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)`.

index	0	1	2	3	4	5	6	7	8	9	10
call count	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.

index	0	1	2	3	4	5	6	7	8	9	10
call count	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)

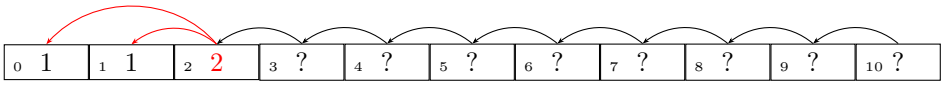


Figure 6.2: The state of the stack when we first recurse to `fib(2)`. The two dependent values are known, so we can compute the value.

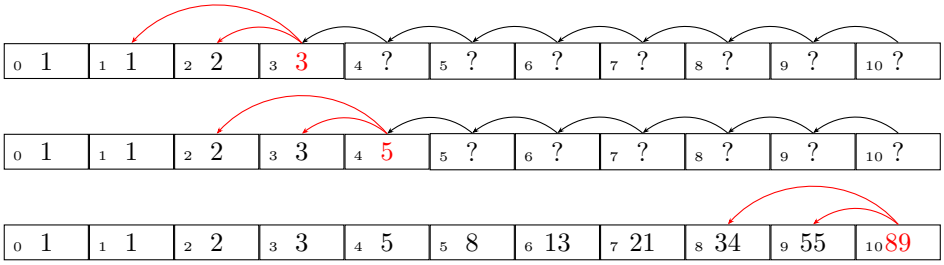


Figure 6.3: Values are filled into the array as the dependencies become fulfilled.

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:

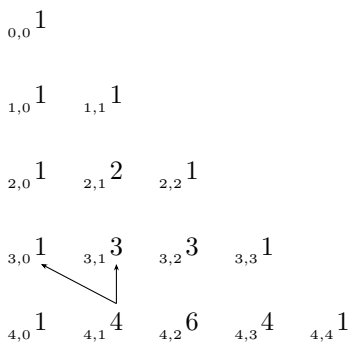
1. What the index into the DP array represents
   - For fibonacci, this was i, the index of a fibonacci number
2. What value is stored at each index in the DP array
   - The actual value of the i-th fibonacci number
3. What is the relation between entries
   - `memo[i] = memo[i - 1] + memo[i - 2];`
4.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

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.

6.3.1 Benefits of Recursion

So far, we've extolled the benefits of DP over recursion with memoization, but is it ever worthwhile to write a recursive solution? Sure.

- 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 order to populate the array. 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, sometimes it is useful or even necessary to revert to recursive ones.

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 te 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:

food	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 (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 = 0; i < food_types; i++) if (q[i] <= s) {
        int recursion = happy(s - q[i]); // value we get if we eat this
            food, so we have s - q[i] servings left

        // 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 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 one food than other any different 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. Similar to fibonacci again, we can cache these results and significantly decrease our runtime.

```
// returns the maximum happiness for S servings
int[] memo;
int happy(int s) {
    memo = new int[s + 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

    memo[s] = 0;

    for (int i = 0; i < food_types; i++) if(q[i] <= s) {
        int recursion = happy(s - q[i]); // if we eat this one, we have
            s - q[i] servings left

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

    return memo[s];
}
```

Whether we needed to go through the exercise of writing out the recursion, we should be able to construct the four elements required to execute the DP.

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: $\text{memo}(s) = max_{i \in \text{foods}}(h_i * q_i + \text{memo}(s - q_i))$, the food which gives us the maximum happiness if eaten and added to the value after recursing on $s - q_i$
4. 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.

```
// returns the maxium happiness for S servings
int[] memo;
int happy(int s) {
    memo = new int[s + 1]; // size array to ensure we can cache values
        <= s
    Arrays.fill(memo, 0); //default base case

    for (int i = 1; i <= s; i++) { // loop over array in reverse order
        of dependencies
        for (int j = 0; j < food_types; j++) if(q[j] <= i) { // can only
            evaluate if the amount of servings represented by i is more
            than minimum quantity of this food
          memo[i] = Math.max(memo[i], h[j] * q[j] + memo[i - q[j]]);
```

```
        }
    }

    return memo[s];
}
```

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 to indicate un-processed nodes as well as cases we skip since they would push us off the array.

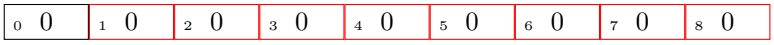


Figure 6.4: The initial state of the array

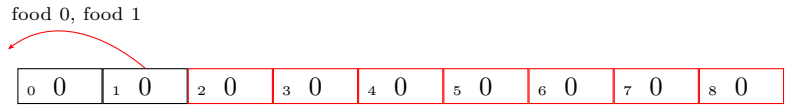


Figure 6.5: evaluation of entry 1. Both food types go off the end of the array, so there is no solution

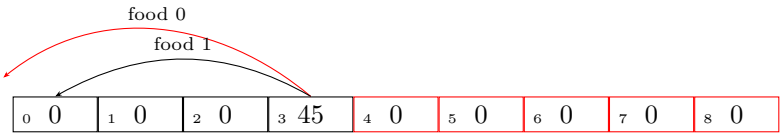


Figure 6.6: Once we get to $S = 3$, we can consume one of the foods, and note the dependency arrow accesses the first element in the array

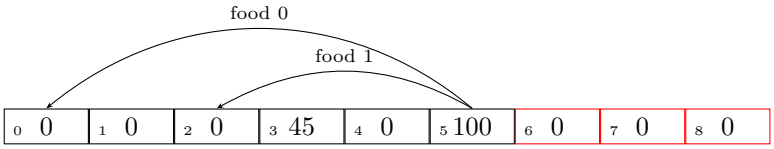


Figure 6.7: Once we get to $S = 5$, we can eat food 0, so get a solution.

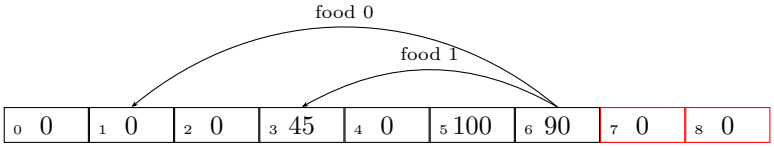


Figure 6.8: We can only eat food 1 here.

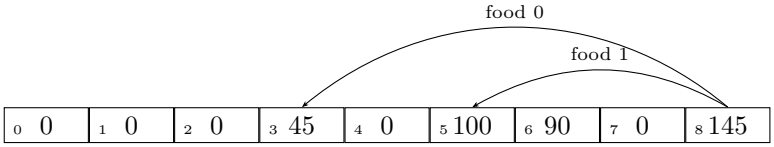


Figure 6.9: We can eat either food here. They both ultimately lead to the same result, which we can return as our solution

The runtime analysis follows similarly to how it did before. The array size is $O(s)$, simple enough, but the work done to populate each entry in the array, unlike earlier, involves walking each type of food. This makes the overall runtime $O(s \times foods)$.

Item Limitations

We'll consider one final variant of this problem, perhaps the most common: one adjusting the previous requirement.

> Each food i has an associated quantity q_i. The food i can only be consumed once, using exactly q_i servings.

While this may seem like a small change, it has major ramifications on the algorithm. The standard way to apply a restriction like this would be to encode whether each particular food has been eaten as an index into our memoization table. Now, when we iterate over all foods for each entry of the table, we skip those which our index has indicated we've already used (usually represented by a bitmask). While this solution would be correct, we now have added an exponential factor to our runtime, giving us $O(s \times 2^{foods} \times foods)$. This will likely be too slow.

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. In the course of evaluation, we'll have to evaluate states that require us to have eaten the following combinations of food:

- 0

- 1
- 2
- 3
- 0 and 1
- 0 and 2
- 0 and 3
- 1 and 2
- etc.

Even though each of the individual lookups is inexpensive, the fact that we have so many lookups and states is compelling. Like the previous variations, we don't really care what order we eat the food in, and yet we're evaluating every order.

Since for a given solution, the order we consume the foods in doesn't change the solution, a very common technique is to apply an arbitrary ordering.

> If you consume food of index i, you may not afterwards consume any food of index $x < i$.

This additional restriction does not change the solution (since we can just reorder the foods that lead to the optimal solution to go increasing index order), but how does it change how we construct our DP? Since the food-eating must go in a specific order (and we can imagine ourselves walking past a buffet, looing at each individual food), we can simply add the food item we're currently evaluating to the state. Our DP factors look as follows:

1. Indices:
   (a) The number of servings we have available to consume
   (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 maximum amount of happiness we can get in that many servings (same as before)
3. Relation: $\text{memo}(s, \text{food}) = max(\text{memo}(s, \text{food}-1), h_{food} \times q_{food} + \text{memo}(s - q_{food}, \text{food} - 1))$. We have two options, we either skip this food, meaning we would recurse on all previous foods with the same number of servings, or we DO eat this food, and we recurse with the appropriately limited servings, and all previous foods.
4. Base Cases: Same as before, we can't evaluate outside the bounds of the array

Runtime

With our new index, we only have $s \times$ foods elements in the array, and we only have 2 lookups to perform for each entry, bringing out exponential runtime down to $O(s \times \text{foods})$.

```
// returns the maxium happiness for S servings
int[][] memo;
int happy(int s) {
    memo = new int[s + 1][food_types + 1]; // size array to ensure we
        can cache values <= s, and for every food type. Note we'll have
        to 1-index the food types so that 0 = no foods remaining, 1 =
        eat the 0-th food, etc.
    for(int i=0;i<=s;i++)Arrays.fill(memo[i], 0);

    for (int i=0; i < food_types; i++) for (int j=1; j <= s; j++) { //
        loop over all food types, and for each food type, check every
        possible amount of servings remaining
        if (q[i] > j) continue; // not enough servings

        // we either don't eat the food, or we do
        // note the 1 indexing of the food
        memo[j][i+1] = Math.max(memo[j][i], h[i] * q[i] + memo[j -
            q[i]][i]);
    }

    return memo[s];
}
```

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.

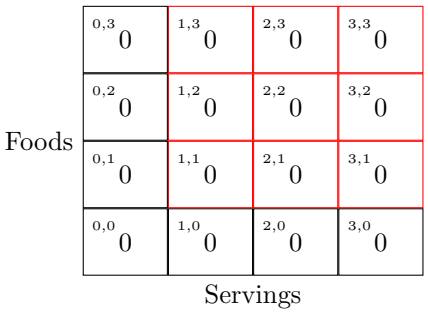


Figure 6.10: The initial state of the array

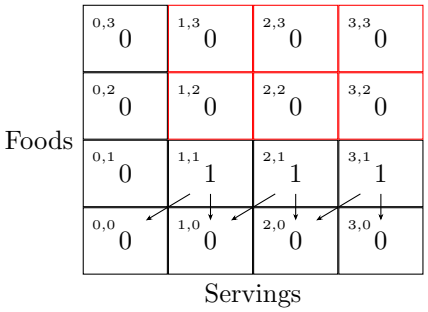


Figure 6.11: When we evaluate each entry for food 0 (index 1), we simulate eating and not eating it and take the maximum. The diagonal arrows represent eating the food (since we have to subtract the servings), and the vertical arrows represent skipping this food, since we have the same amount of servings 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.

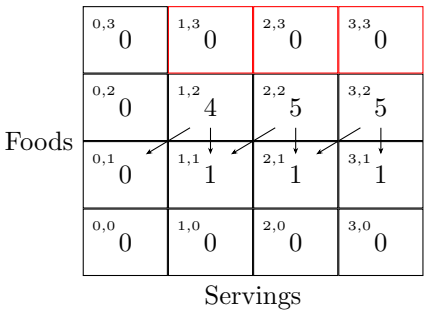


Figure 6.12: We evaluate food 1. We find eating it is always optimal. Each value will end up being the sum of the happiness of food 1 plus the values pointed to by the diagonal arrows, simulating eating this food.

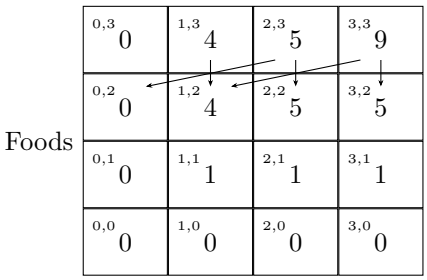


Figure 6.13: We evaluate food 2. When we have fewer than 2 servings available, we skip this food and just use the two previous foods (vertical arrow). When we have 2 servings, we find it is still optimal to skip this food (5 vs 4). When we have 3 servings, we eat this food and come out better (9 vs 5). Note the diagonal arrows go back 2 boxes since food 2 takes 2 servings.

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

   $\text{memo}(d, s) = max(\text{memo}(d + 10, s - 1) * .5, f_s + \text{memo}(d - a_s, s - 1))$

   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.

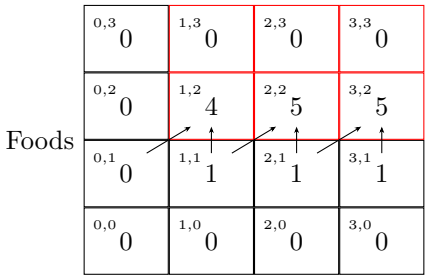
```
// returns the maxium fun for a given amount of max danger
double[][] memo;
double fun(int max_d) {
    memo = new int[max_d + 1][segments + 1];
    for(int i=0;i<=max_d;i++)Arrays.fill(memo[i], 0);

    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];
}
```

6.4.2 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.

6.4.3 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 $O(n^2 * 2^n)$.

6.4.4 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.

6.4.5 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 $n < i \text{and} m < i$, 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 $m << n$.

6.4.6 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 $O(n^2)$.

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.

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.

6.4.7 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.

```
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];
}

\subsubsection{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:
```

```
\begin{quotation}
    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.
\end{quotation}

You may note some similarities to a previous class of problems.
\begin{enumerate}
    \item There are two variables, each of which we are trying to
        maximize or minimize
    \item There are multiple classes of items, each contributing varying
        amounts to the two variables.
    \item We can choose to take or not take some of each item
\end{enumerate}

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.

\begin{enumerate}
    \item 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.
    \item Value: The expected amount of money we have for a given store
        and number of gifts left to buy
    \item 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
        $\text{gift}+1$ 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}
```

```
   \]
    \item Base Cases: $\text{memo}(0,g) = starting_money$. 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
        $\text{memo}(0,0<=i<g)$ 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).
\end{enumerate}

\begin{lstlisting}
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.

6.4.8 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.[1] 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
```

[1] a particularly mean problem writer may require this someday!

```
   }
   memo[i+1][0] += memo[i][1]; // 001+V = 000
   memo[i+1][0] += memo[i][4]; // 100+V = 000
    }

    retrun memo[n][0];
}
```

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 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 groups 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. We wish to maximize the sum of the friendship values of the groups.

6.5.5 Restricted Search

6.5.6 Knuth's Optimization

6.5.7 Convex-Hull Optimization

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.