11.1 Introduction

Dynamic programming can be very confusing until you've used it a bunch of times, so the best way to learn it is to simply do a whole bunch of examples. One way of viewing it is as a much more complicated version of divide-and-conquer a la mergesort or quicksort. In those cases, we could divide the problem into two subproblems, solve it optimally on each subproblem, and then combine the solutions (in the case of mergesort by a merge, in the case of quicksort trivially). This is great when it works, but in some cases it's not so simple. Dynamic programming is a way of rescuing the divide-and-conquer ideas from cases where it seems like it shouldn't work – for example, if the subproblems overlap or if there are a lot of possibilities.

The first example we'll see is *Weighted Interval Scheduling*.

11.2 Weighted Interval Scheduling

In this problem we are given a collection of n requests, where each request i has a start time s_i and a finish time f_i. Each request also has a value v_i. The goal is to find a subset $S \subseteq \{1, 2, \ldots, n\}$ such that no two intervals in S overlap and the value $\sum_{i \in S} v_i$ is maximized. Let's assume that they come sorted by finish time, so $f_1 \leq f_2 \leq \cdots \leq f_n$. Let's set up a little bit more notation: for each $i \in \{1, 2, \ldots, n\}$, we let $p(i)$ be the largest index $j < i$ such that intervals i and j are disjoint. In other words, we know that j must finish before i finishes, but we also require j to finish before before i even starts.

How do we design an algorithm for this? It's pretty clear that obvious techniques such as greedily picking the remaining interval of maximum value don't work. Instead, let's start by reasoning about the optimal solution S^*. We don't know what S^* is, but there are certainly some simple things we can say about it. For example, the last interval n is either in S^* or it's not.

What happens if $n \notin S^*$? Then clearly S^* is also the optimal solution for intervals $\{1, 2, \ldots, n-1\}$. On the other hand, if $n \in S^*$ then clearly S^* cannot contain any jobs between $p(n)$ and n, since they all interfere with job n. Moreover, whatever choices are made in jobs $\{1, 2, \ldots, p(n)\}$ do not affect job n, so in fact we know that S^* is just job n together with the optimal solution for intervals $\{1, 2, \ldots, p(n)\}$.

Let's try to write this down a little more formally. Let $OPT(i)$ denote the value of the optimal solution of jobs $\{1, 2, \ldots, i\}$. We can define $OPT(0) = 0$ just by convention. With this notation, what we just said is that if $n \in S^*$ then $OPT(n) = v_n + OPT(p(n))$, and if $n \notin S^*$ then $OPT(n) = OPT(n-1)$. So whether $n \in S^*$ depends only on whether $v_n + OPT(p(n)) \geq OPT(n-1)$! In other words, $OPT(n) = \max\{v_n + OPT(p(n)), OPT(n-1)\}$.

Note that there was nothing special about n here. If we want to analyze $OPT(j)$ the same analysis

still holds. So we get the recurrence relation

$$OPT(j) = \max\{v_j + OPT(p(j)), OPT(n-1)\}$$

This suggest the following obvious algorithm:

```
Schedule(j) {
  If j=0 return 0;
  else return max(Schedule(j-1), v_j + Schedule(p(j))
}
```

This algorithm clearly gives the correct solution, by our above argument (we can do this formally by induction on j). But what is its running time? It depends on the instance, but in the worst case its running time can be very bad. To see this, consider an instance where $p(j) = j - 2$ for all j. Then the recursion tree grows like the Fibonacci numbers, since Schedule(j) calls both Schedule(j-1) and Schedule(j-2)! This means that the number of recursive calls is exponential, so the running time for this algorithm is exponential.

So it seems like we're dead – there are a huge number of recursive calls. On the other hand, there are only n *distinct* recursive calls, since Schedule is always called with some parameter between 1 and n. So the reason the recursive algorithm is bad is because its computing the exact same thing many, many times. For example, there are a huge number of calls to Schedule(5), each one of which makes more recursive calls. But once we've computed the answer for Schedule(5), why not just remember it and return it instead of recomputing it?

To implement this, we'll have a table M with n locations. Initially each $M[i]$ will be empty, but when we first compute Schedule(i) we'll store the answer in $M[i]$. Then on future calls we can just return the answer from the table.

Slightly more formally, we modify the algorithm as follows.

```
Schedule(j) {
  If j=0 then return 0;
  else if M[j] nonempty then return M[j];
  else {
    M[j] = max(Schedule(j-1), v_j + Schedule(p(j));
    return M[j];
  }
}
```

What's the running time of this version? It's definitely not immediately obvious, but we can analyze it by analyzing the progress made towards filling out the table. Every time we issue the two recursive calls, when we return we fill in what was previously an empty slot in the table. So the number of recursive calls is at most twice the total number of empty slots, which is n. Since the work done in each call other than the recursion is $O(1)$, this means the total running time is $O(n)$.

11.3 Memoization vs Iteration

The above technique, where we simply remember the outcome of recursive calls, is called *memoization*. Sometimes the easiest way to think about dynamic programming is memoization. This is sometimes called a "top-down" dynamic programming algorithm, since we start from the full problem and make memoized recursive calls.

On the other hand, there is a completely equivalent "bottom-up" dynamic programming algorithm. We could simply fill up the table from the smallest to the largest values. This gives the following algorithm:

```
Schedule {
  M[0] = 0;
  for(i = 1 to n) {
    M[i] = max(v_i + M[p(i)], M[i-1]);
  }
  return M[n];
}
```

Now the running time is obvious: it is simply the number of table entries times the time to compute each entry given the previous ones. While this is not always true, it is pretty common for it to be easy to design the algorithm using memoization, but then easy to compute the running time using a bottom-up algorithm. I personally tend to think about dynamic programming problems bottom-up, by first reasoning about the table, but whatever works for you.

11.4 Principles of Dynamic Programming

Informally, what are the properties that a problem needs to have to use dynamic programming? First, we have to be able to break it into subproblems. Then we need the following.

1. There are only a polynomial number of subproblems (table entries)
2. The solution to a subproblem can be easily computed from the solutions to "smaller" subproblems. This is sometimes called the *optimal substructure* property: we can compute the optimum solution to one problem by computing the optimum solution to smaller subproblems. Note: this is very intuitive, but there are many interesting problems that do not have the optimal substructure property!
3. The solution to the original problem can be easily computed from the solution to the subproblems (usually, as in the above problem, it is in fact one of the subproblems itself).