 0. Everything else is infinite

    for (int i = 1; i < memo.length) for (int j=0; j < memo[0].length;
         j++) if (memo[i][j] != Integer.MAX_VALUE) { // iterate over all
         entries in numerical order, then all nodes we could be at. skip
         unreachable nodes
   for (int k = 0; k < memo[0].length; k++) if (i & (1 << k) == 0 &&
        adj_mat[j][k] != Integer.MAX_VALUE) { // iterate over each next
        node (this is a forward looking DP), skip if we already visited
        this node or no edge between j and k
       int next_state = i | (1 << k); // the next state is this state,
            but with the 1 extra bit added for visiting k
       memo[next_state][k] = Math.min(memo[next_state][k], memo[i][j] +
            adj_mat[j][k]); // update distance to next node if better
   }
    }

    return memo[memo.length - 1][adj_math.length - 1]; // return entry
         with all nodes visited ending at last node
}
```

In the code, we can see the size of the memo matrix is $n*2^n$, and since we do an iteration over n nodes in the inner loop (iterating over k as the next node), the overall runtime is $O(n^2 * 2^n)$.

6.4.5 Prefix Sum

Consider the following problem:

> Given an array A of numbers and an index i, what is the sum of A[j], where $j < i$?

This is known as a prefix sum (sum of all the numbers which are a prefix of i), The naive algorithm is to simply walk the array from 0 to i and compute the sum. This is the best we can do for a single query, but very inefficient if we have multiple queries. As with previous problems, we can see why this is wasteful. If we compute A[4], and then get a query for A[5], we have to recompute A[4] as part of computing A[5], so we might as well just save it. The DP follows simply as a sequential computation of each i.

```
// generates prefix sum array
int[] memo
void ps(int[] A) {
  memo[0] = A[0];
  for(int i=1; i<A.length; i++) memo[i]=memo[i-1]+A[i];
}
```

We can now compute all the prefix sums in linear time, and all subsequent lookups are constant.

6.4.6 Inclusion/Exclusion

Consider the following problem:

> We have an $n \times n$ grid. On that grid there are m special points. Given a query of $(x_1, y_1), (x_2, y_2)$, return the number of special points which are contained in the box bounded at the upper left by the first point, and the lower right by the second (inclusive).

There are a couple of solutions which are simply, yet inefficient:

- Walk each of the m points to see if it is contained by the bonding box. $O(m)$ for each query.
- Create a hash set of the special points. Walk each $x_1 < x < x_2, y_1 < y < y_2$ and see if the set contains that point. $O(n^2)$ for each query.

There are some optimizations which may help in some cases, such as sorting the points, or collapsing the grid to only container rows or columns which contain special points, none avoid the fact that each query depends either on m or n. To help us figure this one, lets simplify the problem slightly:

... Given a query of (x, y) return the number of special points contained in the box bounded by $(0, 0)$ and the given point.

We could use the same naive algorithms as above, but run into the same issues. Thinking of the duplicate work, we see that a query of $(4, 4)$ followed by a query of $(5, 5)$ would require recomputing the $(4, 4)$ solution. This is wasteful and sounds very much like the repeat computation we saw in prefix sum. The challenge becomes how to modify the recursive relation.

Given a memo array populated with the number of special points contained between $(0, 0)$ and (n, m) for all $n < i and m < i$, how can we compute memo[i][i]?

Diagram

We see that based on the data we have, we can lookup memo[i-1][i] and memo[i][i-1]. The issue here is the overlap. We can solve that problem by then subtracting memo[i-1][i-1], since it would otherwise be included twice. The final piece of the puzzle is to check if (i, i) is a special point and add 1. The code looks as follows

```
// generates an array on the count of special points <= i,j
int[][] memo
void special(int n, HashSet<Point> special} {
  memo=new int[n+2][n+2]; //we size this larger than usual so we don't
      run off the end of the array when evaluating x or y == 0. Also
      means we need to shift everything by 1 elsewhere
  for(int i=1;i<=n;i++)for(intj=1;j<=n;j++){
    memo[i][j]=memo[i-1][j]+memo[i][j-1]-memo[i-1][j-1];
    if(special.contains(new Point(i-1,j-1))memo[i][j]++; // we
        subtract 1 in the special lookup because the whole array is
        offset by 1 so we don't run off the end of the array
  }
}
```

To understand why we shift the array by 1 in each direction, consider what we need to populate in the rows where i or j == 0. We'd either need to special case populate that row and column, or we could have a dummy "-1" row which is initialized to 0 so the relation looks the same as the rest of the array. Diagram

Since we can't index -1, we shift the array so the entire thing is offset by 1. This means when we're looking at memo[1][1], we're really looking at point [0][0]. This explains why we substract 1 before our lookup into the special set.

Now that we have the solution for our simplified problem, we can extend it to the original problem using the same method of inclusion and exclusion we used in the DP computation. Diagram

We can see we add the large box bounded by the second point and $(0, 0)$, then subtract off the two boxes on the side and top, then have to add back in the smaller box since we substracted it twice.

```
int count(int[][] memo, Point first, Point second) {
  return memo[second.x][second.y] - memo[second.x][first.y] -
      memo[first.x][second.y] + memo[first.x][first.y];
}
```

The DP calculation is $O(n^2)$, but each subsequent lookup is $O(1)$. We can use the interesting points optimization below to decrease the bounds further when $m << n$.

6.4.7 Mini-Max

Consider the following problem:

> Alice and Bob are playing a game. Laid out before them is a deck of n cards, each with a distinct integer between 1 and n inclusive. The cards are laid out in a row in random order. Alice goes first and selects the card on either end of the row, adding it to her total. Then bob does the same. They alternate turns until all cards are taken. Each player is trying to ensure their total exceeds the other's by as much as possible. Assuming each play optimally, what is the value of alice's score minus bob's?

If n were small, we could simply evaluate all possible choices recursively. Unfortunately, n probably isn't small. Greedy solutions may seem promising, but we can imagine possibilities where early decisions may have adverse implications later on (suppose Alice can do well by taking high cards off one side, but later on, it leaves Bob to take an extremely highly valued card). Since we can't factor all possible implications into a greedy heuristic, this probably isn't promising. So instead we examine where we might be doing repeat work in a greedy solution. Imagine a case where Alice selects a card on the left, then Bob selects a card on the right. We now have an array from 1 to $n-1$. Is the solution to that subproblem any different than were Alice to have selected the card from the right and bob from the left? Do we need to evaluate it again? Surely not. This leads us close to a DP solution.

1. Indices: We need to know the range of the array, but we also need to know whose turn it is, since each player is optimizing for themselves. The left hand index will be inclusive, and the right hand exclusive. This helps ensure the difference between the two indices represents the number of cards remaining (and matches things like String.subString).

2. Value: This one is a bit tricky. How do we encode both players scores? Each is trying to maximize their difference to the other players score. We can reduce this to simply storing Alice's score minus Bob's, and on Alice's turn, we will attempt to maximize thie value, and on Bob's, we will attempt to minimize it.

3. Relation: on Alice's turn, with cards $i \to j$ remaining, memo[i][j][alice] = max(memo[i+1][j][bob] + card[i], memo[i][j-1][bob] + card[j-1]). We can either select the left or right hand card. Bob's relation is similar, except we want the minimum, and we subtract the card value (since we're storing the difference to Alice's score). Note that we use j-1 since the right hand side of the range is exclusive. When it's bob's turn, we subtract our chosen card value and try to minimize instead.

4. Base Cases: memo[i][i] = 0.

The iteration order is slightly tricky here. Note that in our recursive relation, we're not decreasing or increasing any of the indices, instead we're decreasing the difference between i and j, namely, the number of cards remaining. This means we have to iterate from 0 cards remaining through n.

The code follows nicely:

```
int game(int[] card) {
    int[][][] memo=new int[card.length+1][card.length+1][2]; // we'll
         say 0 == alice, 1==bob.
    for(int i=0;i<card.length;i++)for (int j=0;j<2;j++)memo[i][i][j]=0;
         // if no cards left, no difference!
    for(int size=1;size<=n;size++)for (int
         start=0;start<n;start++)for(int player=0;player<2;player++){
   if(player == 0) { //alice
       // we try to maximize, and our third index is 0 because we're
            evaluating alice. Either take the first or last card of the
            segment and
       // add it to bob's optimal difference of the remaining cards
       memo[start][start+size][0] =
            Math.max(memo[start+1][start+size][1] + card[start],
            memo[start][start+size-1][1] + card[start+size-1]);
   } else { // bob
       // we try to minimize, so the card we select is subtracted from
            our total. Note that we have swapped the third index in each
            case.
       memo[start][start+size][1] =
            Math.min(memo[start+1][start+size][0] - card[start],
            memo[start][start+size-1][0] - card[start+size-1]);
   }
    }

    return memo[0][card.length][0]; // optimal solution with whole array
         when we have all the cards and it's alice's turn
}
```

Since the array is $n \times n$, and we do constant work for each entry, the total runtime is $O(n^2)$.

6.4.7.1 Simplifying

To make the code more concise, we can eliminate the third DP index and for loop, saving both memory and code size. If we assume every entry in our DP array is implicitly representing Alice's turn, it means our recursive relation can only refer to values when it's alice's turn. To do this, we expand our relation to encompass 4 possibilities instead of just 2:

1. Alice takes the left card and then Bob takes the right
2. Alice takes the left card and Bob takes the new left card
3. Alice takes the right card and Bob takes the left
4. Alice takes the right card and Bob takes the new right card

The simplified code looks like this (Note that we have to add a base case for if there's only 1 card left or we could simulate bob taking cards which aren't there!):

```
int game(int[] card) {
    int[][] memo=new int[card.length+1][card.length+1];
    for(int i=0;i<card.length;i++)memo[i][i]=0;
    for(int i=0;i<card.lenght;i++)memo[i][i+1]=card[i]; // extra base
        case
    for(int size=1;size<=n;size++)for (int start=0;start<n;start++){
   memo[start][start+size] = Math.max(card[start] +
        Math.min(memo[start+2][start+size] - card[start+1], // alice
        left, bob left
                           memo[start+1][start+size-1] -
                                card[start+size-1]), // alice left, bob right
                  card[start+size-1] +
                      Math.min(memo[start+1][start+size-1] - card[start],
                      // alice right, bob left
                           memo[start][start+size-2] -
                                card[start+size-2])); // alice right, bob
                                right
    }
    return memo[0][card.length];
}
```

Even though we've halved our memory, the runtime is not faster, even by a constant, since are doing twice as much work for ecah entry, but there are half as many. We have made the code a bit more concise, though. The nature of taking the *maximum* of the subsequent *minima* is where the name Mini-Max comes from.

6.4.7.2 Simple Game Victor

Consider the following simplification of the problem:

> Each card laid out in the row, instead of having a number, is either red or blue. As the players alternate taking turns, Alice attempts to collect the rded cards. If when all the cards have been collected, Alice has all the red cards, she wins, otherwise Bob does. Assuming optimal play, who will win?

It turns out this is simply a binary version of the above problem. Instead of having to worry about totals and differences, we simply have to store whether a state results in a win for Alice. When he makes a move, obviously Bob is not going to move to a state which results in a win for Alice (since we know she plays perfectly). So for Bob to lose, BOTH of the moves he could make must be losing. When it's Alice's turn, if either of her possible moves force Bob to make a losing move, she wins. When viewed as a binary where a 1 represents a win for Alice, and 0, a loss for Alice, Bob is taking the minimum of the two resulting moves, and Alice is taking the maximum. The following two rules follow.

- Alice wins if one of her moves leads to a loss for Bob. (Maximum of her two potential moves)
- Bob loses if BOTH of his moves lead to a win for Alice. (If either of his two potential moves is a 0, he takes it, so a minimum)

```
bool[][] winning_alice;
bool game(bool[] red_card) {
    winning_alice=new bool[card.length+1][card.length+1];
    for(int i=0;i<card.length;i++)winning_alice[i][i]=true; // alice
        wins if no cards
    for(int i=0;i<card.lenght;i++)winning_alice[i][i+1]=true; // alice
        also wins if just 1 card
    for(int size=1;size<=n;size++)for (int start=0;start<n;start++)
        winning_alice[start][start+size] = losing_bob(red_card,
        start+1,size-1) || losing_bob(red_card,start,size-1); // alice
        wins if either of her moves causes a loss for bob
    return winning_alice[0][card.length];
}

bool losing_bob(bool[] red_card,int start,int size){
    if (red_card(start)) return false; // base cases. Bob doesn't lose
        if he can take a red card
    if (red_card(start+size-1) return false;
    return winning_alice[start+1][start+size] &&
        winning_alice[start+1][start+size]; // both blue cards, he loses
        if both moves winning for alice
}
```

This strategy applies to many types of game theory problems where the players alternate turns.

6.4.8 Expected Value

Consider the following problem:

> Tim likes going shopping. He walks up and down the storefronts and stops to look at the items. Sometimes pick-pockets stand outside the stores, and when Tim stops, they steal some amount of his remaining money. Given the probability of Tim getting robbed outside each store, what is the expected amount of money he has after looking at items at each store (Tim is on in trouble with his spouse so will not purchase anything today, only potentially get robbed). The probabilities of getting robbed outside any pair of stores are independent, and Tim walks past each store in order and only once. Tim is guaranteed to have enough money to not run out even if he is very unlucky and is robbed at every store.

Given we have two options at each store, get robbed or don't get robbed, and we can calculate the expected value on a store-by-store basis (Since the probabilities are independent!), we should be able to formulate a 1-dimension DP, similar to fibonacci:

1. Indices: Since we walk by the stores in order, it's a very strong hint that's our index.
2. Value: The expected amount of money we have after visiting each store.
3. Relation: Unlike our previous DPs, we don't take the max of potential values, but simply apply the expected value definition over all possibilities. Assuming p_i is the probability of getting robbed outside the i-th store, and a_i is the amount taken there, then by definition of expected value $\text{memo}(i) = p_i * (\text{memo}(i-1) - a_i)) + (1 - p_i) * (\text{memo}(i-1))$. We either get robbed or don't.
4. Base Cases: $\text{memo}(-1) =$ starting-money Note that since our base case is out of bounds, we'll have to shift the index into the memo array by 1.

```java
double money(int[] a, double[] p, int starting_money){
    double[] memo = new double[a.length+1]; // allocate one extra entry
        since the indices are shifted by 1
    memo[0]=starting_money; // base case
    for (int i=0;i<a.length;i++)memo[i+1]=p[i]*(memo[i]-a[i]) +
        (1-p[i])*(memo[i]); //iterate through the array and apply
        relation
    return memo[a.length];
}
```

6.4.8.1 Application to other DP types

While in this case, the application of the expected value is straightforward, it can also be applied to some of the standard DPs we have seen previously. Consider the following modification:

> Time really wants to purchase some gifts for his spouse. At each shop, he can stop and purchase a gift worth some amount. If he stops, however, his probability of getting robbed incrases to some heightened value. Assuming bob wants to buy at least g gifts for his spouse (he is cheap so doesn't care which ones he buys so long as be has g of them!), what is his maximum amount of expected amount of remaining money? Tim is smart so has enough money to deal with getting robbed at every store while buying any combination of three gifts.

You may note some similarities to a previous class of problems.

1. There are two variables, each of which we are trying to maximize or minimize
2. There are multiple classes of items, each contributing varying amounts to the two variables.
3. We can choose to take or not take some of each item

If you recognized this as a knapsack without repeats, you are correct. We have our two variables, money and gifts, and we are trying to maximize our money while minimizing the the amount of gifts we have left to buy. At each store we can choose to buy or not, and this affects are gifts or money with some probability.

1. Indices: As with before, which store we are on is one of our indices. Also, one of the things we're trying to optimize. In this case it must be the number of gifts left to buy.
2. Value: The expected amount of money we have for a given store and number of gifts left to buy
3. Relation: The relation looks very similar to the previous knapsack, except in each case, we apply the definition of expected value before selecting the maximum. The cost of each gift is c_i and the probability of getting robbed if a purchase is made is q_i. Note that gift represents gifts remaining to buy, and this is a backwards looking dp, so we use gift + 1 to represent having one MORE gift left to buy when we were at the previous store.

$$\text{memo}(\text{store}, \text{gift}) = \max \begin{cases} p_i * (\text{memo}(\text{store-1}, \text{gift}) - a_i) & \\ +(1-p_i) * (\text{memo}(\text{store-1}, \text{gift})) & \text{don't buy a gift} \\ q_i * (\text{memo}(\text{store-1}, \text{gift}+1) - c_i - a_i)) & \\ +(1-q_i) * (\text{memo}(\text{store-1}, \text{gift}+1) - c_i) & \text{bought the gift} \end{cases}$$

4. Base Cases: memo$(0, g) = starting_money$. Note that we'll have to shift the indices by 1 to account for this base case. There is a caveat here. We need to encode that memo$(0, 0 <= i < g)$ are unreachable states, otherwise it will mess up our calculations. We can simply catch this in the relation and not allow us to consider values which are impossible (such as after store 1, having fewer than $g - 1$ gifts left to buy).

```
double money(int starting_money, int g, int[] c, int[] a, double[] p,
     double[] q){
    double[][] memo=new double[a.length+1][g+1];
    memo[0][g] = starting_money;
    for(int store=0;store<a.length;store++)for(int
         g_left=0;g_left<=g;g_left++){
   memo[store][g_left] = q[i] * (memo[store-1][g_left+1]-c[i]-a[i]) +
        (1-q[i])*(memo[store-1][g_left+1]-c[i]); // bought the gift
   if (g-g_left > store) { // can only get here while NOT buying gift if
        we've visited enough stores already. For instance at store 1, if
        we've bought 1 gift, we HAD to have bought it at this state.
       memo[store][g_left] = Math.max(memo[store][g_left], p[i] *
            (memo[store-1][g_left]-a[i]) +
            (1-p[i])*memo[store-1][g_left+1]; // didn't buy the gift
   }
    }
    return memo[a.length][0];
}
```

Similarly to how we applied probability to knapsack, it could potentially be applied to other problem types such as travelling salesman or game-theory. In our recursive relation, we just have to know that instead of choosing one option or the other, we have to calculate the expected value of choosing one or the other.

6.4.9 Grid Tiling

Consider the following problem:

> You have a pack of 1×2 tiles and a $3 \times n$ sized grid. How many ways are there to completely tile the grid?

While not necessarily common, this type of problem comes up often enough to warrant discussion. The characteristics of these problems are a small number of small, fixed shape tiles, and a grid which is relatively small in one dimension and unbounded in the other. While we may try to recursively enumerate all possible combinations, we find this is too slow. We can see the repeat work. Consider the two following arrangements of the first few tiles:

Diagram

The placement of the dominoes yielded a smaller grid which will have the same number of ways to tile regardless of which of the two initial tilings we chose. There is no reason to evaluate it again. This makes it clear that one of the DP states will have to do with how far we have progressed down the grid. The remaining x-space we have to fill can't be the ONLY state, however, as consider the following intermediate state How would we encode this?

Diagram

Along with how far we are along the grid, can we encode the shape of the boundary in a concise enough way to be viable? We note that since the tiles are at most 2 wide, if we place them in order from left to right, we can always do so in such a way that the difference between the left-most point on the boundary and the rightmost point is at most 1.

Diagram

Since the boundary is only 1-wide, we can encode any boundary shape using a bitmask, with a 1 representing the precense of a tile, and 0 indicating the space is uncovered. This means we have a very limited number of shapes that the boundary can take, namely 2^n. Those possibilities are enumerated here along with their binary representation.

Diagram

We now have the workings of a DP solution.

1. Indices: The shape of the boundary, encoded in binary, along with the index of the left-most uncovered square on the grid.

2. Value: the number of possible ways to arrive at the boundary encoded by the location and shape

3. Relation: This is tricky. While it might seem promising to place 1 tile a time, this leads to issues as soon as we try to place the first horizontal domino: Diagram As soon as it is placed, the boundary spans two columns instead of just 1, and we have no way to account for this in our DP table. To avoid this problem, we have to simultaneously place all the dominoes which complete the row we're on. In the horizontal domino case, this leaves us with 2 other possibilties, each of which results in a valid boundary: Diagram So the relation is as follows: Given a boundary, enumerate over all possible ways to complete the row, and add the number of ways to reach our current state to the state that particular completion leads to.

4. Base Cases: There are 1 way to tile an empty grid

Let's look at all possible boundaries and how we can complete the row.

- 000
  - HV: Update 001 in next column
  - HHH: update 000 two columns over
  - VH: update 100 in next column
- 001
  - HH: update 110 in next column

  - V: update 000 in next column
- 010
  - HH: update 101 in next column
- 011
  - H: update 100 in next column
- 100
  - HH: update 011 in next column
  - V: update 000 in next column
- 101
  - H: update 010 in next column
- 110
  - H: update 001 in next column
- 111: This is not really an item, since it would be 000 in the next column.

While we could have written code to enumerate each of these possibilities, most times the number of cases is small enough to not be necessary.[11] Here we have only 10 specific cases. we'll write code for each.

```
long tile(int n) { // often times need a long since the count grows
     exponentially
    long[][] memo=new long[n+1][8]; // 8 possible shapes in n locations
    memo[0][0]=1; // base case
    for(int i=0;i<n;i++){
   if(i<n-1){ //can't evaluate H cases when only 1 column available
      memo[i+1][1] += memo[i][0]; // 000+HV = 001
      memo[i+2][0] += memo[i][0]; // 000+HHH = 000
      memo[i+1][4] += memo[i][0]; // 000+HV = 100
      memo[i+1][6] += memo[i][1]; // 001+HH = 110
      memo[i+1][5] += memo[i][2]; // 010+HH = 101
      memo[i+1][4] += memo[i][3]; // 011+H = 100
      memo[i+1][3] += memo[i][4]; // 100+HH = 011
      memo[i+1][2] += memo[i][5]; // 101+H = 010
      memo[i+1][1] += memo[i][6]; // 110+H = 001
   }
   memo[i+1][0] += memo[i][1]; // 001+V = 000
   memo[i+1][0] += memo[i][4]; // 100+V = 000
    }

    retrun memo[n][0];
}
```

[11]a particularly mean problem writer may require this someday!

So long as the vertical dimension is small enough, the runtime will be reasonable. If m is the vertical dimension, the size of the array is $O(n \times 2^m)$. The Number of possible tilings of a single row is also exponential in m, leading to a total runtime of very roughly $O(n \times 4^n)$. This runtime demonstrates why the vertical dimension must be very small.

6.5 Optimizations

With an understanding of the types of problems we can solve with dynamic programming, we can now look at some tricks to speed it up when the 'obvious' solution is not sufficient.

6.5.1 Memory Reduction

Suppose you are working on a problem similar to the skiing problem above and get a runtime error, though are sure your code doesn't have any invalid array access, bad input reads, or anything of the like. It's quite possible depending on the input size, that you could have run out of memory! This is an easy fix, though. If we look at our recursive relation, we see that we only ever depend on values 1 segment previous. For a given segment, once we've reached the segment after next, we never refer to the segment again. This is clear from the drawn dependency diagrams which show the arrows only ever going back 1 row or column.

The optimation is simply to only allocate two segments worth of data at a time, representing the previous segment and the current segment. Once we progress to the following segment, we discard the original segment, meaning we only ever hold on to 2 segments worth of data. This makes our memory usage independent of the number of segments, instead of scaling linearly with it. The code is modified as follows:

```
// returns the maxium fun for a given amount of max danger
int fun(int max_d) {
    int[] current = new int[max_d + 1];
    for (int i=1; i <= segments; i++) for (int j=0; j <= max_d; j++) {
   int[] next = new int[max_d+1]; // allocate the column we're about to
        populate
        if (j + 10 <= max_d) next[j] = Math.max(next[j], current[j+10] *
             .5); // get here by walking
        if (j - a[i] >= 0) next[j] = Math.max(next[j], f[i] +
             current[j-a[i]]); // get here by skiing
   current = next; // we're done with current. point it to the new data,
        allowing the old to be freed
    }

    return current[max_d];
}
```

We can get some additional constant speedup by swapping the arrays instead of allocating/freeing/initializing them each time, but usually this is not necessary.

This technique can be applied to any solution for which we can limit how far back we look in the DP array. In fibonacci, for example, we can discard any number more than 2 previous, since it will never get used again.

6.5.2 Dimension Swapping

Lets remember back to our earlier buffet problem with item limits. We sook to evaluate the maximum happiness H attainable with I items and S servings. Our solution involved a $I \times S$ array which stored values of H. This $O(I * S)$ solution is often optimal and sufficient. Consider, though, if $S >> H$. Our runtime will be large (since it depends on S), though in our array, we are storing a relatively small number of unique values for H.

We can do something about that by taking the dual of the problem. Instead of considering the maximum happiness that can be attained for a given number of servings, instead consider the minimum number of servings that are needed to attain a given amount of happiness. If we could do such a thing, we'd reduce our runtime to $O(I * H)$, which based on our supposition that $S >> H$ must be much better.

1. Indices:
   (a) The amount of happiness we've obtained
   (b) The index of the food we're currently standing in front of (we've already evaluated those with index less than that, and will never evaluate them again per the new restriction). It is a good rule of thumb that any time we go in some order, such as by time, or food order, it will be an index of the DP.
2. Value: The minimum amount of servings required to obtain the given happiness
3. Relation: For a given food, we either eat it or not, as before, except we calculate the amount of happiness we need to be at, and use that to look up the minimum servings in the table.

$$\text{memo}(i, h) = \min \begin{cases} \text{memo}(i-1, h+H_i) & \text{eat this item} \\ \text{memo}(i-1, h) & \text{don't eat this item} \end{cases}$$

4. Base Cases: Same as before, we can't evaluate outside the bounds of the array

There are two tricks in the conversion from one problem to the other:

1. We have to figure out how large to make the h dimension of our array. It must be big enough to hold the largest happiness we could possible get. The simplest way to do this is to bound it by the sum of the happiness of all the items. That way in the 'worst case' if we eat every item, the array can still hold it.
2. We still have to return the maximum happiness for a given number of servings. To do this, we simply iterate over the last column of the DP array, examining any entry that has fewer than the requisite servings and return the one with the highest happiness.

```
// returns the maxium happiness for S servings
int happy(int[] S, int[] H, int[] Q, int serving_limit) {
    // calculate the bounds on H
    int sum=0;
    for(int i=0;i<H.length;i++)sum+=H[i]*Q[i];

    memo = new int[food_types + 1][sum+1];
    for (int i=0; i < food_types; i++) for (int j=0; j <= sum; j++) { //
         loop over all food types, and for each food type, check every
         possible happiness amount
   memo[i+1][j]=memo[i][j]; // don't eat the food
   if(j>=H[i]*Q[i])memo[i+1][j]=Math.min(memo[i+1][j],
        memo[i][j-H[i]*Q[i]]); // if we could have eaten the food, check
        if it's fewer servings
    }

    // compute maximum H bound by the serving limit
    int maxh=0;
    for(int i=0;i<=sum;i++)if(memo[Q.length][i] <=
         serving_limit)maxh=Math.max(maxh,i);
    return maxh;
}
```

Remember this optimization for knapsack-like problems where one of the values we're trying to optimize is significantly greater than the other.

6.5.3 Interesting Points

Think back to the inclusion/exclusion problem. Our DP solution involved computing first the number of points between (x, y) and the origin. Consider, though, if our grid size $n \times n$ is significantly greater than the number of points m. Our array is sized to the grid, giving us an $O(n^2)$ runtime. Consider when the values in our DP array change:

Diagram

We see that the elements only ever change when we reach a special point. As such, most of the work is wasted. We can instead "collapse" the grid such that every row and column has a point in it. This reduces the grid, and thus our runtime to $O(m^2)$.

TODO fix this code and add diagram

```
// generates an array on the count of special points <= i,j
int[][] memo
void special(int n, HashSet<Point> special} {
   memo=new int[n+2][n+2]; //we size this larger than usual so we don't
        run off the end of the array when evaluating x or y == 0. Also
        means we need to shift everything by 1 elsewhere
   for(int i=1;i<=n;i++)for(intj=1;j<=n;j++){
      memo[i][j]=memo[i-1][j]+memo[i][j-1]-memo[i-1][j-1];
```

```
        if(special.contains(new Point(i-1,j-1))memo[i][j]++; // we
            subtract 1 in the special lookup because the whole array is
            offset by 1 so we don't run off the end of the array
    }
}
```

6.5.4 Dimension Elimination

Consider the following problem:

> We have a group of P people standing in a line. Each pair of people has a known *friendship value* which indicates how much those people like eachother. We wish to partition the line into N groups, with each group comprising some number of consecutive people in line. The friendship value of a group is defined as the pairwise sum of the friendship value of the members of a group (i.e. the freinship value for all $n * (n-1)$ unique pairs of people in the group). We wish to maximize the sum of the friendship values of all the groups.

The typical and natural way to frame this DP is as follows:

1. Indices: number of groups to form n, the sub-segment of people on which to form those groups (i.e. only consider person x to person y). For simplicity, we will consider the sub-segment as defined by the index of the first person in the sub-segment, l, and the size of the segment, s.

2. Value: the maximum friendship to form one can obtain by splitting the s people, starting at person l, into n groups

3. Relation: For any given l, s, n, triple, we want to look at all possible places to create a division, and for that division, we look at all possible assignments of groups on either side of the division, leading to a double for loop. $\text{dp}[n][l][s] = max_{0<i<s,0<j<n}(\text{dp}[j][l][i] + \text{dp}[n-j][l+i][s-i]$

4. Iteration Order: we are always looking at smaller numbers of divisions, and smaller lengths. So long as we compute those first, then we will always have a result ready when we need it.

5. Base Cases: dp[1][l][s]=frienship(l,l+s) (where friendship() is the pairwise sum of friendship values of all peopler from l to l+s), and dp[n][l][0]=-inf, since we cannot have a group of 0 people.

TODO: Insert diagram here showing how the relation works.

This will surely find a solution. The size of the table is $O(N * P^2)$, and the amount of work we do for each is $O(N * P)$ due to the double for loop (we will assume a constant time query of `friendship()`), leading to total $O(N^2 * P^3)$ runtime. If presented with a similar problem, this will almost surely be too slow, even when considering the fact that the table is relatively sparse. Given that, we should look at what potential repeat work we are doing in this DP.

6.5.5 Secondary DP

6.5.6 Restricted Search

6.5.7 Divide and Conquer Optimization

Consider again the problem we used for Dimension Elimination. As a refresher, we arrived at a DP state of divisions remaining and index in line, and stored the maximum friendship value of the groups. The relation for solving d divisions among the first n involved iterating over all possible locations of the final split s and taking the one which maximized `dp[s][d-1] + pairwise_sum(s+1,n)`. Given that we used secondary DP to allow us to lookup the pairwise sum in constant time, the total work for this DP is the size of the tape ($N * D$) times the work per entry (N), or $O(N^2 * D)$.

Proof a typical cost function satisfies QI:

for all A¡=B¡=C¡=D:

f(A,C) + f(B,D) ¡= f(A,D)+f(B,C) (quadrangle inequality stated...AKA ”wider is worse”)

let comb(A,B,C) == f(A,C) - (f(A,B)+f(B,C)), i.e., the extra terms needed to combine two consecutive intervals

f(A,C) + f(B,D) = f(A,B) + f(B,C) + comb(A,B,C) + f(B,C) + f(C,D) + comb(B,C,D)

f(A,D)+f(B,C) = f(A,B)+f(B,C)+comb(A,B,C)+f(C,D)+comb(A,C,D)+f(B,C)

comb(B,C,D) ¡= comb(A,C,D) ,therefore the inequality holds for this function

Proof the typical DP relation can be simplified if the cost function satisfies QI:

we can similarly prove the monotonicity of opt(e) with e.

suppose opt(e) ¡ opt(e-1) let A=0, B=opt(e), C=opt(e-1), D=e-1, E=e dp(A,C)+f(C,D)¡=dp(A,B)+f(B,D), by definition of opt(e-1) dp(A,C)+f(C,D)¡=dp(A,B)+f(B,C)+f(C,D)+com (using hte definitions in the proof above) conclude: dp(A,C)+f(C,D)-dp(A,B)-f(B,C)-f(C,D)-comb(B,C,D) ¡= 0

dp(A,B)+f(B,E) ¡ dp(A,C)+f(C,E) by the supposition dp(A,B)+f(B,C)+f(C,D)+comb(B,C,D)+f(D,E)+com ¡ dp(A,C)+f(C,D)+f(D,E)+comb(C,D,E) f(D,E)+comb(B,D,E) ¡ dp(A,C) + f(C,D) - dp(A,B)-f(B,C)-f(C,D)-comb(B,C,D)+f(D,E)+comb(C,D,E) f(D,E)+comb(B,D,E) ¡ k + f(D,E) + comb(C,D,E), where k¡=0, from the above conclusion

contradiction: comb(B,D,E) ¡ comb(C,D,E), by the same thing we used to prove QI in the first place above.

So IIRC, QI is a necessary and sufficient condition to apply DaC optimization to relations of this form. or perhaps more intuitively, comb(B,C,D) ¡= comb(A,C,D) and comb(B,C,E) for your value function....

6.5.8 Knuth’s Optimization

6.5.9 Convex-Hull Optimization

Consider the following problem:

Craig runs a dojo, in which he teaches n kung-fu students, ordered 1 to n by ability, a_i. A new protege has arrived at the dojo, and naturally, Craig tells them that to prove he is the best, he must defeat his best current student in a kung-fu battle. Craig cannot simply set up the matchup, however, and the protege must prove he is worthy by beating some lesser students first.

6.6 Identifying DP

While it's been touched on throughout this chapter, here are some hints that a problem might be DP:

- It seems to share something in common with some of the standard DP problems above
- Brute force solutions are too slow, but there is no greedy heuristic
- Trying to come up with a greedy solution yields seemingly never-ending edge cases
- There is repeat work being done somewhere, where the repeated work is independent of the work it took to get there

Even if you think a problem might not be DP, it is useful to look at the input bounds, and see which combination of those inputs might yield a runtime which is fast enough. This is often enough to ponder if those inputs might form the indices of a DP state.

Chapter 7

NP Completeness

Chapter 8

Data Structures

8.1 segment trees

8.2 interval trees

8.3 Fenwick Trees

Consider a problem which takes arbitrary values in a list and requires us to do all of the following:

1. Compute the sum of the values between any pair of indices
2. Update any values in the array
3. Remove values from any point in the array
4. Perform these operations in sub-linear incremental time

A prefix-sum, which allow us to compute the sum of (i, j) as `prefix_sum(j)-prefix_sum(i)`, would take linear time to update or remove a value. The list itself enables us to update and remove any value, but takes linear time to compute the range sum query.

In many cases where we need sub-linear time for seemingly competing operations, a typical approach is to use a tree to reduce each of the competing operations to logarithmic time. The approach is also correct here, where a *Fenwick tree* provides the necessary structure to perform all operations in incremental logarithmic time.

8.3.1 Understanding the Structure

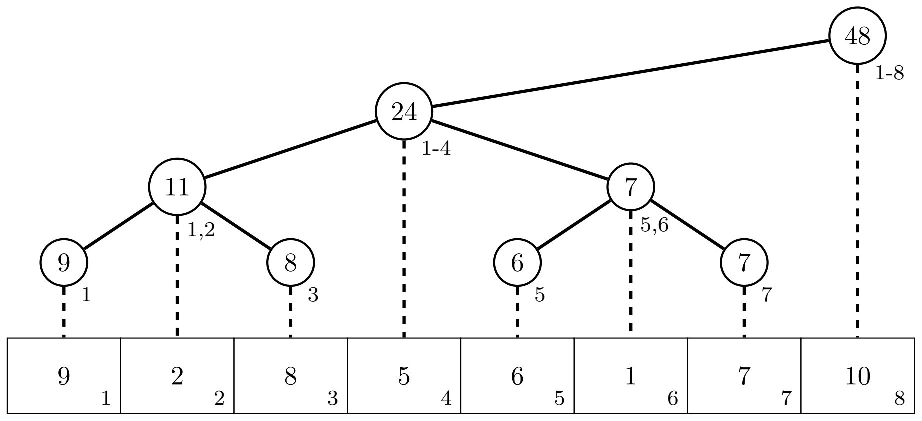
Fundamentally, the Fenwick tree allows us to compute a prefix sum, or modify an element, in logarithmic time. This enables querying individual values (by subtracting the prefix sum of consecutive elements), and removal(by updating the value to 0).

From a high level, we will construct a binary tree with the following rules:

1. An in-order traversal of the tree maps to the indices of the list
2. Each node contains the sum of all elements mapped to its left subtree and the value mapped to the node itself.

Let's see this in action with a tree mapping an array.

- The input array is at the bottom, with 1-indexed elements
- The tree is above, with dashed lines showing the mapping from array to tree nodes
- Tree nodes contain the value as defined by rule 1 above
- The small indices adjacent to tree nodes indicate the nodes comprising the sum contained in that node, as described above



To be perfectly clear where these values come from, we will walk through each node in the tree and explain its value.

1. There is no left subtree, so this node only ends up containing the value of the corresponding array element
2. The sum of all the elements corresponding to the left subtree is 9 (just the first elemenrt), and the value of the corresponding array element is 2. $9 + 2 = 11$
3. There is no left subtree, so this node only ends up containing the value of the corresponding array element
4. The array elements corresponding to the left subtree contain 3 elements, 9, 2, and 8, summing to 19. We then add 5 for ourselves. $19 + 5 = 24$

5. There is no left subtree, so this node only ends up containing the value of the corresponding array element

6. The sum of all the elements corresponding to the left subtree is 6 (just the fifth element), and the value of the corresponding array element is 1. $6 + 1 = 7$

7. There is no left subtree, so this node only ends up containing the value of the corresponding array element

8. The left subtree contains all the elements of the list but the last. We add the 10 for ourselves, and are left with 48.

For now we can assume we have the tree constructed. We'll discuss how to build the tree later in this section.

8.3.1.1 Building a Sum

Now that we have a tree, we can see how we can use it to build a sum. Notably, for some indices (1,2,4, and 8, specifically), the value in the tree node is exactly the prefix sum. How, though can we compute the sum for other indices? Let's consider node 5 as an example.

From a basic level, the prefix sum for index 5 contains the sum of elements 1,2,3,4, and 5. Looking at our tree, can we identify any set of nodes whose coverage includes exactly those 5 elements? Yes. Node 5 and 4 in the tree contain exactly the right set of coverage, and not coincidentally, the prefix sum after the 5th node is $6 + 24 = 30$.

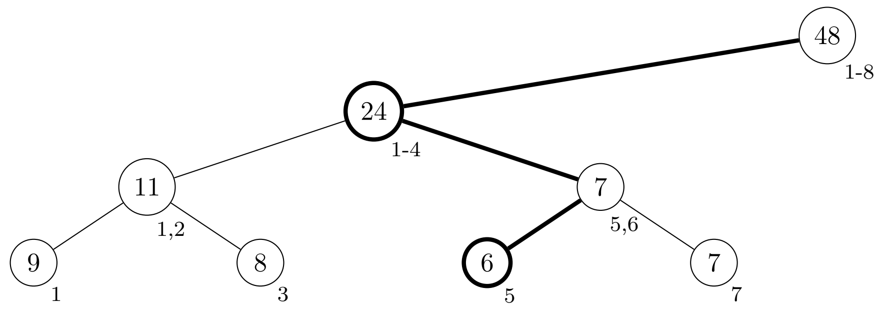
It turns out that if we look at any particular index for which we wish to compute a prefix sum, there exists some selection of nodes in the tree which cover exactly the requisite elements to compute such a sum for that index. Further, those nodes all appear on the path from the node representing that index to the root. The solution for each index is in the table below. Note that for each index, the listed nodes have two properties:

1. They all have the exact coverage to produce the proper prefix sum for the given index

2. They all appear on the path from the node representing this index to the root

index	included nodes
1	1
2	2
3	3,2
4	4
5	5,4
6	6,4
7	7,6,4
8	8

How can we arrive at this mapping for a given index on the fly, though? Lets take a look at the computation for index 5. We show the tree, with bold indicating the path from node 5 to the root, and nodes which are included in the sum.



It may be difficult to find the pattern at first, but note that as we recurse up from the start index, we include a node if we have arrived at it from the **right** subtree, and we do include it if we arrived from the **left** subtree. We can see why this must be true by examining the nodes not included in the above example. Note that each node that is not included in the sum, but is on the path, contains values for which one of the following is true:

1. The value is already included in the sum, and thus would be double counted if we added in this node. This is true for element 5 in the node labeled 5,6, as well as all of 1-5 in the node labeled 1-8.

2. The value should not be included in the sum at all. This is true for element 6 in the node labeled 5,6, as well as 6-8 in the node labaled 1-8.

You can perform the exercise we performed with index 6 beginning at any index and find that this property of only including when arriving from right subtree generates exactly the table above, and allowing us to compute any prefix sum from the tree. The overall algorithm is as follows:

1. Initialize a sum starting with the value at the node representing the element you want the prefix sum for.

2. Recurse up the tree

    - If you reached a node from the left subtree, then add nothing. The elements comprising this node overlap with the sum we have already computed, and contain further nodes from that left subtree that we do not wish to add in.
    - If you reached a node from the right subtree, then the value at this node includes all the values from the left subtree, none of which we have yet added to our sum. Add this value.

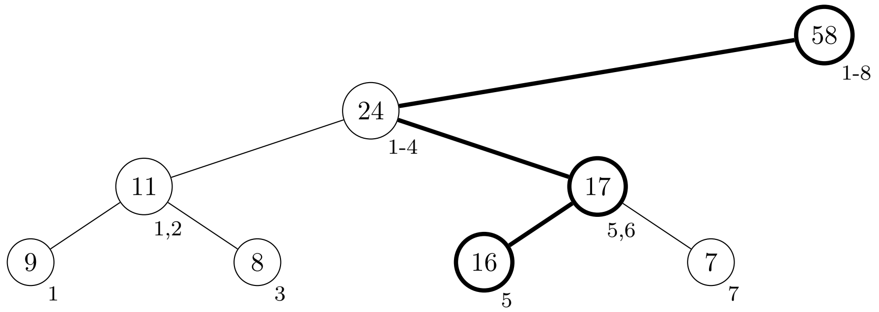
As we progress up the tree, we add in larger and larger chunks of the prefix sum, ensuring at every level we have added in all the necessary values for the subtree rooted at the node we reached. As the tree is balanced, this operation is logarithmic.

For completeness, let's walk through a specific example, computing the prefix sum at $i = 5$.

1. The node we start at contains 6. We begin with this as our sum. $s = 6, v = (5)$
2. We recurse to node 6. We arrived from the left subtree, so we do nothing. This is natural as this node covers the values $i = (5, 6)$, one of which our sum has already included,and the other of which is out of range for the sum we are trying to compute. $s = 6, v = (5)$
3. We recurse to node 4. We arrived from the right subtree, so we add the 24. This node covers the values $i = (1, 2, 3, 4)$, all of which we need. $s = 30, v = (1, 2, 3, 4, 5)$
4. We recurse to node 8. We arrived from the left subtree, so do nothing. As with step 2, we either have already included all values covered by this node, or they are out of range. $s = 30, v = (1, 2, 3, 4, 5)$
5. We have reached the root, so return the proper sum of 30. $9+2+8+5+6 = 30$

8.3.1.2 Updating a value

Using the same understanding of which elements a given node encompasses, we can update the value at a given index using a similar method. When we reach a node whose some should include our value, we modify it accordingly. This means that when we reach a node from the left subtree, we update (as the invariant is that the value at a node must represent the left subtree), but we do not if we reached from the right subtree. As we compute this in a single pass up the tree, it is also logarithmic in time. Here is an example modifying element 5 from 6 to 16 (an increment of 10). As before, we bold the relevant nodes and paths.



As noted, nodes which require update are all reached from the left subtree.

8.3.1.3 Building the tree

Given that we can perform updates in linear time, we can trivially construct a fenwick tree from scratch by initializing a tree of the appropriate size to all 0's, then iteratively "updating" each element of the list in turn to its actual value. The algorithm to build the above tree thus would look as follows:

1. Determine the size of the tree, which will be the next highest power of 2 greater than or equal to the size of the list
2. Construct a tree, with each node initialized to 0, with the appropriate shape, which will be a root node with a complete left subtree, and no right subtree.
3. Perform the "update" operation on index 1.
4. Perform the "update" operation on index 2.
5. etc.

8.3.2 Implementation

While one could construct the graph as described immediately above, and the construction would be performant and correct, it turns out that it is ultimately toilsome and unnecessary. To see why, lets look at which nodes end up added in order to compute the prefix sum for all elements. We will look at both the decimal and binary representations.

index	included nodes	binary index	binary included nodes
1	1	0001	0001
2	2	0010	0010
3	3,2	0011	0011,0010
4	4	0100	0100
5	5,4	0101	0101,0100
6	6,4	0110	0110,0100
7	7,6,4	0111	0111,0110,0100
8	8	1000	1000

While we may not notice a particular pattern in the decimal representation, there is a clear pattern in the binary. Namely, once we add a node, the next node we add is found by converting the least-significant 1's bit to a 0. For example, for node 7, with index representation 0111, we first include 0111, then zero out the last 1 to get 0110, then again zero out the last 1 to get 0100. If we use a flat array to store elements of the tree (remember, each node in the tree maps directly to a single index anyway), this leads to the following algorithm:

```
// assume the tree is stored in an array of size N (some power of 2)
// and indexed by node number
```

```
int ft[N+1];
int query(int e){
   int ans=0;
   for(int i=0;i<31;i++){ // iterate over all bits
      if((e&(1<<i))!=0)ans+=ft[e]; // add the value at the node if this
          bit is a 1
      e&=~(1<<i); // clear this bit
   }
   return ans;
}
```

Because we index into this array based on the binary representation of a number, it is also called a *Binary Index Tree*

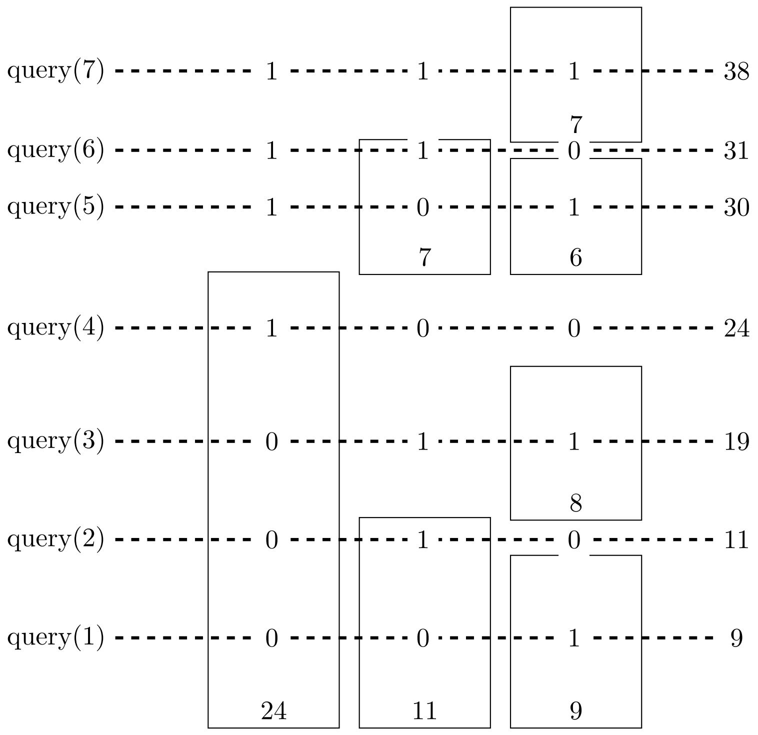


Figure 8.1: A visual representation of how a sum is built with a Fenwick tree. Each box's height is propotionally correct. The binary representatino of each query is overlaid on the dashed line, and corresponds to whether a box is added for that query from that column, which will always be the box the line intersects, or immediately below the 1.

Repeating this exercise for the updates:

index	updated nodes	binary index	binary updated nodes
1	1,2,4,8	0001	0001,0010,0100,1000
2	2,4,8	0010	0010,0100,1000
3	3,4,8	0011	0011,0100,1000
4	4,8	0100	0100,1000
5	5,6,8	0101	0101,0110,1000
6	6,8	0110	0110,1000
7	7,8	0111	0111,1000
8	8	1000	1000

The pattern here is similar to the above, except for each 0 bit greater than the smallest 1 bit, we make it a 1 and zero out the rest of the number. So:

```
void update(int e, int delta){
   ft[e]+=delta;//update ourselves
   bool started=false;
   for(int i=0;i<31;i++){ // iterate over all bits
      if(!started){
         if((e&(1<<i))!=0){//don't do anything until we find a 1 bit
            started=true;
            e&=~(1<<i); // clear this bit
         }
      }else{
         if((e&(1<<i))==0&&(e|(1<<i))<=N){
            ft[e|(1<<i)]+=delta;
         }
         e&=~(1<<i); // clear this bit
      }
   }
}
```

8.3.2.1 Bithacks

We can compress the provded implementations even further using a series of bithacks. In the query case, we note that the following trick ”zeros out” the least significant one bit `e&=e-1`, allowing us to directly compute the next node we must include.

In the update case, if we translate to 0 indexing, the update table looks as follows (Note, we have simply subtracted 1 from all values):

index	updated nodes	binary index	binary updated nodes
0	0,1,3,7	0000	0000,0001,0011,0111
1	1,3,7	0001	0001,0011,0111
2	2,3,7	0010	0010,0011,0111
3	3,7	0011	0011,0111
4	4,5,7	0100	0100,0101,0111
5	5,7	0101	0101,0111
6	6,7	0110	0110,0111
7	7	0111	0111

This has a much easier and apparent pattern than what is discussed above, namely that we just keep filling in 1's from the right side of the number. In this case, the following trick allows us to directly ”one out” the least significant 0 bit `e|=e+1`, allowing us to directly compute the next node we must update. Note that as we used 0-indexing here, but 1 indexing for the query case, we must adjust by 1 when we index into the array itself. Whether we actually store the array as 1 or 0 indexed is an implementation detail.

A full implementation including said hacks is as follows:

```
//Size to some power of 2
#define N (1<<20)
int ft[N+1];
int query(int e){
    int ans=0;
    while(e){
        ans+=ft[e];
        e&=e-1;
    }
    return ans;
}
void update(int e, int delta){
    e--;
    while(e<N){
        ft[e+1]+=delta;
        e|=e+1;
    }
}
```

8.4 Trie

Chapter 9

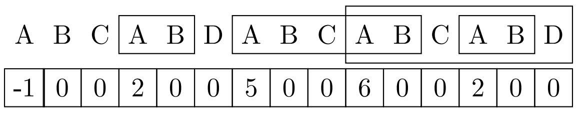
Strings

9.1 Pattern Matching

9.1.1 *Z* Function

The *Z function*[1] of a given index i in a string is the longest substring starting at i which is a prefix of the string itself. More simply, if $x = Z(i)$, then the first x characters starting at i are identical to beginning of the string, but no more.[2]

Let's consider the Z function computed over the string *ABCABDABCABCABD*. Boxes are drawn arond the substrings matching the Z value, which all match the beginning of the string exactly. These are known as *Z boxes.*



A	B	C	A	B	D	A	B	C	A	B	C	A	B	D
-1	0	0	2	0	0	5	0	0	6	0	0	2	0	0

9.1.1.1 Applications

While the utility of such a function may seem esoteric, it is useful both for a few specific applications, some fundamental concepts it exposes, and the simplicity of its implementation, which we will see shortly.

- Finding the longest common extension of any index with the start of the string. This is the definition of the Z function.

- Finding the longest prefix of a second string starting at any index in a given string. This is accomplished by concatenating the two strings with a terminal character which appears in neither of the other strings. More clearly, if we wish to find the longest prefix of a string T which occurs

[1]As described in Dan Gusfield's book *Algorithms on Strings, Trees, and Sequences* as *fundamental preprocessing* of a string.

[2]This is exactly the longest common extension of 0 and i in the string which is discussed more generally later in this chapter.

at any index in another string S, we could create a new string $T\$S$, and compute the Z function over that new string. Note that the \$ character ensures no Z value exceeds the length of T, since no character in S can match the \$.

- Finding all occurrences of a string T in S. Similar to the above, we concatenate with a separator \$ and look for any Z value which is exactly the length of T.

9.1.1.2 Implementation

Trivially, we could compute the Z values in $O(n^2)$ time using a brute force approach. However, with some clever rules, we can reduce this. From a high level, we will compute the Z values in order, and remember some infromation about previously found Z boxes that enable us to quickly compute future Z values using the following rules.

1. Rule 1: If the current index is not found in any known Z box, compute the Z value directly. If we find a Z box, note its left and right endpoints. This rule would cause us to find the Z boxes as indices 3 and 6 in the earlier string.

2. Rule 2: If we are in a Z box, we know that the Z box is duplicated exactly at the start of the string. Look up the Z value at the corresponding index in the beginning of the string.

   (a) Rule 2.1: If the Z value at this corresponding index would keep us within our current Z box, take it as the Z value for the current index. We see this rule apply at index 12.

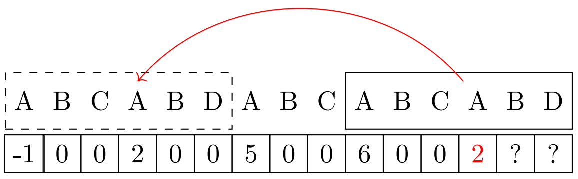


Figure 9.1: The Z value at the corresponding A is 2, so we remain inside the Z box. This means the later A also has a Z value of 2. Note that this must be true, as we know the solid and dashed boxes must be exact copies of eachother, so we could not have a larger Z value in the later Z box than we had in the beginning, so long as the Z value doesn't extend past the bounds of that box.

   (b) Rule 2.2: If the Z value at this corresponding index extends to or past the bounds of the Z box, we cannot rely on 2.1, since more characters might match. We use brute force to examine further characters past the current Z box to find the first mismatch. We then update the current Z box to the one further to the right, since that encapsulates the most current information we have. We see this rule apply at index 9.

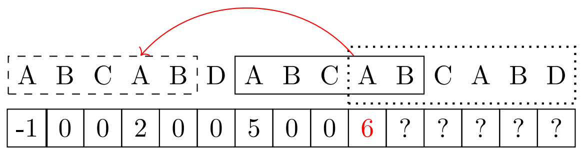


Figure 9.2: The Z value at the corresponding A is 2, but this aligns with the end of the current Z box, so there could be further characters which match. We perform this comparison at the C following the solid Z box, and the corresponding character at the beginning of the string, the C at index 2. We find we match an additional 4 characters, making our total Z value $2+4=6$. We record the new value, and move to the newer, dotted, Z box for all future lookups.

We note that by using these rules, we always track the Z box with the rightmost endpoint that we know about so far. More critically, we note that we only perform a comparsion within a Z box a single time, when we are creating that Z box, and a comparison against a given letter in the string either is part of the creation of a Z box (so that it is never compared against again), or it causes us to complete the computation of the Z value for a given index. This guarantees us $O(n)$ overall time to compute the Z array.

A compact implementation could be as follows.

Listing 9.1: C++

```
vector<int> zfunc(string s){
    vector<int> z(s.length());
    int l=-1,r=-1; // current z-box
    for(int i=1;i<s.length();i++){
        if(r==-1){ // Rule 1
            while(s[z[i]]==s[i+z[i]])z[i]++; // abuse z[i] as our pointer.
            l=i;
            r=l+z[i];
        }else{
            z[i]=z[i-l]; // Rule 2
            if(z[i]>=r-i) {// Special handling for Rule 2.2
                while(s[z[i]]==s[i+z[i]])z[i]++;
                l=i;
                r=l+z[i];
            }
        }
    }
    return z;
}
```

9.1.2 Knuth-Morris-Pratt

Knuth-Morris-Pratt (KMP) is one of the more well-known algorithms for identifying instances of a fixed pattern in a string. Most languages have a built-in, highly-performant way to perform this operation. Further, other algorithms are simpler (such as the above Z-func) or more performant (such as Boyer-Moore). So why is KMP presented here? The method used for KMP is very powerful, and while string matching cna be done in other ways, the method backing KMP can be extended to solve more difficult problems (such as matching multiple patterns). For this reason, it is essential to understand KMP so that the more complex algorithms can be understood.

9.1.2.1 Intuition

First, observe the naive way in which we might match a pattern to a string. We consider the pattern ABABC, and an text ABCABABABC. We'll attempt to align the pattern and the text at every position, and check if there is a match:

Listing 9.2: C++

```
void naive_match(string t,string p){
   for(int i=0;i<=t.length()-p.length();i++){ //all alignments of p in t
      bool match=true;
      for(int j=0;j<p.length();j++){ //check all characters
         if(p[j]!=t[i+j]){
            match=false;
            break;
         }
      }
      if(match)cout<<i;
   }
}
```

We can observe how the algorithm progresses. The failed comparison in each loop is highlighted.

$$\begin{array}{cc|c|ccccccc} \mathrm{A} & \mathrm{B} & \mathrm{C} & \mathrm{A} & \mathrm{B} & \mathrm{A} & \mathrm{B} & \mathrm{A} & \mathrm{B} & \mathrm{C} \\ \mathrm{A} & \mathrm{B} & \mathrm{A} & \mathrm{B} & \mathrm{C} & & & & & \end{array}$$

Figure 9.3: `i==0`

$$\begin{array}{c|c|cccccccc} \mathrm{A} & \mathrm{B} & \mathrm{C} & \mathrm{A} & \mathrm{B} & \mathrm{A} & \mathrm{B} & \mathrm{A} & \mathrm{B} & \mathrm{C} \\ & \mathrm{A} & \mathrm{B} & \mathrm{A} & \mathrm{B} & \mathrm{C} & & & & \end{array}$$

Figure 9.4: `i==1`

$$\begin{array}{cccccccccc} A & B & \boxed{C} & A & B & A & B & A & B & C \\ & & \boxed{A} & B & A & B & C & & & \end{array}$$

Figure 9.5: `i==2`

$$\begin{array}{cccccccccc} A & B & C & A & B & A & B & \boxed{A} & B & C \\ & & & A & B & A & B & \boxed{C} & & \end{array}$$

Figure 9.6: `i==3`

$$\begin{array}{cccccccccc} A & B & C & A & \boxed{B} & A & B & A & B & C \\ & & & & \boxed{A} & B & A & B & C & \end{array}$$

Figure 9.7: `i==4`

$$\begin{array}{cccccccccc} A & B & C & A & B & A & B & A & B & C \\ & & & & & A & B & A & B & C \end{array}$$

Figure 9.8: `i==5`, a match is found

Examining alignment 0, we successfully compared two characters before we failed and mvoed to alignment 1 and performing a further comparison. The second character compared at alignment 0, or in any alignment where we match at least two characters, is B. More importantly, that second character is **not** the first character in the pattern, A. Therefore we can conclude if we ever match the first two characters in an alignment, the pattern will always fail in the subsequent alignment. We see this when `i=0`, implying we cannot match at alignment 1, and we also see this when `i=3`, implying we cannot match at alignment 4.

The fundamental intuition behind KMP is that we can leverage information from comparisons in previous alignments to ”skip” future alignments. By skipping enough alignments and efficiently computing which alignments can be skipped, we can achieve linear overall time.

The algorithm occurs in two large steps:

- Pre-process the pattern, producing a table mapping the location of a comparison failure to the next possible alignment which could potentially match. For instance, this table would map a failure on the second character in the pattern to the alignment two ahead.[3]

[3]Note early on that a failure on the k-th character does not imply we will always move k ahead. This is demonstrated by a failure on the last character of the example pattern, which only allows us to skip two ahead.

- Align the pattern with the first position in the target text and perform a character by character comparison. When that comparison is complete,[4] use the table to determine the next alignment and where in the alignment to resume comparison.

9.1.2.2 Preprocessing: Defining the Overlap Function

The preprocessing table indicates how far we can shift the pattern depending on how far into the pattern a miscompare occurs. We want this shift to be as far as possible (to limit comparisons) while also ensuring we do not miss any potential matches. Our goal will be to ensure once we have successfully compared a character in the text, to never compare that character again.[5]

In order to meet the last criterion, the following must be true. Consider only the part of the pattern which matched. After the shift, the overlapping parts of the pattern must match. If there is a chance they do not match, then we would have to compare characters with our text which we had previously successfully compared.[6]

Consider the pattern ABCDABCE, and a text ABCDABCXXX. We show the first alignment, and the next possible alignment, showing that the overlapping parts of the pattern must match. Note that such an overlap does not guarantee the pattern will match at that alignment; it is a necessary but not sufficient criterion.

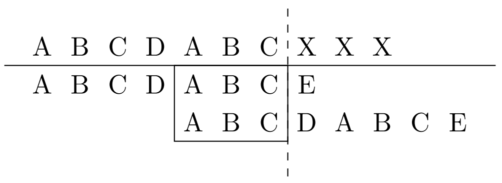


Figure 9.9: We successfully compared the first 7 letters of the pattern and text with a dashed line indicating the end of that succsessful comparison. In order to ensure we never have to compare any of those 7 letters of the text again, we must guarantee the overlapping parts of any future alignment (up to the comparison failure) are equal. We show the next future alignment with a box around that overlap. Any lesser shift results in miscompares to the left of the dashed line. Note that even though the last character of the pattern also overlaps after the shift, it is not considered as it is past the dashed line of furthest successful comparison, and thus does not need to match in order to fulfil the criterion.

We can state the overlap-match more formally:

> Let l be the maximum match of the patern at some index, and $P_{0,l}$ be the sub-pattern representing that match. If $P_{0,l}$ is shifted by

[4] either with a successful match or a miscompare

[5] This will ultimately enable us to prove linear bound.

[6] violating the criterion we are seeking to meet

k and compared against $P_{0,l}$ itself, then the overlap matches only if $P_{0,l-k}$ is the same as $P_{k,l}$. The suffix of the sub-pattern must be a prefix.

In order to ensure we do not accidentally skip a potential match, we will seek to find the minimum shift, therefore the maximum amount of overlap, and ultimately the longest suffix of the sub-pattern which is also a prefix.[7] The size of this longest suffix which is also a prefix will be known as the *overlap* function. This overlap function will be used to compute the minimum shift such that all characters of the text we have previously compared still match, and that no matches have been skipped.

We can see the values for two examples.

ABCDABCE		
Matched Substring	Overlap	Explanation
∅	∅ (0)	If no characters match, by defition we will shift by 1.
A	∅ (0)	There is no suffix of A which is a prefix. We can skip all overlapping shifts.
AB	∅ (0)	There is no suffix of AB which is a prefix. We can skip all overlapping shifts.
ABC	∅ (0)	There is no suffix of ABC which is a prefix. We can skip all overlapping shifts.
ABCD	∅ (0)	There is no suffix of ABCD which is a prefix. We can skip all overlapping shifts.
ABCDA	A (1)	The longest suffix which is also a prefix is A. We can shift by 4 to reach that matching overlap.
ABCDAB	AB (2)	The longest suffix which is also a prefix is AB. We can shift by 4 to reach that matching overlap.
ABCDABC	ABC (3)	The longest suffix which is also a prefix is ABC We can shift by 4 to reach that matching overlap.
ABCDABCE	∅ (8)	There is no suffix which is a prefix. We can skip all overlapping shifts.

[7]but shorter than the entire sub-pattern,which would result in no shift!

ABCABABABC		
Matched Substring	Overlap	Explanation
∅	∅ (0)	If no characters match, by defition we will shift by 1.
A	∅ (0)	There is no suffix of A which is a prefix. We can skip all overlapping shifts.
AB	∅ (0)	There is no suffix of AB which is a prefix. We can skip all overlapping shifts.
ABC	∅ (0)	There is no suffix of ABC which is a prefix. We can skip all overlapping shifts.
ABCA	A (1)	The longest suffix which is also a prefix is A. We can shift by 3 to reach that matching overlap.
ABCAB	AB (2)	The longest suffix which is also a prefix is AB. We can shift by 3 to reach that matching overlap.
ABCABA	A (1)	The longest suffix which is also a prefix is A. We can shift by 5 to reach that matching overlap.
ABCABAB	AB (2)	The longest suffix which is also a prefix is AB We can shift by 5 to reach that matching overlap.
ABCABABA	A (1)	The longest suffix which is also a prefix is A We can shift by 7 to reach that matching overlap.
ABCABABAB	AB (2)	The longest suffix which is also a prefix is AB We can shift by 7 to reach that matching overlap.
ABCABABABC	ABC (3)	The longest suffix which is also a prefix is ABC We can shift by 7 to reach that matching overlap.

9.1.2.3 Preprocessing: Computing the Overlap Function

As the goal of the algorithm is linear time, we must be able to compute the overlap function in linear time as opposed to the naive quadratic implementation. While it should be apparent that one can compute the overlap from the Z-func,[8] our goal is not simplicity but comprehension. The alternative[9] method shown here better extends to further algorithms such as Aho-Corasick and therefore its presentation is a pre-requisite for understanding such.

[8]The overlap is equivalent to the Z-box with the left most endpoint which covers a given letter. We can extract all the necessary values by caterpillaring through the pattern, alternatively iterating until the right end of the current Z-box is found, and then iterating until the next Z-box left endpoint is found.

[9]actually the original

From a high level, we will compute the overlaps from left to right. To compute $Olap(i+1)$, we will examine all suffixes which end at i which have a corresponding overlappling prefix in decreasing order of length, and identify the first[10] one which has a matching character following the two instances.

We make this far clearer with a picture where we attempt to compute $Olap(i+1)$ from $Olap(i)$ and all other lesser values.

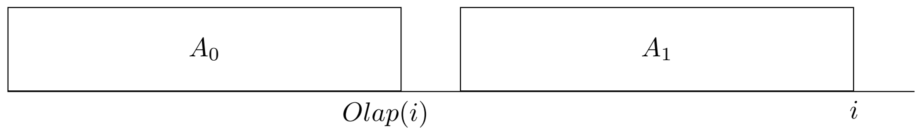


Figure 9.10: Consider $Olap(i)$. We see the overlap itself A appears in totality at both the beginning and end of the substring.

We will consider A as our first candidate and compare succeeding letters.

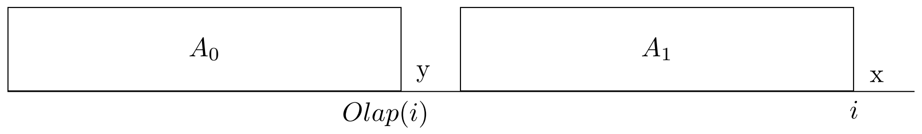


Figure 9.11: The letters after A_0 and A_1 do not match, and therefore we cannot simply extend A by 1. If those two letters had matched, we would simply set $Olap(i+1) = Olap(i) + 1$ and increment to solve for the next i.

As our current candidate failed, we must find the next longest candidate overlap.

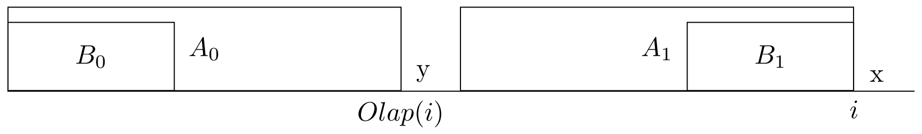


Figure 9.12: The substring B represents the next longest overlap which ends at i and occurs at the beginning of the string.

We know we have to identify B, but how can we do so efficiently? Incrementing through each suffix in A and checking whether that substring is also a prefix is too slow. The key lies in the fact that as we know that the two A boxes are exact copies of eachother, there are actually two other instances of B.

[10]and therefore longest

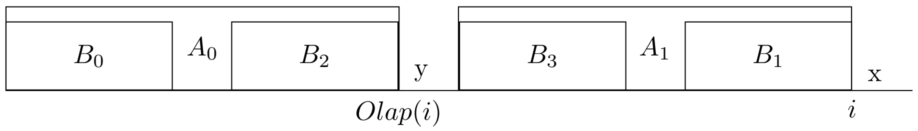


Figure 9.13: Since the two A boxes are identical, there must be at least 4 copies of B in the string: at the bounds of each A box. Note: The subscripts on the label for each box do not carry semantic meaning, and are simply for identification.

How can we use this information to quickly find B? If we look strictly at A_0, we note that there is a copy of B at both the start and end. Is there a way to look up the longest string which is both a suffix and prefix of A? Of course! It is exactly the $Olap$ function we have computed at the right endpoint of A. As the right endpoint of A is $Olap(i)$, the size of B must be $Olap(Olap(i))$ or $Olap^2(i)$.

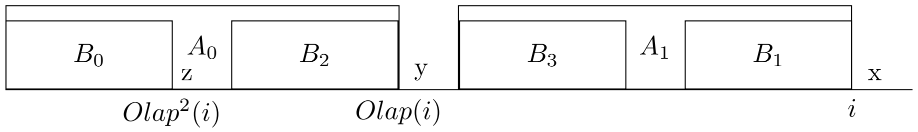


Figure 9.14: Iteratively applying the $Olap$ function enabled us to find the length of the next-longest suffix ending at i which is also a prefix. We again see that there is no match.

The fun doesn't stop there. Our next-longest overlap which is shorter than B occurs at least 8 times in the string, twice in each B-box. For clarity, we only draw the relevant instance.

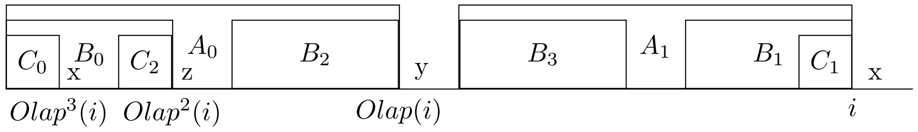


Figure 9.15: We iteratively apply $Olap$ to find the next longest suffix ending at i which is also a prefix has length $Olap^3(i)$. Examining the succeeding characteres of C_0 and C_1 shows they are both the same. We can therefore set $Olap(i+1) = Olap^3(i) + 1$.

This leaves the following overall algorithm to compute $Olap(i + 1)$ from all lesser values of i:

1. Chose l as the longest suffix ending at i which is also a prefix by looking up $Olap(i)$.
2. Examine if the character at $i + 1$ is the same as the character at $l + 1$.

  - If so, set $Olap(i+1) = Olap(i) + 1$ and return.

3. Find the next longest suffix ending at i which is also a prefix by applying $l = Olap(l)$.

4. Loop to step 2, or exit with a base case if there are no more suffixes to examine.

Implementation

Listing 9.3: C++

```
//assume a pattern, p.
//olap is the maximum overlap of a substring of length i.
int olap[p.size()+1]={};

// - start with box from previous index
// - recurse until the next char matches or we hit a base case
// - save box if we found one which matched
for(int i=2;i<=p.size();i++){
   int box=olap[i-1];
   while(box&&p[i-1]!=p[box])box=olap[box];
   olap[i]=(p[box]==p[i-1])?box+1:0;
}
```

Rough Proof of Runtime We iterate through all values of i, which is at best linear time. It would seem that the recursives step in the inner loop would yield a runtime which is superlinear. We can bound the number of times we perform this inner loop as follows:

- Consider the value of $Olap(i)$ and how it changes as we iterate through i. As its value increase by at most 1 from one value of i to the next, the total amount it increases over the course of the function is at most n.

- Each call to $Olap$ for a given value of i decreases the value of the overlap function for the subsequent i. Specifically, each call decreases the maximum possible value of the subsequent $Olap$ by at least 1. If 3 calls are made, the value decreases by at least 1.[11]

- As the value of $Olap$ only increases by n and is never negative, the decreases can also not exceed n. Any more than 2 calls to $Olap$ for a given i incrementally decreases the value. Therefore the total number of calls to $Olap$ cannot exceed $3n$, which is $O(n)$.

[11]If $Olap$ is called once for a given value of i, then $Olap$ increaeses by 1. If it is called twice, the $Olap$ increases by at most 0. If it is called 3 times, $Olap$ decreaes by at least 1.

9.1.2.4 Alignment

Once the overlap function is known for all i, we can use it to compute all matches. This is largely the before described process which we formalize here.

1. Align the pattern at index 0.
2. Iterate through the pattern, starting at the index of the last successful comparison, comparing character to character
   - If we reach the end of the pattern, indicate a match
3. Based on the last successfully compared character in the pattern, i, shift to the next alignment, which is $i - Olap(i)$, and begin comparison after the last successfully compared character in the text.

We can use our original naive match as an example of the improved method.[12]

$$\begin{array}{cc|c|ccccccc} A & B & C & A & B & A & B & A & B & C \\ A & B & A & B & C & & & & & \end{array}$$

Figure 9.16: `i==0`. Comparison fails after 2 characters in p. $Olap(2) = 0$, so we shift by $2 - Olap(2) = 2$ and begin comparison at the first unsuccessfully compared character, the C.

$$\begin{array}{cc|c|ccccccc} A & B & C & A & B & A & B & A & B & C \\ & & A & B & A & B & C & & & \end{array}$$

Figure 9.17: `i==2`. Comparison fails on the first character. By definition, we will shift by 1 and must begin comparison on the next character.

$$\begin{array}{ccccccc|c|cc} A & B & C & A & B & A & B & A & B & C \\ & & & A & B & A & B & C & & \end{array}$$

Figure 9.18: `i==3`. We successfully compared the first 4 characters of p before reaching a failure. $Olap(4) = 2$, so we shift by $4 - Olap(4) = 2$, and begin comparison at the first unsuccessfully compared character, the A. Note that we will never compare any of the 4 successfully compared characters of the text again.

[12]While a reader should be able to compute it, the values for *Olap* in this pattern are 0,0,0,1,2,0.

```
A B C A B A B A B C
          A B A B C
```

Figure 9.19: `i==5`. We began comparison at the last A, and matched the remaining ABC, finding a match. Note that we only had to compare 3 total characters in this step, as we previously compared the first two when i was 3. If the text were longer, we would shift by $5 - Olap(5)$ before continuing comparison at the character after the C.

Rough Proof of Linearity Each character in the text is only compared successfully once, for a total of $O(n)$ successful comparisons. On a failed comparison, the pattern is shifted to a new alignment. As the number of alignments is $O(n-p)$, this limits the total number of failed comparisons to $O(n)$ as well. The overlap function is also computed in linear time, leading to an overall runtime of $O(n)$.

9.1.2.5 Implementation

The implementation, while straightforward, is finicky due to the potential for off by one errors and other corner cases. For simplicitly, instead of explicitly ”shifting” the pattern and aligning to the text, we track the current pointer into both the text and the pattern, and adjust those pointers as necessary.

Listing 9.4: C++

```cpp
//assume a text, t, and a pattern, p.

//preprocess
int olap[p.size()+1]={};
for(int i=2;i<=p.size();i++){
   int box=olap[i-1];
   while(box&&p[i-1]!=p[box])box=olap[box];
   olap[i]=(p[box]==p[i-1])?box+1:0;
}

//align
//the outer loop moves i through t one char at a time
//inside the loop, we shift p until it matches at i, or hit a base case
//if we have gotten j all the way through p, it indicates a match, so
//we have to shift to set up for next match
for(int i=0,j=0;i<t.size();i++){
   while(j&&t[i]!=p[j])j=olap[j]; //shift
   if(t[i]==p[j])j++; //advance in p
   if(j==p.size()){ //match found!
      cout<<i+1-p.size()<<"\n";
      j=olap[j];
   }
}
```

9.1.3 Multi-pattern Matching/Aho-Corasick

With KMP, we were able to optimize matching by iteratively aligning a pattern with a text, and then optimizing by only checking certain alignments. *Aho-Corasick* allows us to search for multiple patterns at once using similar intuition.

Naively, if we were to run KMP on each of P patterns, we would find we take $O(nP)$ time, as we need to interate through the text independently for each pattern. We will see how this time can be reduced in the following high level stages.

1. Match all patterns in a single pass of the text
2. Optimize the matching to yield a runtime proportional to $O(n+p)$, where n is the length of the text, and p is the total length of all patterns
3. Address the corner case where one pattern is a substring contained in another

9.1.3.1 Single-Pass Matching

When we first introduced KMP, we saw in section **??** that we could naively attempt to align a pattern at each index in the text. We can modify this implementation to cope with multiple patterns in a single pass of the text using a trie, and aligning each index in the text with the root of the trie. Once there aligned, we will walk the trie to see if we reach the end node of some pattern.

We will walk through an example of this using the following words:[13]

- booboo
- booster
- oboe

The trie containing these three words appears as follows:

[13]Note that none of these are substrings of any other, a point which will be addressed in section **??**

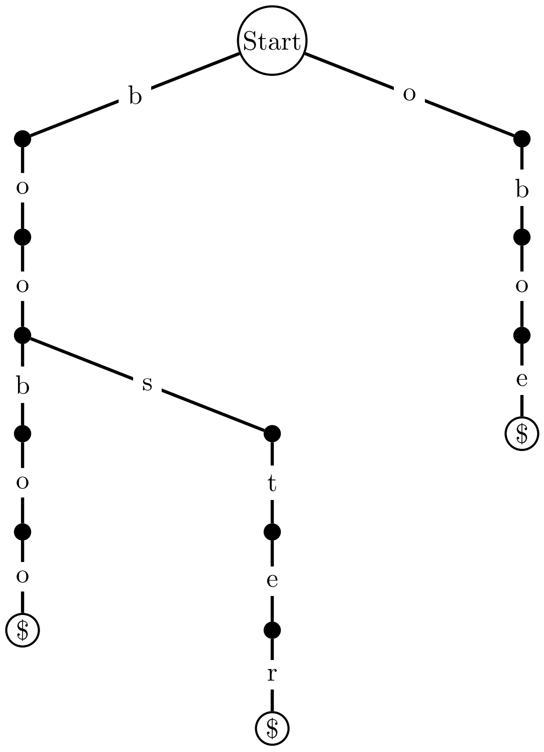


Figure 9.20: Nodes where a pattern terminates are marked with a '$'.

Consider matching these words against the text "obeobooboe". We attempt to walk the trie starting at each index into the text.

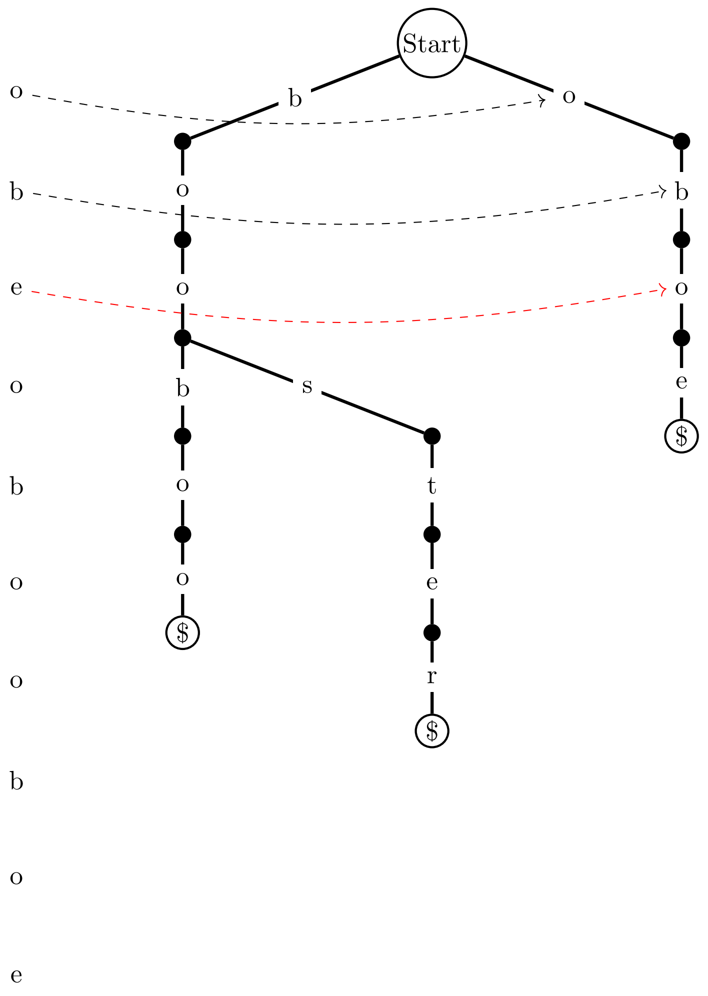


Figure 9.21: We walk the trie from the root. Upon comparing the third character, we find there is no outgoing edge where we can continue to match against. This is indicated by the red arrow, where the only outgoing edge from our current node is 'o', but the text is 'e'. There is no pattern that matches at this alignment.

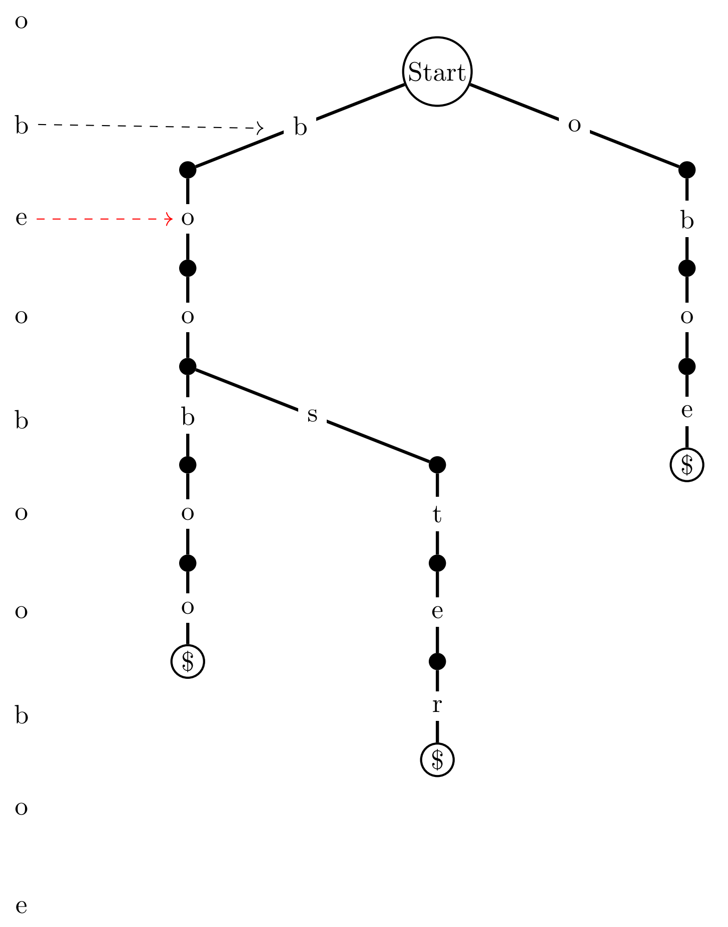


Figure 9.22: At the next alignment, we again fail to find a match.

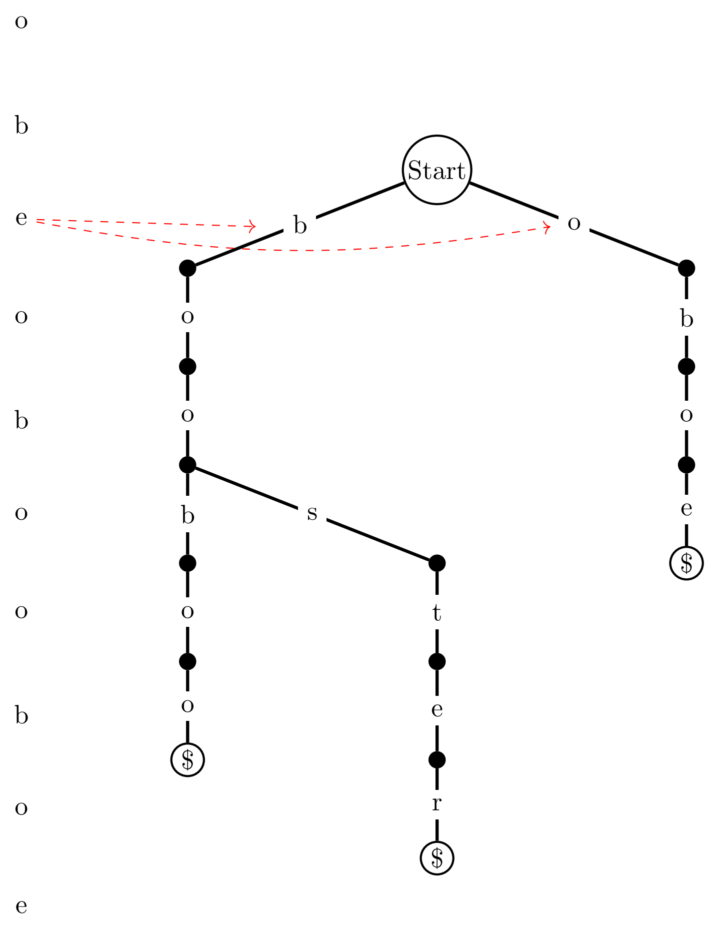


Figure 9.23: Checking the next alignment doesn't get far. This character matches none of the edges emenating from the current node.

We can continue checking alignments, and see that all of them fail until we reach one at the end:

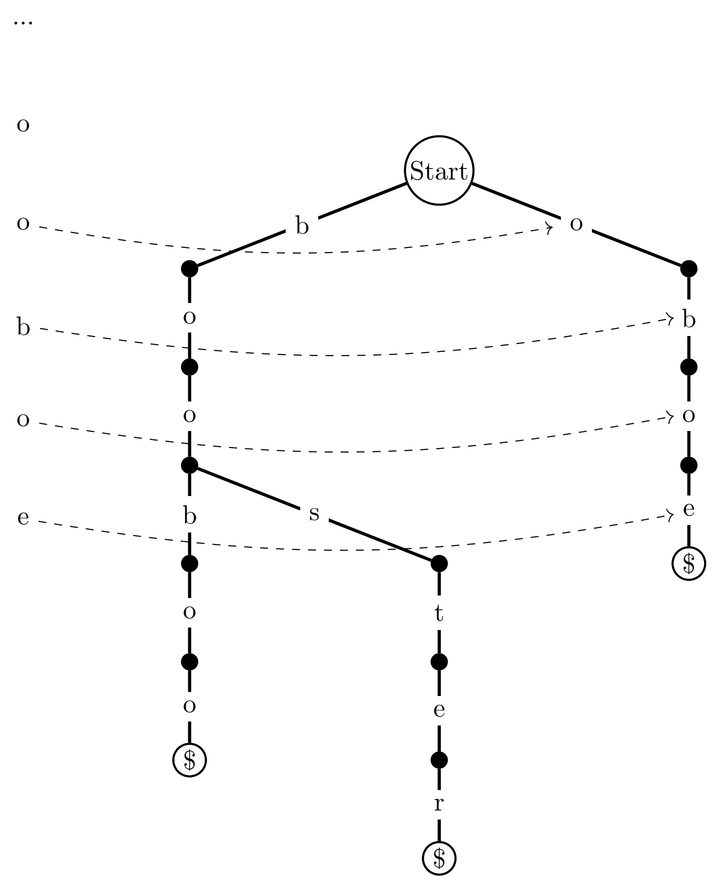


Figure 9.24: At the final alignment, we reach the end node of some pattern, indicating the text matches the pattern at this alignment.

After trying all possible alignments against the tree, we have found one alignment which match some pattern in the trie. These constitute all matches of the three patterns to the text. As we have tried each of n alignments, and each alignment requires $O(n)$ time to walk the tree, our runtime is $O(n^2)$.[14]

9.1.3.2 Preprocessing: Defining Overlap Edges

If we examine the alignments starting at index 1 (*be...*) and 2 (*eo...*), we notice we can optimize as we did with KMP. The failed character (e) does not appear at the start of any pattern, and therefore the next alignment (which starts with e) cannot match any pattern, and can be skipped. To achieve this with KMP, we defined the *Olap* function which provides the longest suffix of the pattern where our comparison last succeeded which is also a prefix of the pattern.[15]

[14]It is also $O(np)$.

[15]See the KMP section for further details on the intuition behind this.

After shifting by *Olap*, we were able to resume comparison at the same position in the text.

We can apply similar logic to our trie in the following manner:

1. Compute the longest prefix of any pattern which is a suffix of the pattern we were matching when comparison failed

2. Identify where in the trie to resume comparison to ensure we don't successfully compare a character of the text more than onces[16]

With KMP, the second condition was met trivially. When we shift by the *Olap*, we know where to begin comparison again by simple subtraction.[17]. This is more difficult with a trie, as after the shift, we don't necessarily know which pattern or patterns we should continue matching against, or put another way, which branch of the trie we are in. To solve this purpose, we will extend the *Olap* function to return instead of a length, a node in the trie. From each node in the trie, if a comparison fails, instead of subtracting the *Olap*, we will traverse this special *Olap* edge and resume comparison.[18]

The edge is formally defined as follows.

> Consider the sequence of letters required to traverse from the root to some node as that nodes label. Consider all suffixes of this label, and further, such suffixes which themselves are the label of some other node in the trie. The Olap edge is the edge from the former node to the node which is labeled by the longest such suffix.

Each node has an Olap edge (the degenerate case points back to the root), and it is unique.[19] If suffix edges were drawn on a trie which only had a single pattern, the edges would traverse exactly from a node representing character i to the node representing character $Olap(i)$, as defined by KMP. In KMP, we resume comparison with the character immediately after that defined by the *Olap*, and here, we resume comparison with the characters immediately after the node defined by the Olap edge.

Our trie is redrawn here with all Olap edges shown.

[16]As with KMP, achieving linear time requires such a strict condition

[17]See the Alignment section

[18]This edge is called various things in various sources, such as "Suffix edge" or "failure links". We choose our terminology to demonstrate the relationship to KMP.

[19]Every string has at most one node representing it in the trie, and all considered suffixes have distinct length.

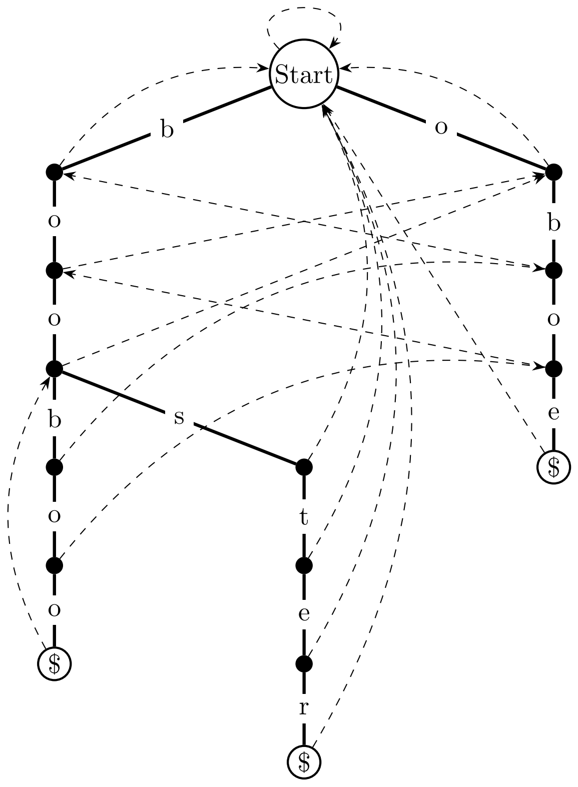


Figure 9.25: Nodes with their Olap edges drawn in dashed lines. Each node has an edge to the node such that the overlap of the destination nodes label and the suffix of the label of the source node is maximal.

The visualization of the Olap edges is very cluttered, so let's view a description of each such edge in table form:

Label	Overlap Label	Explanation
$\emptyset$	$\emptyset$	If no characters matched, and therefore we are still at the start node, we will shift by one and continue comparison at the root.
b	$\emptyset$	There is no suffix of b in the trie. We resume comparison at the root.
bo	o	The longest suffix of bo which is the prefix of some pattern is o. We resume comparison matching oboe.
boo	o	The longest suffix of boo which is the prefix of some pattern is o. We resume comparison matching oboe.
boob	ob	The longest suffix of boob which is the prefix of some pattern is ob. We resume comparison matching oboe.
boobo	obo	The longest suffix of boobo which is the prefix of some pattern is obo. We resume comparison matching oboe.
booboo	boo	The longest suffix of booboo which is the prefix of some pattern is boo. We resume comparison matching booboo or booster.
boos	$\emptyset$	There is no suffix of boos in the trie. We skip all overlapping shifts and resume at the root.
boost	$\emptyset$	There is no suffix of boost in the trie. We skip all overlapping shifts and resume at the root.
booste	$\emptyset$	There is no suffix of booste in the trie. We skip all overlapping shifts and resume at the root.
booster	$\emptyset$	There is no suffix of booster in the trie. We skip all overlapping shifts and resume at the root.
o	$\emptyset$	There is no suffix of o in the trie. We resume comparison at the root.
ob	b	The longest suffix of ob which is the prefix of some pattern is b. We resume comparison matching booboo or booster.
obo	bo	The longest suffix of obo which is the prefix of some pattern is bo. We resume comparison matching booboo or booster.
oboe	$\emptyset$	There is no suffix of oboe in the trie. We skip all overlapping shifts and resume at the root.

The table descriptions are naturally very similar to those presented for KMP. The major difference is instead of a raw length, we produce a node itself. The number of alignments which end up ”skipped” is implied by the difference in length between the label and overlap label.

9.1.3.3 Preprocessing: Computing Overlap Edges

While it may seem difficult to compute these edges, the logic by which we compute the *Olap* function for KMP applies almost directly.

The following algorithm computes $Olap_{edge}(n_c)$ for some node n_c, representing a node n's child, following an edge labeled with character c, assuming $Olap_{edge}$ has been computed for n and all its direct ancestors.

1. Choose p as the node whose label contains the longest suffix of n by looking up $Olap_{edge}(n)$.
2. Examine if p has an outgoing edge c, pointing at some p_c.
    - If so, set $Olap_{edge}(n_c)$ to p_c and return.
3. Find the node whose label contains the next longest suffix of n by applying $p = Olap_{edge}(p)$.
4. Loop to step 2, or exit with a base case if there are no more suffixes to examine.

The proof of correctness is nearly identical to the justification of the KMP algorithm. At each step, the label of p is the longest suffix which terminates at n, and therefore the longest suffix which terminates at p and exists in the trie must be exactly the next longest such suffix of n.[20]

Similarly, the proof of linearlity follows from that of KMP.

1. Consider the depth of the node pointed to by $Olap_{edge}$ as we progress down a branch of the trie. It increases by 1 from one node to the next. Therefore the total amount it increases over the progression from the root to some leaf node is at most the depth of that leaf.
2. Each recursive call to $Olap_{edge}$ for a given value of n_c reduces the depth of the candidate p_c by at least 1.
3. As depth cannot be negative, and the total increase for any branch is linear, the total number of calls to $Olap_{edge}$ to compute that function for the entire branch is also linear. Therefore, the total number of calls to $Olap_{edge}$ for the entire tree must be $O(p)$.

[20]The reader is encouraged to review the computation of the *Olap* function for KMP if this is not fully understood. The boxes there-presented apply equally to branches of the trie as they do to a single pattern.

9.1.3.4 Alignment

Once the overlap edge is known for all nodes, we can use it to compute all matches. This is done simply by comparing each character of the text in turn with our current location in the trie. If an edge exists, we follow it, otherwise we follow the $Olap_{edge}$. The alignment is implicitly defined by the depth in the trie and the charatcter we are currently comparing.[21]

We return to our example.

[21] Recall that our goal is to only successfully compare each character once, so we can maintain an index of that character to imply the alignment without actually storing it.

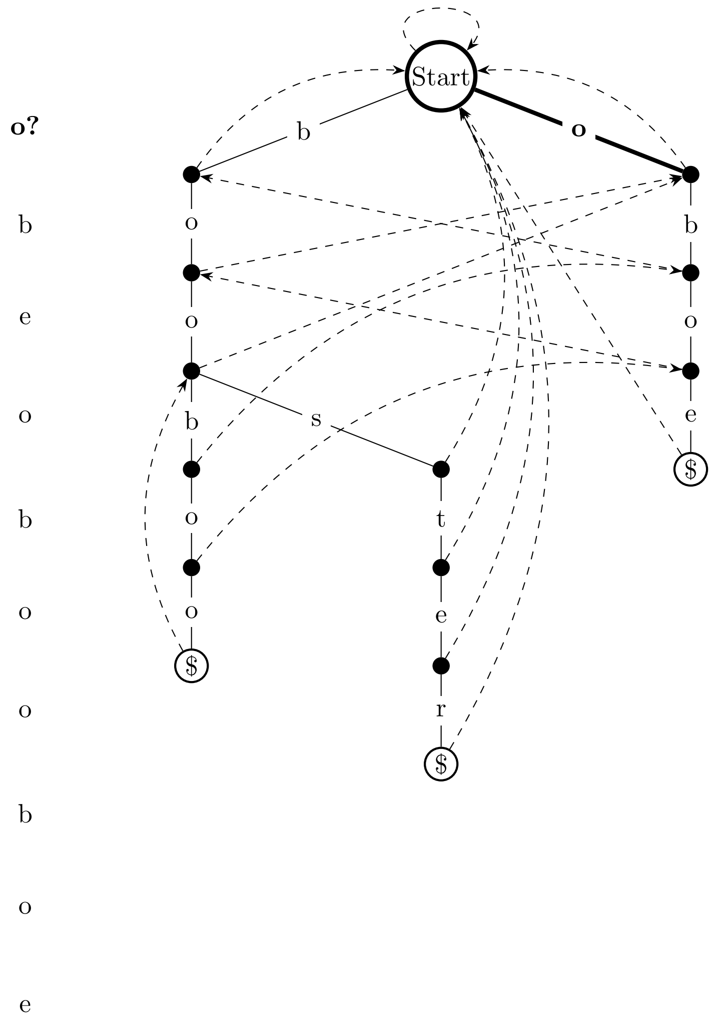


Figure 9.26: In the first alignment, and starting at the root, we match an outgoing edge o.

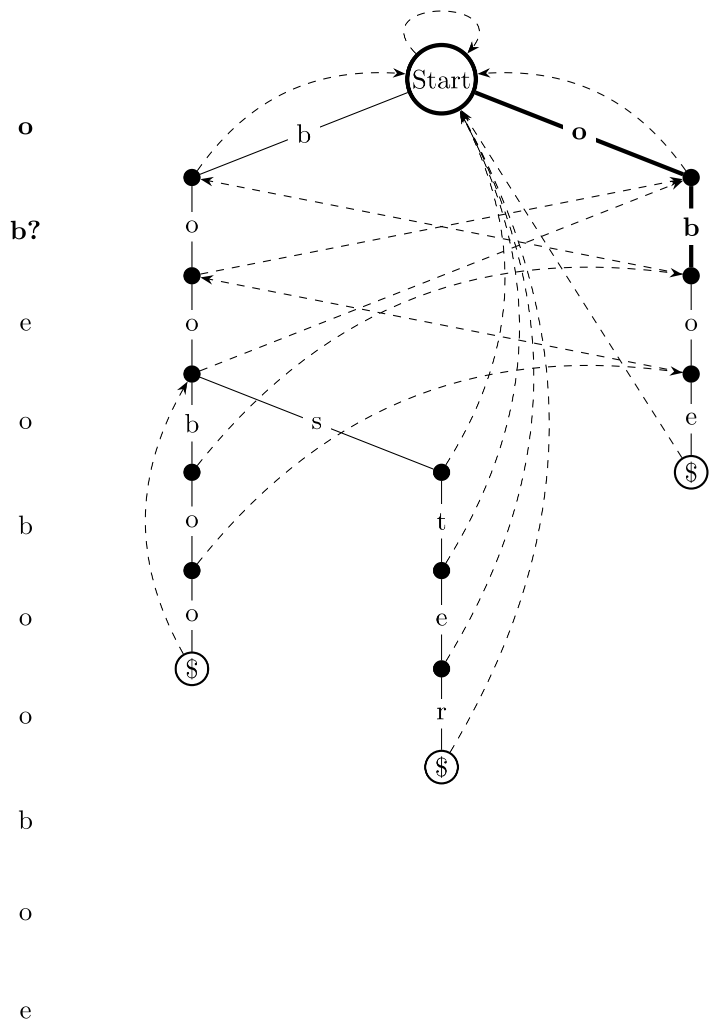


Figure 9.27: Continuing from the o node, we match an outgoing edge b.

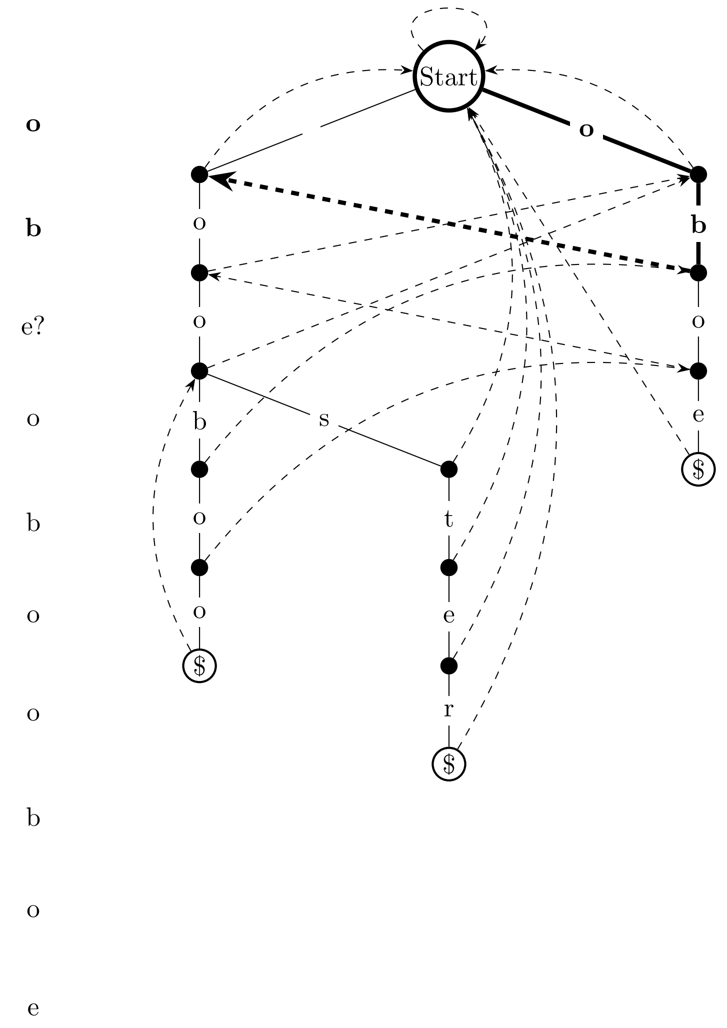


Figure 9.28: Continuing from the ob node, we do not find a match for e, so we follow the overlap edge.

Note that as we have followed an overlap edge, the depth of the node we are currently at has decreased. As we are still comparing the same character in the text, as indicated by the question mart, the alignment of the text against the trie shifts. This is seen in the following image, where instead of the 'o' being the first character matched to some edge, the 'b' is.

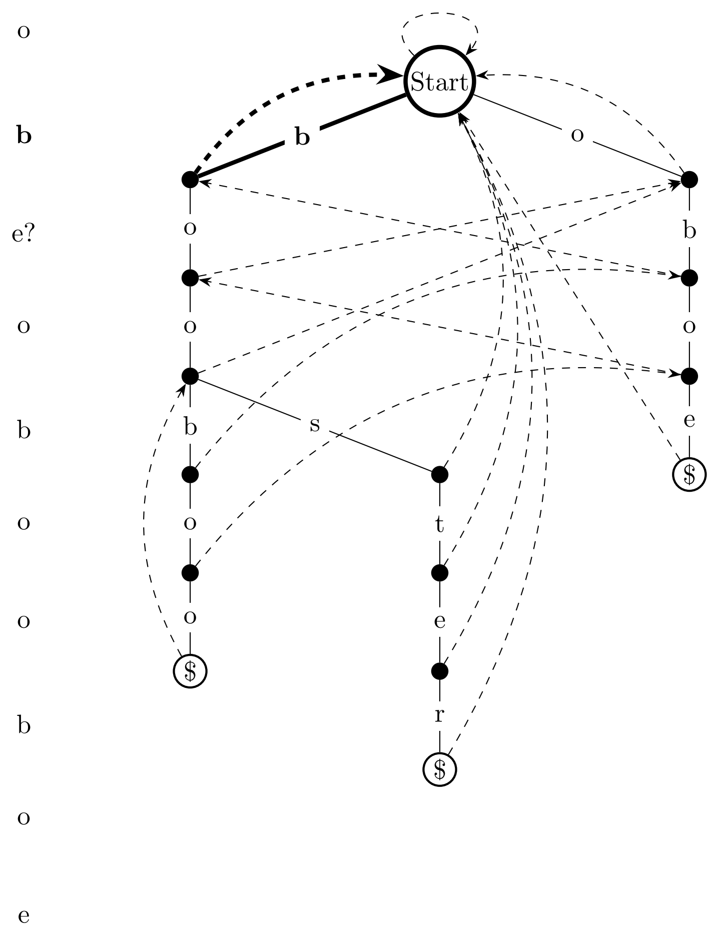


Figure 9.29: Continuing from the b node, we do not find a match for e, so we follow the overlap edge back to the root.

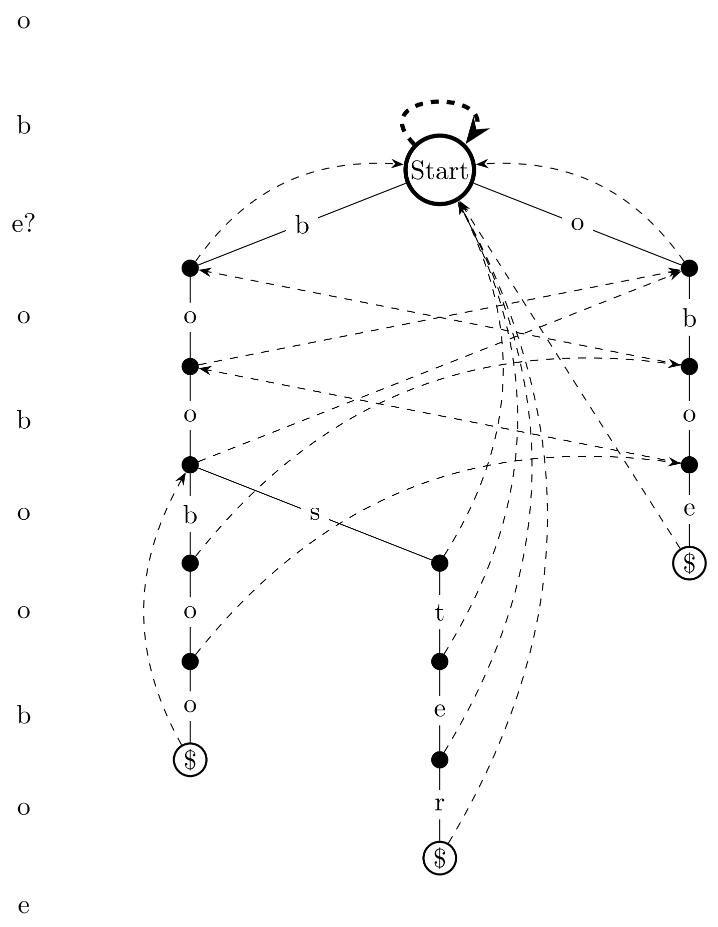


Figure 9.30: Continuing from the root node, we do not find a match for e, so we follow the overlap edge back to the root.

As we are traversing the root node to itself, we know that no pattern can match at this alignment in the text. We explicitly increment the alignment of the text and the trie. This is the only time in the algorithm this happens explicitly. It is regulaly implied by the decrease in depth of the current node we are comparing to relative to the text.

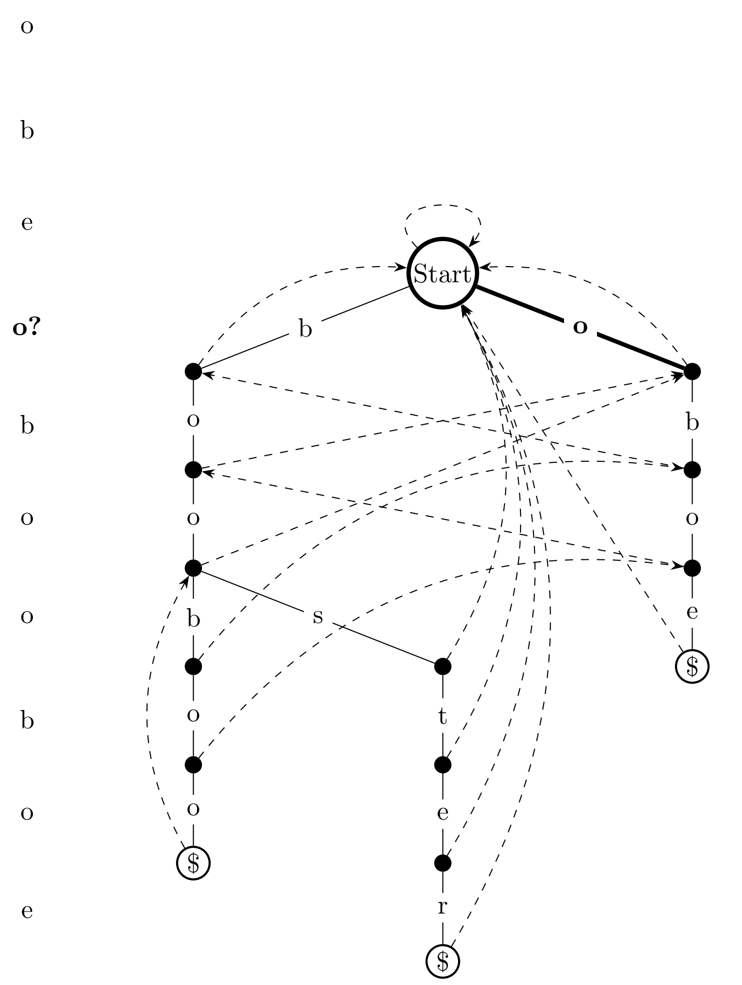


Figure 9.31: Starting at the root, we match the outgoing edge o.

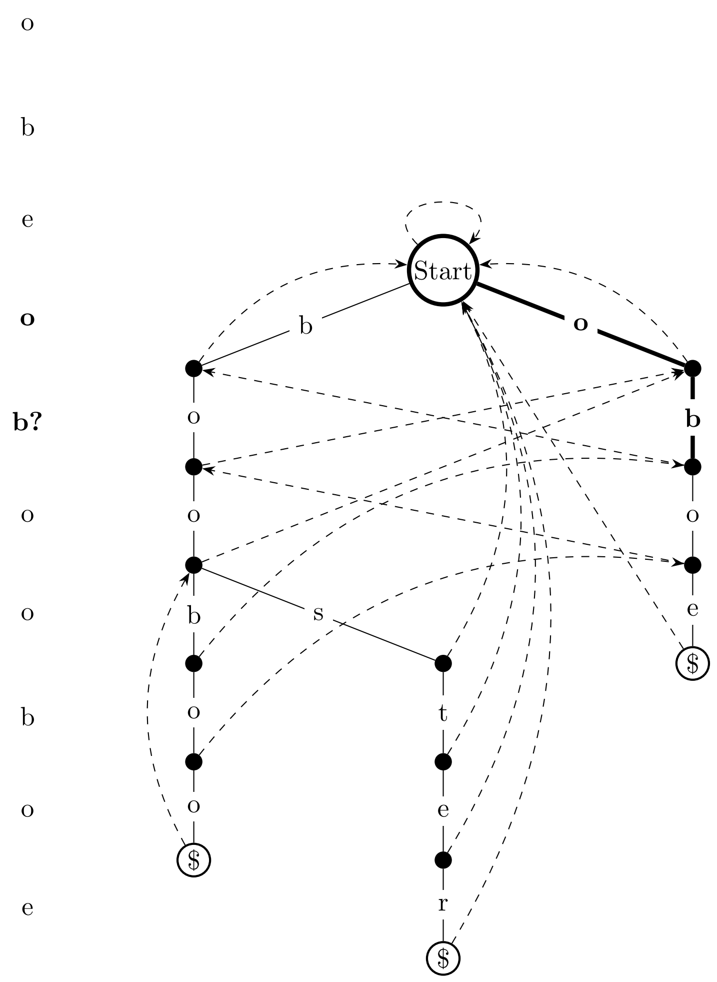


Figure 9.32: Continuing from the o node, we match an outgoing edge b.

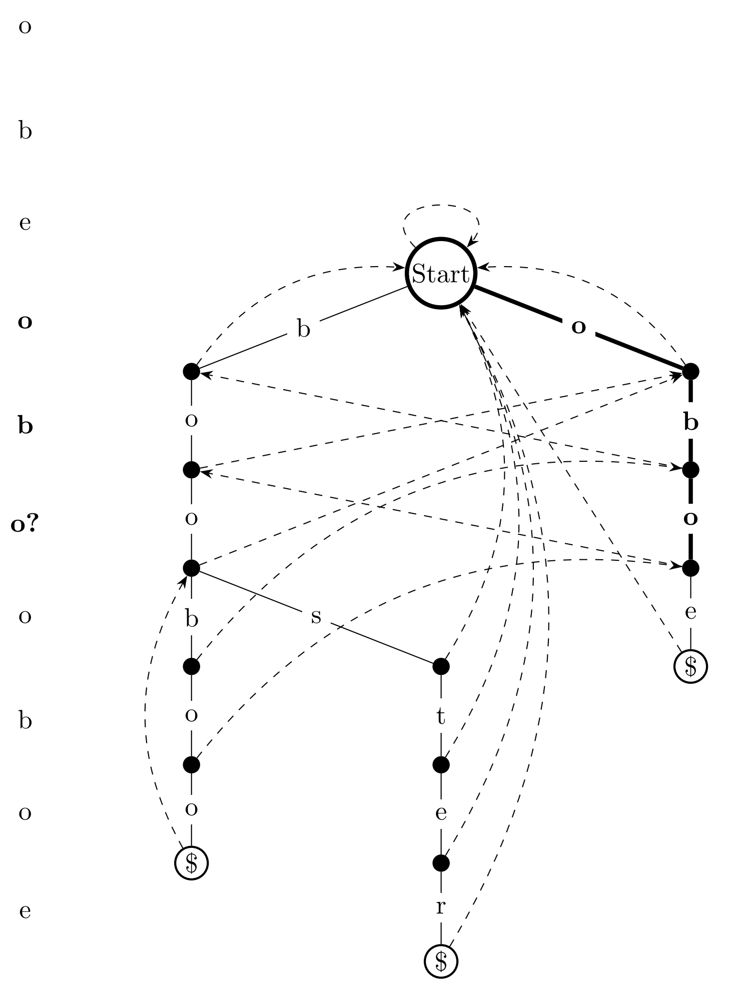


Figure 9.33: Continuing from the ob node, we match an outgoing edge o.

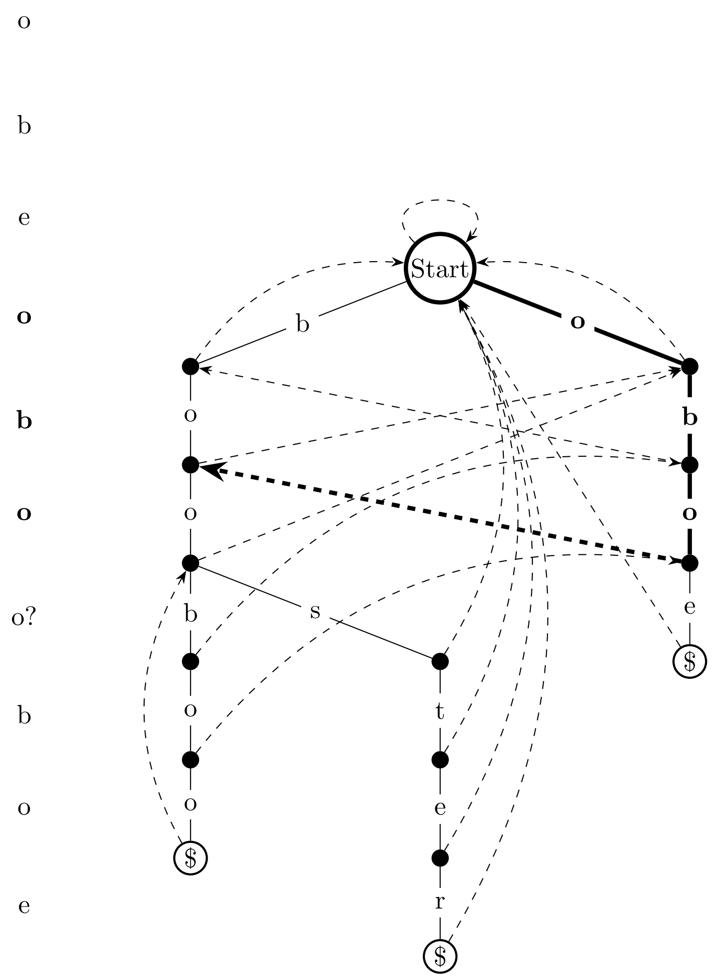


Figure 9.34: We do not find an outgoing edge for the o, so we follow the overlap edge.

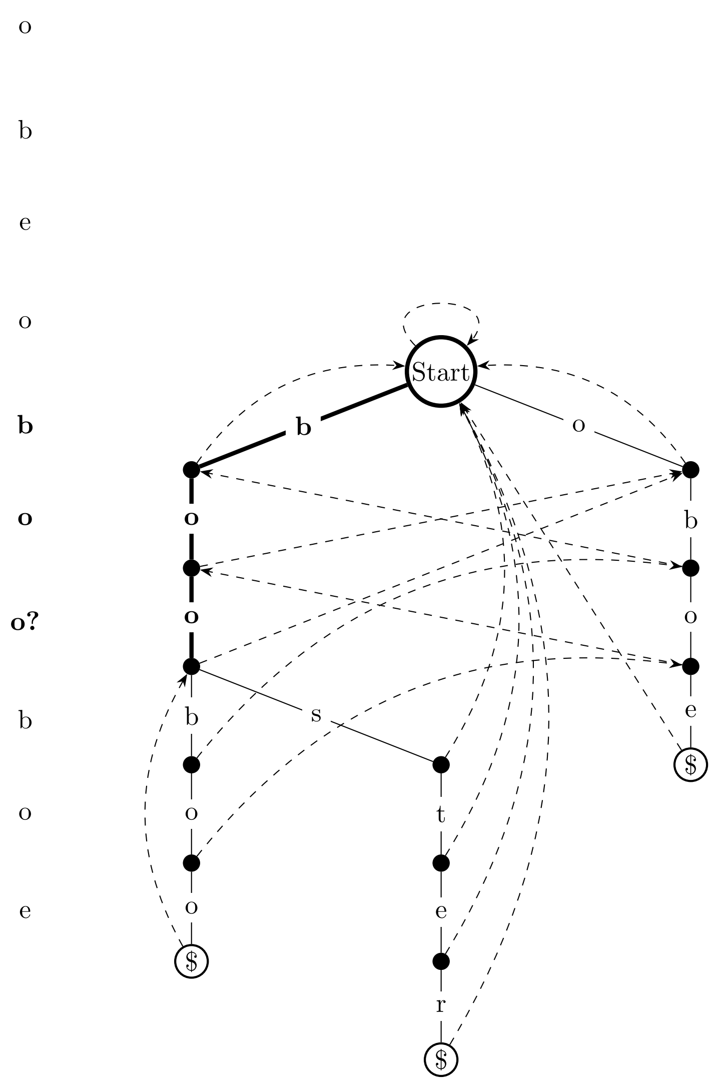


Figure 9.35: After implicitly following the overlap edge, we now find a match for the o.

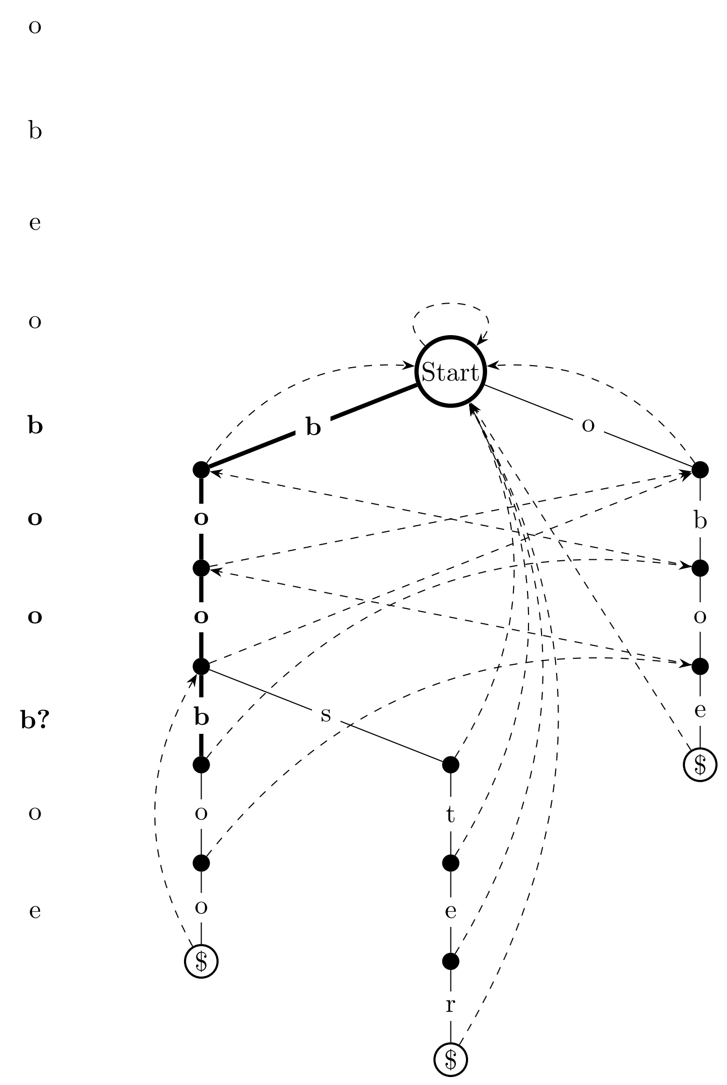


Figure 9.36: We find a match for the b.

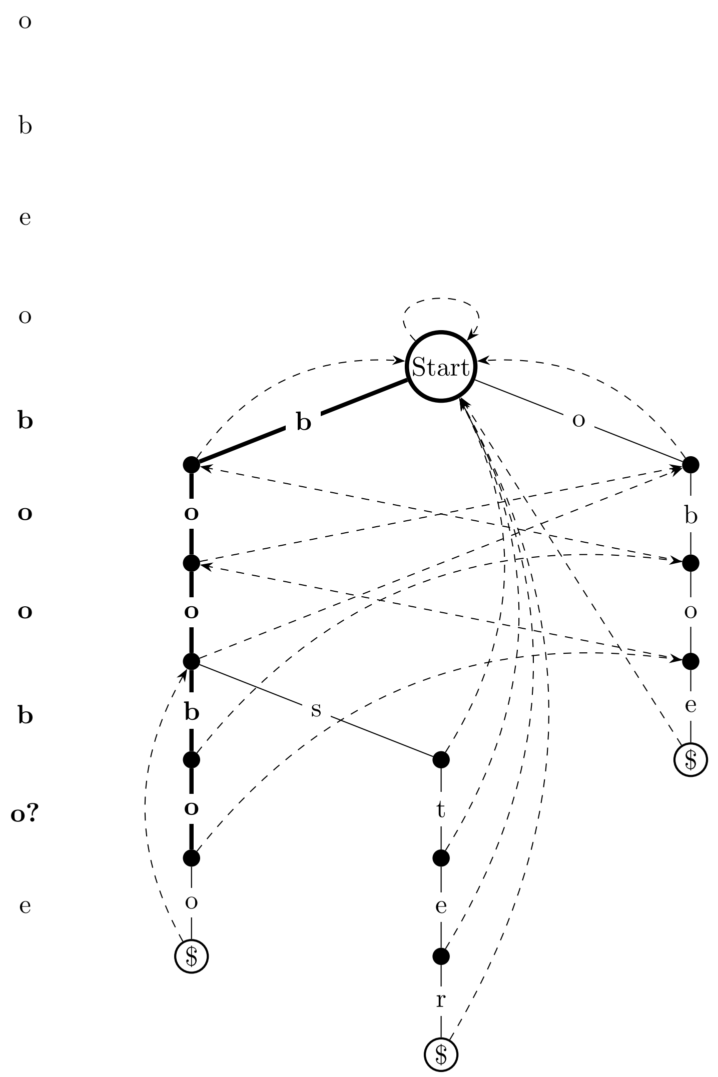


Figure 9.37: We find a match for the o.

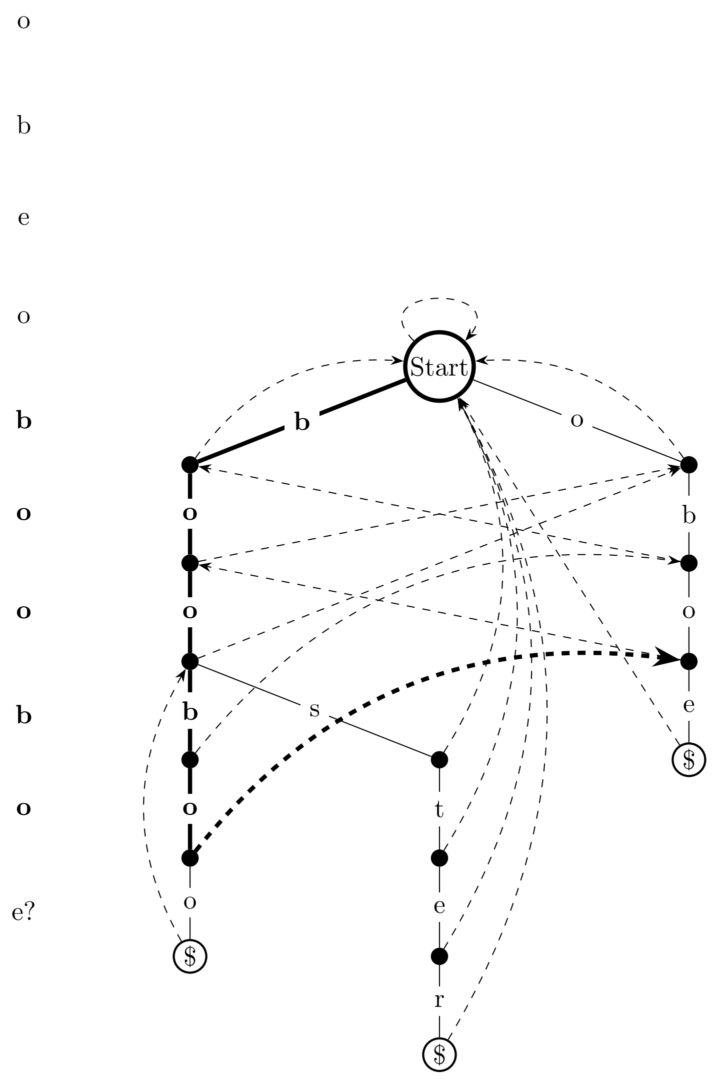


Figure 9.38: We do not find a match for the e, so follow the overlap edge.

After following the overlap edge in this stage, note that as the depth of the node we are on decreased by 2, we have effectively skipped one alignment of the text relative to the tree, saving valuable computation.

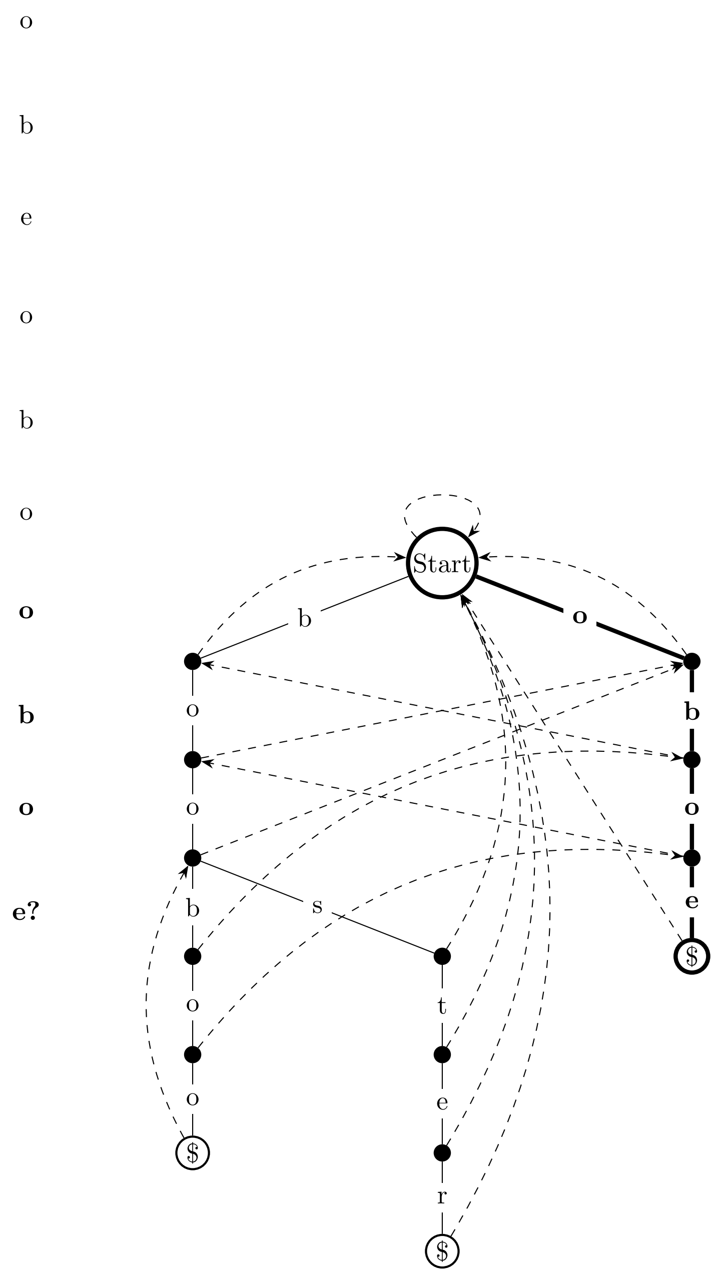


Figure 9.39: We find a match for the e.

At this stage in the algorithm, we have arived at the terminal node of some pattern. We should at this point record the match, including which pattern we found, and at which index we found it. While our text ends here, were it to continue, we would note there are no outgoing edges from that node, and follow the overlap edge, as usual.

Proof of Linearity Each comparison of a character either results to progressing to the next character in the text, or following an overlap edge. As the overlap edge always decreases the depth of the current node by at least 1, the alignment of the text to the trie is also shifted by at least one. As there are only $O(n)$ alignments, the total runtime of this step is also $O(n)$. Combined with the time to compute the overlap edges, we have performed the algorithm in $O(n + p)$ time.

9.1.3.5 Corner Case: Patterns Containing Other Patterns

Recall we made the assumption that no pattern is a substring of some other pattern. Our previous example was precisely constructed to avoid this possibility. If it were the case, executing the algorithm as described above may miss matches. We can see this trivially by adding the pattern "ob" to our search. The modified trie appears as follows:

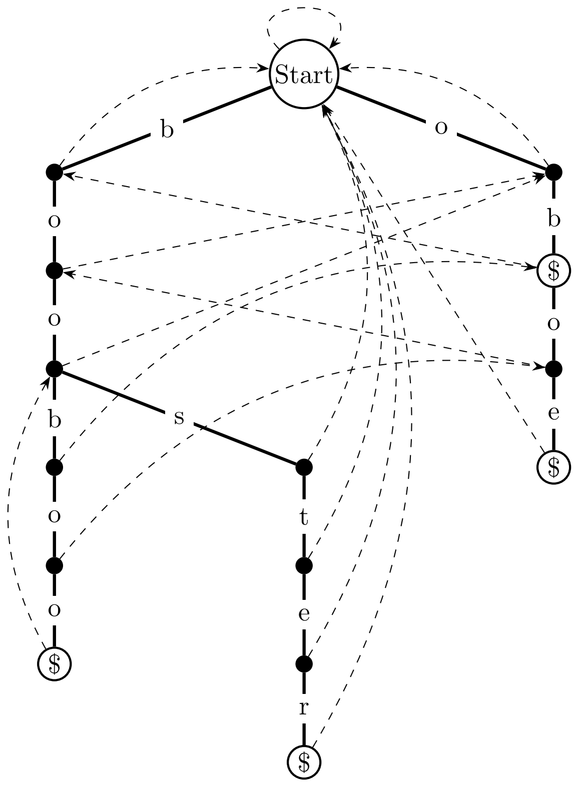


Figure 9.40: The ob node now indicates a pattern terminates there.

It is easy to see this pattern may be missed. Consider matching simply the word "booboo." Executing the algorithm as described will discover the match of booboo itself,, as well as the match of the entire word, but will miss the the occurrence of 'ob'.

To resolve this issue, we need a way to indicate while traversing distal parts of the tree, that we may have traversed a match of some other contained pattern. In this case, for every instance of ob in the trie, we must be able to know that ob is a match in itself. More generally, if the suffix ending at a given node is a complete match elsewhere in the trie, we must indicate so. To accomplish this, we will link all such nodes with a third type of edge we'll call a pattern edge.[22]

A pattern edge will be drawn from a node to a terminal node whose label is both a suffix of the source node, and for which the length of that match is the longest of any other matching suffix in the trie. By this definition, we can glean two things:

1. Each node has at most one pattern edge.[23]

2. By recursively following pattern edges from a given node, we can identify all patterns which are suffixes of the label of the that node.[24]

The trie amended with the pattern edge appears as follows:

[22]Also known as a dictionary link or output link in other texts. There is no good justification for not choosing one of these terminologies here, but we haven't used precendented terminology yet, so why start now?

[23]There is at most one pattern of a given length which is wholly contained and terminates at any particular index in any individual pattern.

[24]If two patterns of different lengths terminate at the same index in some third pattern, the shorter of those two patterns must be a suffix of the longer, and thus have a pattern edge, or by induction, a chain of pattern edges to reach from the longer to the shorter.

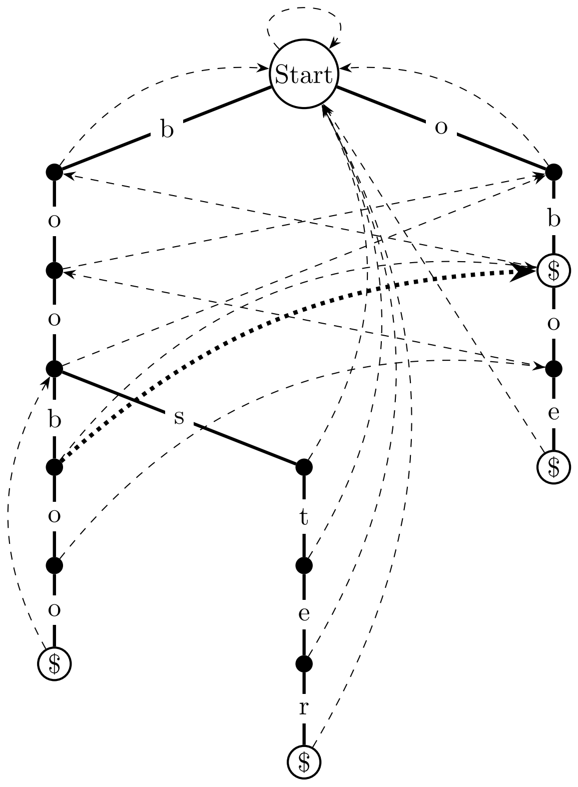


Figure 9.41: The dotted edge indicates there is a remote node in the tree which has a suffix "ob". Traversals of that node must indicate that a match was found.

If we knew all such pattern edges, we could amend the algorithm as follows:

> When traversing a node, if there is an outgoing pattern edge, recursively follow it and record all matches. Once complete, resume matching at the original node.

In the case of our example matching "booboo," we traverse the node "boob" and detect the pattern edge. We note the match represented by the destination "ob" node, as well as there are no pattern edges originating from that node. We then proceed matching from the original "boob" node. Care must be taken when following an overlap edge after following a pattern edge to ensure that the pattern is not matched a second time, nor are subsequent edges duplicately followed, as might be the case if matching booboe.[25]

Upon adding this modification, we add another term to the runtime, k, the number of matches. As we may have to follow a pattern edge to form a match

[25]Note that while in this case the pattern and overlap edges are the same, that is not always the case. We can guarantee, however, the overlap edge is at least as deep as the pattern edge, and that if they are not the same, then the source and destination of the overlap edge both contain exactly the same pattern edge.

without advancing either the current character in the text or the alignment of the text to the trie, it must be added, leading to a runtime of at least $O(n+p+k)$.

Computing Pattern Edges The question then remains how to efficiently compute the edges. As the pattern edges are a subset of all suffixes of a given node which exist in the trie, we can do this while computing overlap edges using the following rule:

1. If $Olap_{edge}(n)$ is the terminal node of a pattern, draw a pattern edge. It is, by definition, the longest suffix which exists in the trie, and thus there can be no longer pattern which is a suffix.
2. If $Olap_{edge}(n)$ is not a terminal, but has a pattern edge, draw a pattern edge fron n to the destination of that preexisting pattern edge.[26]
3. Otherwise, do not draw a pattern edge.[27]

As this process adds only a constant time evaluation at each node, it does not impact the overall runtime of what is now a complete Aho-Corasick algorithm.

9.1.3.6 Implementation

Despite being substantially similar in idea,[28] the implementation of aho-corasick is far trickier, as it involves the construction of a tree[29] rather than a simple integer valued function. To accomadate this, we perform the algorithm in three high level steps:

1. Build the trie, without the overlap or pattern edges
2. Process the trie, adding the overlap and pattern edges
3. Align the text with the trie, finding any matches as we go

For simplicitly, the trie is encoded with a lookup table, with a given character indexing to a specific outgoing edge. While this results in compact code, it does result in slower traversal of all children, as all possible characters must be examined to check whether there is an outgoing edge. This is non-optimal in cases where there are large alphabets or otherwise sparse tries. For any alphabet size, it is unlikely a hashed structure performs faster, so if more performance is needed, a compressed adjaceny matrix-like representation may be useful.

Each node in the trie contains three additional values:

[26]Suppose this node is not the longest complete pattern which is a suffix of n. Either it is longer than the destination of the overlap edge(in which case the definition of the overlap edge is contradicted as this suffix is longer), it is the same length as the destination of the overlap edge (violating the assertion that node is not a terminal), or it is shorter than the length of the destination of the overlap edge (in which case it also must be the pattern edge of $Olap_{edge}(n)$).

[27]The proof that this finds all terminal edges is a direct corollary to the proof of case 2. If any pattern existed, it must be findable via the overlap edge.

[28]formally, KMP is a specific case of Aho-Corasick

[29]with multiple types of edges

1. The overlap edge
2. The pattern edge
3. The index of the pattern which terminates at this node, if one exists

In order to leverage non-null values to imply existence, the entirety of the trie is 1-indexed. The root itself sits at index 1, and all terminal values are 1-indexed into the list of patterns.

The use of a single value betrays one issue: we cannot deal if we have multiple identical patterns. If we must do so, we can create a second array to store a linked list of all identical patterns, outputting all of them. This code is largely independent of the underlying algorithm, and can be ommitted if unneeded.

The loop to perform the alignment with the trie can be done in many ways, but this implementation chooses the following:

1. Advance through overlap edges until we find a node where the next character matches, or are otherwise at the root
2. Advance to the node which matches the next character
3. Output all patterns which might terminate at this node, including pattern edges

This choice of implementation enables clean handling of several corner cases in the algorithm, mostly surrounding outputting all matches at most, and exactly once. If other implementations are chosen, the following cases should be considered:

- Matches which occur at the last character in the text, including pattern edges
- Ensuring matches are not output twice if a pattern edge is the same as an overlap edge
- Ensuring matches are not output twice if a node and its overlap destination have the same pattern destination
- Nodes which have pattern edges but are not terminal nodes for patterns themselves

Finally, a complete implementation:

Listing 9.5: C++

```cpp
//assume a text, t, and vector of patterns, ps.

int dups[ps.size()+1]={0};

//Build the trie
vector<array<int,AS+3>>tr(2,{0}); //children, overlap, pattern, terminal
```

```
for(int i=0;i<ps.size();i++){
   int on=1;
   for(int j=0;j<ps[i].length();j++){
      if (!tr[on][ps[i][j]-F]){ //create new node if doesn't exist
         tr[on][ps[i][j]-F]=tr.size();
         tr.emplace_back();
      }
      on=tr[on][ps[i][j]-F];
   }
   if(tr[on][AS+2])dups[i+1]=tr[on][AS+2];
   tr[on][AS+2]=i+1; //Set terminal
}

//Populate the olap and pattern edges with a BFS
queue<int>q;
for(q.push(1);!q.empty();q.pop()){
   int on=q.front();
   for(int i=0;i<AS;i++)if(tr[on][i]){ //look at each outgoing edge
      int next=tr[on][i],d=tr[on][AS];
      while(d&&!tr[d][i])d=tr[d][AS]; //recurse on olap edge
      tr[next][AS]=d?tr[d][i]:1; //assign either value or root
      tr[next][AS+1]=tr[tr[next][AS]][AS+2]?
         tr[next][AS]:tr[tr[next][AS]][AS+1]; //find pattern edge
      q.push(next);
   }
}
tr[1][AS]=1; //set the root olap edge last.

//do the matching
vector<pair<int,int>>out;
for(int i=0,on=1;i<t.size();i++){
   while(on!=1&&!tr[on][t[i]-F])on=tr[on][AS]; //advance past all olaps
   if(tr[on][t[i]-F])on=tr[on][t[i]-F]; //move ahead if we can
   for(int j=on;j==on||tr[j][AS+2];j=tr[j][AS+1])
      if(tr[j][AS+2])
         out.push_back({tr[j][AS+2]-1,i});//all matches which end here
}

//out has last char of match, so find first
for(auto& i:out)i.second-=ps[i.first].size()-1;
//populate dups
for(int i=0;i<out.size();i++)
   if(dups[out[i].first+1])
      out.push_back({dups[out[i].first+1]-1,out[i].second});
```

9.2 Palindromic Substrings

Palindromic substrings are those which read the same forward and backwards. Put another way, a substring of length N is palindromic if for all i, the i-th character of the substring matches the $N-1-i$-th character. We are often interested in finding the longest palindromic substrings within a string. To do so, we will find the longest palindromic substring substring centered about each index in the string, and can then select the greatest of these if desired.[30]

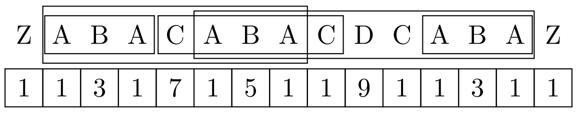


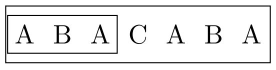
Figure 9.42: A string with all non-trivial longest palindromes boxed. The length of the longest palindrome centered at each character is printed below.

The longest-palindrome array could be computed in quadratic time with the following naive algorithm:

Listing 9.6: C++

```
vector<int> longest(string s){
   vector<int> lp(s.length(),1);
   for(int i=0;i<s.length();i++){
      while(i+lp[i]+1<s.length()&&i-lp[i]-1>=0&&s[i+lp[i]+1]==s[i-lp[i]-1])lp[i]+=2;
   }
   return lp;
}
```

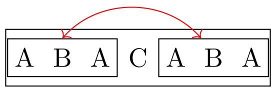
Speed-up Intuition Consider just 7 of the characters in the above example. Suppose we have already found the palindrome centered at the second and fourth letters.



What can we discern about the 6th character? We know that the strings on either side of the the 'C' are reversed. We also know that a palindrome, when reversed, is itself. Therefore, the palindrome centered on the 'B' to the left, when reversed about the 'C' will be identical to the string centered at the 'B' to the right.

We can therefore declare there is a palindrome of (at least) side 3 centered at the second 'B' without having compared a single character.

[30]Note this only considers odd-length palindromes. See the implementation section for dealing with even-length ones.



If we drive this intuition to its conclusion, we notice we never have to compare any characters on the right hand side of any previously found palindrome. As we saw above, palindromes on one sound can be found on the other. Similarly, if there is not a palindrome centered at a character on one side, there will not be on the other.[31]

Leveraging these properties over the entire string enables us to skip enough comparison to beat the naive quadratic time.

9.2.1 Manacher's Algorithm

Manacher's Algorithm articulates how we can apply the above intuition to identify all palindromes in an input string in $O(1)$ time.

From a high level, we will compute the longest palindrome values in order, and remember some information about previously found palindromes, specifically the palindrome which reaches most right into the string, that enable us to quickly compute future values using the following rules. We will refer to this right-most palindrome as a *palindrome box*.

1. Rule 1: If the current index is not currently in the palindrome box, compute the largest palindrome. If we find a palindrome, set the palindrome box as this palindrome must necessarily be right-most. This rule would be used in our example when we reach the first 'B' or 'D'. Note that neither lies inside any previously found box.

2. Rule 2: If we are in the palindrome box, we know that the right side is reversed exactly on the left side of that box's center. Look up the longest palindrome at the index mirrored about the center.

   (a) Rule 2.1: If the palindrome at this corresponding index would keep us within our palindrome box, take it as the longest palindrome for the current index. We see this rule apply at index 12.

   (b) Rule 2.2: If the palindrome at this corresponding index extents to or past the bounds of the palindrome box, we cannot rely on 2.1, since more characters might match. We brute force to examine further characters past the current palindrome box to find the first non-palindromic character. We then update the current palindrome box to the one further to the right.

Astute readers may recognize the above rules. Manacher's Algorithm is, in fact, just the Z-algorithm with a small modification. Instead of tracking the longest string starting at a character which matches the start of the string, we

[31]Except perhaps at the far edge, as we will see.

are tracking the longest string starting at a character which matches the string going in reverse from the current Z box. This adjustment does not impact either the arguments for correctness or runtime, only where we get our candidate value for each index. In the Z algorithm, we got it from the corresponding index at the beginning of the string based on our Z box, but now we get it from the corresponding index going backwards from the current palindrome box.

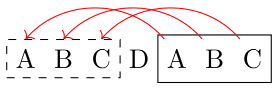


Figure 9.43: Here we see a representation of the Z algorithm. The solid box is our current Z box, and the dahsed box is the duplicate of of the Z box at the start of the string. The red arrows indicate where we lookup to start evaluation of this index.

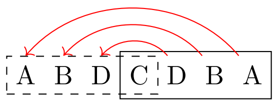


Figure 9.44: Here we see the Z box representation of the algorithm. The solid box is our current Z box, and the dahsed box is the duplicate of of the Z box, but reversed from the start of the current Z box. The red arrows indicate where we lookup to start evaluation of this index.

9.2.1.1 Example

As this algorithm is less intuitive than the canonical Z algorithm, we will walk through an example.

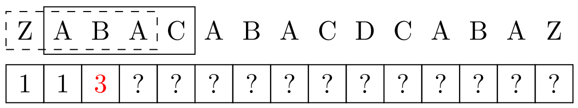


Figure 9.45: The first 3 indices all follow rule 1, explicitly computing the longest palindrome. At this third index, we find our first palindrome box. We will use the Z algorithm view, with solid and dashed boxes indicating the current palindrome box, and its reversed duplicate.

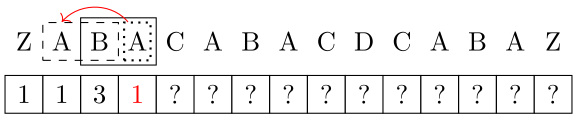


Figure 9.46: As we are currently inside the palindrome box, we lookup our corresponding index, and find a value of 1. As this brings us to the end of the edge of the current palindrome box, rule 2.2 applies, so we must explicitly check the next character to see if there is a match. In this case, B does not equal C, so the value of 1 remains. The dotted box shows the "candidate" palindrome abutting the edge of the solid palindrome box.

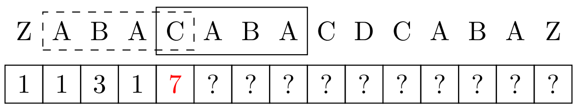


Figure 9.47: At the next index, we are outside the palindrome box, so rule 1 applies, forming a new palindrome box.

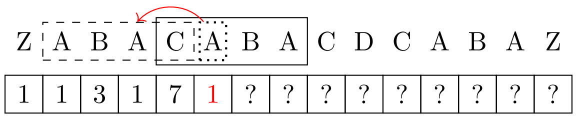


Figure 9.48: At the next index, we look up our corresponding index. This value does not reach the end of the palindrome box, so rule 2.1 applies and no further comparison is needed.

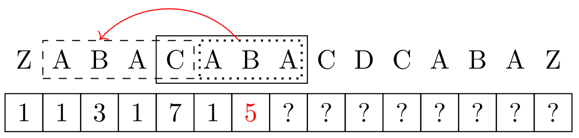


Figure 9.49: When we lookup the candidate for the next index, we find it abuts the end of the palindrome box, so rule 2.2 applies. We perform an explicit check of the subsequent characters, and find the matching C's on either end, but not the next pair. This results in a total size of 5. As this palindrome extends beyond the current palindrome box, we have identified a new palindrome box.

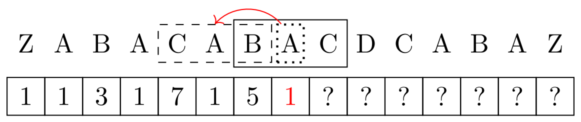


Figure 9.50: Lookup of the candidate in the new palindrome box does not reach the end of the box, so rule 2.1 applies.

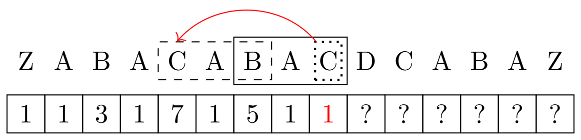


Figure 9.51: Lookup of the candidate in the new palindrome box does reach the end of the box, so rule 2.2 applies. No further comparison succeeds, however.

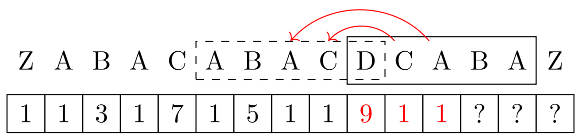


Figure 9.52: As the next index is outside the palindrome box, rule 1 applies and the new palindrome box is calculated explicitly. The subsequent two indices follow rule 2.1.

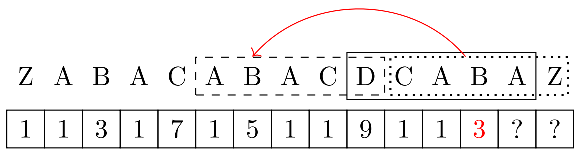


Figure 9.53: Lookup of the next index would yield a palindrome of length 5. This candidate would exceed the edge of the palindrome box, so rule 2.2 applies. We can only guarantee a match up to the edge of the palindrome box, so begin comparison at the edge. We find the subsequent character (the ’Z’) does not match, and therefore the palindrome centered here is in fact, only the 3 characters, reaching the end of the palindrome box.

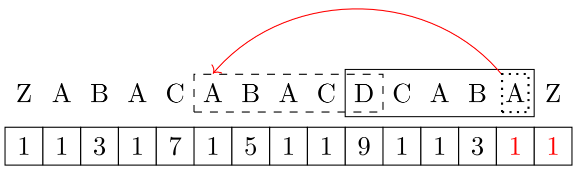


Figure 9.54: The last A follows rule 2.1 due to abutting the end of the palindrome box. No further match is found, leaving 1. The last character is outside the palindrome box and uses rule 1.

9.2.1.2 Implementation

The implementation tracks largely with the implementation of the Z algorithm, but for the adjustment of the lookup index as well as some additional bounds checking to ensure a palindrome would not run off the beginning of end of the string.

Even-length Palindromes Thus far, we have only considered odd-length palindromes, where some character was explicitly at the center. How then, do we find Even lenth palindromes?

One approach is to simply run the algorithm twice, returning two arrays. On the first pass, the odd length palindromes are found as described and stored in one array. On the second pass, even length palindromes are found by adjusting the algorithm to consider the center as between characters, rather than on one. These results are found in the second array. Both arrays must be considered to identify all palindromes.

We use a second approach. In the input string, we simply insert a dummy character between each input characters. This means that every palindrome in our input string is an odd-length palindrome in the modified string, centered on one of the dummy characters. Odd length palindromes centered at index `x) are reported at index \lstinline`2*x) while the even-length palindromes are reported in the interleaved indices.

We should note this would lead to each palindrome being reported as twice as long as it is in the input string, as there are twice as many characters. Fortunately, the Z algorithm natively reports the number of *matching characters*, which is only half the length of ultimate palindrome, less the one overlapping character. These two effects cancel eachother out, leading to the correct output, after an off-by-one adjustment.

Listing 9.7: C++

```
vector<int> manacher(string s){
    char d[2*s.length()+2]={};
    vector<int> z(2*s.length()-1,1);
    for(int i=0;i<s.length();i++)d[2*i+1]=s[i];
    int c=0;//The center of the palindrome box
    for(int i=0;i<z.size();i++){
      //Rule 2. Identify a candidate palindrome, limited to the bounds
      //of the palindrome box
        if(i<c+z[c])z[i]=min(c+z[c]-i,z[c-(i-c)]);
      //Rules 1 and 2.2. Identify the widest match we can.
      //Note the bounds check for the start and end of the string.
        while(i-z[i]+1>=0&&i+z[i]+1<2*s.length()+2&&d[i+z[i]+1]==d[i-z[i]+1])z[i]++;
      // Set the palindrome box, if we have found a new one
        if(i+z[i]>c+z[c])c=i;
    }
    for(int i=0;i<z.size();i++)z[i]--;
    z[0]=1;
    return z;
}
```

9.3 Regexes

While regex libraries are often not performant enough for heavy string processing tasks, there are some problems for which understanding how to apply them can lead to simplified implementation. Consider the following problem:

> The language of chickens involves just two syllabuls: *buh* and *gock*. In order to form more complex thoughts, chickens may concatenate those two syllables together in arbitrary orders, and arbitrary numbers of times, such as *buhbuhbuhbuhbuhgock* or *gockgockgockbuhbuh*. Given a string, return whether it could have been uttered by a chicken or not.

While we could write a parser to consume those syllables, we can trivially solve this problem by checking whether the input string matches the regex `(buh|gock)+`.

Listing 9.8: C++

```cpp
bool is_chicken(string s){
   return regex_match(s,
      regex("(buh|gock)+"));
}
```

Listing 9.9: Java

```java
import java.util.regex.*;
bool is_chicken(String s){
   return Pattern
      .compile("(buh|gock)+")
      .matcher(s)
      .find();
}
```

Below follows a very concise description of the standard regex language. Note that the exact specification will be language dependent, though many things are common.

- Regexes are matched one character at a time.
- `^` and `$` are special characters that match the front and end of the string respectively. Otherwise, matches can come from anywhere.
- To match one of multiple characters, we can use brackets. `^a[lt]$`, for instance, would match *al* and *at*, but not *all* or *alt*. Note that even though there are multiple characters in the brackets, we still only match a single instance of all of those characters, not multiple.
- Matching a range of characters can be done using a dash. `[a-z]` would match a single lower-case letter, for instance. `[a-zA-Z]` would match a single letter of any case.
- Matching all but a group of characters can be done by starting the bracketed group with a `^`. `[^a-z]` would match any single character except for a lower case letter.

- Performing a match on any one of a group of strings (as opposed to characters), is done using parentheses and pipes. As we saw above, `(buh|gock)` would match *buh* or *gock* exactly once.

- A `*` character indicates the previous element (either a single character, or a group, if encompassed by parentheses), can be repeated many times, or not at all. `[aeiou]*` would match a string of zero or more vowels, and `(on|off)*` would match a string of any number or combination of the words *on* and *off*, including none.

- A `+` character functions like the above `*` character, but the previous element must occur one or more times.

- A `?` character functions as the above two, but the previous element must occur exactly zero or one time.

- In many languages, a number in brackets indicates the previous element should occur a specific number of times. `{3}` would indicate the element should occur exactly three times, and `{4,6}` would indicate the element should occur four to six times.

- A `.` character matches any character. Therefore, `.*` matches any string, as it is any character, repeated 0 or more times (note, not necessarily the same character repeated, but any character).

- A backslash followed by a number indicates a backreference, and represents some string previously matched. Each group in the regex surrounded by parentheses has an absolute ordering, generally numbered by the order in which the opening parenthesis occurs, from 1 to n. taking that index and placing it after the backslash tells the engine to match whatever string that group of parentheses happened to match, and match it again. For instance `(buh|gock)-\1` would match either of *buh* or *gock*, followed by a dash, and followed by whichever of those two syllables matched previously. This regex would match *buh-buh* and *gock-gock*, but not *buh-gock* or *gock-buh*. The same is true for square brackets.[32] Backreferences can usually nest inside parenthesis, leading to very complicated patterns.

- Each language has special characters. `\s` typically matches any whitespace character,[33] `\d` matches a digit, and `\w` matches alphanumeric characters. Capital versions of those characters typically invert the match (i.e. any non whitespace character, non-digit, or non alphanumeric character).

- Typically regexes involve characters which are already special in each language, so escaping becomes very complicated when specifying. For instance, `regex r("\\\\")` matches a single backslash. Note that `\\` is the

[32]Note this is the opposite behavior we get were we to use one of the above repetition operators, where we can use different members of a group for each repetition. Placing a repetition operator after a backreference still locks the back reference to whichever string was matched originally.

[33]Hence `\s+` matching any amount of whitespace

string literal for a single backslash, therefore, the string itself only ends up containing two backslashes. Similarly, the backslash is a special character in the regex engine, so it needs two to end up matching a single backslash, leading to needing four total in the string literal. Matching whitespace is correctly done with `regex r("\\s+")`, which will pass `\s+` into the regex engine, which is the special character for whitespace repeated. In general, you need twice as many backslashes as the regex language would define when defining the regex using a string literal.

As noted, the exact features that are available in a specific language differ, so one should refer to the language specification for exact details beyond what is covered here.

9.4 Suffix Trees

A suffix tree is a structure where every path from the root to a leaf encodes some suffix of an input string, and all suffixes are encoded along some such path. It can be used to solve several problems, some of which are covered earlier in the chapter, including:

1. Checking if a pattern exists in a string
2. Finding the location of all instances of a pattern within a string
3. Finding the lonest repeated substring within a string
4. Finding the lonest common substring (LCS) between two input strings
5. Finding the longest palindromic substring within an input string
6. Finding the minimum rotation of an input string
7. Constructing the suffix array

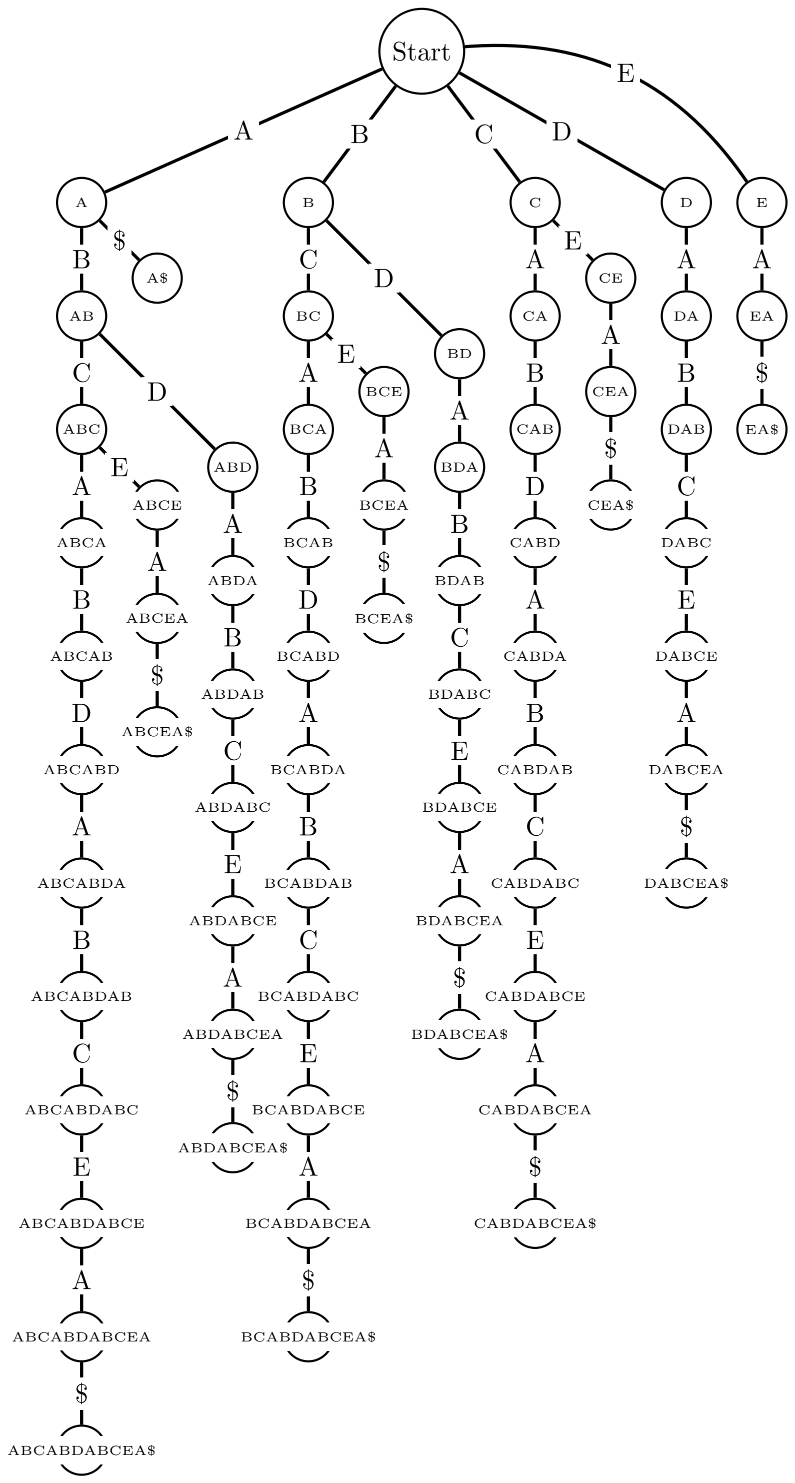
9.4.1 Building Suffix Trees

In this section, we will present a naive suffix tree, demonstrate how, through a series of optimizations, we can operate on, and then build it, in linear time. Following sections will demonstrate how to use it to solve the above listed problems. Throughout, we will use the following string in our demonstrations:

ABCABDABCEA

9.4.1.1 The Naive Tree

In the naive tree, each edge emenating from a node is labeled by a letter, which identifies the next node along the path. The tree for the above string looks as follows:


Start
A
B
C
D
E
A
$
A$
B
AB
C
ABC
E
ABCE
A
ABCEA
$
ABCEA$
A
ABCA
B
ABCAB
D
ABCABD
A
ABCABDA
B
ABCABDAB
C
ABCABDABC
E
ABCABDABCE
A
ABCABDABCEA
$
ABCABDABCEA$
D
ABD
A
ABDA
B
ABDAB
C
ABDABC
E
ABDABCE
A
ABDABCEA
$
ABDABCEA$
B
C
BC
A
BCA
B
BCAB
D
BCABD
A
BCABDA
B
BCABDAB
C
BCABDABC
E
BCABDABCE
A
BCABDABCEA
$
BCABDABCEA$
E
BCE
A
BCEA
$
BCEA$
D
BD
A
BDA
B
BDAB
C
BDABC
E
BDABCE
A
BDABCEA
$
BDABCEA$
C
A
CA
B
CAB
D
CABD
A
CABDA
B
CABDAB
C
CABDABC
E
CABDABCE
A
CABDABCEA
$
CABDABCEA$
E
CE
A
CEA
$
CEA$
D
A
DA
B
DAB
C
DABC
E
DABCE
A
DABCEA
$
DABCEA$
E
A
EA
$
EA$

Several notes about the tree:

- Any prefix can be found by starting at the root and traversing to a leaf node following the edges labeled by the next character in the string.
- Nodes are labeled for convenience, giving the path followed to arrive at that node from the root.
- The terminal charachter \$ must be included at the end of the string. Otherwise, we would not know for sure whether a suffix completes at a given node. Consider the node **A**. Without the terminal character, we would not know that there is a suffix which ends there. Instead, we see a path labeled with \$, and can know conclusively. The exact character used doesn't matter all that much, so long as we can guarantee it is a character not contained in the input string.
- As a consequence of the previous bullet, for a string length N, there are exactly N leaf nodes.

The tree can be constructed by looping over each suffix of the input string and traversing the existing tree. If, when at a node, there is no path leaving the node with the next letter of the string, we create a new node, and add the edge before traversing to it. As this is not a binary tree, instead of having left and right children, we have potentially one edge for each letter of the alphabet and the terminal character. We can store the node at the other end of these edges in either a map, or an array, indexed using the ASCII character.

Even though this tree is correct and could be used to solve the prior listed problems, it will never be able to do so, nor even be constructed, in linear time. There are simply too many nodes. Note the long branchless chains in the tree which are repeated multiple times. It turns out, worst case, the number of nodes in this tree is $O(n^2)$. This can be seen by constructing the tree for a string with no repeated letters (say, ABCDEFGH).

9.4.1.2 Optimization 1: Compression

The first optimization involves creating fewer nodes. Namely, we observe that instead of creating a node for each letter in the suffix, we might create nodes at only places where there is a split in the tree. In such a situation, an edge may contain multiple letters, instead of just a single one, even if we still index it by only the first letter. Lets see how the creation of such a tree works.

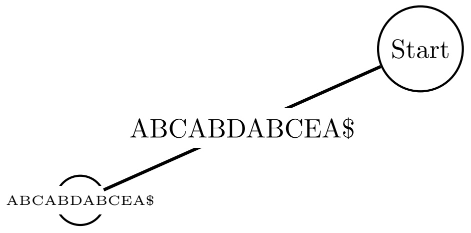


Figure 9.55: Step 1: We add the first suffix to the tree. There will necessarily be no splits in the tree, so there will be a single edge with the entire suffix leading to the only leaf node.

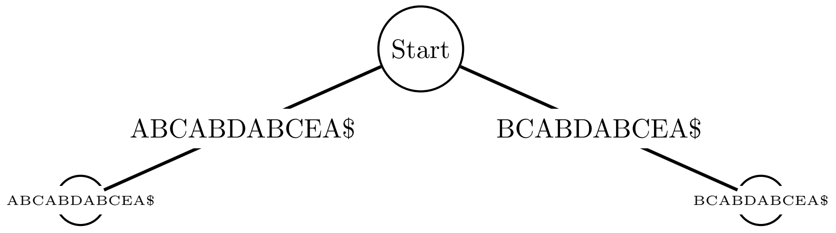


Figure 9.56: Step 2: We add the second suffix to the tree. There is no overlap with the previous edges, so it ends up juts being a single edge.

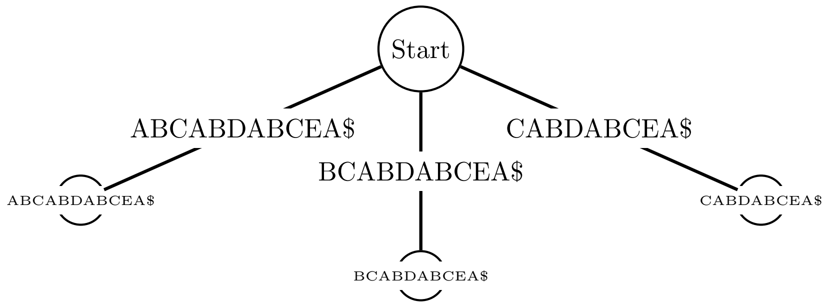


Figure 9.57: Step 3: The same process as step 2.

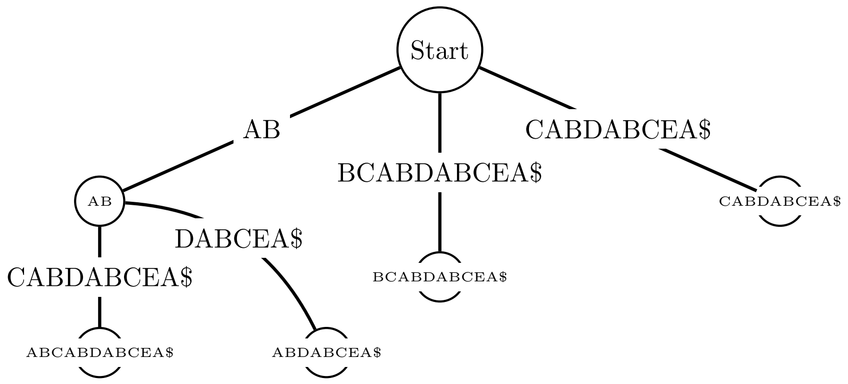


Figure 9.58: Step 4: When adding the suffix ABDABCEA$, we see that an edge beginning with A already exists at the root. So we compare the letters in that edge with our suffix until we find a difference. When we compare the C in the edge with the D in the new suffix, we find they differ. We create a new node at the point where they differ, and create two new edges. Note that if we happen to come across an intermediate node while in the process of comparing, we simply procress down the appropriate edge from that node based on the next character, as will be the case when we insert ABCEA later. We will compare A and B, arrive at a node, then progress down the CABD... edge to compare the remaining letters.

As we've now seen how to add new suffixes, we'll skip to the completed, compressed tree.

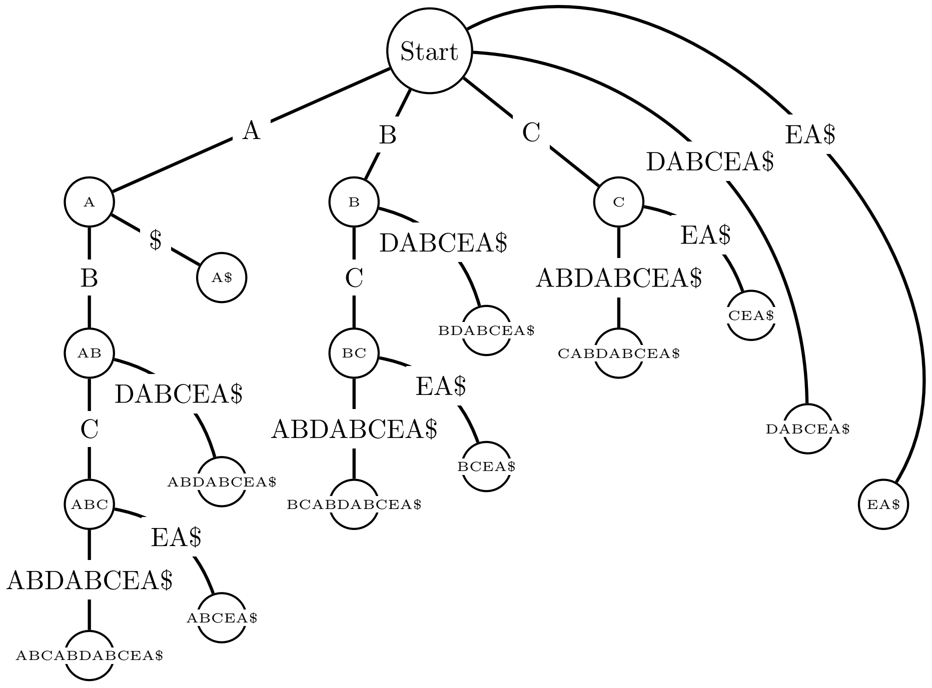


Figure 9.59: The completed, compressed suffix tree. Note that it contains all the same suffixes as the previous suffix tree, but with far fewer nodes.

After the creation of this compressed form, we note that there are far fewer nodes, but exaclt how many fewer? As noted before, we have exactly N leaf nodes. We note here, though, that every internal node, we create a subtree which contains at least two leaf nodes which were not previously in the same subtree. This limits the total number of interior nodes to a maximum of $N-1$, limiting the total number of nodes in the tree to $2 * N - 1$, which is $O(N)$: a reduction by a factor of N from the uncompressed form, even in the worst case.

Despite this, the construction still may be too slow. Consider, again, the worst-case tree with all unique letters. In that case, we are still left with $O(N)$ branches, which each must store $O(N)$ characters, making even the storing the of the edge data super-linear.

9.4.1.3 Optimization 2: Indexing

The next optimization involves avoiding the cost of copying and storing the multitude of characters on each edge. We notice that the string on each edge comprises of some substring of our input. Given that, instead of storing the substring itself, we can simply store a pair of indices pointing to the input string. These indices indicate the start and end of the substring which represents that edge. If multiple such substrings exist, any may be used. We reprise the compressed tree, now including such indices. The substring itself is noted only for clarity, it is not copied or stored with the edge.

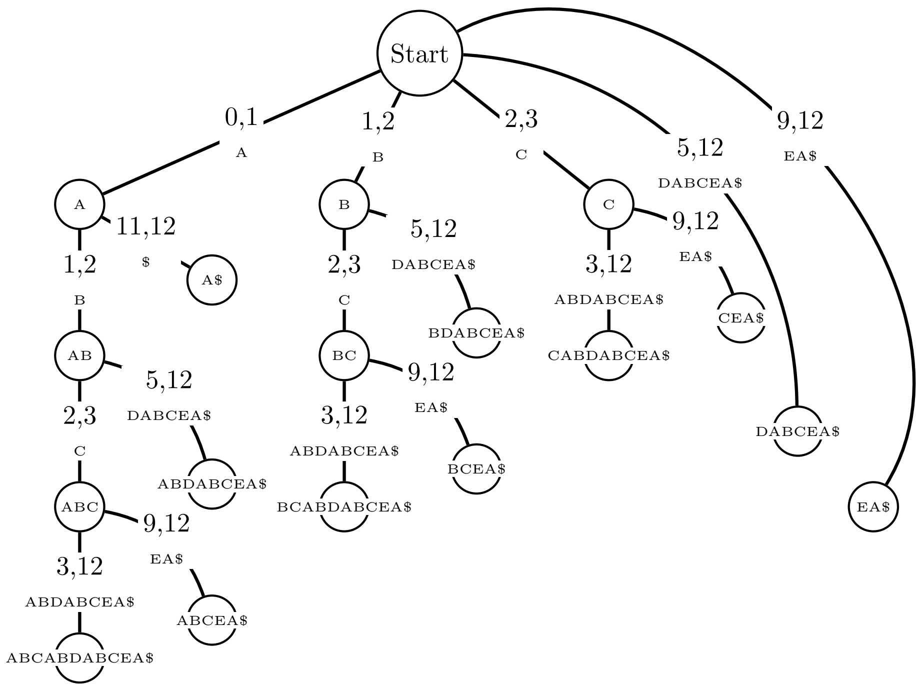


Figure 9.60: Same as before, but with the stored indices. The substrings those indices resolve to, as well as the substrings representing each node, are only presented for clarity.

9.4.1.4 Optimization 3: Per character, not per suffix

Up until now, we have dodged a fundamental issue with our methods in constructing the tree. By iteratively adding each distinct suffix of our input to a tree, we run into a wall. Fundamentally, there are $O(N)$ suffixes, and in the worst case, each one may require $O(N)$ comparisons (consider a highly repetitive string such as AAAAAAAAAAB to see why this is the case). In order to construct a tree in linear time, we must find a way to do so in a small number of, if not single pass over the input string.[34]

[34]There are algorithms that use a suffix-first approach, but they are much more conceptually difficult. See: Weiner's algorithm.

The key intuition is that instead of iterating over each suffix and adding it to the tree, we can build the suffix tree one character at a time. We will create a complete suffix tree for the substring encompassing the first character, then we will extend the input string one character at a time, and attempt to updatte the tree to accommodate it. Then, once we have reached the $ character, we will have successfully build the suffix tree for the complete input.

We will present a few iterations of building a tree with this optimization, but will not construct the full tree, as we'll see there are a few other "problems" we must solve in order to form a complete algorithm.

Here's how it works in practice.

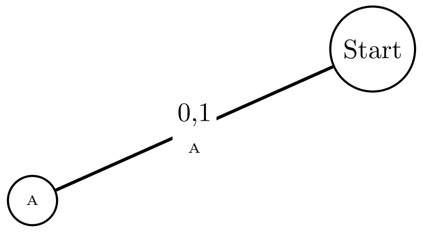


Figure 9.61: Step 1: After processing the first A, we have the single substring that ends at that character encoded in the tree. [A]

In order to add the second character, B, we must know at what nodes terminated a substring for the previous character. After the first step, this is only node A, which is noted in square brackets in the caption. We'll refer to these as *open* substrings. While we represent these based on their label here, this could feasibly use any pointer to the node, such as an index.

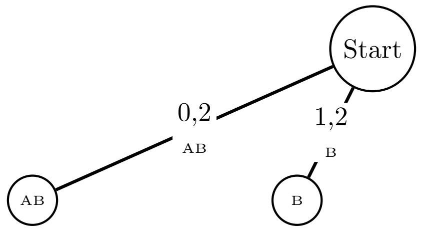


Figure 9.62: Step 2: In order to process the first B, we add a B to the open substring from the previous step by changing the label from (0,1) to (0,2) to indicate the extrac character, and then add B from the root. We now have two open substrings. [AB,B]

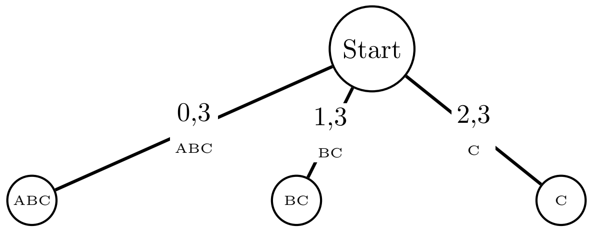


Figure 9.63: Step 3: Similar to step 2, we update (0,2) to (0,3), and then update (1,2) to (1,3), and add the new open substring to the root. [ABC,BC,C]

We'll pause here.

While we've demonstrated creating the tree by character, instead of by suffix, we also see that the construction still has a flaw that will ultimately force us into super-linear time. As we add each character, we increment the number of open strings, and for each new character, we have to update the number of open strings. We note that this most commonly results in incrementing the "end" value of each of the applicable edge labels. Without finding a way to eliminate these updates, we will be unable to reach linear time.[35]

9.4.1.5 Optimization 4: Open Ended Substrings

As we were updating the open substrings, we note that the value we were updating the labels to always ended up being to the same index, representing a substring ending at the character we just added. Instead of using a fixed value for the end index of these open substrings which we must increment each step, we can instead use a floating value which simply represents *a substring ending at whatever the previous character we added to the tree.* We will use a ? to represent this floating index.

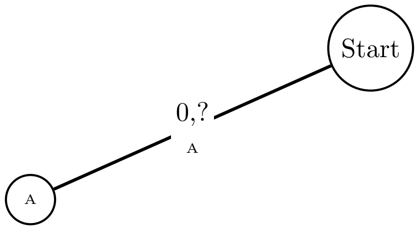


Figure 9.64: Step 1: After processing the first A, we have the single substring that ends at that character encoded in the tree. Note that we indicate its end with the floating index to indicate this edge is still open.

[35]If you're following the formal description of Ukkonen's algorithm, this is Rule 1.

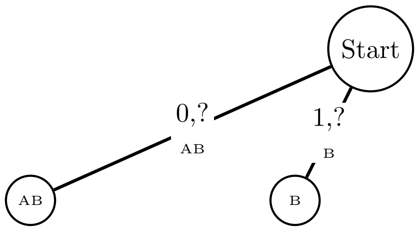


Figure 9.65: Step 2: To add the B, we only added a new open substring to the root. Note that the labels on the edge (0,?) did not change at all, but the value represented by that edge did, since the index which ? represents incremented by one. We have implicitly updated 2 edges worth of substrings while only physically updating one of them.

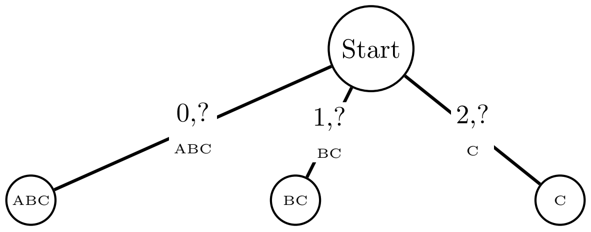


Figure 9.66: Step 3: Similar to step 2, we have only physically added a single edge to the tree, but the remaining open edges have been implicitly updated due to the floating endpoint.

We now would run into an issue when we attempt to add the second A. The A edge already exists, and without "cheating" we won't know if we are supposed to split the edge there or not, as it ends up depending on future characters. In order to allow us to efficiently cope with this, we use a non-commital suffix.

9.4.1.6 Optimization 5: Non-Committal Suffixes

The fundamental problem when we inserted the second A into the suffix tree was that we did not know if the insertion of the A would cause the tree to split. If the next character is the same, we do not want to split, but if it is not, we must. However, we cannot look arbitrarily ahead in the word without violating our per-character paradigm. We must come up with a way to encode that we have an open substring which may or may not result in a split in the edge while also not looking ahead in the string.[36]

This is accomplished by creating a special open suffix, a pointer into the tree indicating the point on a specific edge we will attempt to insert the next

[36]Splitting an edge in this case is rule 2 in the canonical description of Ukkonen's algorithm.

character. This is called the *active point*[37], and is represented using 3 values:

- the parent node of the edge containing the active point
- the character representing the specific edge from the parent node which contains the active point
- the specific character on that edge which is the active point, represented by the number of characters along that edge the active point occurs

When we reach a character we are attempting to insert into the tree, we first check if the character succeeding the active point matches. If it does, we move the active point, otherwise, we split the edge in question. The reset point is initialized to point at the root node, covering the cases when we have not seen repeat letters.

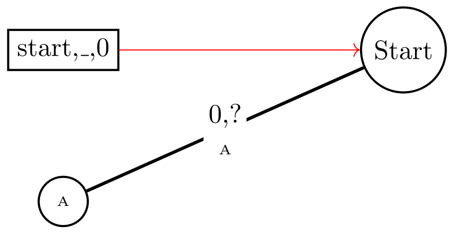


Figure 9.67: Step 1: Inserting the A is identical to before, however we note that we now have an active pointer pointing to the start node.

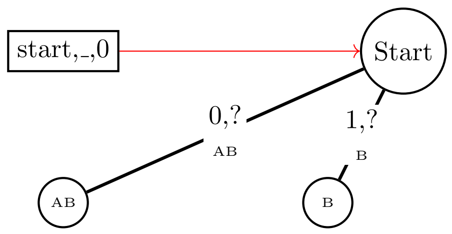


Figure 9.68: Step 2: Inserting the B is also identical to before.

[37]The active point allows us to encode rule 3 from the canonical description of Ukkonen's algorithm

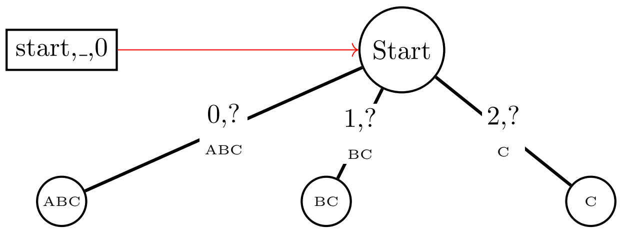


Figure 9.69: Step 3: Inserting the C is also identical to before.

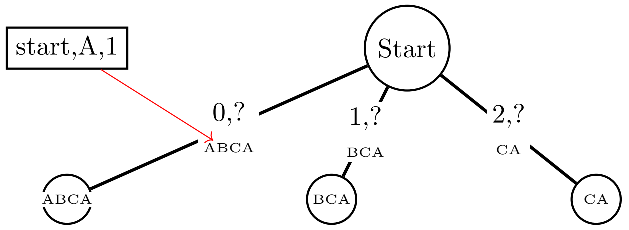


Figure 9.70: Step 4: When we go to insert the second A, we note that we already have a path for A. We don't know, however, whethether we will have to split the edge (without cheating by looking ahead). We modify the active point to the A on the appropriate edge, which is the first character on the A branch coming from the start node.

So far so good. All 4 of the suffixes are either represented by open edges, or the active point.

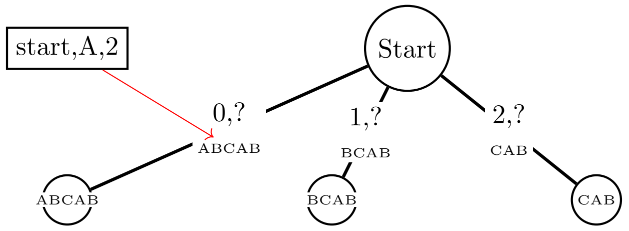


Figure 9.71: Step 5: When we go to insert the second B, we see that the next character after the active point is already a B, so we simply move the active point, in this case, by incrementing the number to 2 (to indicate the second character on the A edge from start).

Despite seemingly going swimmingly, we find we have a problem. Unlike after step 4, we no longer have all the necessary suffixes in the tree. We have, of course, the 3 open edges and the active point, but we note that none of these constructs encode the suffix B. The middle edge has BCAB, but no provision for the single character. We could add a second active point, but we can see how that may become unweildy. Instead, we will simply cache that we haven't added it, represented by "how many suffixes haven't we added yet".[38]

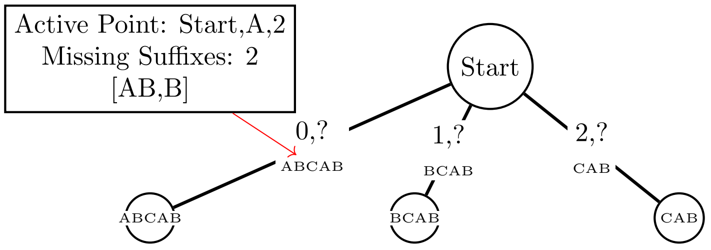


Figure 9.72: Step 5 (take 2): We now have successfully included all suffixes, either on our open edges, or the missing suffixes count. The active point lets us know where the first missing suffix would go. Note that we do not actually save the list of suffixes, only the count.

When we insert the D, we will find a mismatch at the next character of the active point. Remember that the purpose of the active point was to delay splitting an edge until we knew we needed to. So, now that we know we will need to split the edge, we can proceed to do so, and the active point resets back to the start. Now that the active point is not being used, we can proceed to insert the missing suffixes, which have expanded to include the D.

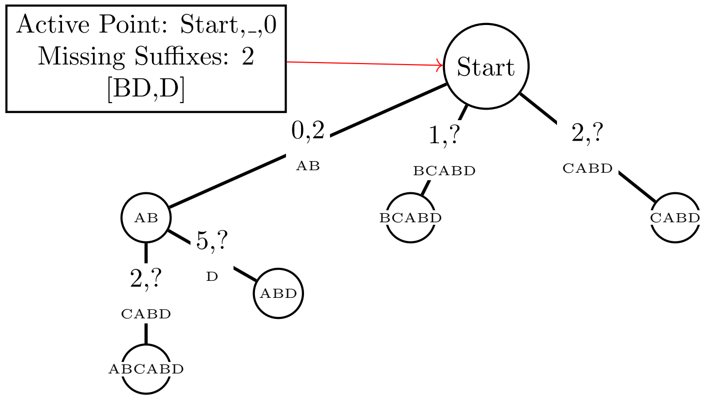


Figure 9.73: Step 6 part 1: We've split the edge, reset the active point, and noted the missing suffixes. Note that we simply increase the missing suffix count, and the last two suffixes are BD and D; we don't actually store those two suffixes, however, so there is no cost to update them. Note that all suffixes are accounted for with the four open edges, and 2 missing suffixes.

We now proceed to add the missing suffixes, as the active point is no longer

[38]Also notice that all the "missing" suffixes are actually in the tree already, they just don't have a leaf node. These are known as *implicit*.

in use. Starting with BD:

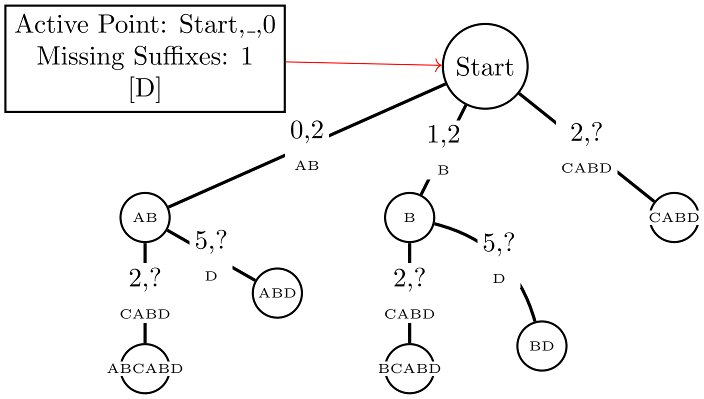


Figure 9.74: Step 6 part 2: We traversed down the tree to insert BD, and found a place to split an edge, so did so.

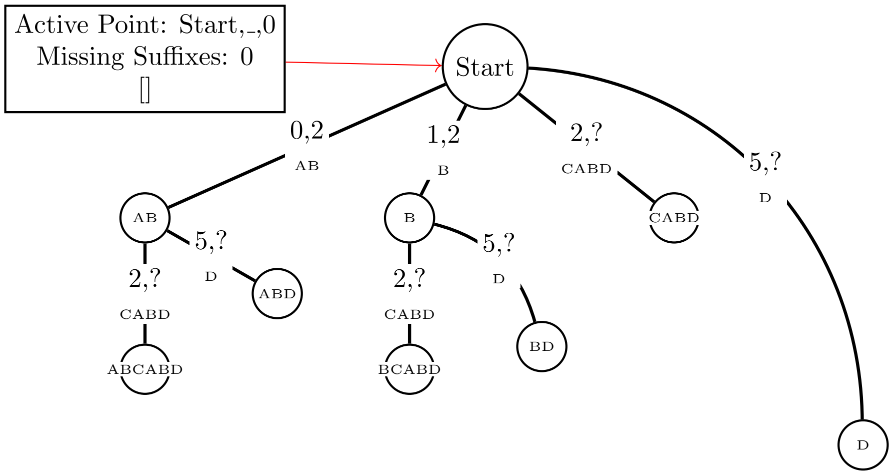


Figure 9.75: Step 6 part 3: We traverse down the tree to insert the next missing suffix, D, but find we already don't have an outgoing edge from the root, so insert it. As there are no longer any missing suffixes, we proceed to step 7.

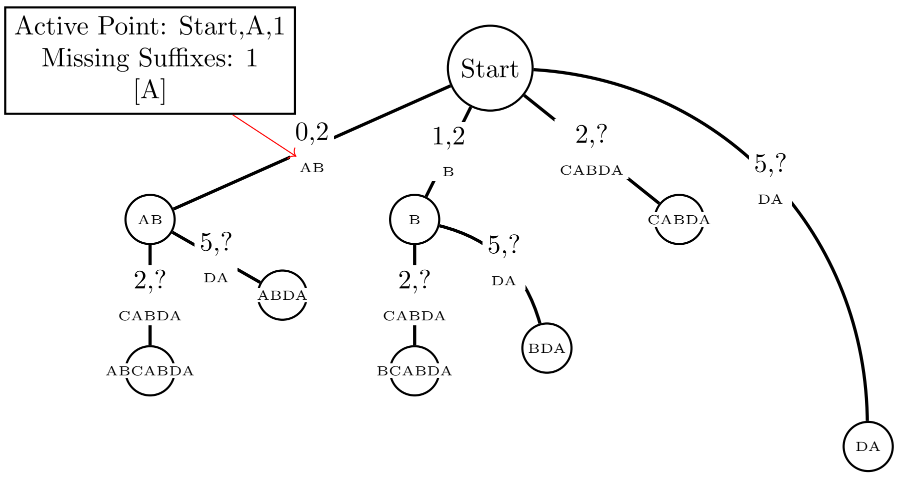


Figure 9.76: Step 7: We are inserting an A, and it already exists, so we simply move the active point.

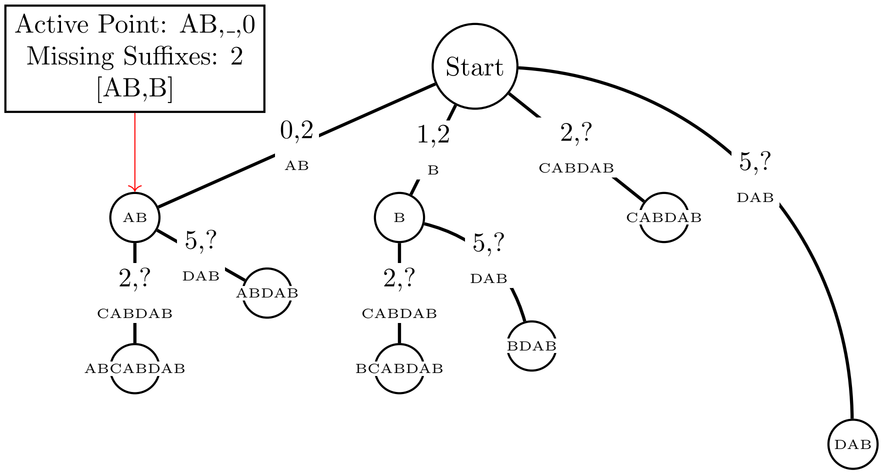


Figure 9.77: Step 8: We are inserting the B, and it already exists, but the active point reaches the end of an edge, so we update it to point to the node at the end of that edge. Note that we have also recorded that we are missing a suffix again.

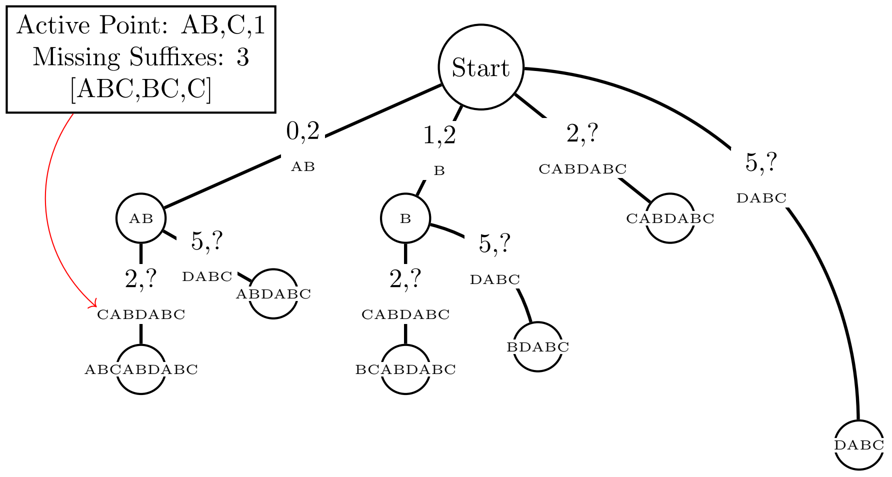


Figure 9.78: Step 9: We are inserting the C, and we can accommodate this by simply moving the active point and noting the extra missing suffixes.

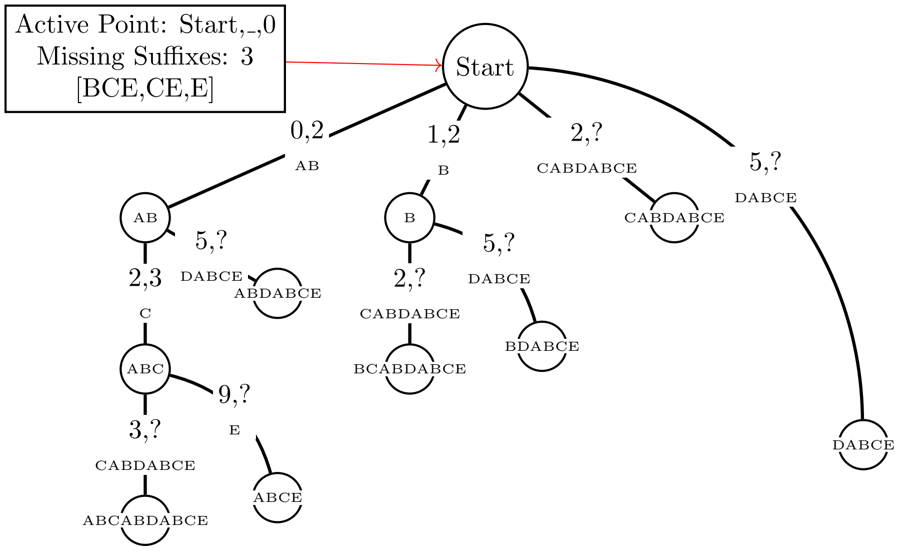


Figure 9.79: Step 10 part 1: We are inserting the E, which does not match the character succeeding the active point, so we must split the edge, return the active point back to the root, then work on inserting the missing suffixes.

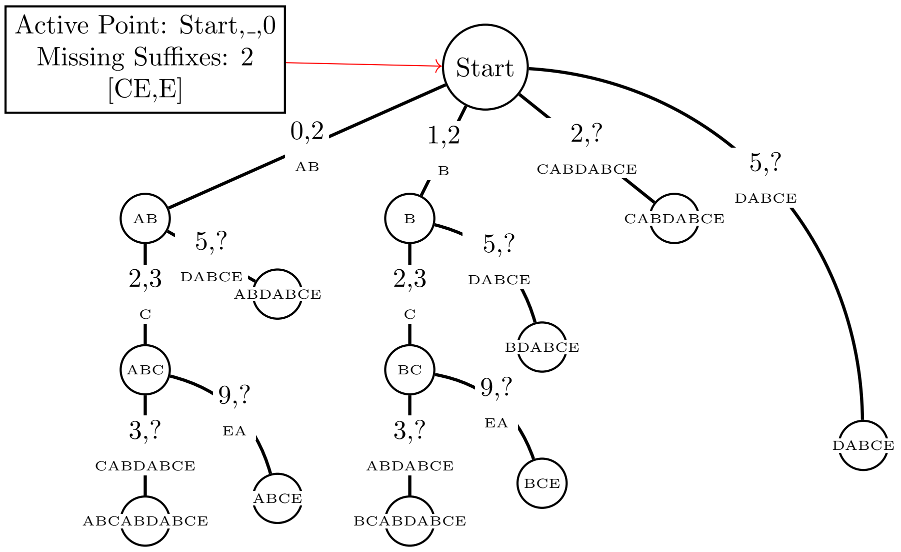


Figure 9.80: Step 10 part 2: We traverse down the tree and insert suffix BCE

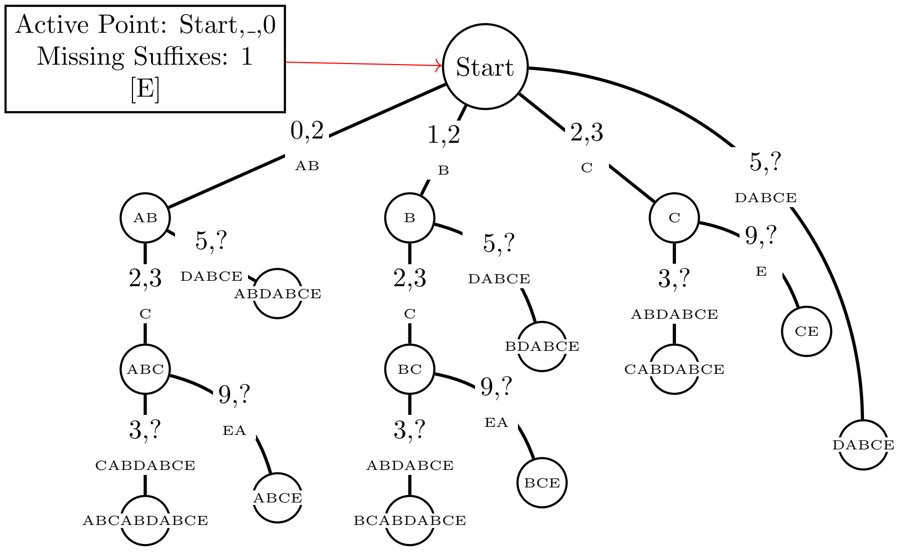


Figure 9.81: Step 10 part 3: We traverse down the tree and insert suffix CE.

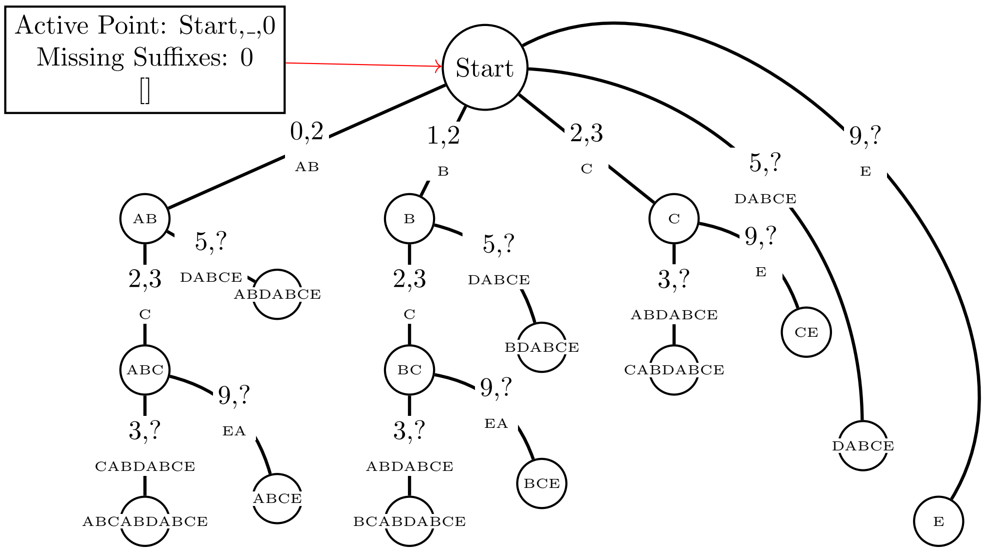


Figure 9.82: Step 10 part 4: We traverse down and insert the final suffix E, and can proceed to step 11.

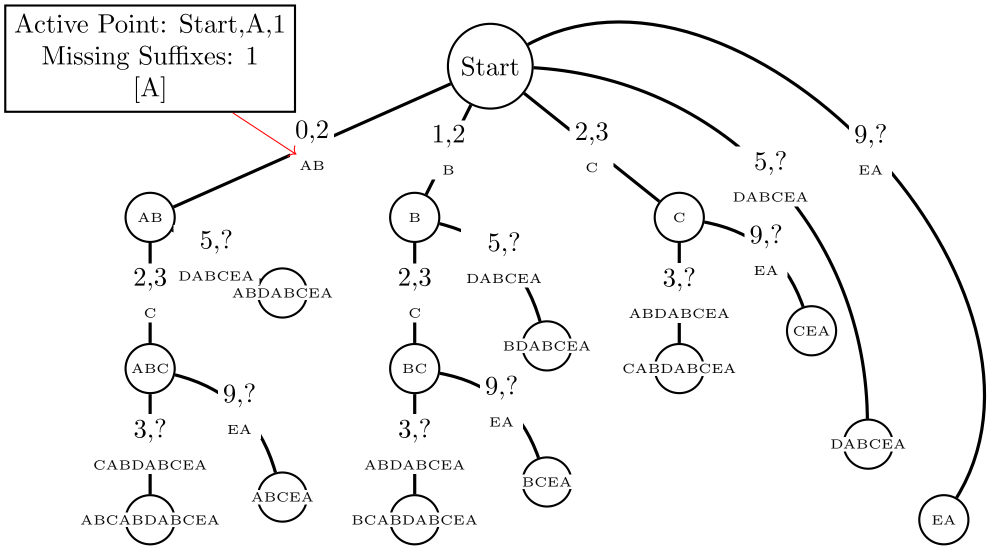


Figure 9.83: Step 11: Inserting the final A is simply a move of the active point.

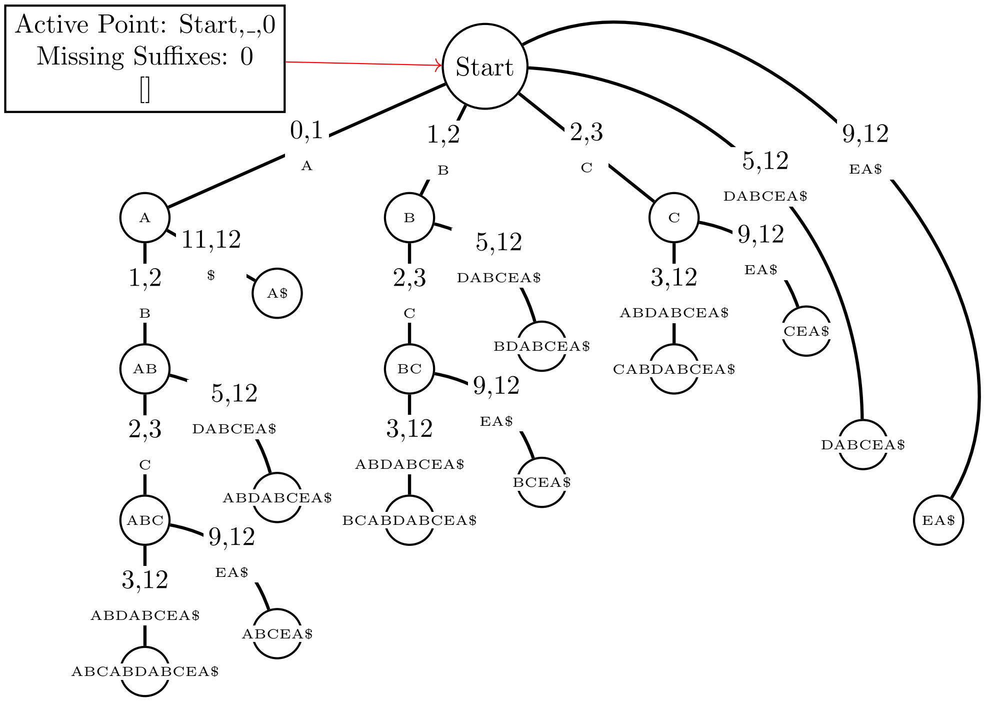


Figure 9.84: Step 12: Inserting the final \$, that terminal character does not match the character folling the active point, so we split the edge and reset the active point to the start node. The tree is now complete. Note: If we so chose, we could insert a \$ node coming off the start node. We convert all ? to a "real" value for completenes.

The algorithm is correct and seems to be efficient in how it builds the tree. There is one issue, however, which prevents linear time. In the stages where we inserted the missing suffixes, we had to traverse down the tree to insert each suffix individually. As there are $O(N)$ potential missing suffixes to insert, and each may take $O(N)$ comparisons, our runtime is still quadratic in the worst case.

9.4.1.7 Optimization 6: Suffix Links

In order to address the problem of inserting the missing suffixes, we first take note of some significant similarities in different regions of the tree.

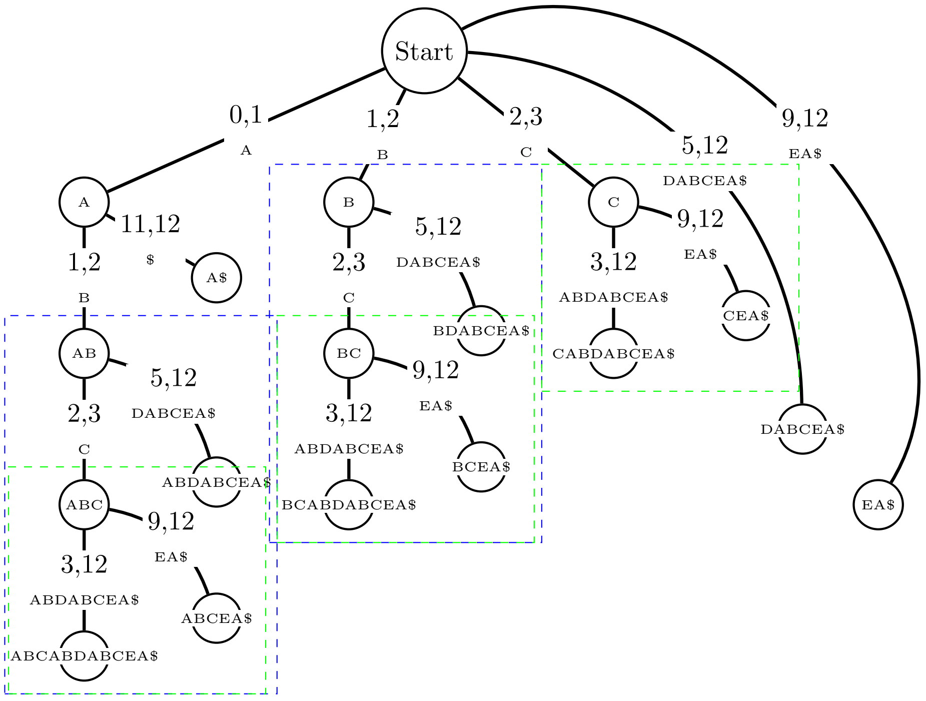


Figure 9.85: There is a "blue" subtree of 5 nodes which contains identical edges occurring 2 times, and a "green" subtree of 3 nodes which occurs 3 times.

In the previous subsection, we note that whatever operation we ended up taking in the AB subtree, we later ended up taking in the B subtree. In step 6, we split the ABCAB edge, then immediately made the same split in the B subtree. In step 10, we split the ABC subtree, then immediately split the equialent node in the BC subtree and C subtree.

By defining the proper relationship between these similar subtrees, we can avoid the previous issue of having to traverse the entire tree while inserting missing suffixes. These relationships are known as *suffix links*. When we split an edge to create a new leaf node, we remember the internal node created at the split, and if we create another leaf node during this step, we draw a suffix link from the "remembered" internal node to the equivalent one. Then, in subsequent steps, if an edge we split comes from a node which has a suffix link, instead of resetting the active point to the root, we instead follow the suffix link and start our traversal there. Suffix links connect the similar subtrees we observe, and avoid us having to traverse from the root down a particular subtree more than once when inserting missing suffixes.

A bit more formally, lets consider a suffix which creates a split on an edge, and thus new internal and leaf nodes: $\alpha\beta$X, where α is some character, β is the substring after α until the newly created internal node, and X is the character forcing the split. Assuming β is non-empty, then there will end up being a suffix link from node $\alpha\beta$ to node β. Note that as we are inserting $\alpha\beta$X, then, since we are iterating character by character, then $\alpha\beta$ and β will already be existing paths in the tree. The node β will come from one of two sources:

1. Created in the next "part". This is the common case, and we draw the suffix link when we create the node β, to split off the leaf node to βX
2. Already existing, and we draw the suffix link when we reach β in our traversal to insert βX

Lets see it in action, starting at step 6, part 2. Remember that in step 6, part 1, we had created an internal node AB, in order to add suffix ABD. We know at this point that we will end up with a suffix link from node AB to node B in the next part.

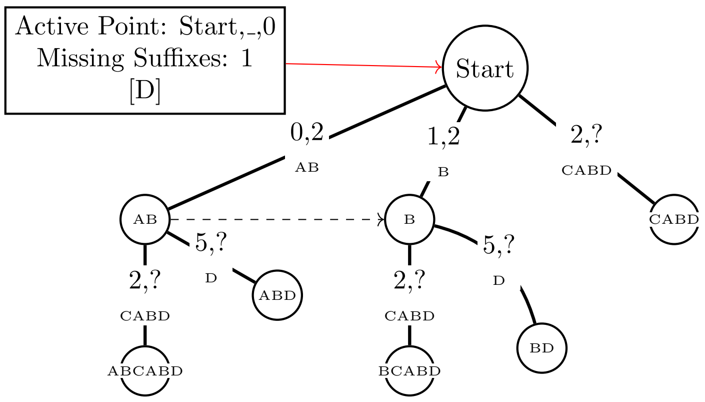


Figure 9.86: Step 6 part 2: We traversed down the tree to insert BD, and found a place to split an edge, so did so, creating node B. As noted above, we know we will need a suffix link from AB to B, so we create it.

The suffix link we drew in the previous picture ends up being the node at the root of the two "blue" subtrees from the previous figure. The implication is that moving forward, every change which occurs in the AB subtree will also end up occurring in the B subtree. This must be true as there are is no such suffix ABX which does not also imply a suffix BX. Note that the inverse is not true, as there may be a suffix BX which was not immediately preceeded by an A.[39] This explains why the suffix link is directed. We see the utility of the link in step 10, and will demonstrate construction of the tree starting at that point.

[39]We will see an example of this shortly.

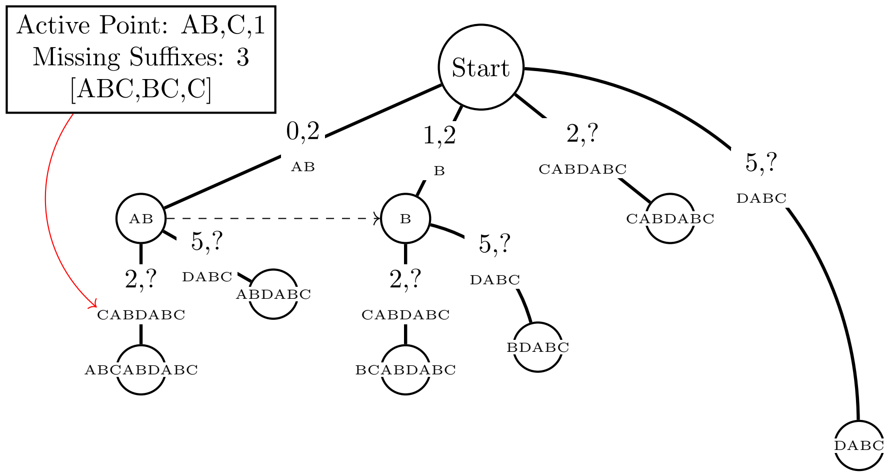


Figure 9.87: The tree after step 9.

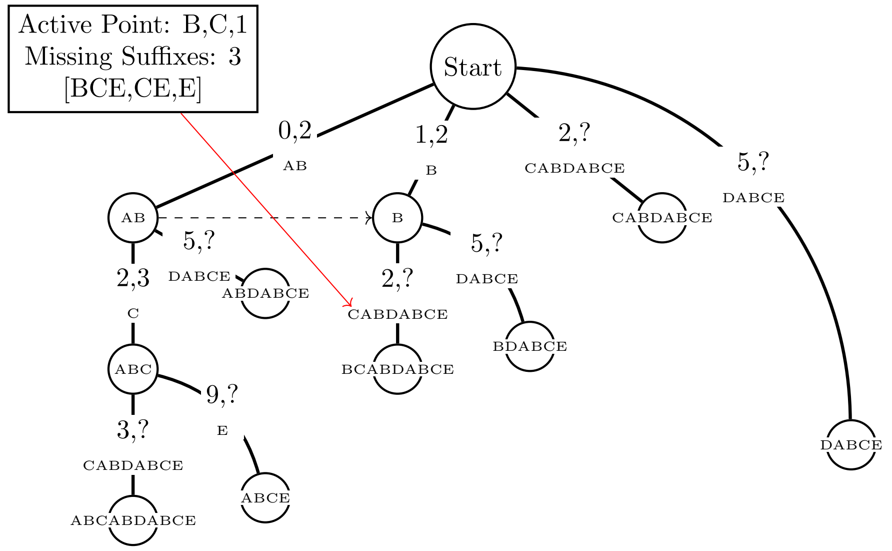


Figure 9.88: Step 10 part 1: As with before, we split the edge to insert the E. We note that as the newly created node is ABC, we will end up with a suffix link to a node BC. Instead of resetting the active point to the suffix, however, we note that the node the active point was using as a base has a suffix link, and cause the active point to "hop" to the location in the new subtree, where as previously discussed, we will end up making the same edit.

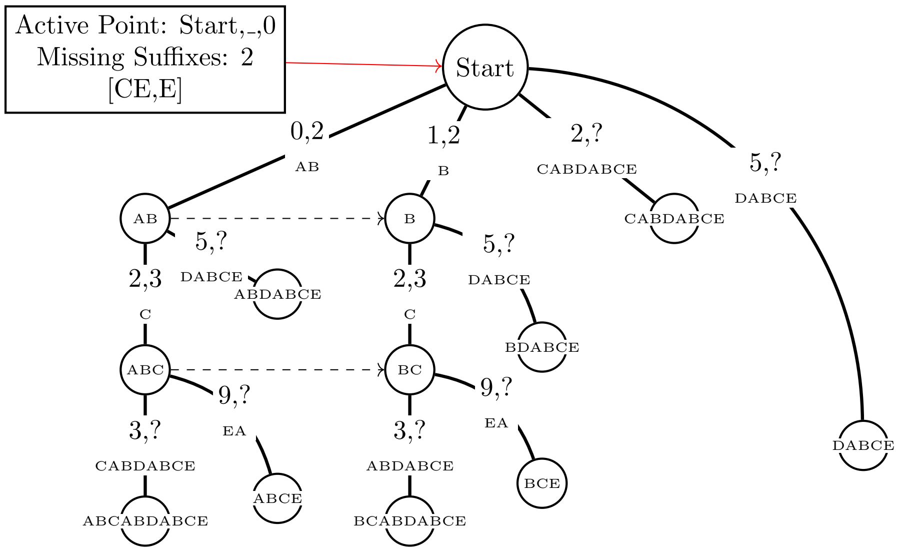


Figure 9.89: Step 10 part 2: Several things happen in this part. As before, we split the edge, since the E does not match the character following the active point. We've now created node BC, which is the endpoint of the newly needed suffix link. We also note that BC will end up needing a suffix link to a node C. Lastly, as there is no suffix link coming out of the B node, we revert the active point to the start. Note that the ndoes ABC and BC which now have a suffix links are roots of two of the green subtrees.

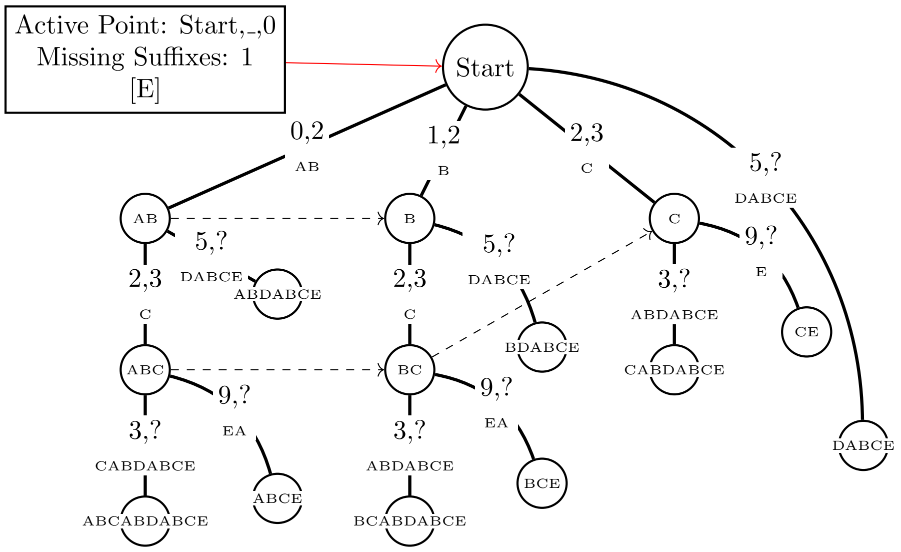


Figure 9.90: Step 10 part 3: We traverse down the tree and insert suffix CE. As with the previous part, the newly created node is the endpoint required for the suffix link we needed, so we create it. Note that this has now linked all three green subtrees.

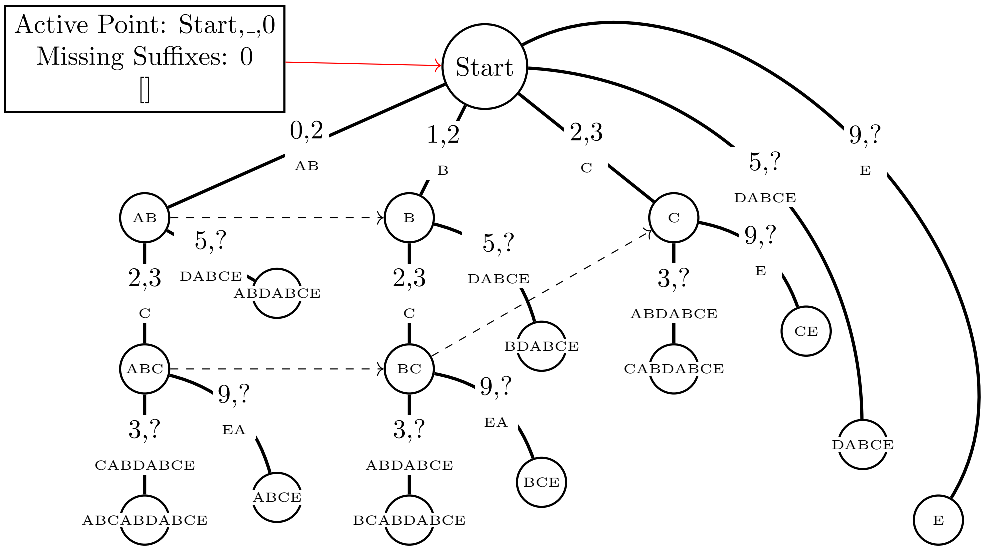


Figure 9.91: Step 10 part 4: We traverse down and insert the final suffix E, and can proceed to step 11.

Steps 11 and 12 are identical to before.

9.4.1.8 Optimization 7: Fast Traversal

Again, while the algorithm seems efficient, there is one step which is still problematic: Traversing down the tree to find a split point. In the worst case, we have to traverse $O(N)$ characters down from the root to find a split point. The final optimization helps us do this more efficiently.

Critically, as noted before, we can notice that while we are manipulating the active point, all the missing suffixes do exist in the tree, but do not yet have associated leaf nodes. Further, any split which occurs does so just before the final letter of the suffix we are inserting. Using these two facts, we can traverse any edge in constant time. If we are at the root (or any node), trying to insert a given string, and there is an edge that comes from that node which matches the first character of the string, since we know that the string will be in the tree, we can simply "skip" to the next node without having to do a direct comparison. The cases here are:

1. Edge is shorter than or same size as string: Move on to the next node at the end of the edge, use edge labels to determine which character to look at next in the string.[40]

[40]The edge labels are indices, so computing the length is simply substraction.

2. String is shorter than edge: We have found the location where we will need to insert a split on this edge without having to explicitly perform comparison

This traversal is used in both places we perform such a traversal: after we have followed a suffix link, or after we mave moved the active point to the root.

9.4.2 Rough Proof of Linearity

We have a linear number of steps, and most elements during a step are constant time. The only hang up is parts where we create a node, which may happen multiple times during a step, and may involve a traversal, even if it is fast. The total number of nodes in the tree is linear, so the actual creation is not ultimately problematic, but ensuring we can find the location to create the node might be. With all above optimizations, it can be shown that we only traverse a linear total number of nodes for the entirety of the algorithm, so assuming we traverse each edge in constant time with fast traversal, gets us linear time overall. The number of traversals can be shown to be linear based on the fact that suffix links (or "resets to the root") all change the tree depth of the current active point by moving up at most one level towards the root. As such movements only happen a linear number of times, the maximum number of nodes we can "climb" the active point is linear, and the heigh of the tree is linear, so the number of downward traversals must also be bound linearly.[41]

9.4.3 Complete Example

Now that all optimizations have been given, we will perform one more complete example to demonstrate the algorithm. We will use the string BCBDABCABD$, and will attempt to represent most of the implementation notes. We will not draw explicit suffix links to the root, however.

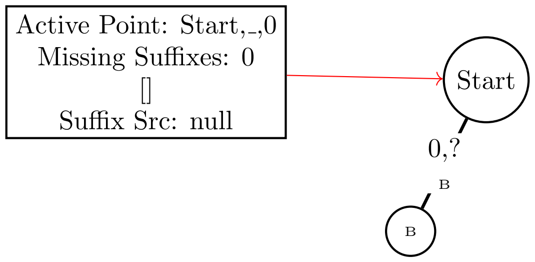


Figure 9.92: Step 1: Root does not have an edge B, so create it.

[41]For the formal proof, check out Dan Gusfield's book *Algorithms on Strings, Trees, and Sequences.*

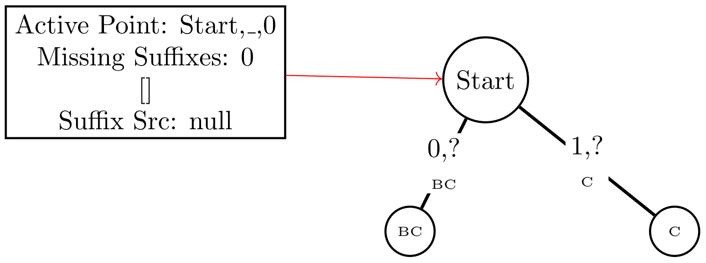


Figure 9.93: Step 2: Root does not have an edge C, so create it.

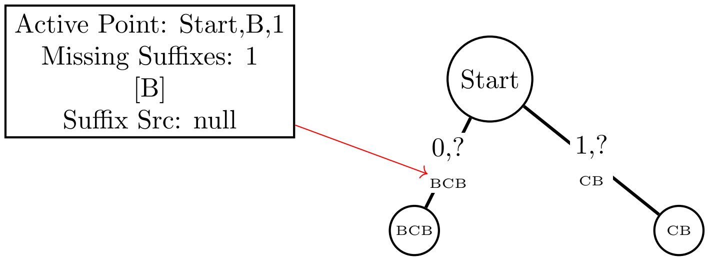


Figure 9.94: Step 3: Root does have an edge B, so move the active point

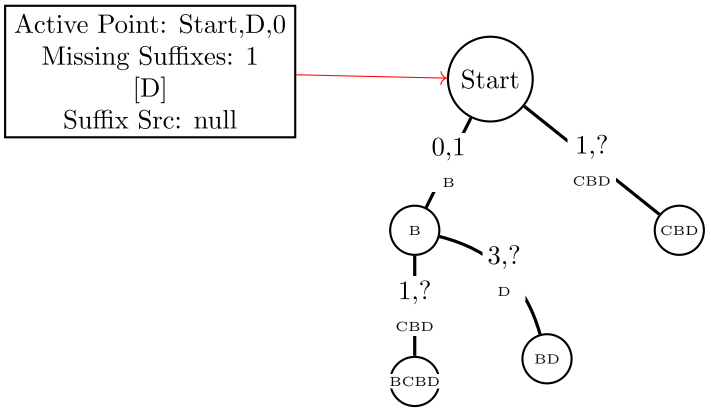


Figure 9.95: Step 4 part 1: The D mismatches, create a new edge. As there is no suffix link, the active point reverts to the root, where we adjust the active edge to D, and subtract 1 from the acttive length. The new node B represents a string of length 1, so we do not set it as the suffix src. Note: we would do so if we intended to create suffix links to the root.

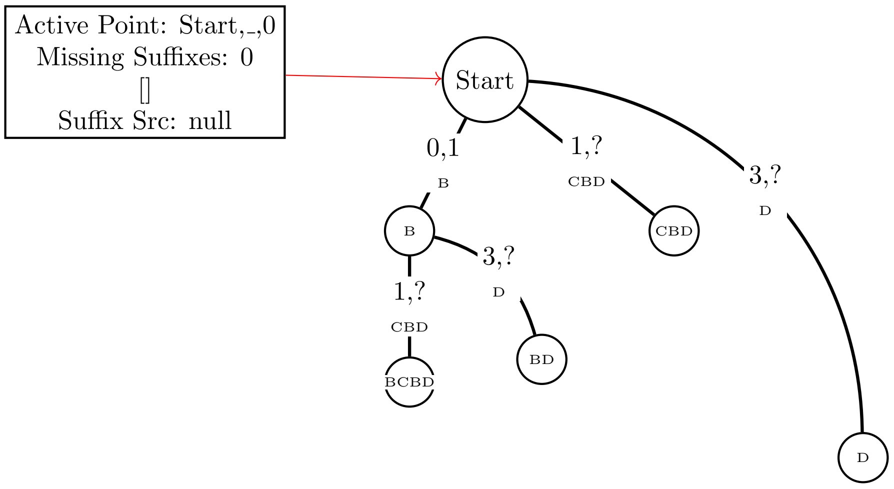


Figure 9.96: Step 4 part 2: Root does not have active edge D, so create it.

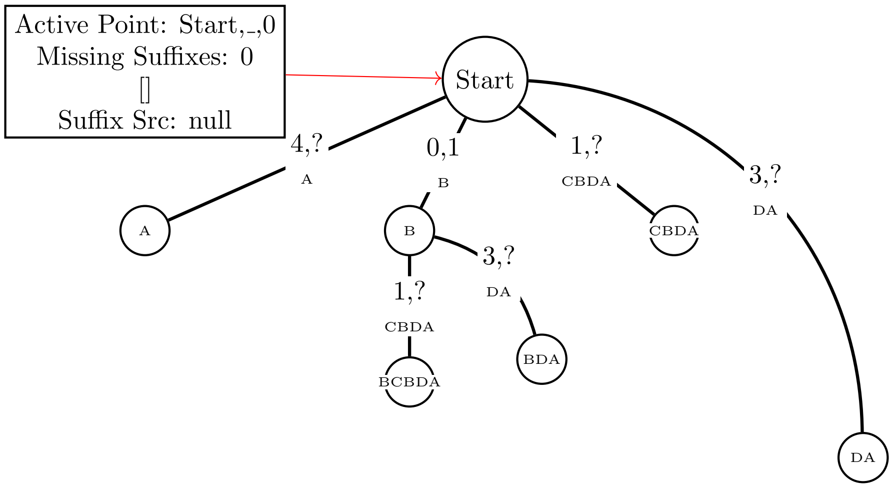


Figure 9.97: Step 5: Root does not have an edge A, so create it.

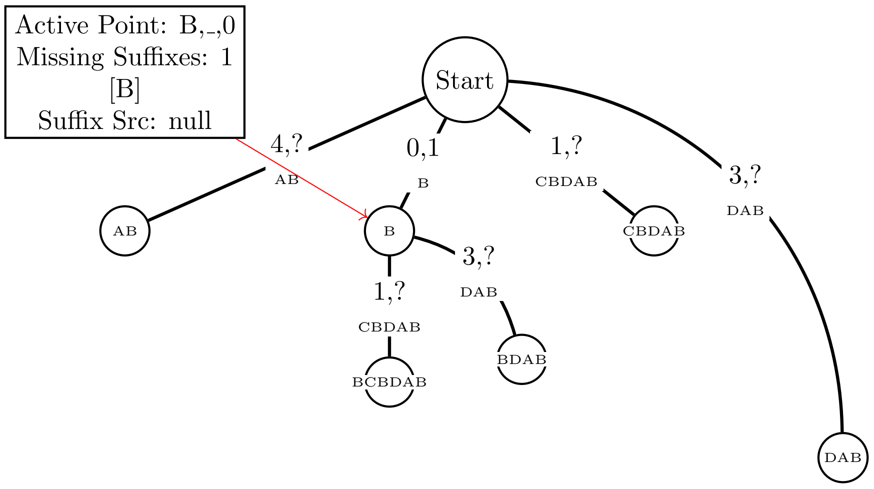


Figure 9.98: Step 6: Root does have an edge B, so move the active point on edge B, length 1. The length of the active edge matches the length of the edge it points to, so move the active point to the next node, and adjust the edge and length accordingly.

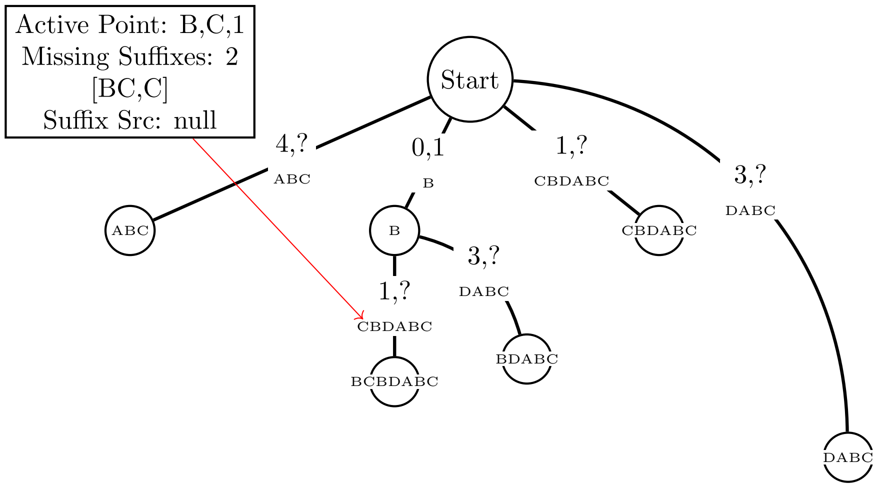


Figure 9.99: Step 7: The active node does have an edge C, so move the active point.

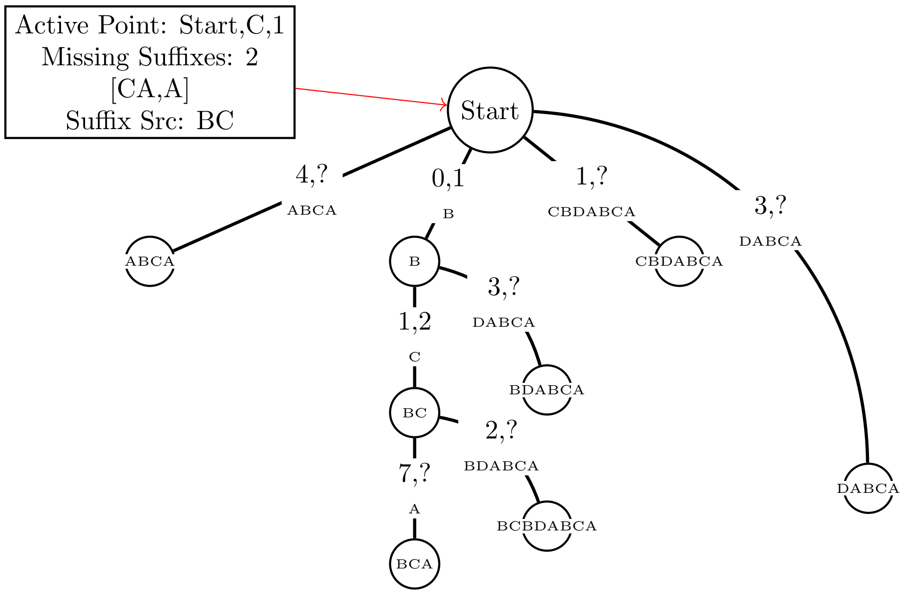


Figure 9.100: Step 8 part 1: The A does not match the next character after the active point. We split the edge at node BC. As BC is length greater than 1, we set it as the suffix src. We know it will end up with a suffix link to a node C. As the active node, B, has no outgoing suffix link, the active point reverts to the root. The next suffix is CA, so the active edge is C, and the length is 1 (i.e. we have to traverse to C (a string of length 1) to find where to insert leaf A).

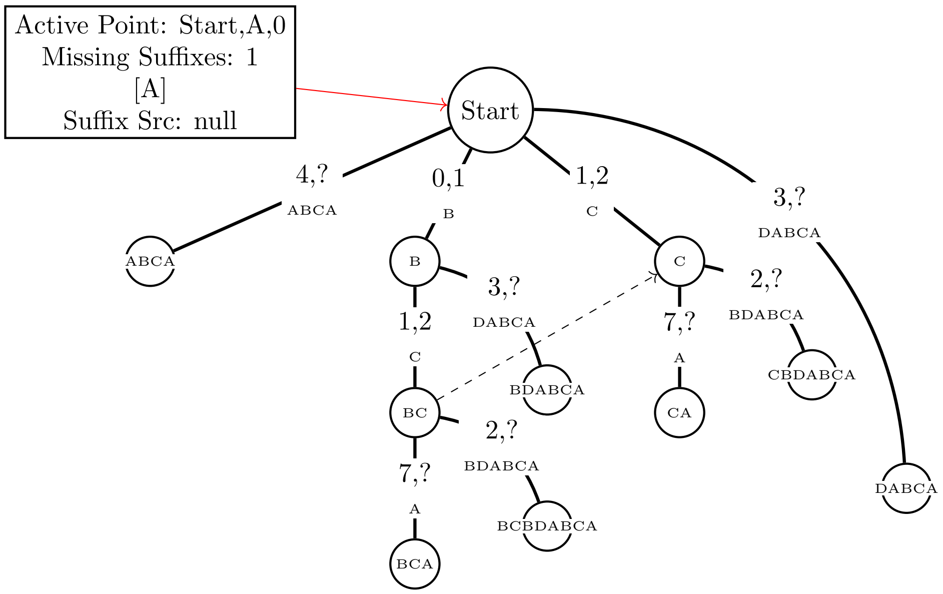


Figure 9.101: Step 8 part 2: We see that the active edge does exist from the active node, and the length of that edge (7) is less than the active length (1). We therefore have found our split location. We split the edge. We see that there was a suffix src, so we draw the suffix link from BC to B. This new node represents a string of length 1, so it is not set as the suffix src. The active node remains at the root, and we update it for the next suffix to insert.

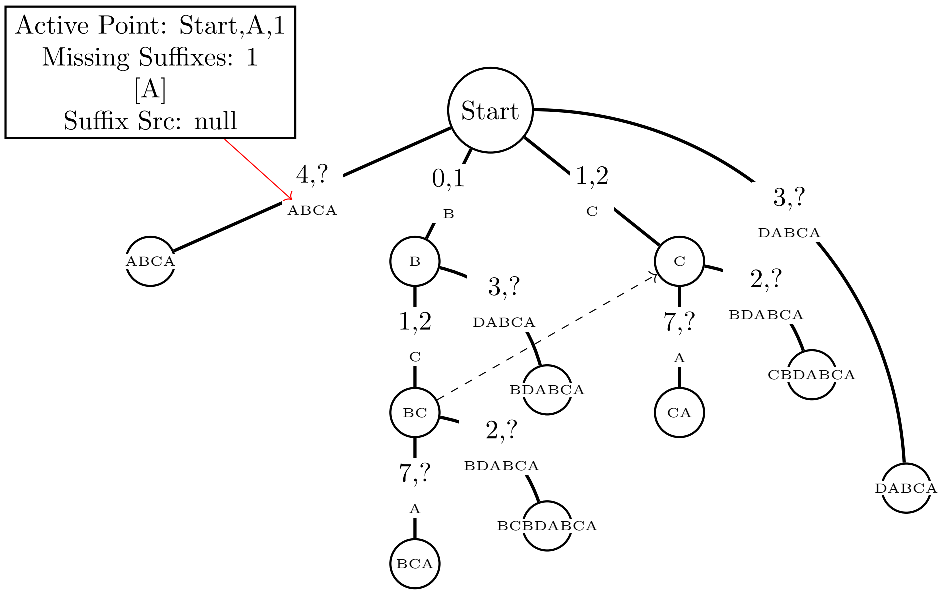


Figure 9.102: Step 8 part 3: There is only the A remaining, which means there is no traversal to do before inserting it (represented by the active length having been 0). We therefore simply note that the activenode already an edge A, so we simply move the active point.

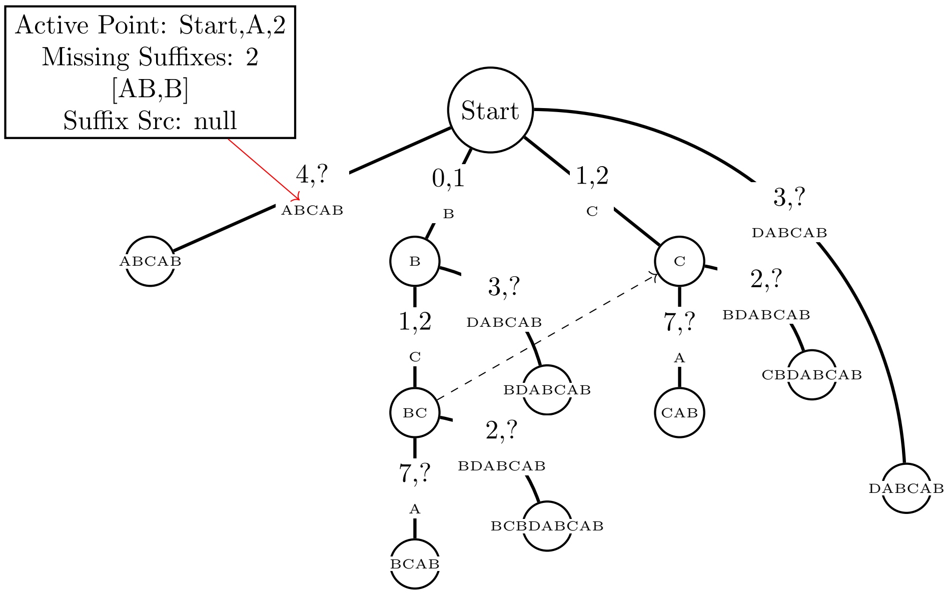


Figure 9.103: Step 9: The next character, B, matches the character after the active point, so we simply move it.

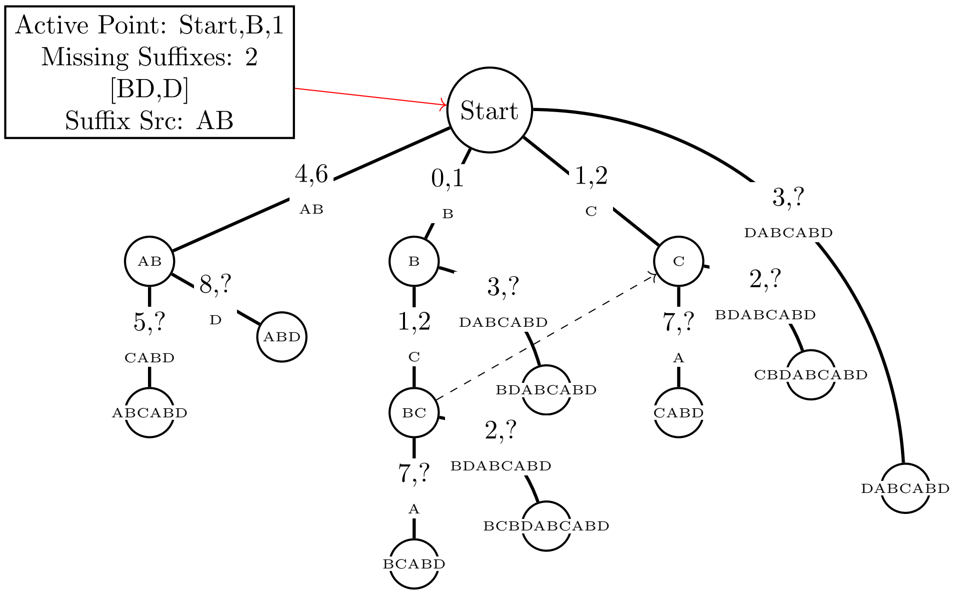


Figure 9.104: Step 10 part 1: The next character, D, does not match the character after the active point. We split this edge. We note that AB as the suffix src, as it will end up with a suffix link to B, and we update the active point to the root, since we don't have an outgoing suffix link to follow.

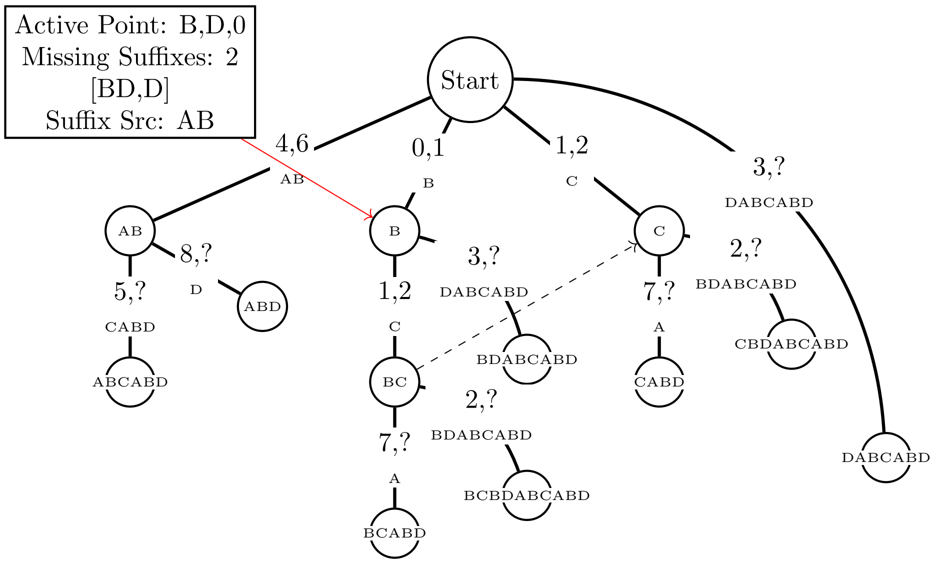


Figure 9.105: Step 10 part 2: The active edge has a length shorter or equal to the active length. We therefore move the active point to that node.

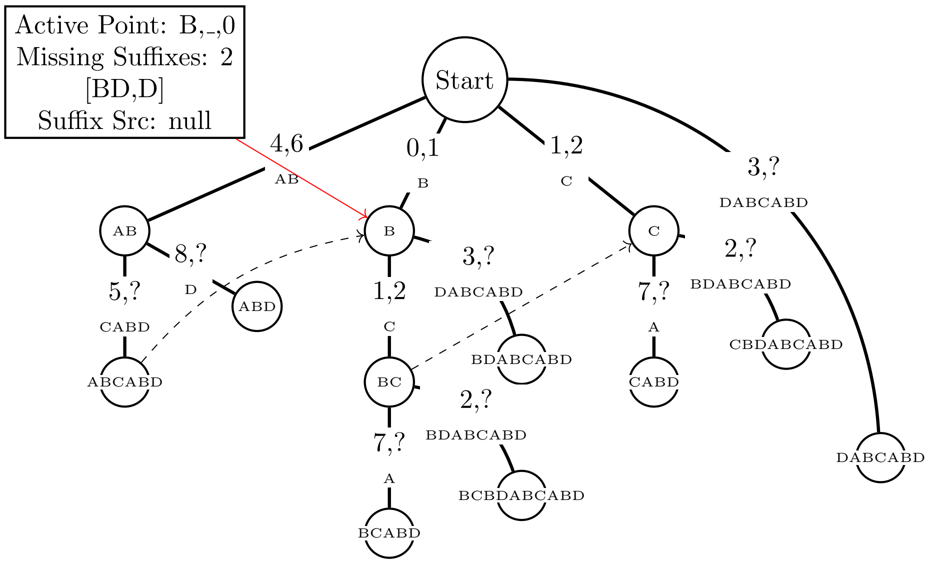


Figure 9.106: Step 10 part 3: We have found the location (B) in the tree where we would add BD. We know this since our active length is 0. A node already exists here, however, so there is no need to perform a split. We will draw our suffix link from the suffix src, however.

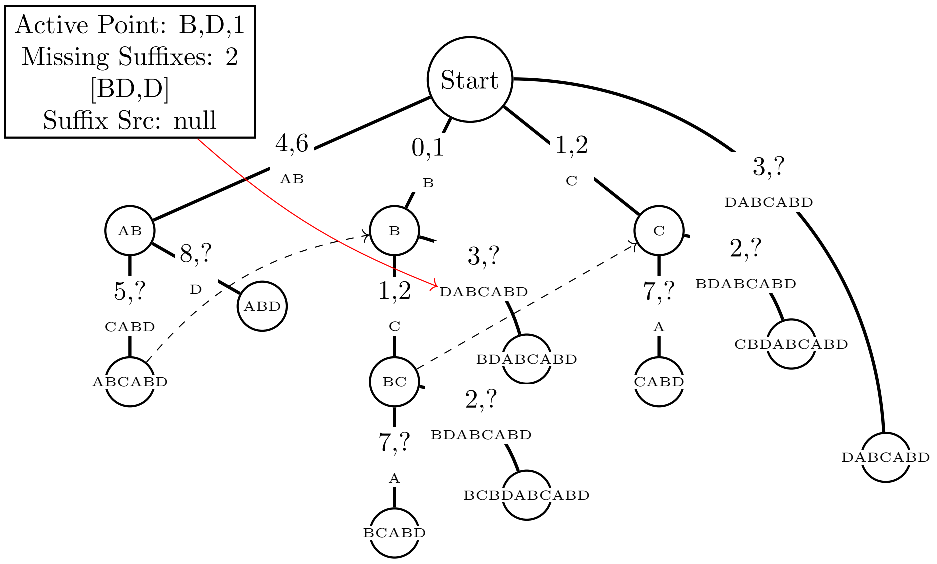


Figure 9.107: Step 10 part 4: When we go to finally create the leaf, we find node B already has an edge for D, so we simply move the active point, and are complete with this step. It may seem unusual that after a split we do not end up back at the root, but as with the original justification for non-committal suffixes, we do not know if we will have to make a split or not without looking ahead. As all suffixes are still properly tracked in either leaves or the missing suffix list, we are all set.

Step 11 is the insertion of the terminal character \$. As this section is already too long with diagrams, the parts of step 11 are described here.

1. \$ does not match the next character after the active point, so we split the edge, creating node BD. We set BD as the suffix src. As there is no outgoing suffix link from the ative node, the active point reverts to the root with edge D and length 1.

2. There is an outgoing edge D from the root, and it is longer than the active length 1. We split this edge to form node D, create the leaf node for \$, and draw the suffix link from our suffix src BD.

9.4.4 Implementation

The implementation of Ukkonen's algorithm is frought with corner cases that each must dealt with to avoid incorrect trees or accidental quadratic runtime. We provide some high-level guidance on implementing the algorithm followed by pitfalls.

The implementation must hold many pieces of state to encode the tree and other information necessary for its construction.

- An array storing the tree itself. It maps one node to another via the first character on the edge to that node.
- An array storing each node's parent node.
- An array storing the suffix link of each node.
- Arrays storing the information about each edge. These ultimately store a substring of the input string based on its start and end index. The arrays are indexed by the destination node of the edge. If the substring runs to the end of the string, the end index is represented by `INT32_MAX`.
- The active point, encoded by the active node, the active branch (representing the destination node of the branch we are on),[42] and the active length or number of characters along the active branch we have traversed.
- The number of currently used nodes. As we are encoding the tree as an array, rather than individual structs, we have to know the index of the next element to use. The number of used nodes allows us to identify the next entry to use.
- The number of remaining suffixes we have not yet finalized in the tree. Recall that when we have an active point not pointing at the root, it means we have processed some character, but haven't yet generated all the nodes that represent that suffix (all the way to its leaf node). This data informs us how many the current active point represents. This allows us to know which suffix we are inserting next when we do have a miscompare at the active point.
- The most previously created node. This is necessary as it will be the source node of the next drawn suffix link.

There are many ways in which the algorithm can be implemented overall. The author does the following in each phase of the algorithm:

1. Properly set the active point by adjusting the values appropriately if the active length is longer than the active edge, or if we are at a node which has a matching outgoing branch, but have not yet set the active branch.

[42]We could just as validly use the first character of this branch.

2. If the next character and matches at the active point, proceed to the next character.

3. If the next character does not match, take two actions:

    (a) If we are currently in the middle of a branch, create a new node in the middle of this branch. This is the parent node of the new branch. Otherwise, the node we are currently at will be the parent node of the new branch.

    (b) Create a new leaf linked to the just identified parent, representing the new branch and this suffix.

    When this occurs, we must properly link the parent node of the new branch as the destination of a new suffix link (if necessary), and then follow any existing suffix link for the next suffix insertion. Recall after a miscompare, we must insert any remaining suffixes rather than advancing to the next character of the input string.

Even when understanding these steps there are many corner cases that must be dealt with appropriately. We provide a non-exhaustive list of some common ones:

- After following a suffix link, we find that the next intended split point already exists. This pre-existing node may[43] or may not[44] have the branch we would have otherwise created. Both cases should be handled.

- New branches can arise from the same node multiple times. Suffix links must be followed properly in such cases.

- Multiple cases occur when adjusting the active point after a new branch creation:

    – If the suffix link reaches some non-root node, the length and branch stay the same, though may end up adjusted if the length ends up longer than the length of the new active branch.

    – If the suffix link reaches the root node, and the branch does not exist, the length and branch are now 0. This can only happen when inserting the last character of a given suffix.

    – If the suffix link reaches the root node, and the branch does exist, the length decreases by 1, and the active branch is represented by the first character of the next remaining suffix to insert.

As expected, most of the corner cases revolve around managing suffix links. The most important fact to remember is every internal node must have a suffix link, even if it points back to the root. The source of the suffix link is either

[43]"bcbdabcabd" must have a suffix link from the ab node to the b node after insertion of suffixes abd and bd.

[44]"abadbax" must have a suffix link from ba to a after the insertion of bax and x.

the parent of the edge which we split, or the node where we generated the new branch, if that node previously existing. After following the link, the location where the next split occurs or may not exist already and all cases must be handled.[45]

An effective way to debug an implementation is to generate a random string, concatenate it to itself, and validate the longest repeated substring in the concatenated string is the original random string.[46] This may not reveal all problems, but will likely resolve most. Scaling the length of the random string enables finding the shortest inputs which are likely to break the code, enabling easier debugging.

Other important notes:

- All input characters must be within some fixed range. The value of `#define A`, and the offset used, `int c=s[i]-'a';` may have to be adjusted to accomadate different ranges.
- Entered strings should have a termination character. As with other characters, these should be within the range of the defined alphabet size. As specified (with an alphabet size of 27), '' should be used as it is the next character following the set of lowercase letters.[47]'.
- The memory required to build a suffix tree is substantial for many problems that require it. This may result in confusing runtime errors if the allotted stack or heap size for the program is not large enough. Be sure to execute the program with enough available memeory.

Listing 9.10: C++

```
#define N 100
#define A 28 //enough for two terminating chars
struct suffix_tree{
  string s;
  int st[2*N][A]={}; //the adjacency list of the tree
  int parent[2*N]={};
  int suffix_link[2*N]={};
  int next_node=1; //the next unused node index
  int active_node=0,active_branch=0,active_length=0;
  int remaining=0; //count of suffixes not fully inserted in the tree
  int last_created=0; //last split node, used as source of suffix link
  //substring of the edge ending at this node
  int start_index[2*N]={},end_index[2*N]={};
  suffix_tree(string s):s(s){
    for(int i=0;i<s.length();i++){ //main loop over each char
      last_created=0; //don't create suffix links across rounds
```

[45]Note: Many online resources exist, including visualizations, do not properly create suffix links in all cases.

[46]Note that there are corner cases where it is longer, but these can be validated via brute force.

[47]Subsequent terminators would be '—' and '

```
        int c=s[i]-'a'; //convenience
        remaining++;
process: //entrypoint after following suffix links

        //Adjust active point if is longer than this edge
        while(active_branch&&
              active_length>=
              end_index[active_branch]-start_index[active_branch])
        active_length-=end_index[active_branch]-
                     start_index[active_branch],
        active_node=active_branch,
        active_branch=st[active_node][s[i-active_length]-'a'];

        //Set active branch if we can
        if(!active_length&&st[active_node][c])
           active_branch=st[active_node][c];

        //found a match. move active point
        if(active_branch&&
           c==s[start_index[active_branch]+active_length]-'a'){
           active_length++;
           //corner case where next suffix already exists.
           if(last_created)suffix_link[last_created]=active_node;
        }else{ //Mismatch. create some new nodes
           int temp=active_node; //cache the previous parent
           if(active_length){ //in the middle of an edge. Create new
                parent
              start_index[next_node]=start_index[active_branch];
              end_index[next_node]=
                 start_index[active_branch]+active_length;
              start_index[active_branch]=end_index[next_node];
              st[active_node][s[start_index[next_node]]-'a']=next_node;
              st[next_node][s[start_index[active_branch]]-'a']=
                 active_branch;
              parent[next_node]=active_node;
              parent[active_branch]=next_node;
              temp=next_node++;
           }
           // create the new leaf node
           start_index[next_node]=i;
           end_index[next_node]=INT32_MAX;
           st[temp][c]=next_node;
           parent[next_node++]=temp;
           if(last_created)suffix_link[last_created]=temp;
           last_created=temp;
           remaining--;
           //reset the active point
           active_node=suffix_link[active_node];
           //keep same, or decrement if back to root
           active_length=active_node?active_length:max(remaining-1,0);
```

```
            //no active length means no active branch. otherwise, set it
            //to the correct next branch based on being at the root or
            //not the root
            active_branch=active_length?
               active_node
                  ?st[active_node][s[start_index[temp]]-'a']
                  :st[0][s[i-active_length]-'a']
               :0;
            if(remaining)goto process;
         }
      }
   }
};
```

9.4.4.1 Helper Functions

Several bits of auxiliary information are necessary for implementing the following use cases. As these are not required for every use case, they are not included with the default implementation. They should be called once after tree construction if the use case requires it.

Depth Several use cases require knowing the depth in one of two ways:

1. the number of characters required to traverse to any specific node
2. the number of nodes required to traverse to any specific node

These two values can be computed in a single linear pass of the tree.

Listing 9.11: C++

```
int depth[2*N]={},depth2[2*N]={};
void calc_depth(){
   queue<int>q;
   q.push(0);
   while(!q.empty()){
      int on=q.front();
      q.pop();
      for(int i=0;i<A;i++)if(st[on][i]){
         int next=st[on][i];
         depth[next]=depth[on]
                  +min(end_index[next],(int)s.length())
                  -start_index[next];
         depth2[next]=depth2[on]+1;
         q.push(next);
      }
   }
}
```

Leaf Indices The construction of the tree allows us to find, for a given node, where it lies within the original string based on the depth and end index which arrives at that node. It does not however, allow us to map from a suffix in the original string to a node in the tree without walking the tree each time. We can resolve this problem by building an array which maps a suffix in the original string to the leaf node which represents that suffix by walking all the nodes, caching leaf nodes and where they point to.

Listing 9.12: C++

```cpp
int leaf_node[2*N]={};
void calc_leaves(){
   for(int i=0;i<2*N;i++)
      if(end_index[i]==INT32_MAX)
         leaf_node[s.length()-depth[i]]=i;
}
```

Binary Lifting Metadata As we will see, computing the LCA of two nodes is a common necessity. Therefore we need the binary lifting metadata to do this in sub-linear time. This is computed as it would be in any other tree.

Listing 9.13: C++

```cpp
vector<int>bl[2*N];
void calc_bl(){
   queue<int>q;
   q.push(0);
   while(!q.empty()){
      int on=q.front();
      q.pop();
      for(int i=0;i<A;i++)if(st[on][i]){
         bl[st[on][i]].push_back(on);
         for(int j=0,h=on;j<bl[h].size();h=bl[h][j++])
            bl[st[on][i]].push_back(bl[h][j]);
         q.push(st[on][i]);
      }
   }
}
```

Printing Nodes One last common helper is to print the string that a given node represents. We know the length of the string to arrive at any node based on its depth, and we also know the index into the input string which the path to that node ends at (based on the end index). This leaves a rather straightforward way to extract the substring itself.

Listing 9.14: C++

```
string to_string(int node){
  return s.substr(end_index[node]-depth[node],depth[node]);
}
```

9.4.5 Usage

Now that we have created a suffix tree in linear time, we can discuss how to solve the previously-listed problems.

9.4.5.1 Pattern Existence

The suffix tree does not simply contain all suffixes. A suffix only occurs if we end a traversal at a leaf node. As any substring of an input must be the prefix of the suffix starting at the same location, it must be found in the traversal of the suffix tree towards the leaf node representing that suffix. Consider the substring DAB. This substring is a prefix of DABCEA, which is a suffix of ABCABDABCEA, and therefore is found in the trafersal towards that suffix's leaf node in the suffix tree.

So check if a pattern exists in a string, we can traverse the tree using the pattern. If we are still traversing when we consume all the characters of the pattern (i.e. we have not found a mismatch), the pattern must exist in the string.

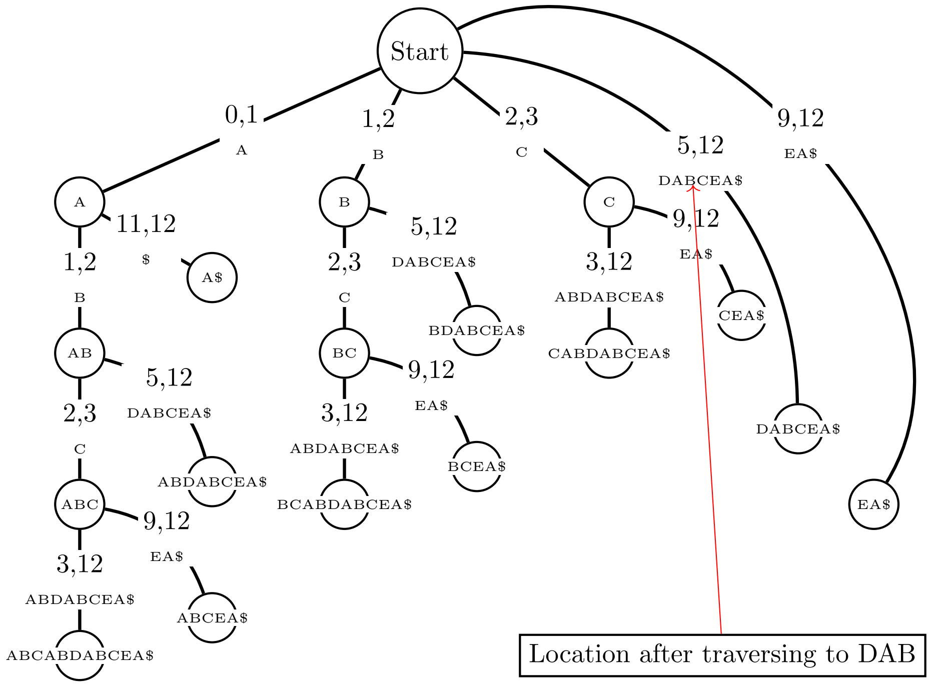


Figure 9.108: We traverse to DAB and find it is represented on some edge in the tree. Therefore we can declare it exists in the original string.

As finding a single instance reduces to finding all instances of a pattern, the implementation is shown in that section.

9.4.5.2 Finding All Instance of a Pattern

Every leaf node in the tree represents a suffix, and every suffix starts at a different location. The number of times a pattern occurs in a string is equal to the number of suffixes which start with that pattern. Therefore, the number of times a pattern occurs in a string is equal to the number of leaf nodes below the traversal to the pattern. For instance, AB occuurs three times in the input string, and we see it has 3 leaf nodes below its traversal.

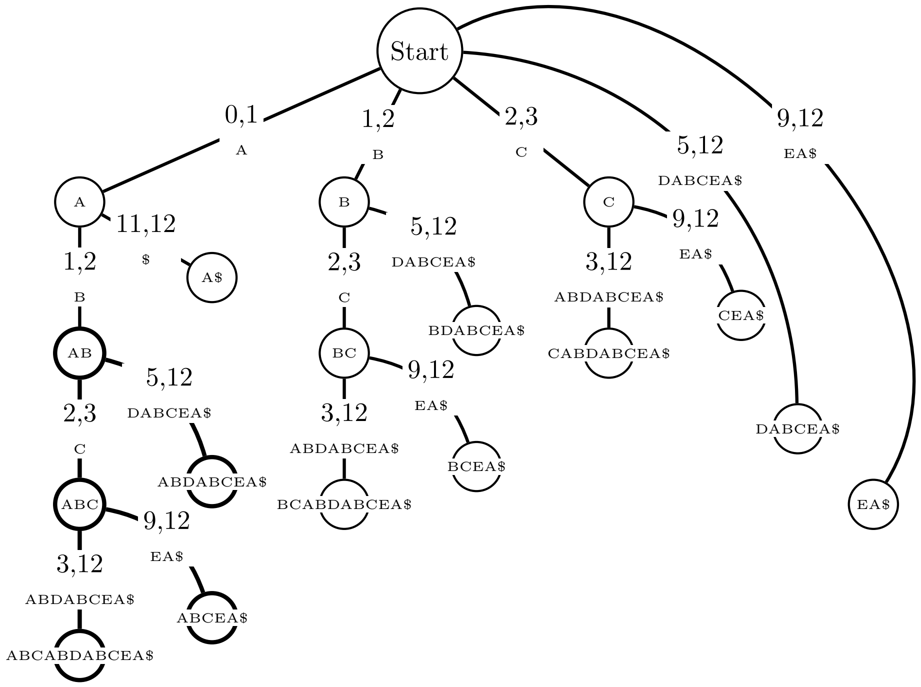


Figure 9.109: The subtree below AB is highlighted. Note there are 3 leaf nodes, representing the 3 locations of AB in the input string.

We can identify the leaf nodes with a simple BFS from the location of the pattern in the tree. Note that there at most $2 * N$ nodes, so this is sufficiently efficient. If multiple lookups are required, the number of leaf nodes could be precomputed for all nodes in the tree.

If we must recover the actual locations of the pattern, we can note the total length of the suffix of each leaf node we reach in our BFS traversal, which tells us where that suffix, and thus the searched pattern, starts. This is unnecessary if we have previously labeled all leaf nodes with their start index.

Listing 9.15: C++

```
//returns index of all locations where the pattern occurs
vector<int> find_all(string pattern){
   vector<int> ans;
   int an=0,al=0,ab=0;
   for(int i=0;i<pattern.length();i++){ //Traverse to end of string
      int c=pattern[i]-'a';
      if(!ab&&!st[an][c])return ans;
      if(!ab)ab=st[an][c];
      if(s[start_index[ab]+al++]-'a'!=c)return ans;
      if(al==end_index[ab]-start_index[ab])an=ab,ab=0,al=0;
   }
   queue<int>q;//walk through remaining children
   q.push(al?ab:an);
   while(!q.empty()){
      int on=q.front();
      q.pop();
      if(end_index[on]==INT32_MAX)ans.push_back(s.length()-depth[on]);
      else for(int i=0;i<A;i++)if(st[on][i])q.push(st[on][i]);
   }
   return ans;
}
```

9.4.5.3 Longest Repeated Substring

Using similar rules to above, we can determine the longest repeated substring. Note again that the number of leaf nodes below a given point in the tree represent the number of times that substring occurs. Therefore, the longest repeated substring is the deepest node in the tree which has at least two leaf nodes (i.e. the deepest internal node). This can be done in a single BFS of the tree.

Listing 9.16: C++

```
int long_rep(){ //returns node that represents longest repeated string
   int ans,ma=-1;
   for(int i=0;i<2*N;i++)if(end_index[i]<INT32_MAX&&depth[i]>ma){
      ans=i;
      ma=depth[i];
   }
   return ans;
}
```

9.4.5.4 Longest Common Substring

The longest common substring between two input strings is equivalent to the longest repeated substring in the concatenation of those two strings which does not cross the boundary between them, and where the two instances are on opposite sides of the boundary. Example: The longest common substring of ABCDAB and CDEABCDE is ABCD. The longest repeated substring in the concatenation of those, ABCDABCDEABCDE, is ABCDE, but this requires us to cross the boundary between the two strings, which is not allowed. ABCD is the longest which does not cross that boundary.

We know how to compute the longest repeated substring, but how do we ensure it adheres to the other conditions? This is solved by leaving a terminal character in place between the two substrings. We can extend the idea of the $ character, and concatenate the two strings, but enusre they each have a unique terminal: ABCDAB$CDEABCDE&. Now, the longest repeated substring is guaranteed not to cross the boundary, as the $ terminal character is guaranteed to only appear once in the concatenated string. If we were to build a suffix tree of this and search, we would find a longest repeated substring, but how do we also guarantee the two substrings occur on opposite sides of the boundary? When comparing the two leaf nodes, we can simply check that start location of thet two suffixes place them on either side of the boundary.

The high level algorithm is as follows:

1. DFS through the tree
2. Note the distance to the root as we perform the DFS
3. When returning from the recursion, note whether there are leaf nodes at or below this node which are on the left side of the boundary, right side of the boundary, or both.
4. Store the maximum depth node which was found which has "both"

This concept of storing multiple strings with different terminal characters together is known as a *generalized suffix tree*. Note that we can make its usage slighlty more convenient by noting that every suffix starting in the first substring will contain the entire second substring. We can truncate these extra characters so that the leaf node terminates at the proper terminal character (either $ or & in our example).

When implementing LCS, be sure to use appropriate string terminators and adjust the alphabet size in the base suffix tree class appropriately. The following implementation assumes input is of the form "string1string2—". We assume the depth is computed beforehand. The two child arrays store whether a node has a leaf node which is on the left or right hand side of the boundary.

Listing 9.17: C++

```
int s2_index;
bool s1_child[2*N]={},s2_child[2*N]={};
void lcs_helper(int on){
   if(end_index[on]==INT32_MAX){
      if(depth[on]<=s.length()-s2_index)s2_child[on]=true;
      else s1_child[on]=true;
   }
   for(int i=0;i<A;i++)if(st[on][i]){
      lcs_helper(st[on][i]);
      s1_child[on]|=s1_child[st[on][i]];
      s2_child[on]|=s2_child[st[on][i]];
   }
}
int long_common_substr(){//assume inserted as S{T|. ans is a node
   for(int i=0;i<s.length();i++)if(s[i]=='{')s2_index=i+1;
   lcs_helper(0);
   int ans,md=-1;
```

```
  for(int i=0;i<2*N;i++)if(depth[i]>md&&s1_child[i]&&s2_child[i]){
    ans=i;
    md=depth[i];
  }
  return ans;
}
```

9.4.5.5 Longest Common Extension

Consider the following question: Given indices i and j in an input string, how many consecutive characters following i and j match eachother? This is known as the longest common extension (LCE) problem. Assuming we have identified the leaf nodes corresponding to the suffixes starting at the two indices (which we could do in a single pass of the tree if necessary), the LCE is the longest common prefix of those two suffixes, which can be found trivially by walking the tree from the root towards the two leaves until the path splits.

A single query of LCE is not particularly interesting (and could be done without the suffix tree), however, the useful case is when we have many queries, and for which a linear lookup may not be sufficient. To accommodate this, we notice that the point where the path to the two leaves splits is the lowest common ancestor of the two leaf nodes. We can utilize any standard LCA algorithm, however the linear time processing with constant time lookup algorithm is most appropriate here.

The following implementation uses a standard binary lift to compute the LCA and assumes the binary lifting metadata has been precomputed.

Listing 9.18: C++

```
int long_common_ext(int a,int b){ //returns a length
  a=leaf_node[a],b=leaf_node[b];
  if(depth2[a]<depth2[b])swap(a,b);
  for(int i=bl[a].size()-1;i>=0;i--)
    if(i<bl[a].size()&&depth2[bl[a][i]]>=depth2[b])
      a=bl[a][i];
  for(int i=bl[a].size()-1;i>=0;i--)
    if(i<bl[a].size()&&bl[a][i]!=bl[b][i])
      a=bl[a][i],b=bl[b][i];
  return depth[a!=b?parent[a]:a];
}
```

9.4.5.6 Longest Palindromic Substring

The longest palindromic substring is the longest substring within an input which is a palindrome. This is equivalent to the longest common substring between an input string and its reverse with an additional condition that the substring occurs in a matching locations in the string and its reverse. To see why this condition is necessary, consider the example ABADXYDABAZ. The longest palindromic substring is ABA, however, the longest common substring between this string, and its mirror (ZABADYXDABA), is ABAD, which is not a palindrome.

The issue is that ABAD and its mirror BOTH occur in the input string. This causes us improperly match those two strings. When looking at the string and its mirror, we note two things:

1. Each character has a "pair" in the mirrored string. For a character at index i, this is found at $N - i - 1$ in the mirror.

2. The common substrings in each the input and its mirror comprising the candidate palindrome must contain the same characters.

3. the candidate ABAD contains characters 0,1,2,3 in the input, which should be characers 10,9,8,7 in the mirror, but instead is found at 1,2,3,4, and therefore is invalid.

Therefore the complete location condition is that if index i is the start of the candidate of length l in the first string, then the start of the candidate in the mirror must occur at index $N - i - l$. We see that candidate ABA, at index 0, length 3, properly occurs at $11 - 0 - 3 = 8$ in the mirrored string, and is therefore a solution. Using longest common substring, we can easily identify the incorrect ABAD, but how can we impose the location condition to ensure we only find ABA? One possible solution is to cache on each internal node the start index of the leaf nodes you can reach from there for both the forward and reverse string. Then, as we later traverse the tree, we can see if any pair of those suffixes adhere to the location condition. The problem with this scenario is that it is super-linear. Even storing the leaf node "cache" on each node is quadratic overall.

Instead of walking the tree, we can consider all possible mid-points of a longest palindrome. There are exactly $2 * N - 1$: N odd centers, and $N - 1$ even. From each midpoint, we wish to determine the widest palindrome with that midpoint. This is equivalent to finding the LCE of the substring going to the right of the midpoint and the one going to the left. Assuming we have constructed a suffix tree as described above (with the forward, and mirrored string), then this is simply the LCE of the substring going to the right in the forward string, and the substring going to the left in the mirrored string. We can identify those two leaves, and execute the above LCE algorithm to determine the widest palindrome centered at that point in constant time. Looping over all possible centers gives us the necessary lienar time.

The implementation assumes the string is created as string1rev(string1)—.

Listing 9.19: C++

```
//assume inserted as S{rev(S)|. returns start and length
pair<int,int> long_palindrome(){
  pair<int,int>ans(0,0);
```

```
   for(int i=0;i<s.length()/2-1;i++){
      int t=long_common_ext(i,s.length()-i-2);//odd
      if(t>ans.second)ans={i-t+1,2*t-1};
      t=long_common_ext(i,s.length()-i-1);//even
      if(t>ans.second)ans={i-t,2*t};
   }
   return ans;
}
```

9.4.5.7 Minimum Rotation

As described earlier, the minimum rotation of a string is the least lexicographic substring of length N which occurs if we allow said substring to ”wrap around” to the beginning if we reach the end of the input. This is equivalent to the least such substring if the input is concatenated to itself. Example: the minimum rotation of BACAA is AABAC, which is also the longest length 5 substring in BACAABACAA.

Therefore, if we construct a suffix tree of such a concatenation, we can follow a path from the root, along the least lexicographic edge which comes from each node, until we have found a substring which is N characters long. Note that we are guaranteed not to get ”stuck” in a branch with not enough characters, as every suffix of fewer than N characters also has a ”partner” with N more characters, and therefore there must be somewhere to traverse. (e.g. suffix AA has a partner AABACAA. Suffix CAA has a partner CAABACAA).

Listing 9.20: C++

```
int minrot(){//returns start index of minimum rotation
   int on=0;
   while(end_index[on]!=INT32_MAX)for(int i=0;i<A;i++)if(st[on][i]){
      on=st[on][i];
      break;
   }
   return s.length()-depth[on];
}
```

9.4.5.8 Counting Unique Substrings

We know that the suffix tree contains each substring of the input string exactly once. Therefore, we can count all the substrings in a single walk of the tree. In an uncompressed suffix tree, this would simply be the number of nodes, but given the compressed structure, we must increment the count not for every node, but for every substring which could terminate in the middle of the edge to that node. This means we simply add the length of the edge when we traverse to a node, instead of just 1.

Listing 9.21: C++

```
long long substr_cnt(){
   long long ans=0;
   queue<int>q;
   q.push(0);
   while(!q.empty()){
      int on=q.front();
      q.pop();
      for(int i=0;i<A;i++)if(st[on][i]){
         int next=st[on][i];
         ans+=min(end_index[next],(int)s.length())-start_index[next];
         q.push(next);
      }
   }
   return ans-s.length();
}
```

9.4.6 Suffix and LCP Array

As we have seen, suffix trees can solve a vast array of string problems in constant time. Despite that, they have three major shortcomings:

- Suffix trees have a significant memory cost, requiring several arrays sized at twice the length of the string to cacahe all the necessary metadata for each node.
- Suffix trees have a significant constant factor, leading to potentially slow runtimes despite the linear complexity. Tree traversal is slow in general, and suffix trees will always have double the number of nodes as characters in the string.
- Suffix trees are a lot of code to type, which is difficult in contests where templates are not allowed, and may be prohibitive in contests where no references are allowed at all.

To that end, we explore other data structures which solve similar problems, but without the above issues. One such structure is the *Suffix Array.*

Suffix arrays are sorted lists of all suffixes of a string. For example, the suffix array of the string "ABRACADABRA" is as follows:

Suffix	Index
A	10
ABRA	7
ABRACADABRA	0
ACADABRA	3
ADABRA	5
BRA	8
BRACADABRA	2
CADABRA	4
DABRA	6
RA	9
RACADABRARA	2

How might such a structure be useful? Consider if we wished to find every instance of a given pattern in the string. As we saw with suffix trees, we know every instance of the pattern in the string is a prefix of some suffix. As the suffixes are ordered alphabetically, we could binary search for the first and last suffixes which have the pattern as a prefix, and know for sure all suffixes between those two also contain the pattern.

Finding the longest common substring is equivalent to finding the pair of consecutive suffixes in the array who share the longest common prefix. It turns out knowing the length of the common prefix is so essential to utilizing suffix arrays, it is almost ubiquitously computed with the suffix array and given its own name, the longest common prefix array, or *LCP Array.*

We extend our example to include this value:[48]

Suffix	Index	LCP
A	10	1
ABRA	7	4
ABRACADABRA	0	1
ACADABRA	3	1
ADABRA	5	0
BRA	8	3
BRACADABRA	1	0
CADABRA	4	0
DABRA	6	0
RA	9	2
RACADABRARA	2	0

9.4.6.1 Construction

Astute readers will observe that the inices which represent the suffix arrays can be acquired by simply walking a suffix tree in alphabetical order, and the LCP values are just the heights of the LCA of consecutive leaf nodes. Therefore, we can "trivially" construct the suffix and LCP arrays in linear time by building a suffix tree and walking it. Such a technique, while correct, is at least as bad as constructing the suffix tree in the first place, and therefore defeats much of the benefit of the suffix array.

While there are ways to directly compute suffix and LCP arrays in linear time, we instead focus on an $O(n * log(n))$ algorithm, which due to its intuitive nature and straightforward implementation has made it a ubiquitous choice to complement the more complex suffix tree construction.

We will first see how to construct the suffix array, then subsequently construct the LCP array from the suffix array.

Sorting Rotations It should first be apparent that the order of the suffix array is the same as the order of the rotations of the string. We can make this explicitly clear by inserting a terminator character, "$", which is lexicographically less than any character in the string.

Sorting these rotations naively with any old comparison sort requires $O(n * log(n))$ comparisons. With linear time for each comparison, this results in $O(n^2 * log(n))$.[49] In such cases, moving to radix sort is often a potential improvement path. Recall, however, that radix sort is proportional both to the number of items and the length of each item. As both are n in this case, the overall runtime is $O(n^2)$.

For the remainder of this algorithm construction, we will consider sorting rotations and suffixes as equivalent.

[48]Consecutive suffixes BRA and BRACADABRA share a common prefix of length 3. Therefore the LCP value is 3.

[49]This could be improved with hashing to $O(n * log^2(n))$.

Radix Sort Fundamentals To move forward, lets take a deeper look at the fundamental properties which underly radix sort, specifically most significant digit first radix sort.[50]

Each step of an MSD radix sort has two pieces of information which are used:

1. The exact piece of information which enables a given item to be placed in the correct bucket. This is typically a single character or digit. Buckets are labeled with all posible values this piece of information can take.
2. Which bucket an item was in after the previous iteration. In an MSD radix sort, items can only swap places with other items which were in the same bucket as last time.[51]

After a given iteration, we use these two pieces of information to build an invariant, namely, after an iteration i, items are sorted by their first l digits or characters.

Lets demonstrate this with all rotations of our example string. In the first iteration ($i = 1$), we place words in buckets based on their first character. The invariant indicates (correctly) that words are sorted by order of their first letter.

Bucket	Rotations
$	$ABRACADABRA
A	ABRACADABRA$ ACADABRA$ABR ADABRA$ABRAC ABRA$ABRACAD A$ABRACADABR
B	BRACADABRA$A BRA$ABRACADA
C	CADABRA$ABRA
D	DABRA$ABRACA
R	RACADABRA$AB RA$ABRACADAB

In the second iteration, within the scope of a given $i = 1$ bucket, we place words in buckets based on their second character. The invariant incidates words are now sorted by their first two letters. To communicate this, we simply label buckets with the two characters which prefix all strings that bucket contains.

[50]MSD

[51]Consider the numbers 198 and 245. After 198 is placed into the 1 bucket for the hundreds digit, and 245 in the 2 bucket, they cannot swap places later when tens digits are compared, where $9 > 4$.

Bucket	Rotations
\$A	\$ABRACADABRA
A\$	A\$ABRACADABR
AB	ABRACADABRA\$ ABRA\$ABRACAD
AC	ACADABRA\$ABR
AD	ADABRA\$ABRAC
BR	BRACADABRA\$A BRA\$ABRACADA
CA	CADABRA\$ABRA
DA	DABRA\$ABRACA
RA	RACADABRA\$AB RA\$ABRACADAB

In the third iteration, within the scope of a given $i = 2$ bucket, we place words in buckets based on their third character. Words are now sorted by their first three characters.

Bucket	Rotations
\$AR	\$ABRACADABRA
A\$A	A\$ABRACADABR
ABR	ABRACADABRA\$ ABRA\$ABRACAD
ACA	ACADABRA\$ABR
ADA	ADABRA\$ABRAC
BRA	BRACADABRA\$A BRA\$ABRACADA
CAD	CADABRA\$ABRA
DAB	DABRA\$ABRACA
RA\$	RA\$ABRACADAB
RAC	RACADABRA\$AB

This process continues until after the last iteration, the strings are sorted based on their entirety.

Optimizing for Repeated Strings Imagine we wished to speed up the above algorithm in the general, non-repeated case. We could do so by processing l characters at a time. This would reduce the number of rounds we have to execute by a factor of l, but at best, we must do l extra work to place a given string in a bucket. Even more so, we must either have 27^l buckets,[52] or extra time to sort the buckets.[53] Neither ends up with a better runtime.

The fundamental problem is after we have bucketed based on l characters, we have no information about the next l characters of each string, and thus must still examine them all.

[52]incurring prohibitive memory cost
[53]If lazily allocated

This declaration is not true when we are sorting rotations of a string. When bucketing rotations based on the l characters starting at index i, we must have previously bucketed based on the l characters starting at index $i - l$. Therefore, the l characters beginning at i in some rotation r must have already been examined in rotation $r + l$ at index $i - l$.

For example, consider $l = 3$. When we are bucketing rotation $r = 0$ ABRACADABRA$ in the second iteration, we are considering characters ACA. These same three characters must necessarily be the first three characters of rotation $r = 3$ ACADABRA$ABR.

This may seem inconsequential, but consier the following two facts:

1. In round i/l, we can immediately determine the bucket a rotation r should go in. As the l characters at i in r are the same as the l characters at $i-l$ in $k + l$,[54] the bucket r does into in this round is the same as the bucket $k + l$ went into in the previous round.[55]

2. The total number of buckets is $O(n)$. A string contains $O(n)$ l-tuples, and rotating by a single character introduces at most one additional, leading to $O(n)$ total.

These two factors enable us to actually realize speedup by bucketing. Let's consider sorting our example with $l = 3$.

The sorted list of all size-3 buckets we need is AB, AA, ABR, ACA, ADA, BRA, CAD, DAB, RA$, RAC. After the first round, we bucket based on the first 3 characters in each rotation. We note both the index of the buckets (in lexicographic order) and the rotations.

Bucket	Index	Rotation	Index
$AB	0	$ABRACADABRA	11
A$A	1	A$ABRACADABR	10
ABR	2	ABRACADABRA$	0
		ABRA$ABRACAD	7
ACA	3	ACADABRA$ABR	3
ADA	4	ADABRA$ABRAC	5
BRA	5	BRACADABRA$A	1
		BRA$ABRACADA	8
CAD	6	CADABRA$ABRA	4
DAB	7	DABRA$ABRACA	6
RA$	8	RA$ABRACADAB	9
RAC	9	RACADABRA$AB	2

The second round will bucket based on the 3 characters starting at index 3 in each string. Let's examine bucket 2, ABR.[56] The first string we consider,

[54]assuming we generate rotations by removing the first character and placing it at the end of the string

[55]of course, within the appropriate bucket from the previous round

[56]As one of two buckets which has more than one string in it.

rotation 0, has letters ACA at index 3. These are necessarily the same characters at index 0 in rotation 3. As that rotation went into bucket 3 in the first round, we know that rotation 0 must go in bucket 3 in this round.[57] Note that we only did constant time work to determine this bucket, and didn't look at a single extra character in rotation 0. We can apply this logic to all strings to perform the next bucketing in $O(n)$ total time.

For convenience, we label each bucket with the full string which specifies it, as well as the 3-tuple indices which label each step to arrive there.

Bucket	Index	Rotation	Index
\$AB,RAC	0,9	\$ABRACADABRA	11
A\$A,BRA	1,5	A\$ABRACADABR	10
ABR,A\$A	2,1	ABRA\$ABRACAD	7
ABR,ACA	2,3	ABRACADABRA\$	0
ACA,DAB	3,7	ACADABRA\$ABR	3
ADA,BRA	4,5	ADABRA\$ABRAC	5
BRA,\$AB	5,0	BRA\$ABRACADA	8
BRA,CAD	5,6	BRACADABRA\$A	1
CAD,ABR	6,2	CADABRA\$ABRA	4
DAB,RA\$	7,8	DABRA\$ABRACA	6
RA\$,ABR	8,2	RA\$ABRACADAB	9
RAC,ADA	9,4	RACADABRA\$AB	2

We could repeat this process, each in constant time, were there any remaining buckets with more than one string in them, but there are not so we terminate.

Avoiding Excess Bucket Iterations Naively, for each bucket from the parent iteration, we would create a set of new buckets and iterate through them to perform the stort. As there are $O(n)$ buckets after each round, splitting each into $O(n)$ new buckets could result in $O(n^2)$ runtime. We can avoid this by:

1. Adding each item into a global bucket for the new index.
2. Traversing the buckets in order, and placing each item in this order back in the bucket it originated from.

Let's take a quick look at what this looks like when sorting the second set of three characters. We first bucket based on the second three characters:

[57] larger umbrella of its current bucket 2, of course

Bucket	Index	Rotation	Index	Previous Bucket
$AB	0	BRA$ABRACADA	8	5
A$A	1	ABRA$ABRACAD	7	2
ABR	2	RA$ABRACADAB	9	8
		CADABRA$ABRA	4	6
ACA	3	ABRACADABRA$	0	2
ADA	4	RACADABRA$AB	2	9
BRA	5	A$ABRACADABR	10	1
		ADABRA$ABRAC	5	4
CAD	6	BRACADABRA$A	1	5
DAB	7	ACADABRA$ABR	3	3
RA$	8	DABRA$ABRACA	6	7
RAC	9	$ABRACADABRA	11	0

We then read each back into the bucket it came from which it came from previously, in the order of this new bucketing.[58] Note that in this case the "previous bucket" is just the first 3 letters, but subsequent passes, this label must encompass all six values, so we generate a new label for use in the subsequent rounds. Note the new labels can be generated by simply checking whether consecutive entries match at both the old and new index, and generating an incremental new index if one of the two is different.

Bucket	Old Index	New Index	Rotation	Index
$AB,RAC	0,9	0	$ABRACADABRA	11
A$A,BRA	1,5	1	A$ABRACADABR	10
ABR,A$A	2,1	2	ABRA$ABRACAD	7
ABR,ACA	2,3	3	ABRACADABRA$	0
ACA,DAB	3,7	4	ACADABRA$ABR	3
ADA,BRA	4,5	5	ADABRA$ABRAC	5
BRA,$AB	5,0	6	BRA$ABRACADA	8
BRA,CAD	5,6	7	BRACADABRA$A	1
CAD,ABR	6,2	8	CADABRA$ABRA	4
DAB,RA$	7,8	9	DABRA$ABRACA	6
RA$,ABR	8,2	10	RA$ABRACADAB	9
RAC,ADA	9,4	11	RACADABRA$AB	2

While we note that as each rotation is in a distinct bucket, they mustbe sorted, we show the procedure of bucketing on the third triplet just for clarity.

[58]Astute readers will note this is effectively doing a mini LSD radix sort with two passes.

Bucket	Index	Rotation	Index	Previous Bucket
$AB	0	ADABRA$ABRAC	8	5
A$A	1	CADABRA$ABRA	7	8
ABR	2	DABRA$ABRACA	9	9
		BRACADABRA$A	1	7
ACA	3	RA$ABRACADAB	9	10
ADA	4	$ABRACADABRA	11	0
BRA	5	ABRA$ABRACAD	7	2
		RACADABRA$AB	2	11
CAD	6	A$ABRACADABR	10	1
DAB	7	ABRACADABRA$	0	3
RA$	8	ACADABRA$ABR	3	4
RAC	9	BRA$ABRACADA	8	6

And then back to the previous buckets, in the order of these new buckets.

Bucket	Old Index	New Index	Rotation	Index
$AB,RAC,ADA	0,4	0	$ABRACADABRA	11
A$A,BRA,CAD	1,6	1	A$ABRACADABR	10
ABR,A$A,BRA	2,5	2	ABRA$ABRACAD	7
ABR,ACA,DAB	2,7	3	ABRACADABRA$	0
ACA,DAB,RA$	3,8	4	ACADABRA$ABR	3
ADA,BRA,$AB	4,0	5	ADABRA$ABRAC	5
BRA,$AB,RAC	5,9	6	BRA$ABRACADA	8
BRA,CAD,ABR	5,2	7	BRACADABRA$A	1
CAD,ABR,A$A	6,1	8	CADABRA$ABRA	4
DAB,RA$,ABR	7,2	9	DABRA$ABRACA	6
RA$,ABR,ACA	8,3	10	RA$ABRACADAB	9
RAC,ADA,BRA	9,5	11	RACADABRA$AB	2

Using this method, we perform only a linear amount of total work each pass.

Runtime The initial iteration takes $O(n*l)$ time, as we have to explicitly examine the first three characters of each rotation. Each subsequent iteration takes $O(n)$ incremental time. As there are n/l total iterations, the runtime of the bucketing is $O(nl + n^2/l)$. When $l = \sqrt{n}$, this becomes $O(n^{1.5})$.

We must also consider the initial sorting of the buckets. This time ends up dominating, at $O(\sqrt{n} * nlog(n))$.[59]

Exponentially Growing Iterations If we consider the computation of the above runtime, the two steps which conspire to prevent us from increasing l arbitrarily high are the initial sort step and the first bucketing when we must examine the characters, both of which grow linearly in l. A small l is ideal for

[59]We might optimize by adjusting l to better balance the initial sort with the bucketing, but we'll beat that imminently anyway

those two initial phases, but a large l is ideal for the remainder of the algorithm as it limits the total number of iterations.

Instead of a fixed l size, can we grow l in each round? There are two key issues:

1. With a fixed l, we were able to sort all buckets up front. We now have a different set of buckets each round which we have to sort.

2. With fixed l, we were directly able to look up the right bucket to use, since we knew some other rotation had already bucketed the same l-tuple in the previous iteration. This no longer applies now that the buckets have changed each iteration.

Let's deal with the issues in order.

Bucket Sorting In each iteration, we have buckets of size, say, l, and let's suppose we have already ordered them from 1 to l. In the next iteration, we have buckets of size $2l$. Naively, we would perform some sort on all of these (up to $O(n)$), incurring a large cost. This is unnecessary, however.

If we note in the above section on avoiding bucket iterations, we encoded each bucket tuple as a new index in order, these new indices inherently generated an ordering of all length l substrings which occur in the input. There is therefore no further work to be done to sort the next length l labels, since we just did it.

Identifying The Labels Since we have relabeled all length l buckets, we can still simply look up the bucket which was used for by rotation $r + l$ to know which new bucket rotation r should go into.

Putting it all together The above properties leads to a complete algorithm which enables us to radix sort the rotations with exponentially growing bucket labels.

- Bucket each of the the rotations by the first character ($l = 1$). The strings are now sorted by the first character.
- Bucket each of the rotations by the next l characters remembering which previous bucket they were in.
- Place items back in their original buckets based on this new bucket order.
- Compute new bucket labels, representing the sorted indices of the buckets for the next iteration.
- Double the size of l and go back to step 2.

Runtime We incur only $log(n)$ iterations over the course of the algorithm. In each iteration, we perform a constant number of passes to bucket n items and relabel n buckets. Both these operations are $O(n)$ overall, and therefore the complete algorithm is $O(n * log(n))$.

9.4.6.2 Complete Example

To ensure complete understanding, we walk through the example from start to finish.

In the first step, we bucket each rotation based on the the first letter.

Bucket	Index	Rotation	Index
$	0	$ABRACADABRA	11
A	1	ABRACADABRA$	0
		ACADABRA$ABR	3
		ADABRA$ABRAC	5
		ABRA$ABDACAD	7
		A$ABRACADABR	10
B	2	BRACADABRA$A	1
		BRA$ABRACADA	8
C	3	CADABRA$ABRA	4
D	4	DABRA$ABRACA	6
R	5	RACADABRA$AB	2
		RA$ABRACADAB	9

We now bucket based on the second letter, while remembering which bucket the rotation was in. Recall that we don't need to actually look at the second letter in a given rotation r, we simply need to check what bucket $r + 1$ was in.

Bucket	Index	Rotation	Index	Previous Bucket
$	0	A$ABRACADABR	10	1
A	1	$ABRACADABRA	11	0
		RACADABRA$AB	2	5
		CADABRA$ABRA	4	3
		DABRA$ABDACA	6	4
		RA$ABRACADAB	9	5
B	2	ABRACADABRA$	0	1
		ABRA$ABRACAD	7	1
C	3	ACADABRA$ABR	3	1
D	4	ADABRA$ABRAC	5	1
R	5	BRACADABRA$A	1	2
		BRA$ABRACADA	8	2

And back into the previous buckets in this new order. We note the new labels on the bucekts as we do this.

Bucket	Index	New Index	Rotation	Index
\$A	0,1	0	\$ABRACADABRA	11
A\$	1,0	1	A\$ABRACADABR	10
AB	1,2	2	ABRACADABRA\$	0
			ABRA\$ABRACAD	7
AC	1,3	3	ACADABRA\$ABR	3
AD	1,4	4	ADABRA\$ABRAC	5
BR	2,5	5	BRACADABRA\$A	1
			BRA\$ABRACADA	8
CA	3,1	6	CADABRA\$ABRA	4
DA	4,1	7	DABRA\$ABRACA	6
RA	5,1	8	RACADABRA\$AB	2
			RA\$ABRACADAB	9

We now increase to $l = 2$ and bucket on the second two letters. Note we alreayd have a sorted list of all two letter combinations from the new index, but can also determine the bucket for a given rotation r by looking at the current bucket of rotation $r + 2$. For example, rotation 5, which has the second pair of characters "AB" will end up in bucket 2, which is the current bucket of rotation $r + 2 = 7$, which must start with "AB".

Bucket	Index	Rotation	Index	Previous Bucket
\$A	0	RA\$ABRACADAB	9	8
A\$	1	BRA\$ABRACADA	8	5
AB	2	A\$ABRACADABR	10	1
		ADABRA\$ABRAC	5	4
AC	3	BRACADABRA\$A	1	5
AD	4	ACADABRA\$ABR	3	3
BR	5	\$ABRACADABRA	11	0
		DABRA\$ABRACA	6	7
CA	6	RACADABRA\$AB	2	8
DA	7	CADABRA\$ABRA	4	6
RA	8	ABRACADABRA\$	0	2
		ABRA\$ABRACAD	7	2

And back into the previous buckets in this new order, again noting the new bucket labels.

Bucket	Index	New Index	Rotation	Index
$ABR	0,5	0	$ABRACADABRA	11
A$AB	1,2	1	A$ABRACADABR	10
ABRA	2,8	2	ABRACADABRA$	0
			ABRA$ABRACAD	7
ACAD	3,4	3	ACADABRA$ABR	3
ADAB	4,2	4	ADABRA$ABRAC	5
BRA$	5,1	5	BRA$ABRACADA	8
BRAC	5,3	6	BRACADABRA$A	1
CADA	6,7	7	CADABRA$ABRA	4
DABR	7,5	8	DABRA$ABRACA	6
RA$A	8,0	9	RA$ABRACADAB	9
RACA	8,6	10	RACADABRA$AB	2

We increase to $l = 4$ and bucket on the second set of four letters. We extract the bew bucket index from the rotation $r + 4$.

Bucket	Index	Rotation	Index	Previous Bucket
$ABR	0	ABRA$ABRACAD	7	2
A$AB	1	DABRA$ABRACA	6	8
ABRA	2	ACADABRA$ABR	3	3
		BRA$ABRACADA	8	5
ACAD	3	$ABRACADABRA	11	0
ADAB	4	BRACADABRA$A	1	6
BRA$	5	CADABRA$ABRA	4	7
BRAC	6	RA$ABRACADAB	9	9
CADA	7	ABRACADABRA$	0	2
DABR	8	RACADABRA$AB	2	10
RA$A	9	ADABRA$ABRAC	5	4
RACA	10	A$ABRACADABR	10	1

And back into the previous buckets in this new order, again noting the new bucket labels.

Bucket	Index	New Index	Rotation	Index
$ABRACAD	0,3	0	$ABRACADABRA	11
A$ABRACA	1,10	1	A$ABRACADABR	10
ABRA$ABR	2,0	2	ABRA$ABRACAD	7
ABRACADA	2,7	3	ABRACADABRA$	0
ACADABRA	3,2	4	ACADABRA$ABR	3
ADABRA$A	4,9	5	ADABRA$ABRAC	5
BRA$ABRA	5,2	6	BRA$ABRACADA	8
BRACADAB	6,4	7	BRACADABRA$A	1
CADABRA$	7,5	8	CADABRA$ABRA	4
DABRA$AB	8,1	9	DABRA$ABRACA	6
RA$ABRAC	9,0	10	RA$ABRACADAB	9
RACADABR	10,8	11	RACADABRA$AB	2

While we could progress and bucket onthe next 8 characters, as each rotation is in its own bucket, thy must be sorted and thus can terminate if we choose.

9.4.6.3 Implementation

From a high level, the implementation largely follows the above described algorithm. We first discuss a few key optimizations.

Cheating the Buckets Recall each iteration of the algorithm. The first step we took was to bucket based on the l characters starting at index l. In the first iteration, after the initial bucketing, we bucket based on the 1 character at index 1. This bucketing is used as the order to place the rotations back in their initial bucket.[60]

Lets take a look at the order of the rotations before and after this initial bucketing.[61]

Index Before	Index After
11	10
0	11
3	2
5	4
7	6
10	9
1	0
8	7
4	3
5	4
2	1
9	8

The pattern should be apparent. The order of the indices after bucketing is simply the initial index minus 1. This pattern continues in the $l = 2$ case, where we simply subtract two.

[60] before relabeling, of course

[61] These are the indices in the first two tables in the concrete example section.

Index Before	Index After
11	9
10	8
0	10
7	5
3	1
5	3
1	11
8	6
4	2
6	4
2	0
9	7

Consider why this pattern makes sense. If a rotation r is minimal when sorting by the first character, then rotation $r-1$ must be minimal when sorting by the second character, as the previous rotation shifts that first character into the second position.[62] Therefore, rather than explicitly bucketing to perform the first step in each iteration, we can simply subtract l from the indices output from the last iteration.

Bucketing Without Lists In a naive implementation of the remaining bucket step in each iteration,[63] we would generate a list for each bucket, append to it for any rotation we found for that bucket, and then merge. It would be faster, however, to place each element in its exact location in an output array. This avoids the relatively high overhead of lists relative to arrays as well as the merge step.

The challenge of finding the exact location of the next element is twofold:

1. We must know where the given bucket starts in output array, and
2. We must know how many elements have previously been placed in this bucket.

This can be solved by summing up the number of elements in each bucket. The prefix sum of these values represents the first index at which an element in this bucket should be placed. For instance, before the first iteration, there is 1 item in the $ bucket, and 5 items in the A bucket. The first element for bucket B therefore goes in slot 6.

[62] $ABRACADABRA shifts to A$ABRACADABR, putting $ in the position we're sorting on.

[63] where we place items back into their original bucket, based on the order of generated by the now-optimized first bucketing step

Bucket	Count	Prefix Sum
$	1	0
A	5	1
B	2	6
C	1	8
D	1	9
R	2	10

In order to identify the offset into a given bucket, we simply increment this value each time we place an item into the bucket. After the first insertion into the B bucket, we increment its value to 7.

With a small amount of variable juggling, we can do this with constant extra memory.[64]

Other Notes

- In our first bucketing step, instead of explicitly computing labels for each bucket, we simply use the ASCII value of each character.
- The size of the array used to count the number of buckets is bound by the maximum label on any bucket. In the first iteration, this is the number of ASCII values (256), and in all other iterations, this is n. The array is sized to hold both of these.[65]
- When computing new bucket labels, we obtain our label from this step (where we bucketed l characters starting at l) directly from the saved `temp_bucket` array. To obtain the label for the first half of the string (l characters starting at 0), it seems we should have to have a cache of the previous iteration. Fortunately, as the l characters at index 0 in rotation r appear as the l characters starting at l in rotation $r + l$, we can recover this value as the element in `temp_bucket` for rotation $r + l$.[66]

Listing 9.22: C++

```
struct suffix_array{
    string ss; //the input string with a terminator
    int n; //size of the duplicated string
    vector<int> sa; //the suffix array itself
    suffix_array(string s):ss(s+"$"),n(ss.length()),sa(n,0){
        vector<int> q, //Queue for the radix sort
                    bucket, //the bucket each element of the sa is in
                    temp_bucket, //temporary copy of the bucket array
                    cs(n+256,0); //array used for counting number of
```

[64]alternatively, we can compute the prefix sum in the "standard" way, and it will represent the last index available for a given bucket. We can then populate all the buckets by processing the elements in reverse.

[65]We sum them, but a max of the two would be sufficient.

[66]Note this is the same logic we used to eliminate the initial bucketing step in each iteration

```
                          //elements in each bucket
        for(int i=0;i<n;i++)q.push_back(i);
        for(char c:ss)bucket.push_back(c);
        //The main loop. l is as described.
        //i is the size of the sorted label at the end of the iteration
        for(int i=1,l=0;l<n;i<<=1,l=i>>1){
            //perform the first bucketing step of the iteration with
            //simple subtraction, as described above
            for(int j=0;j<n;j++)q[j]-=l-(q[j]<l?n:0);

            //compute the number of elements in each bucket
            fill(cs.begin(),cs.end(),0);
            for(int j:bucket)cs[j]++;
            // and the prefix sum
            for(int j=0,s=0,t=cs[0];j<cs.size();s=t,t=cs[++j])
                cs[j]=j?cs[j-1]+s:0;

            //perform the second bucketing step,
            //adjusting the counts as we go
            for(int j:q)sa[cs[bucket[j]]++]=j;

            //compute new buckets. Generate a new index only if one of
            //the two indices doesn't match
            temp_bucket=bucket;
            bucket[sa[0]]=0;
            int bucket_index=0; //index of next bucket
            for(int j=1;j<n;j++)bucket[sa[j]]=
                temp_bucket[sa[j]]!=temp_bucket[sa[j-1]]||
                temp_bucket[(sa[j]+l)%n]!=temp_bucket[(sa[j-1]+l)%n]
                    ?++bucket_index
                    :bucket_index;
            q=sa;
        }
    }
};
```

9.4.6.4 LCP Array Construction

Recalling earlier, the LCP array stores the longest common prefix between consecutive suffixes in the suffix array. Naively this takes $O(n^2)$ time, which is not fast enough for what we need.

Instead of computing the LCPs in order of the suffix array, what if we instead compute them in the order of the original string? Let's consider an example. Assume we have already computed the LCP between ABRA\$ and ABRACADABRA\$ as 4.

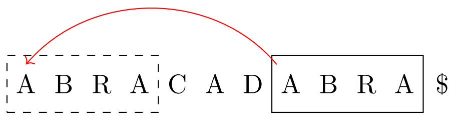


Figure 9.110: The suffix following ABRA$ is ABRACADABRA$, as noted by the red arrow. The LCP between these two consecutive suffixes is 4, as highlighted by the boxes.

Instead of now considering the next suffix in the suffix array (ABRACADABRA$), let's instead consider the next suffix in the string itself (BRA$). As BRA$ is a suffix of ABRA$ which is one shorter in length, and BRACADABRA$ is a suffix of ABRACADABRA$ which is one shorter in length, then we can declare that the LCP between the two starting with B is one shorter than the two starting with A.

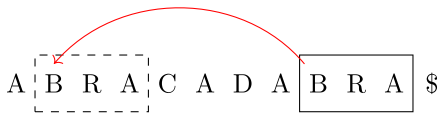


Figure 9.111: The arrow moved one to the right, and we chopped off left most character of each box, demonstrating the LCP between those two suffixes is 3.

We have one issue, however. Just because ABRA$ and ABRAC... are consecutive elements in the suffix array does not mean that BRA$ and BRAC... are as well. What we do know, however, is that if a suffix does come between them, it must match in those same three characters.[67] As such, we know that even if the two suffixes are not consecutive, the LCP value must be at least the "seed" value of 3 we obtained from the previous iteration.

To solve this problem, we simply explicitly check characters against the next suffix in the actual suffix array to see if the match is actually longer. In this case, BRA$ and BRAC... are actually consecutive, and the $ miscompares with the C, leaving our value as 3.[68]

This leads to the overall algorithm:

1. Iterate through the suffixes in the order of the original string.
2. Choose a seed value ones less than the LCP of the previous suffix (or 0 if negative).
3. Compare characters between this suffix and the subsequent suffix in the suffix array until a miscompare is found.

[67]If one of those characters differ, the interloping suffix would necessarily sort before or after both of the existing candidates.

[68]We would then continue with the suffix RA$ and a seed value of 2.

Runtime In each iteration of the algorithm, the value of the seed decreases by at most one from iteration to iteration, bounding the total decrease over all iterations to $O(n)$. As the seed is bound between 0 and n, the maximum increase over the course of the algorithm must also be $O(n)$, leading to overall linear time.

Implementation The implementation is tacked on as an extension to the generation of the suffix array itself.

Listing 9.23: C++

```
vector<int> lcp(n,0),rsa(n,0);
for(int i=0;i<n;i++)rsa[sa[i]]=i; //generate a reverse lookup of the SA
for(int i=0,seed=0;i<n;i++,seed=max(seed-1,0))if(rsa[i]<n-1){
    while(i+seed<n&& //The first suffix hits end of string
         sa[rsa[i]+1]+seed<n&& //The second suffix hits end of string
         s[i+seed]==s[sa[rsa[i]+1]+seed])seed++; //increment if matches
    lcp[rsa[i]]=seed;
}else seed=0;
```

9.4.7 Usage

Most of the tasks articulated under the suffix tree section can be solved with suffix arrays. We focus on subset of those tasks here.

9.4.7.1 Finding All Instances of a Pattern

Recall that every instance of a pattern in a string is the start of a suffix of a string. Therefore, finding all instances of a pattern in a string reduces to finding all suffixes which start with that string. As we have a sorted list of suffixes, we can determne the first suffix which starts with the pattern, and the last, and the total number of instances of the pattern is the count of suffixes between those endpoints (inclusive). The endpoints can be found with binary search in $O(p * log(n))$.

Listing 9.24: C++

```
pair<int,int> find_all(string p){
   int lhs=0,rhs=0,d=1;
   while(d<sa.size())d<<=1;
   d>>=1;
   while(d){
      if(lhs+d<sa.size()&&ss.compare(sa[lhs+d],p.size(),p)<0)lhs+=d;
      if(rhs+d<sa.size()&&ss.compare(sa[rhs+d],p.size(),p)<=0)rhs+=d;
      d>>=1;
   }
   return{lhs+1,rhs+1};
}
```

By using string hashing techniques, the comparisons can be done in $O(log(p))$ time, leading to $O(log(n+p))$.

9.4.7.2 Longest Repeated Substring

The longest repeated substring is simply the consecutive elements in the suffix array with the largest common prefix, or simply the largest element in the LCP array.

Listing 9.25: C++

```cpp
pii long_rep(){//returns length, and start index
   pii ans={0,0};
   for(int i=0;i<lcp.size();i++)if(lcp[i]>ans.first)ans={lcp[i],sa[i]};
   return ans;
}
```

9.4.7.3 Counting Unique Substrings

To solve this problem, we should consider how a given suffix contributes to the count of unique substrings. The first simplification is we only have to consider substrings which are prefixes of the suffix as all substrings are the prefix of some other suffix. Put another way, only consider the substrings which start at the same character as the suffix we are considering.

Lets first consider all substrings which start here (`n-sa[i]`) and consider which we have seen before. It turns out if the end of the substring is within the LCP of the previous suffix, we would have seen it when we examined that suffix. For example, when comparing BRA$ and BRACADABRA$, the first three substrings at the start of the latter (B, BR, and BRA) all must appear at the start of the previous suffix, since the LCP is 3. This must otherwise be the first time we are seeing each other substring at the start of this suffix, leading to the simple calclation of `n-sa[i]-lcp[i-1]`).

As we do not wish to count substrings which end with $, we subtract n from the total, and as we have already discounted substrings which start with $ by skipping the 0-th rotation, we have accidentally subtracted the substring $ twice, so we must add one to account for it.

Listing 9.26: C++

```cpp
ll substr_cnt(){
   ll ans=0;
   for(int i=1;i<n;i++)ans+=n-sa[i]-lcp[i-1];
   return ans-n+1;
}
```

9.5 Hashing

Using naive methods, string comparisons take $O(l)$, where l is the length of the strings. When two strings are compared exactly once, this is true. However when a string is used in multiple comparisons, we can do significantly better using hashing.

In general, when we compute the hashcode of a structure, we reduce the entire contents of that structure to a single number in $O(n)$ time,[69] which can be compared with other structure hashes in $O(1)$ time afterwards. With a good hash function, the chance of two unequal structures hashing to the same value is incredibly small. We can apply this technique on strings, computing a multitude of string-comparison based algorithms exceptionally quickly, intuitively, and with a small amount of code.[70]

9.5.1 Hashing Theory

In almost all cases, we expect hash functions to have certain properties:

- We expect hash functions to be *consistent*. If $A = B$, then we require $hash(A) = hash(B)$
- We expect hash functions to be *uniform*. Without this properly, then $hash(X) = 1 \forall X$ could be a valid, consistent hash function, though clearly useless. When a hash function is uniform, it allows us to make a probabalistic guarantee that if $hash(A) \neq hash(B)$, then $A \neq B$. Violations of this guarantee are known as *collisions* and the probabality of such an outcome is tunably small.

Depending on the use case, we may also prefer a third propery, *composability*. Composability is a property which allows the computation of hash of the combination of two previously-hashed objects without further examining the contents of either object. With two objects A, B and composable hash might be defined as $hash(A, B) = f(hash(A), hash(B))$ where f is some function which does not depend on the contents of either A and B. With such a hash, if we have already eaten the linear time cost of hashing of two subcomponents, we can compute the hash of their concatenation faster than time linear in the size of the combined object.[71]

In the case of strings, if we can devise such a composable hash, we would be able concatenate two strings and compute the combined hash in constant time.[72] Furthermore, if the function for computing the combined hash is invertible, we would be able to compute the hash of one part of a string only by knowing the

[69] where n is the size of the structure

[70] Compared to a suffix tree

[71] Note that in some circumstances, such as cryptography, this property is extremely detrimental, and a secure hash should not be computable without the contents itself.

[72] assuming the two parts were already hashed

hash of the whole string, and the hash of the remaining part. Assume we have such a function, then the following is possible:[73]

$$hash(S_{1,l}) = f(hash(S_{1,x}), hash(S_{x+1,l}))$$
$$f^{-1}(hash(S_{1,l}), hash(S_{1,x})) = hash(S_{x+1,l})$$

This may seem like an innocuous transformation, however note that if we have such an invertible function and have precomputed the hash of every substring starting at index 1, then we can, in constant additional time, compute the hash of any arbitrary substring. Naively, computing the hash of every prefix in order to have the requirements for the second equation takes quadratic time, however we depend on our defition of composability directly. This allows us to incrementally compute such a ”prefix” hash in constant incremental time, and linear time total.

$$hash(S_{1,x+1}) = f(hash(S_{1,x}), hash(S_{x+1,x+1}))$$

Here we assume that we can hash a single character string in constant time, and if it is so, we can extend our hash prefix in constant time.[74]

9.5.1.1 The String Hash Function

As noted, composability, invertibility, and the ability to hash a single string character in constant time are the properties needed[75] for a useful string hash. The following function is up to the task.[76]

$$hash(S) = \sum_{i=1}^{l} S_i P^i \mod M$$

where

- S_i is an positive integer unique to the charater at index i in S and all such identical characters
- P is a prime larger than any value S_i
- M is a large prime

Consistency and uniformity are relatively intuitive to see, so we focus on composability. Consider two strings S and T. Their hashes are computed as follows:

$$hash(S) = S_1P^1 + S_2P^2 + S_3P^3 \mod M$$

[73]Assume S is a string, and $S_{x,y}$ is the substring of S from index x to y, inclusive.

[74]Note that this is fundamentally similar to ”typical” prefix sums, whereby the composability and invertability of the function allows us to precompute all prefix sums of a sequence in linear time, and compute any range sums in constant time

[75]along with consistency and uniformity

[76]Note that this is effectively a numerical representation of the string in base P, mod M

$$hash(T) = T_1P^1 + T_2P^2 + T_3P^3 \mod M$$

If we concatenate the two strings, we obtain the following hash which demonstrates composability.

$$hash(ST) = S_1P^1 + S_2P^2 + S_3P^3 + T_1P^4 + T_2P^5 + T_3P^6 \mod M$$

$$hash(ST) = hash(S) + P^3 * hash(T) \mod M$$

$$hash(ST) = hash(S) + P^{len(S)} * hash(T) \mod M$$

As the combined hash depends on addition and multiplication, it is trivially invertible, fullfilling all the required properties.

9.5.2 Implementation

9.5.2.1 Selection of Parameters

The above function is quite generic, but to operate well a good selection of the parameters is necessary.[77]

- A convenient selection of S_i is a trivial application of ASCII math, using `s[i]-'a'`.[78] This naive implementation leads to a high possibility of collision for short strings comprised of 'a'. For instance the strings "a" and "aa" would have unacceptably identical hashes. To alleviate this problem, we offset by 1, leaving a much better function of `s[i]-'a'+1`.
- As the hash is effectively a number in base P, P should be larger than the cardinality of the alphabet. For maximum assurance of collision avoidance, this value should ideally be prime. For strictly lowercase letters,[79] 31 is viable. For both upper and lower case, 61. The determination of a usable P for any other alphabet is left as an exercise for the reader.[80]
- The modulus M should be a large prime. For implementation convenience, M should be kept to less than 2^{31}, enabling any multiplication to fit in a 64-bit signed integer. 1000000007 and 1000000009 are good choices.

[77] Note that functions which can generate collisions for a known selection of P and M exist. Therefore in contests which allow submission of antagonistic test cases, one should determine these values at runtime based on some dynamic parameter.

[78] Note `a` should be replaced by the least lexicographic character in the possible alphabet.

[79] or strictly uppercase

[80] Be sure the mapping of character to integer from the previous bullet maps the first character to the integer 1, and that the mapping of the greatest character in the alphabet is still less than the chosen prime.

9.5.2.2 The Birthay Paradox

Using a modulus of approximately 10^9 yields a nominal collision probability of 10^{-9} for any particular comparison. Due to the birthday paradox, it takes relatively few comparisons before a collision is likely. One solution is to to increase the value of M to increase the number of bits in the hash. This compromises our ability to perform math within the space of a 64-bit integer. To cope with this, we can simultaneously compute the hash using two (or even three) 31-bit distinct moduli, enabling signficiantly more resolving power with relatively little additional complexity.

9.5.2.3 Code

We present here an implementation of the computation of a string's prefix hash. The output is the hash value of every prefix of the string. As demonstrated above, this runs in linear time. Note that we precompute the powers of P for simplicity.

Listing 9.27: C++

```
//assume a string s
int64_t M=1000000009;
uint64_t P=31;
uint64_t ps[s.length()],h[s.length()+1];

//precompute powers of P
ps[0]=1;
for(int i=1;i<s.length();i++)ps[i]=ps[i-1]*P%M;

//compute prefix hashes
h[0]=0;
for(int i=0;i<s.length();i++)h[i+1]=(h[i]+(s[i]-'a'+1)*ps[i]%M)%M;
```

The usage of a hash array that is one greater than the size of the string simplifies the code needed for substring extraction by aligning substring endpoints with the needed elements in the hash array, as well as to avoid the corner case where the left endpoint has index 0, which may cause out of bounds errors.

9.5.2.4 Substring Extraction

Using the reversible nature of the hash function and the precomputed prefix hash, we can easily extract the hash value of an arbitrary substring.[81]

$$hash(T) = \frac{hash(ST) - hash(S)}{P^{-1*len(S)}} \mod M$$

As this function may be oft executed, and depends on multiplying by the inverse of $P \mod M$, it may be appropriate to precompute all such values.

[81]This is just solving for $hash(T)$ in the earlier definition of the string hash function.

Computing Modular Inverse The computation of $P^-x \mod M$ is often stymying, and more difficult than other modular arithmetic. We can compute it as follows.

$$P^x P^{-x} = 1 \mod M$$

$$P^x P^{-x} + kM = 1$$

As P^x and M are integral and coprime, their GCD is 1, meaning extended euclidean algorithm can be applied to solve for P^{-x} and M in logarithmic time.

Listing 9.28: C++

```
uint64_t euclid(uint64_t a, uint64_t b, int64_t &x, int64_t &y) {
    if(!b)return x=1,y=0,a;
    uint64_t d=euclid(b,a%b,y,x);
    return y-=a/b*x,d;
}

uint64_t pi[s.length()];
pi[0]=1;
int64_t q,k;
euclid(P,M,q,k);
q=(q%M+M)%M;
for(int i=1;i<s.length();i++)pi[i]=pi[i-1]*q%M;
```

Note that the value of `q` returned by euclid may be negative. This necessitates using a signed variable for the mod `M` to ensure we can properly extract its positive equivalent.

Extracting actual substrings is done as follows:

Listing 9.29: C++

```
//assume we have precomputed pi and h
//also assume left index is inclusive, and right is exclusive
substring_hash=(h[right]-h[left]+M)%M*pi[left]%M;
```

Another Option Depending on the use case, extracting the exact value of $hash(T)$ may not be necessary. Consider an attempt to compare two substrings, T and T' (with prefixes of S and S'). Based on what has been discussed, the comparison should be

$$\frac{hash(ST) - hash(S)}{P^{-1*len(S)}} \stackrel{?}{=} \frac{hash(S'T') - hash(S')}{P^{-1*len(S')}}$$

Instead, we can multiply both sides by $P^{max(len(S),len(S'))}$. This leaves us in a situation where we only need to multiply one side by a power of P, and eschew any computation of the modular inverses at all.

The choice of whether or not to use this method or the modular inverse is a choice by the implementer depending on the exact nature of the specific problem.

9.5.2.5 Hashing Hashes

In the following algorithms, we will see instances where we have to compare a hash against a set of hashes. A common idea is to place the string hashes in a hashset, providing constant lookup. While this may seem appealing, the constant factor on working with hashsets is quite high. It may be faster to explicitly add the hashes to an array, then sort and binary seaerch rather than hash the hash.

On the other hand, the elements we are adding to the hashset would already be hashes, so hashing those values is an expensive and unnecessary step. An implementation which directly uses the interger value of the string hash instead of further hashing it may be faster than the other methods.

9.5.3 Algorithms

Hashing can be used to solve many of the problems that other constructs, such as suffix trees, can solve. See those sections for more complete problem descriptions.

9.5.3.1 String Comparison

Comparing two strings for equality is straightforward, as we simply compare hashes.[82] We can also use the prefix hashes to lexicographically compare strings in order to either sort or perhaps find a minimum. Fundamentally this can be done by identifying the length of the longest common prefix of the two strings, and then comparing the first unmatched character. We can find the longest common prefix by binary searching over the prefix length and comparing the hash of that length prefix within each string. If the two hashes match, we know the longest common prefix is longer than (or equal to) the given candidate length, otherwise shorter. Once the length of the longest prefix is identified, we can compare the subsequent characters in each of the two strings.

As we can extract any arbitrary substring in constant time, assuming a precomputed prefix hash, we can perform this comparison in logarithmic time as opposed to the naive linear time without using hashes.

9.5.3.2 Pattern Existence

Fundamentally, searching for a pattern P in S resolves to checking if there is a substring in S, of length equal to the length of P, which is equal to P. Assuming we have computed the hash of P and the prefix hashes of S, we can iterate through each substring of S of length $len(P)$ and compare the hash of P to

[82] which may involve substring extraction from a precomputed prefix hash

the hash of that substring. As there are $O(len(S))$ such substrings, and the substring hash is constant, we are left with linear overall time.

9.5.3.3 Finding All Instances of a Pattern

Similar to pattern existence, assuming we have precomputed the appropriate hashes, we can simply return all substrings of the appropriate length which match the pattern hash, instead of just whether it exists.

9.5.3.4 Longest Repeated Substring

The key intuition for finding the longest repeated substring is that we can consider the hash of all substrings of a given length in $O(len(S))$ time. Given that, we ask the question, "Is there any duplicated substring of length k?" and binary search of the largest such k.

To examine each candidate length k, we iterate through each substring of that length and add the hash to some structure. If duplicate hashes exist for that k, then there are duplicate substrings of length k. The choice of a hashset, treeset, or array of the hashes which is later sorted is left as an implementation choice.

Assuming the usage of a treeset and $len(S) = l$, the runtime is $O(l * log^2(l))$

9.5.3.5 Longest Common Substring

The computation of the longest common substring follows very closely to the above longest repeated substring. We can examine each hash of length k in one string, and then by walking all hashes of length k in the other string, see if there is a duplicate. Binary searching on k yields the maximum such value.

9.5.3.6 Longest Common Extension

The computation of the longest common extension is similar to the above alogirhtms, where we can binary search for the length of such an extension, using the hashes to compare in constant time.

9.5.3.7 Longest Palindromic Substring

The longest palindrome idea is similar to the suffix tree-based algorithm, in that we examine each possible palindromic midpoint, and compute the longest common extension of forward and reverse substrings starting from that point. To accomplish this, we compute the prefix hash of the string, and the prefix hash of the reversed string. For each midpoint (which is a single character for odd-length palindromes, or consecutive characters for even), we binary search over the length of the longest palindrome there centered, where the hash we use for comparison starts at the selected candidate midpoint in both the forward and reversed string.

Considering every midpoint, and computing the longest common extension is $O(l * log(l))$ time.

9.5.3.8 Minimum Rotation

Examining each rotation of a string is equivalent to concatenating the string to itself finding the least lexicographic substring of length l.[83] Using the logarithmic method for string comparison, we can find the minimum such substring by simply examining all of them. This takes $O(l * log(l))$ time.

[83]This is similar to the construction for minimum rotation in suffix trees.

9.6 Finding Minimum Rotations

The minimum rotation of a string is the least lexicographic substring of length N which occurs if we allow said substring to "wrap around" to the beginning if we reach the end of the input. An example using BACAA:

Rotations	Lexicographic Order
BACAA	4
ACAAB	3
CAABA	5
AABAC	1
ABACA	2

While we can solve this using suffix trees, construction of a suffix tree is costly in terms of code length and memory, and has a high constant-factor to its execution time. Despite this, the existence of linear-time suffix tree construction algorithms means there must be linear-time minimum rotation algorithms. Many such are known, however we focus on one which is based on the underlying suffix tree algorithm.

Consider building an entire suffix tree as described above. If we are naively attempting to find the minimum rotation with it, we must construct the entire tree and walk the minimum branch at each node. However, what if we discard steps and branches which we can determine have no impact on the minimum rotation? Let's observe the optimizations we can make.

Only One Root Branch During construction of the tree, at some point we will likely create a second branch from the root. One of these two branches will have a lexicographically-lesser first character. We do not care about the other one, and thus do not need to store it.

Active Point As we only have a single branch, if the active point is not at the root, it must be on on that branch.[84] This means the active point can be stored as a single value indicating how many characters on the single branch we have compared successfully.

Suffix Links In Ukkonen's algorithm, once we have a miscompare at the active point, we would create a new branch, and insert all suffixes of the path to the newly created node, using suffix links to optimize this insertion. This becomes more difficult as we have not maintained any suffix links.

Case 1: Larger Branch Consider the case where this new branch is lexicographically greater than the existing one. We do not wish to maintain this branch, as it is not minimal. Is it possible that some suffix of this rotation

[84]Note that were the insertion to follow a different branch, we would have just discarded it, due to optimization 1.

might create a lesser branch? It turns out to not be the case. As every path out of a node representing the source of a suffix link is necessarily represented at the destination of a suffix link, if a branch point has some path less than a newly created branch, then so must every node reachable via suffix links from this node.

As such, we know no suffixes of the rotation can be less than a previously seen rotation and can immediately resume insertions starting at the subsequent character from the root without having to follow any links.

Case 2: Lesser Branch On the other hand, consider the case where this new branch is lexicographically lesser than the existing one. We know the rotation we are inserting at this moment is less than the current minimum, but how can we know whether, or which of the suffixes of this rotation may be even smaller?

Consider the case AACAADAACAABX. When inserting the suffix AA-CAAB into current minimum AACAAD, we find the B is lesser than the D, and therefore AACAAB is less than AACAAD. If we simply followed case 1, we would continue on to insert the rotation starting at X. This would be incorrect, however, as AACAAB contains a suffix (AAB) which is less than AACAAB, and therefore a lesser rotation.

Fortunately, we can simplify the problem. We know any suffix link that would lead to a different branch than the current minimum branch cannot be better overall.[85] We therefore only have to consider suffix links which would terminate on the minimum branch. This occurs any time the suffix of the path we are inserting matches the prefix of the path ittself.

Consider the case of AACAAB at the time we would insert the B, lead to by a path of AACAA. AA is a suffiix of that path, and a prefix. Therefore during suffix tree construction, we end up inserting suffix AAB, and would end up folling suffix links from AACAA to AA eventtually. The same is true of a single A, which is both a suffix and prefix.

Put formally, When inserting a string S into a suffix tree and ending at the root, we create exactly $|S|$ new branches. A branch point of the suffix starting at index i in S is an ancestor of the deepest branch point (representeing S itself) if and only if the first and last i characters of S are equal.

If we could determine such points, we can simply compare the subsequent character (the B in our example) with the character following that prefix to determine whether the branch created at this point would be minimal.[86] The

[85]Consider the branch point from the current minimum branch to whatever this candidate destination branch is. All characters up to that branch point are common, at the split, the character on the minimum branch must be less than the character on the other branch. If a suffix link would take us to that other branch, the path to that node must include those common shared characters followed by the same greater character for which the branch was created. Any path that terminates in this subtree must therefore be greater than the minimum path.

[86]In the example, AAB would compare less than AAC. Subsequently, AB would not compare less than AA, covering the two suffix/prefix pairs.

problem then remains, how can we identify such branch points without storing all links?

Fortunately, the Overlap function defined while developing KMP alalows us to compute, in linear time, the suffixes of a given string which are also prefixes. By using the same logic we used in developing that function, we can identify all such nodes.

In our example, the OLAP values for AACAAB are 01012. After inserting the B, we know we have an suffix/prefix pair of length 2 (AA), and then can recursively lookup the longest suffix of AA which is also a prefix, to find A.[87]

9.6.0.1 The Algorithm

Now that we have described in theory how we can reduce Ukkonen's algorithm to just the bits that are necessary for computing the minimum rotation, lets describe the algorithm a bit more concisely. Throughout the course of the algorithm, we will cache:

- The index of the current minimum rotation, representing the branch int the suffix tree that starts at that index
- The index current character we are inserting
- The index of the current "candidate" suffix/rotation we are attempting to insert into the tree

The overall flow goes as follows for each character we insert:

1. If the character does not match at the active point[88], and is greater than the character on the exiting branch, the candidate suffix and all its own suffixes can be discarded. The next candidate is the subsequent character in the string, which we will then attempt to insert.
2. If the character matches at the active point, simply proceed to the next character. The actiive point moves implicitly.
3. If the character does not match at the active point, and is less than the character on the existing branch, we have a new minimum.
    (a) The candidate is less than the previous minimum, so update the minimum accordingly.
    (b) Compute the overlap function over the candidate up to the index of the character just inserted.

[87]This is the exact same method we use to construct the overlap in the first place. See the KMP section for details

[88]Which is the point along the minimum branch which is the delta between the current character and the current candidate

(c) For each suffix the overlap function indicates is also a prefix, evaluate if this rotation is less than any previous minimum by evaluating the last character.[89]

(d) As we have checked each suffix/rotation which might produce a lesser branch and updated the minimum accordingly, the next candidate is the subsequent character in the string, which we will then attempt to insert.

Rough Proof of Linearity It should be clear that steps one and two above are constant time. This leaves the last step. The overlap function is computed multiple times over the course of the algorithm. Fortunately, the third step is only executed once per miscompare, over the current length of the candidate substring. As the next candidate begins after this range, no single character is considered as part of a candidate range more than once. Any work that is linear over candidate range is linear over the entire algorithm.

Another way to see this is from the proof of runtime of Ukkonen's algorithm itself. The proof of linearity in that case depends on suffix links moving up the tree with a decrease in deptth of no less than 1. This means the resolution of a a new branch via suffix links is linear in the depth of the branch point. In our case, we can freely use that time to compute the overlap function and evaluate necessary branch points without impacting the runtime.

9.6.0.2 Implementation

The implementation follows from the description, with some slight inconsequential changes.

- The input string is concatenated to itself to eliminate the need consider the wrap-around.
- Rather than computing the overlap function only when we have found a miscompare, we compute it along the way. As the same work is done, this does not impact the runtime.

Listing 9.30: C++

```cpp
int minrot(string s){
    string t=s+s;
    int o[t.size()]={},mi=0,can=1;
    for(int i=1,k=0;i<t.size();i++,k=i-can){
      //compute the overlap function at this index relative to the
          candidate
        for(o[k]=(k?o[k-1]:0);o[k]&&t[i]!=t[can+o[k]];o[k]=o[o[k]-1]);
        o[k]=(k&&t[can+o[k]]==t[i])?o[k]+1:0;
```

[89]Recall we are only concerned about branch points which are ancestors of eachother, and if that is the case, the length of the prefix/suffix defines both the branch point, and that all characters up to the last one must match.

```
      //this branch is greater, dump it
        if(t[i]>t[mi+k])can=i+1;

      //this branch is better. check each suffix/prefix pair to see if
           it's minimum
        else
             if(t[i]<t[mi+k])for(mi=can,can=i+1,o[k]=k;o[k];o[k]=o[o[k]-1],mi=(t[i]<t[mi+o[k]])?i-o[k]:
    }
    return mi;
}
```

Chapter 10

Other Common Techniques

10.1 Interesting Points

10.2 non-constructive indexing

10.3 String Search

10.3.1 Regular Expressions

10.4 Caterpillaring

10.5 precomputation

10.6 scangrid and quick-updates