CIS 320: Algorithms September 27, 2021

Lecture 7

Lecturer: Aaron Roth *Scribe: Aaron Roth*

Dynamic Programming I: Weighted Interval Scheduling

So far we've been lucky in our "divide-and-conquer" examples. This was manifested in part by how easy it was to choose sub-problems to divide our initial problem into. By and large, we always wanted to divide our problem in half if we could, and didn't have to agonize over whether there might be better splits. We were also able to just solve the recurrences we came up with, and we ended up with fast algorithms! But sometimes things don't work out like that: its not obvious how to break up our problem, and even if we do come up with a recursive algorithm, its straightforward implementation still runs in exponential time. Today we'll begin studying a more powerful version of "divide an conquer" called "dynamic programming", that can be used when it is not obvious *which* sub-problems to divide our problem into. The more powerful approach will also have the potential to have more dramatic improvements in running time over the naive solution. And yet the basic premise is very simple: we start by identifying structure that lets us write down a recursive solution to our problem. The naively implemented recursive solution may actually have exponential running time, but what we are looking for is wastefulness in the naive recursion: if it turns out that the same sub-problem is solved many different times, we can simply cache the solution, and re-use it in the recursion without needing to recompute it. The result is running time scaling with the number of *distinct* sub-problems needed in the recursion, not the total size of the recursive tree. Before going on, we should acknowledge that the technique we are about to learn has two different names, depending on the implementation details: "dynamic programing" and "memoization". Both names are terrible, so ignore them for now. What we will be doing is implementing a recursive algorithm while caching previously solved sub-problems ("memoization"), and then simply unrolling the recursion to solve the sub-problems in a bottom up manner ("bottom up dynamic programming").

We'll give our first example with the problem of "weighted interval scheduling":

Definition 1 *An instance of the weighted interval scheduling problem is given by a collection* $I = \{w(i), s(i), f(i)\}_{i=1}^{n}$ *of* n jobs $(w(i), s(i), f(i))$. *Each job has a real valued* weight $w(i)$, start time $s(i)$, *and* finish time $f(i)$. *Think of a job as representing an interval of utilization of some resource spanning* $[s(i), f(i)]$

A feasible solution *to the weighted interval scheduling problem is a subset* $S \subseteq [n]$ *of jobs such that for each* $i \in S$, *the intervals* $[s(i), f(i)]$ *are non-overlapping. The* weight *of a solution* S *is* $w(S) = \sum_{i \in S} w(i)$. *The goal of the weighted interval scheduling problem is to find a feasible solution* $S \subseteq [n]$ *of maximal weight.*

Consider e.g. the problem of scheduling different families to take vacations in a time-share. The intervals $[s(i), f(i)]$ represent their proposed vacation dates, and $w(i)$ represents how much they will value (or pay for) the privilege. Weighted interval scheduling is the problem of scheduling them to maximize sum happiness (or profit).

In solving this problem, we will imagine that our jobs are sorted in ascending order of finish times: $f(1) \leq f(2) \leq \ldots \leq f(n)$. If this is not already the case, we can make it so by sorting in $O(n \log n)$ time. Moreover, let us write $p(i)$ for the job with latest finishing time that is *before* job i's start time (if there is none, we define $p(i) = 0$). $p(i) = \max\{j : f(j) \leq s(i)\}$. We'll assume we have pre-computed the values of $p(i)$ for all i; once again, this can be done in time $O(n \log n)$. Finally, for $k \leq n$, lets write $\mathrm{OPT}(k)$ to denote the value of the optimal solution to the weighted interval scheduling problem $\{w(i), s(i), f(i)\}_{i=1}^{k}$ — i.e. the problem that contains only the first k jobs.

Lets make an observation that is so simple it seems unlikely to be useful. Suppose S^* is the solution to a given weighted interval scheduling problem on the first n jobs. Then we are in one of two cases. *Either*:

1. $n \notin S^*$: In this case, we must have $\text{OPT}(n) = \text{OPT}(n-1)$, *or*
2. $n \in S^*$: In this case, we must have $\text{OPT}(n) = \text{OPT}(p(n)) + w_n$. This is because we know that S^* contains n, so no feasible solution can contain any point $j > p(n)$, and all solutions that do *not* contain a point $j > p(n)$ are feasible, so we might as well choose the optimal such one.

This observation immediately gives us the following recursive algorithm for computing the *value* of the optimal solution:

Algorithm 1 Compute-OPT(I, k)

```
if k = 0 then
  Return 0.
else
  Return max(Compute-OPT(I, k − 1), Compute-OPT(I, p(k)) + w_k)
end if
```

This algorithm returns the correct solution. Our reasoning above is enough to prove the following claim by induction on k:

Theorem 2 *Given any instance I of the weighted interval scheduling problem, Compute-OPT(I, k) returns the optimal feasible solution $S^* \subseteq [k]$.*

Proof This is an induction on k. In the base case ($k = 0$), the solution is $\text{OPT}(0) = 0$, which is what our algorithm returns. In the inductive case, we may assume that for any $k' < k$, Compute-$\text{OPT}(I, k') = \text{OPT}(k')$. Since $k - 1, p(k) < k$, we therefore have that the algorithm returns:

$$\text{OPT}(k) = \max\left(\text{OPT}(k-1), \text{OPT}(p(k)) + w_k\right)$$

which is correct by our reasoning above. ■

Unfortunately, the depth of the recursion tree generated by Compute-OPT(I, n) is n, and its branching factor is 2, so the recursive algorithm as we have written it runs in exponential time. Can we do better?

The insight that lets us improve is that the recursion is spectacularly wasteful! Despite the fact that the algorithm makes exponentially many recursive calls, most of them are repeats. In fact, there are only $n+1$ distinct calls that are ever made, since the algorithm only takes as input a single varying parameter (k), which ranges from $k \in \{0, \ldots, n\}$. The trick is just to record the solutions to each sub-problem as we compute them, and read them off (rather than recomputing them) if we need them again. The following algorithm simply implements the naive recursion in Algorithm 2, with this caching step. The below algorithm makes use of an array $M[0, \ldots, n]$ in which every entry is initially empty.

Algorithm 2 M-Compute-OPT(I, k)

```
if k = 0 then
  Return 0.
else if M[k] is not empty then
  Return M[k]
else
  Let M[k] = max(M-Compute-OPT(I, k − 1), M-Compute-OPT(I, p(k)) + w_k)
  Return M(k)
end if
```

This is the "memoized" version of Algorithm 2. It returns exactly the same solution, but it never repeats a recursive call that it has made before. Its not hard to see that now the running time is improved from exponential to linear:

Theorem 3 *For any instance I of the interval scheduling problem, M-Compute-OPT(I, n) runs in time $O(n)$.*

Proof Each call of M-Compute-OPT(I,k) runs in $O(1)$ time and makes up to 2 recursive sub-calls. So to bound the running time we must bound the total number of recursive subcalls made. For each k, M-Compute-OPT(I, k) makes its recursive calls only if $M[k]$ is empty. However, after the first such time, it fills in $M[k]$. Hence for each k, calls to M-Compute-OPT(I, k) make at most 2 recursive calls over the entire run of the algorithm. Thus the total number of recursive calls is bounded by $2n$. ■

We're almost done! It only remains to tick a few more boxes. We've implemented a recursive algorithm, which was the most natural way to think about the problem at the start. But recursion can have more overhead than we want in practice, so we might want a straightforward iterative algorithm. This is easy once we observe that all our recursive algorithm is doing is filling out the array M, and that to find the solution to $M[k]$, we only need values of $M[k']$ for $k' < k$. Thus we can just fill out the table from the smallest to largest entry. This kind of implementation is sometimes called bottom-up dynamic programming. This algorithm does exactly the same thing, and now its running time is transparently

Algorithm 3 Iterative-Compute-OPT(I)

```
M[0] = 0
for k = 1 to n do
  M[k] = max(w_j + M[p(k)], M[k − 1])
end for
Return M[n].
```

linear.

Finally, the algorithms we have given so far return the *value* of the optimal solution, but not the solution itself. But the solution is easy to recover as well, once we have computed our array M, by simply retracing the path of the recursion. Recall when we were deriving our recursion, we noted that $\text{OPT}(k) = \text{OPT}(k-1)$ exactly when the optimal solution *did not* include k, and if the optimal solution included k, then we had $\text{OPT}(k) = w_k + \text{OPT}(p(k))$. So given M, we can check which is the case to read off the solution. Here we give the recursive version; you might write down the iterative version as an exercise:

Algorithm 4 Extract-Solution(k)

```
if k = 0 then
  Return ∅.
else
  if w_k + M[p(k)] ≥ M[k − 1] then
    Return {k} ∪ Extract-Solution(p(k))
  else
    Return Extract-Solution(k − 1)
  end if
end if
```

Running Extract-Solution(n) makes at most n recursive calls that take $O(1)$ time each, and so we have:

Theorem 4 *Given the array M generated by M-Compute-OPT(I) (or Iterative-Compute-OPT(I)), Extract-Solution returns the optimal solution to the weighted interval scheduling instance I in $O(n)$ time.*

Thus, we have an $O(n \log n)$ time solution in total (taking into account the possible need to sort the jobs in I initially).