Algorithms for Programming Contests

Kevin Kauffman
Duke University

January 29, 2019

Abstract

learn to do programming!

Contents

Chapter 1

Introduction

Chapter 2

Basic Techniques

2.1 IO

2.2 Bit manipulation

2.3 Basic Structures

2.4 language Tools

2.5 Brute Force Search

2.5.1 Fixing a Variable

2.6 Recursion and Backtracking

2.7 Custom Comparation

Chapter 3

Math and Geometry

Chapter 4

Graphs

4.1 Types and Terminology

4.2 Representation

4.3 Algorithms

4.3.1 Search

DFS

BFS

Dijkstra

MST

Others

- All Pairs
- negative weights
- reverse distances
- dense graphs
- path reconstruction

4.3.2 Flow and Match

In-Out Nodes

graph-split transformation

4.3.3 Componentization

SCC

BCC

2-sat

4.4 State Explosion

4.5 FSMs

4.6 DAGs

4.7 Trees

4.8 counting loops

4.9 union find

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 DP. 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: $1, 1, 2, 3, 5, 8 \ldots$. 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.

```
// returns the n'th fibonacci number
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:

1. Evalute `fib(49)`
    (a) Evalute `fib(48)`
        i. Evalute `fib(47)`
            A. ...
2. Evaluate `fib(48)`
    (a) Evalute `fib(47)`
        i. ...
3. Add result of (1) and (2)

We can see even in this small breakdown that we are computing the same thing multiple times! The reality is quite ugly, and we can see the excess of times we evaluate each `fib(n)` after making a call to `fib(10)`.

0	1	2	3	4	5	6	7	8	9	10
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 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 the value again. The function looks like this:

```
// returns the n'th fibonacci number
int[] memo;
int fib(int n) {
    memo = new int[n + 1]; // size array to ensure we can cache all
        values <= n
    Arrays.fill(memo, -1); // use -1 to indicate "unknown"
    return fib_helper(n);
}

int fib_helper(int n) {
```

```
    if (memo[n] != -1) return memo[n]; // if we already computed, don't
        recompute

    if (n == 0 || n == 1) memo[n] = 1;
    else memo[n] = fib(n - 1) + fib(n - 2); // save newly compute value

    return memo[n];
}
```

This small optimization, simply saving the result and returning, causes an immense speedup. Computation is fast enough that we run out of stack space with large `n` before the algorithm takes a particularly long time to run. The call counts reflect this speedup; the 19 calls being a far cry from the earlier 177.

0	1	2	3	4	5	6	7	8	9	10
1	2	2	2	2	2	2	2	2	1	1

Runtime

The runtime analysis of this optimization is quite simple, and consists of two parts:

1. We execute the main body of the `fib_helper` 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 which equals -1, and that execution causes the entry to not equal -1, limiting the number of executions to the size of the array
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, we find the stack limitation prevents us from evaluating particularly large values of n. To solve this, lets look at the state of the array as the algorithm progresses.

0 1	1 1	2 ?	3 ?	4 ?	5 ?	6 ?	7 ?	8 ?	9 ?	10 ?

Figure 6.1: The initial state of the array (with base cases populated)

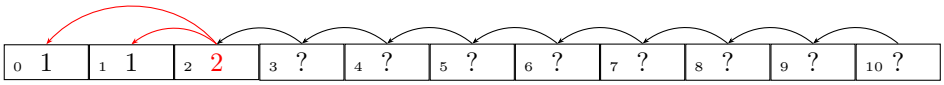


Figure 6.2: The state of the stack when we first recurse to `fib(2)`. The two dependent values are known, so we can compute the value.

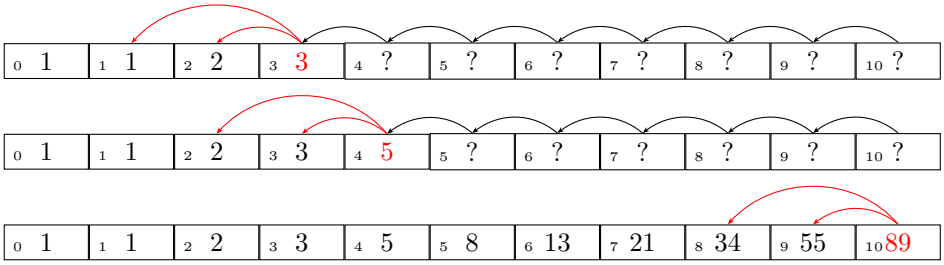


Figure 6.3: Values are filled into the array as the dependencies become fulfilled.

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 can determine the order in which the array will be populated, we can instead eliminate the recursion and directly calculate the values.

```
// returns the n'th fibonacci number
int[] memo;
int fib(int n) {
    memo = new int[n + 1]; // size array to ensure we can cache all
        values <= n
    memo[0] = 1; // initialize the base cases
    memo[1] = 1;
    for (int i = 2; i <=n; i++) { // loop over the order the array would
        get populated
        memo[i] = memo[i - 1] + memo[i - 2]; // exact same computation
            as before
    }

    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.

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 4 common building blocks:

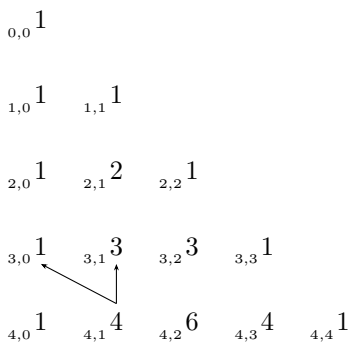
1. What the index into the DP array represents
   - For fibonacci, this was i, the index of a fibonacci number
2. What value is stored at each index in the DP array
   - The actual value of the i-th fibonacci number
3. What is the relation between entries
   - `memo[i] = memo[i - 1] + memo[i - 2];`
4. What are the base cases
   - `memo[0, 1] = 1;`

Once we've answered these four questions, we can translate our candidate solution to code.

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.



Two arrows show, matching the definition, that `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
3. Relation: `ncr(i,j)= ncr(i-1, j)+ ncr(i-1, j-1)`
4. Base Cases: This one is a bit trickier. Based on the relation, we know where all the dependency arrows point. So if we look at the triangle, where do we have to define base cases? The Arrows pointing directly upward run into a "wall" when `n == r`, and the diagonal arrows do the same when `r == 0`. This allows us to define the following two base cases that will bound any of the relationships
   (a) `ncr(i,i)= 0`
   (b) `ncr(i,0)= 0`

With these four things defined, we can proceed to write our code in a very similar manner to fibonacci.

```
// returns n-choose-r
int[][] memo; // use 2-D array since 2 dimensions
int fib(int n, int r) {
    memo = new int[n + 1][n + 1]; // size array to ensure we can cache
        all values <= n
    // initialize the base cases
    for (int i = 0; i <= n; i++) {
        memo[i][0] = 1;
        memo[i][i] = 1;
    }

    // loop over the array in the direction opposite the dependency
        arrows
    for (int i = 2; i <= n; i++) for (int j = 1; j < i; j++) {
        memo[i][j] = memo[i-1][j] + memo[i-1][j- 1]; // the recursive
            relation
    }

    return memo[n][r];
}
```

6.3.1 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 $O(n^2)$, and the fact that we only populate half of it ($j < i$), it does not change the runtime. The relation involves a constant number of lookups, meaning the overall runtime is still quadratic.

6.4 Standard DP Problems

6.4.1 Knapsack

Consider the following problem:

> 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?

There are three distinct characteristics of this problem that strongly indicate it falls into this class of *knapsack* problems.

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

We'll consider three variations of this problem that result in different solutions.

Greedy Knapsack

This problem is actually quite trivial as stated. There is no limit to the amount of any individual food item we can take, and since everything is done servings, regardless of our choices, we can fully eat up to our serving limit. As such, there is no reason to not greedily select whichever food has the highest happiness per serving, and consume S servings of it.

Even if we modify the problem slightly, and place a limit on amount of each amount of food we can take, we can still greedily select that highest happiness food, and eat it until it runs out before consuming the next highest food (and so on). We will be able to fully consume our limit regardless of the food selection.

A third slight variant, perhaps entailing a serving limit which is non-integral, but allowing fractional servings of individual food items has the same result. We can fully consume our limit regardless of selection, so should taked the highest happiness first.

The common theme of all these variants is that regardless of the slightly differing constraints, in each case we are guaranteed to hit the limit exactly. If that is the case, then the greedy solution will apply.

Knapsack with Repeats

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. The food i must be consumed in multiples of q_i.

While it may seem innocuous, this small additional restriction means that there is no longer a guarantee we can hit S exactly. Here's an example:

i	happiness	quantity
0	100	10
1	95	4

If $S = 10$, then taking the 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 (without the ability to eat 2 servings of either of the available foods). If we instead select the slightly less happy item 1, however, we will be able to consume 12 servings of it, leading to a far greater happiness of 1140. The greedy solution fails and we need something a bit more clever.

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 (n == 0) return 1; // base case...no servinces, can't eat!

    int ans = Integer.maxValue;

    // 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 = 0; i < food_types; i++) if (q[i] <= s) {
        int recursion = happy(s - q[i]);
        if (recursion == Integer.maxValue) continue; // no valid
            solution, skip

        // if eating q[i] of this food gives us a better answer, do it!
        ans = Math.max(ans, h[i] * q[i] + helper(s - q[i]));
    }

    return ans;
```

```
}
```

While this solution would be correct, it is undoubtedly too slow for any reasonable limits. Think back to 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, is eating 1 then 2 any differently than eating them in the opposite order? Just like with fibonacci, we see that in our execution, given a serving count and amount of happiness we have while at that serving count, we don't care what we ate to get there, the solution of the remainder of the problem will be unaffected. We can thus use this to start generating the 4 elements we need for our DP. Using this information, we can memoize.

```
// returns the maximum happiness for S servings
int[] memo;
int fib(int s) {
    memo = new int[n + 1]; // size array to ensure we can cache all
        values <= s
    Arrays.fill(memo, -1); // use -1 to indicate "unknown"
    return helper(s);
}

int helper(int s) {
    if (memo[s] != -1) return memo[s]; // if we already computed, don't
        recompute

    if (n == 0) memo[s] = 1; // base case...no servinces, can't eat!
    else {
        // in brute force solution, we would just try everything, we do
            here, and take the minimum
        for (int i = 0; i < food_types; i++){
            // if eating q[i] of this food gives us a better answer, do
                it!
            memo[i] = Math.max(memo[i], h[i] * q[i] + helper(s - q[i]));
        }

    }

    return memo[s];
}
```

1. Indices: The solution for a given number of servings does not change. We can use this as our index.
2. Value: We store the maximum happiness which can be obtained in so many servings
3. Relation: Our previous relations were quite straightforward. Lets take a

step back and write a recursive backtracking solution with memoization and see if we can figure it.

6.4.2 Mini-Max

6.4.3 Grid Tiling

6.4.4 Inclusion/Exclusion

6.4.5 Travelling Salesman

6.5 Optimizations

6.5.1 Memory Reduction

6.5.2 Dimension Swapping

6.5.3 Simension Elimination

6.5.4 Restricted Search

6.5.5 Knuth's Optimization

6.5.6 Convex-Hull Optimization

6.6 Identifying DP

6.6.1 Value of Recursion

Chapter 7

NP Completeness

Chapter 8

Data Structures

8.1 segment trees

8.2 interval trees

8.3 fenwick trees

8.4 Trie

Chapter 9

Other Common Techniques

9.1 Interesting Points

9.2 non-constructive indexing

9.3 String Search

9.3.1 Regular Expressions

9.4 Caterpillaring

9.5 precomputation

9.6 scangrid and quick-updates