DS 320 Algorithms for Data Science — **Dynamic Programming**
Spring 2024 — Prof. Kira Goldner

Main Steps

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 $\text{OPT}(i) =$ (or $\text{OPT}(i, j)$, 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. Your pseudocode will reflect this:

Algorithm 1 Pseudocode(Input)

```
Initialize memo array/table of dimensionality that matches the number of parameters of your
subproblem. Make sure the size is enough to contain all of the subproblems you want to fill out,
including base cases. (Refer to steps 1, 2, and 4.)
Do any pre-processing that you may need to do.
for varied parameters of your subproblem moving in the correct direction do
    Fill out the base cases of your memo table. (Make sure this matches your step 4!)
end for
for varied parameters of your subproblem moving in the correct direction do
    Fill out your memo table using your recurrence (Make sure this matches your step 2!)
end for
return the solution to your original problem. (Make sure this matches your step 5!)
```

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

1. Pre-processing: computing base cases, sorting, etc.
2. 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!
3. Postprocessing: Return statement, etc.

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.

An Example: Weighted Interval Scheduling

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. We would like to find a set S of compatible jobs whose total weight is maximized.

Step 1: Subproblem. Let us assume that our n jobs are ordered by non-decreasing finish times f_i. For each job i, let $\text{OPT}(i)$ denote the maximum weight of any set of compatible jobs that all finish by f_i.

Step 2: Recurrence. 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: Proof of Recurrence. To prove this is correct, consider the optimal solution for $\text{OPT}(i)$. There are two cases: either job i is used in the solution for $\text{OPT}(i)$, or it is not.

Case 1: If i is *not* used in the optimal solution for $\text{OPT}(i)$, 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 $\text{OPT}(i-1)$.

Case 2: If i *is* used in the optimal solution for $\text{OPT}(i)$, then since jobs $p(i)+1$ through $i-1$ all conflict with job i, the remaining jobs selected for $\text{OPT}(i)$ are drawn from 1 through $p(i)$. Removing i from the optimal solution for $\text{OPT}(i)$ yields a compatible solution on jobs $1 \ldots p(i)$. So by the optimality of $\text{OPT}(p(i))$, $\text{OPT}(i) - w_i \leq \text{OPT}(p(i))$. But similarly, adding job i to $\text{OPT}(p(i))$ 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 $\text{OPT}(i)$ is a maximization, the larger of these two cases is the weight of $\text{OPT}(i)$.

Step 4: Base Cases. 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: Final Answer. The solution to the original problem will be $\text{OPT}(n)$.

Step 6: Algorithm. This produces the following algorithm.

```
sort jobs by increasing finish times
compute function $p(i)$ for $i$ from 1 to $n$
set memo(0) = 0 and memo(1) = $w_1$
**for** $i$ from 2 to $n$ **do**
    set memo($i$) = $\max\{\text{memo}(i-1), w_i + \text{memo}(p(i))\}$
**end for**
**return** memo($n$)
```

Step 7: Runtime. 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 $O(n \log n)$.

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. One is to use the computed values $\text{OPT}(i)$ to reverse engineer an optimal set S of jobs to select. Alternatively, we can change the algorithms so we build lists as we go:

```
sort jobs by increasing finish times
compute function $p(i)$ for $i$ from 1 to $n$
set memo(0) = 0 and memo(1) = $w_1$
set $S(0) = \emptyset$ and $S(1) = \{1\}$
**for** $i$ from 2 to $n$ **do**
    **if** memo($i-1$) > $w_i$ + memo($p(i)$) **then**
        set $S(i) = S(i-1)$ and memo($i$) = memo($i-1$)
    **else**
        set $S(i) = \{i\} \cup S(p(i))$ and memo($i$) = $w_i$ + memo($p(i)$)
    **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). Can you see how to modify this so that it only uses $O(n)$ space?