Introduction to Algorithms **Outline for Dynamic Programming**
CS 482 Spring 2006

Main Steps

Solutions using dynamic programming all have a number of common points.

Step 1: Define your sub-problem. Describe in English what your sub-problem means, whether it look like $P(k)$ or $R(i, j)$ or anything else. For example, you might write "Let $S(k)$ be the largest number of monkeys you can cram into the first k boxcars".

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.

Step 4: State your base cases. Sometimes only one or two bases cases are needed, and sometimes you'll need a lot (say $O(n)$). The later 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: Present the algorithm. This often involves initializing the base cases and then using your recurrence to solve all the remaining sub-problems. 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 6: Running time. As usual.

What about the proof? Well if you've done steps 1 through 6, 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 6 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 $S(3, 7)$ to solve $S(6, 6)$ before your algorithm has solved $S(3, 7)$, 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.

Let us assume that our n jobs are ordered by non-decreasing finish times f_i. Let $S(i)$ be the maximum weight of any set of compatible jobs, all of which finish by f_i.

Define $p(j)$ to be the job with the largest index less than j which is compatible with job j. In other words, the largest k such that $f_k \leq s_j$. If no such job exists define $p(j) = 0$.

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

Consider the optimal solution for $S(i)$. Either job i is used or it is not. If it is not, then we have a maximum weight set of compatible jobs from among jobs 1 through $i-1$. By definition, this is $S(i-1)$. Alternatively, i is used for $S(i)$. Since jobs $p(i)+1$ through $i-1$ all conflict with job i, this means that the remaining jobs selected for $S(i)$ are drawn from 1 through $p(i)$. Removing i from the optimal solution for $S(i)$ yields a compatible solution on jobs $1 \ldots p(i)$. So by the optimality of $S(p(i))$, $S(i) - w_i \leq S(p(i))$. But similarly, adding job i to $S(p(i))$ is (by the definition of $p(\cdot)$) compatible, and only uses jobs up through i. Hence $S(i) - w_i \geq S(p(i))$. Therefore $S(i) = w_i + S(p(i))$. Finally, since $S(i)$ is a maximization, the larger of these two terms is correct.

As base cases, we define $S(0) = 0$ and $S(1) = w_1$.

The algorithm works as follows:

sort jobs by increasing finish times.
compute function $p(i)$ for i from 1 to n (this 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 linear pass if we have already created a second list of jobs sorted by increasing start times).

set $S(0) = 0$ and $S(1) = w_1$
loop over i from 2 to n
    set $S(i) = \max\{S(i-1), w_i + S(p(i))\}$
endloop

Sorting takes $O(n \log n)$ time. The computation of $p(i)$ takes another $O(n \log n)$ time and the main loop are both linear, and 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 $S(i)$ to reverse engineer an optinal set A 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 $S(0) = 0$ and $S(1) = w_1$
set $A(0) =$ and $A(1) = \{1\}$
loop over i from 2 to n
    if $S(i-1) > w_i + S(p(i))$
        set $A(i) = A(i-1)$ and $S(i) = S(i-1)$
    else $S(i-1) \leq w_i + S(p(i))$
        set $A(i) = \{i\} \cup A(p(i))$ and $S(i) = w_i + S(p(i))$
    endif
endloop

Of course, this version of the algorithm now requires using $O(n^2)$ 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 onle uses $O(n)$ space?