## 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 divide-and-conquer ideas from cases where it seems like they shouldn't work – for example, if the subproblems overlap.

Fundamentally, dynamic programming is a method of solving a problem by breaking it down into a collection of "smaller" subproblems, and then using the solutions to these subproblems to build a solution to the actual problem of interest. So far this sounds a lot like divide-and-conquer, but in dynamic programming we will divide into subproblems which overlap significantly, and moreover, a naive divide-and-conquer algorithm would end up "re-solving" the same subproblem many times. We overcome this by solving each of those subproblems just once, and storing their solutions. The next time the same subproblem occurs, instead of recomputing its solution, we simply look up the previously computed solution. So the two key requirements of any dynamic programming algorithm are that 1) there aren't too many subproblems, and 2) the optimal solution to any subproblem can be computed efficiently if we are given the optimal solutions to "smaller" subproblems.

Dynamic programming is used all over the place. It was originally developed in the context of control theory, and immediately found uses in economics. It was later realized that it was useful for a huge variety of combinatorial optimization problems. Nowadays, the most famous users of dynamic programming (particularly at JHU) tend to be people in bioinformatics and natural language processing. This is because dynamic programming tends to be a very good fit for algorithms on strings, which are fundamental objects in both bioinformatics (DNA/RNA) and NLP.

Quick diversion: Dynamic programming was invented by Richard Bellman (although there were people like von Neumann who had used similar ideas earlier). The first question people usually ask about dynamic programming is "why is it called dynamic programming". The somewhat depressing but true answer, from Bellman's autobiography:

> An interesting question is, Where did the name, dynamic programming, come from? The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then,

about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

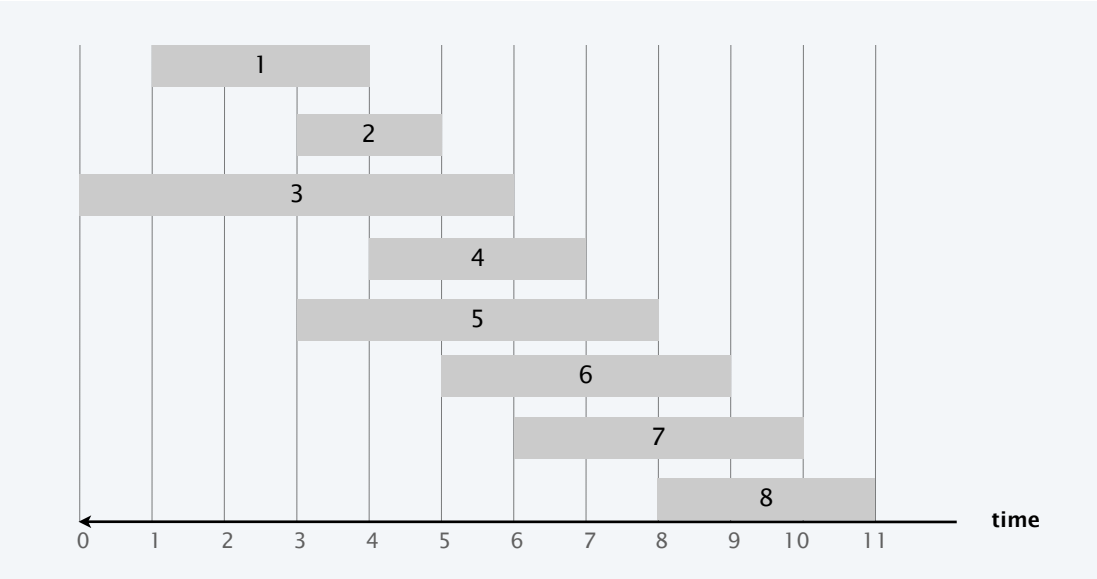
The first example we'll see is *Weighted Interval Scheduling*.

## 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. These requests are sometimes called "jobs", due to the original scheduling motivation (we have one CPU that can process one job at a time, and we are trying to maximize the value of the jobs that we schedule).

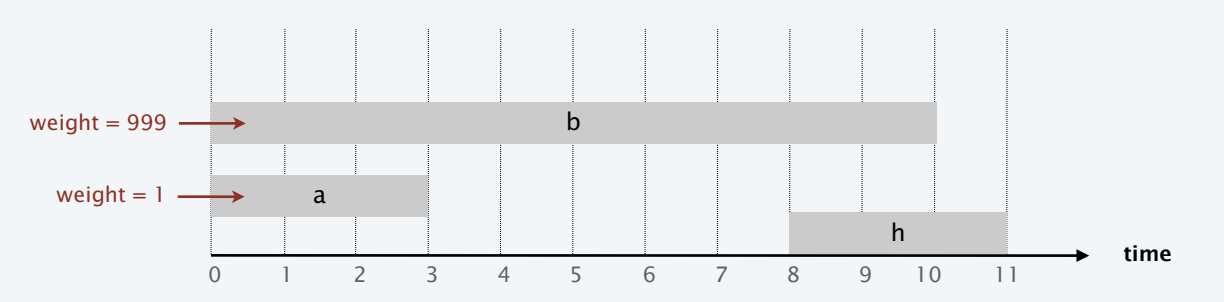
Let's assume that the requests come sorted by finish time, so $f_1 \leq f_2 \leq \cdots \leq f_n$. Let's set up a little bit more notation: for each $i \in \{1, 2, \ldots, n\}$, we let $p(i)$ be the largest index $j < i$ such that intervals $i$ and $j$ are disjoint. In other words, we know that $j$ must finish before $i$ finishes, but we also require $j$ to finish before before $i$ even starts.

For example, our input might look like the following, where each job also has an arbitrary value (not pictured):




In this case $p(1) = 0, p(2) = 0, p(3) = 0$, and $p(5) = 0$ (by convention this means that it conflicts with all previous jobs). For the other jobs, $p(4) = 1, p(6) = 2, p(7) = 3$, and $p(8) = 5$.

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. (Good exercise to do at home: try to design a greedy algorithm and find an example where it doesn't give the optimal solution). For example, the greedy algorithm where we pick greedily by earliest finishing time does work if all values are 1, but works very poorly with arbitrary values:




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, $OPT(n) = \max\{v_n + OPT(p(n)), OPT(n-1)\}$.

We can prove this a bit more formally by contradiction. First, by the above discussion and the definition of $OPT$, we know that there is a feasible solution of value $v_n + OPT(p(n))$, and another feasible solution of value $OPT(n-1)$. Thus $OPT(n) \geq \max\{v_n + OPT(p(n)), OPT(n-1)\}$.

On the other hand, assume for contradiction that $OPT(n) > \max\{v_n + OPT(p(n)), OPT(n-1)\}$. Let $S^*$ denote the solution which has value $OPT(n)$. If $n \notin S^*$ then by definition $S^*$ is feasible for the intervals $\{1, 2, \ldots, n-1\}$, so this together with out assumption that $OPT(n) > OPT(n-1)$ implies that there is a solution for the intervals $\{1, 2, \ldots, n-1\}$ of value larger than $OPT(n-1)$. This contradicts the definition of $OPT(n-1)$. Hence $OPT(n) \leq OPT(n-1) \leq \max\{v_n + OPT(p(n)), OPT(n-1)\}$ if $n \notin S^*$.

Similarly, if $n \in S^*$ then by definition $S^* \setminus \{n\}$ is a feasible solution for the intervals $\{1, 2, \ldots, p(n)\}$. Thus there is a solution for the intervals $\{1, 2, \ldots, p(n)\}$ of value $OPT(n) - v_n > v_n + OPT(p(n)) - v_n = OPT(p(n))$, where the inequality is by our starting assumption that $OPT(n) > \max\{v_n + OPT(p(n)), OPT(n-1)\}$. Thus there is a solution for the intervals $\{1, 2, \ldots, p(n)\}$ of value strictly larger than $OPT(p(n))$, which contradicts the definition of $OPT(p(n))$. Hence $OPT(n) \leq OPT(n-1) \leq \max\{v_n + OPT(p(n)), OPT(n-1)\}$ if $n \in S^*$.

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

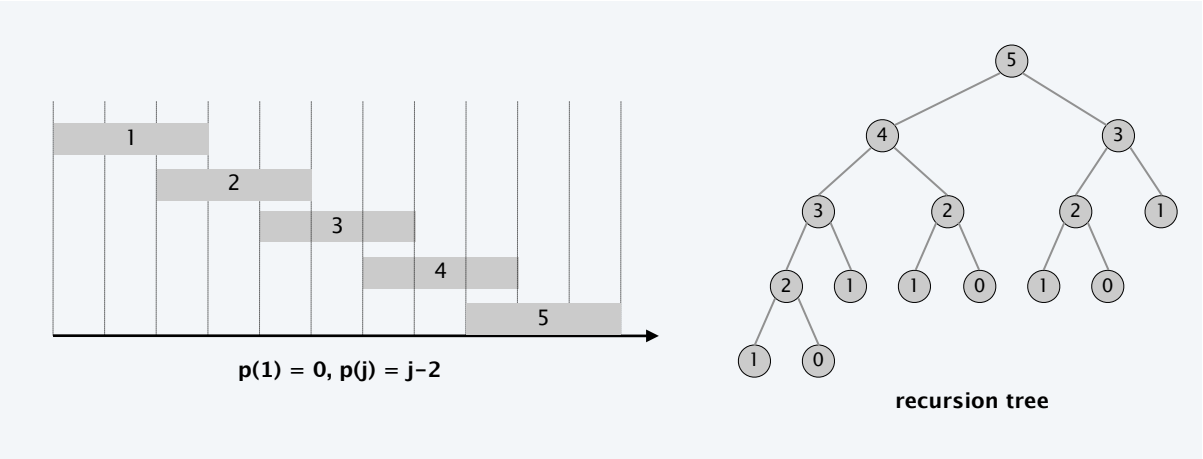
$$OPT(j) = \max\{v_j + OPT(p(j)), OPT(j-1)\} \tag{11.2.1}$$

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. To see this a bit more formally, let's prove by induction on $j$ that the value returned by Schedule($j$) is equal to $OPT(j)$. When $j = 0$ this is certainly true. For the inductive step, suppose it holds true for all $j' < j$. Then the value returned by Schedule($j$) is equal to max(Schedule($j-1$), $v_j$ + Schedule($p(j)$)), which by the induction hypothesis is equal to $\max(OPT(j-1), v_j + OPT(p(j)))$, which by (11.2.1) is equal to $OPT(j)$.

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(3), each one of which makes more recursive calls. But once we've computed the answer for Schedule(3), 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. What happens on a single call to `Schedule`? We either return an existing value in the table ($O(1)$ time) , or make two recursive calls and then fill in a table entry which was empty ($O(1)$ time to make the calls, fill in a table entry, and return, although more time could be spent inside the recursive calls). So the running time is $O(1)$ times the number of recursive calls. But every time we make two recursive calls we fill in a table entry, so the total number of recursive calls is at most $2n$. Thus the total running time is $O(n)$.

This is dynamic programming! We combined the optimal solutions of subproblems in order to find the optimal solution of a larger problem.

Side note: this was under the assumption that jobs are already sorted by finishing time. If they're not, then we can spend $O(n \log n)$ time and sort them. Good exercise at home: what goes wrong if we try to use this algorithm on an unsorted instance?

### 11.2.1 Finding the Solution

As some of you may have noticed, the dynamic programming algorithm we just designed does not actually return the optimal solution: it returns the *value* of the optimal solution. This is essentially trivial to fix. One way would be to keep track of the solution as we go. This works, but is a little inelegant and takes up extra space (for each table entry $M[j]$ we essentially have to store not just the *value* of the optimal solution on job $\{1, 2, \ldots, j\}$ but also the solution itself). You also have to be *very* careful that you don't spend too much time copying all the solutions. Another way, which is a little more elegant and is essentially what we always do, is just to do a second pass once the table has been filled out. This algorithm would look like the following:

```
Solution(j) {
  If j=0 then return ∅;
  else if v_j + M[p(j)] > M[j-1] return {j} ∪ Solution(p(j));
  else return Solution(j-1)
}
```

## 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 is fine.

## 11.4 Principles of Dynamic Programming (Section 15.3 of CLRS)

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. Usually these subproblems are determined by some choice, which we know we need to make but we don't know *how* to make. For example, in weighted interval scheduling, the choice is whether or not to include the final job. Second, we need for this choice to lead to smaller subproblems, which look like smaller instances of the original problem.

We have this, we need the following.

1. There are only a polynomial number of subproblems (table entries)
2. The optimal solution to a subproblem can be easily computed from the optimal solutions to "smaller" subproblems. This is sometimes called the *optimal substructure* property: we can compute the optimum solution to one subproblem 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).