DS 320 Algorithms for Data Science **Lecture #13 Worksheet**
Spring 2025 Prof. Kira Goldner and Jeffrey Considine

Dynamic Programming I: Weighted Interval Scheduling

Algorithms Recap

- Greedy: Myopically take what's "best" according to some metric at each successive step.
- Divide & Conquer: Naive/brute force is already polynomial, but by splitting into subproblems and solving recursively, we can give a speed-up.
- **Now:** 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.

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

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.

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.

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 $\text{OPT}(n)$ or $\max_i \text{OPT}(i)$.

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.

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.

Space:

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 $\text{OPT}(3,7)$ to solve $\text{OPT}(6,6)$ before your algorithm has solved $\text{OPT}(3,7)$, you may want to rethink things.

Compute the actual schedule: