DS 320 Algorithms for Data Science	Lecture #13
Spring 2023	Prof. Kira Goldner

Dynamic Programming I: Weighted Interval Scheduling

Algorithms Recap

- Greedy: blindly takes what's best and it turns out to be optimal.
- Divide & Conquer: Naive/brute force is already polynomial, but by splitting into subproblems and solving recursively, we can give a faster algorithm.
- Now: Dynamic programming. Kind of like more clever D&C, and the opposite of greedy—almost brute-forcing and checking everything, but in a more efficient way.

The Problem

Suppose we are given n jobs. Each job i has a start time s_i, a finish time f_i, and a weight w_i. Our goal is to choose a set S of compatible jobs whose total weight $\sum_{i \in S} w_i$ is maximized.

There are 7 main steps to a dynamic programming algorithm-proof pair.

Step 1: Define your sub-problem. Describe in words what your sub-problem means. This should be in the form of OPT(i) = (or OPT(i, j), etc.) followed by an English description which defines OPT. For each index in your (i, j, etc.) of OPT, you *must* define what that index means.

Let us assume that our n jobs are ordered by non-decreasing finish times f_i. For each job i, let OPT(i) denote the maximum weight of any set of compatible jobs that all finish by f_i.

Step 2: Present your recurrence. Give a mathematical definition of your sub-problem in terms of "smaller" sub-problems.

In order to present the recurrence, we first need some extra notation. Define $p(j)$ to be the job with the largest index less than j that is compatible with job j; in other words, $p(j)$ is the largest k such that $f_k \leq s_j$. If no such job exists, define $p(j) = 0$.

Our recurrence is $\text{OPT}(i) = \max\{\text{OPT}(i-1), w_i + \text{OPT}(p(i))\}$.

Step 3: Prove that your recurrence is correct. Usually a small paragraph. This is equivalent to arguing your inductive step in your proof of correctness.

To prove this is correct, consider the optimal solution for OPT(i). There are two cases: either job i is used in the solution for OPT(i), or it is not.

Case 1: If i is *not* used in the optimal solution for OPT(i), then the maximum-weight set of compatible jobs among jobs 1 through i is just the maximum-weight set of compatible jobs among jobs 1 through $i-1$; by definition, this is OPT($i-1$).

Case 2: If i *is* used in the optimal solution for OPT(i), then since jobs $p(i)+1$ through $i-1$ all conflict with job i, the remaining jobs selected for OPT(i) are drawn from 1 through $p(i)$. Removing i from the optimal solution for OPT(i) yields a compatible solution on jobs $1 \ldots p(i)$. So by the optimality of OPT($p(i)$), $\text{OPT}(i) - w_i \leq \text{OPT}(p(i))$. But similarly, adding job i to OPT($p(i)$) is (by the definition of $p(\cdot)$) compatible, and only uses jobs up through i. Hence $\text{OPT}(i) - w_i \geq \text{OPT}(p(i))$. Therefore $\text{OPT}(i) = w_i + \text{OPT}(p(i))$.

Finally, since OPT(i) is a maximization, the larger of these two cases is the weight of OPT(i).

Step 4: State and prove your base cases. Sometimes only one or two or three bases cases are needed, and sometimes you'll need a lot (say $O(n)$). The latter case typically comes up when dealing with multi-variate sub-problems. You want to make sure that the base cases are enough to get your algorithm off the ground.

As base cases we define $\text{OPT}(0) = 0$ and $\text{OPT}(1) = w_1$, since the max-weight subset of no jobs and one job weigh 0 and w_1, respectively.

Step 5: State how to solve the original problem. Given knowledge of OPT for all relevant indices, how do you compute the answer to the problem you originally set out to solve, on the full input? This will usually be something like OPT(n) or $\max_i \text{OPT}(i)$.

The solution to the original problem will be OPT(n).

Step 6: Present the algorithm. This often involves initializing the base cases and then using your recurrence to solve all the remaining sub-problems in some specific order. You want to ensure that by filling in your table of sub-problems in the correct order, you can compute all the required solutions. Finally, generate the desired solution. Often this is the solution to one of your sub-problems, but not always.

This produces the following algorithm.

Algorithm 1 WeightedIS(s, f, w).

Input: Arrays of start times s_i, finish times f_i, and weights w_i for jobs i indexed from 1 to n.
sort jobs by increasing finish times
compute function $p(i)$ for i from 1 to n
set $\text{OPT}(0) = 0$
for i from 2 to n **do**
  set $\text{OPT}(i) = \max\{\text{OPT}(i-1), w_i + \text{OPT}(p(i))\}$
end for
return OPT(n)

Step 7: Running time. Break your runtime into three parts:

- **a.** Pre-processing: computing base cases, sorting, etc.
- **b.** Filling in memo: This can be further broken down into
  - (a) Number of entries of your memo table.
  - (b) Time to fill each entry. Be careful of things like taking maxes over n elements!
- **c.** Postprocessing: Return statement, etc.

Sorting takes $O(n \log n)$ time. The computation of $p(i)$ can clearly be done in $O(n^2)$ time; if we want to do better, we can either binary search for each i, or all $p(i)$ values can be computed in a single pass if we have already created a second list of jobs sorted by increasing start times. Finally, the main loop is linear; therefore the total running time is $O(n \log n)$.

What about the proof? Well if you've done steps 1 through 7, there isn't really much left to do. Formally you could always use induction, but you'd pretty much be restating what you already wrote. And since that is true of almost all dynamic programming proofs, you can stick to just these 7 steps. Of course, if your proof in step 3 is incorrect, you'll have problems. Same goes if for some reason your order for filling in the table of sub-problem solutions doesn't work. For example, if your algorithm tries to use OPT$(3, 7)$ to solve OPT$(6, 6)$ before your algorithm has solved OPT$(3, 7)$, you may want to rethink things.

The Optimal Solution

Note that we computed the value of an optimal schedule, but not the schedule itself. To actually compute the actual schedule, we have a few options. We *could* change the algorithms so we build lists as we go:

Algorithm 2 WeightedIS-Sol(s, f, w).

Input: Arrays of start times s_i, finish times f_i, and weights w_i for jobs i indexed from 1 to n.
sort jobs by increasing finish times
compute function $p(i)$ for i from 1 to n
set $\text{OPT}(0) = 0$
set $S(0) = \emptyset$
for i from 2 to n **do**
    if $\text{OPT}(i-1) > w_i + \text{OPT}(p(i))$ **then**
        set $S(i) = S(i-1)$ and $\text{OPT}(i) = \text{OPT}(i-1)$
    else
        set $S(i) = \{i\} \cup S(p(i))$ and $\text{OPT}(i) = w_i + \text{OPT}(p(i))$
    end if
end for
return $S(n)$

This version of the algorithm now requires $O(n^2)$ space, since for each index we store a set of jobs (which may be as large as n). Alternatively, we can use the computed values OPT(i) to reverse engineer an optimal set S of jobs to select.

Algorithm 3 computeSol(i, memo).

```
Input: Index i and filled out memo table.
if i == 0 then
    return ∅
else
    if w_i + OPT(p(i)) > OPT(i − 1) then
        return computeSol(p(i), OPT) ∪ {i}
    else
        return computeSol(i − 1, OPT)
    end if
end if
```

How much space does this require?