CSC 364S Notes **University of Toronto, Spring, 2004**

Dynamic Programming Algorithms

The setting is as follows. We wish to find a solution to a given problem which optimizes some quantity Q of interest; for example, we might wish to maximize profit or minimize cost. The algorithm works by *generalizing* the original problem. More specifically, it works by creating an array of related but simpler problems, and then finding the optimal *value* of Q for each of these problems; we calculate the values for the more complicated problems by using the values already calculated for the easier problems. When we are done, the optimal value of Q for the original problem can be easily computed from one or more values in the array. We then use the array of values computed in order to compute a *solution* for the original problem that attains this optimal value for Q. We will always present a dynamic programming algorithm in the following 4 steps.

Step 1:
Describe an array (or arrays) of values that you want to compute. (Do not say how to compute them, but rather describe what it is that you want to compute.) Say how to use certain elements of this array to compute the optimal value for the original problem.

Step 2:
Give a recurrence relating some values in the array to other values in the array; for the simplest entries, the recurrence should say how to compute their values from scratch. Then (unless the recurrence is obviously true) justify or prove that the recurrence is correct.

Step 3:
Give a high-level program for computing the values of the array, using the above recurrence. Note that one computes these values in a bottom-up fashion, using values that have already been computed in order to compute new values. (One does not compute the values recursively, since this would usually cause many values to be computed over and over again, yielding a very inefficient algorithm.) Usually this step is very easy to do, using the recurrence from Step 2. Sometimes one will also compute the values for an auxiliary array, in order to make the computation of a solution in Step 4 more efficient.

Step 4:
Show how to use the values in the array(s) (computed in Step 3) to compute an optimal solution to the original problem. Usually one will use the recurrence from Step 2 to do this.

Scheduling Jobs With Deadlines, Profits, and Durations

In the notes on Greedy Algorithms, we saw an efficient greedy algorithm for the problem of scheduling unit-length jobs which have deadlines and profits. We will now consider a generalization of this problem, where instead of being unit-length, each job now has a *duration* (or processing time).

More specifically, the input will consist of information about n jobs, where for job i we have a nonnegative real valued profit $g_i \in \mathbb{R}^{\geq 0}$, a deadline $d_i \in \mathbb{N}$, and a duration $t_i \in \mathbb{N}$. It is convenient to think of a schedule as being a sequence $C = C(1), C(2), \cdots, C(n)$; if $C(i) = -1$, this means that job i is not scheduled, otherwise $C(i) \in \mathbb{N}$ is the time at which job i is scheduled to begin.

We say that schedule C is *feasible* if the following two properties hold.

(a) Each job finishes by its deadline. That is, for every i, if $C(i) \geq 0$, then $C(i) + t_i \leq d_i$.

(b) No two jobs overlap in the schedule. That is, If $C(i) \geq 0$ and $C(j) \geq 0$ and $i \neq j$, then either $C(i) + t_i \leq C(j)$ or $C(j) + t_j \leq C(i)$. (Note that we permit one job to finish at exactly the same time as another begins.)

We define the profit of a feasible schedule C by $P(C) = \sum_{C(i) \geq 0} g_i$.

We now define the problem of Job Scheduling with Deadlines, Profits and Durations:

Input A list of jobs (d_1, t_1, g_1),..., (d_n, t_n, g_n)

Output A feasible schedule $C = C(1), ..., C(n)$ such that the profit $P(C)$ is the maximum possible among all feasible schedules.

Before beginning the main part of our dynamic programming algorithm, we will sort the jobs according to deadline, so that $d_1 \leq d_2 \leq \cdots \leq d_n = d$, where d is the largest deadline. Looking ahead to how our dynamic programming algorithm will work, it turns out that it is important that we prove the following lemma.

Lemma 1 *Let C be a feasible schedule such that at least one job is scheduled; let $i > 0$ be the largest job number that is scheduled in C. Say that every job that is scheduled in C finishes by time t. Then there is feasible schedule C' that schedules exactly the same jobs as C, and such that $C'(i) = \min\{t, d_i\} - t_i$, and such that all other jobs scheduled by C' end at or before time* $\min\{t, d_i\} - t_i$.

Proof: This proof uses the fact the the jobs are sorted according to deadline. The details are left as an exercise.

We now perform the four steps of a dynamic programming algorithm.

Step 1: Describe an array of values we want to compute.
Define the array $A(i,t)$ for $0 \le i \le n$, $0 \le t \le d$ by

$$A(i,t) = \max\{P(C)|C \text{ is a feasible schedule in which only jobs from } \{1,\cdots,i\} \\ \text{are scheduled, and all scheduled jobs finish by time } t\ \}.$$

Note that the value of the profit of the optimal schedule that we are ultimately interested in, is exactly $A(n,d)$.

Step 2: Give a recurrence.
This recurrence will allow us to compute the values of A one row at a time, where by the ith row of A we mean $A(i,0),\cdots,A(i,d)$.

- $A(0,t) = 0$ for all t, $0 \le t \le d$.
- Let $1 \le i \le n$, $0 \le t \le d$. Define $t' = \min\{t, d_i\} - t_i$. Clearly t' is the latest possible time that we can schedule job i, so that it ends both by its deadline and by time t. Then we have:

If $t' < 0$, then $A(i,t) = A(i-1,t)$.
If $t' \ge 0$, then $A(i,t) = \max\{A(i-1,t), g_i + A(i-1,t')\}$.

We now must explain (or prove) why this recurrence is true. Clearly $A(0,t) = 0$.

To see why the second part of the recurrence is true, first consider the case where $t' < 0$. Then we cannot (feasibly) schedule job i so as to end by time t, so clearly $A(i,t) = A(i-1,t)$. Now assume that $t' \ge 0$. We have a choice of whether or not to schedule job i. If we don't schedule job i, then the best profit we can get (from scheduling some jobs from $\{1,\cdots i\}$ so that all end by time t) is $A(i-1,t)$. If we do schedule job i, then the previous lemma tells us that we can assume job i is scheduled at time t' and all the other scheduled jobs end by time t', and so the best profit we can get is $g_i + A(i-1,t')$.

Although the above argument is pretty convincing, sometimes we want to give a more rigorous proof of our recurrence, or at least the most difficult part of the recurrence.
THIS ARGUMENT CAN BE OMITTED ON THE FIRST READING.
Say that $1 \le i \le n$, $0 \le t \le d$, and $t' = \min\{t, d_i\} - t_i \ge 0$; we want to prove that $A(i,t) = \max\{A(i-1,t), g_i + A(i-1,t')\}$. To prove such an equality, it is usually convenient

to prove two *inequalities*:
$A(i,t) \geq \max\{A(i-1,t), g_i + A(i-1,t')\}$ and
$A(i,t) \leq \max\{A(i-1,t), g_i + A(i-1,t')\}$.

- To prove $A(i,t) \geq \max\{A(i-1,t), g_i + A(i-1,t')\}$, we prove that *both* $A(i,t) \geq A(i-1,t)$ and $A(i,t) \geq g_i + A(i-1,t')$ hold.
  - To show that $A(i,t) \geq A(i-1,t)$, consider a feasible schedule C of profit $A(i-1,t)$ that schedules only jobs from $\{1, \cdots, i-1\}$ so that all jobs finish by time t. Clearly C schedules only jobs from $\{1, \cdots, i\}$ so that all jobs finish by time t, so C's profit is less than or equal to the best such profit, so $A(i-1,t) \leq A(i,t)$.
  - To show that $A(i,t) \geq g_i + A(i-1,t')$, Consider a feasible schedule C of profit $A(i-1,t')$ that schedules only jobs from $\{1, \cdots, i-1\}$ so that all jobs finish by time t'. Let C' be the feasible schedule that extends C by scheduling job i beginning at time t'. Then C' is a feasible schedule that schedules only jobs from $\{1, \cdots, i\}$, all ending by time t, and $P(C') = g_i + A(i-1,t')$. C' has a profit that is less than or equal to the best such profit, so $g_i + A(i-1,t) \leq A(i,t)$.

- To prove that $A(i,t) \leq \max\{A(i-1,t), g_i + A(i-1,t')\}$, we prove that *either* $A(i,t) \leq A(i-1,t)$ or $A(i,t) \leq g_i + A(i-1,t')$ holds. Let C be a feasible schedule that only schedules jobs from $\{1, \cdots, i\}$ such that all jobs end by time t, and such that $P(C) = A(i,t)$. We consider two cases:
  - **Case 1:** C does not schedule job i. Then C schedules only jobs from $\{1, \cdots, i-1\}$ so that all jobs finish by time t, so C's profit is less than or equal to the best such profit, so $A(i,t) \leq A(i-1,t)$.
  - **Case 2:** C schedules job i. Then by the previous lemma, there is a feasible schedule C' such that C' schedules the same jobs as C, C' schedules job i beginning at time t', and C' schedules all its other jobs so that they end by time t'; we have $P(C') = P(C) = A(i,t)$. Let C'' be the feasible schedule that is the same as C', except that C'' doesn't schedule job i; so $P(C'') = A(i,t) - g_i$. Since C'' is a feasible schedule such that only jobs from $\{1, \cdots i-1\}$ are scheduled and such that all jobs end by time t', its profit is less than or equal to the best such profit, so $A(i,t) - g_i \leq A(i-1,t')$, so $A(i,t) \leq g_i + A(i-1,t')$.

Step 3: Give a high-level program.
We now give a high-level program that computes values into an array B, so that we will

have $B[i,t] = A(i,t)$. (The reason we call our array B instead of A, is to make it convenient to prove that the values computed into B actually are the values of the array A defined above. This proof is usually a simple induction proof that uses the above recurrence, and so usually this proof is omitted.)

```
for every t ∈ {0, ···, d}
    B[0, t] ← 0
end for
for i : 1..n
    for every t ∈ {0, ···, d}
        t' ← min{t, d_i} − t_i
        if t' < 0 then
            B[i, t] ← B[i − 1, t]
        else
            B[i, t] ← max{B[i − 1, t], g_i + B[i − 1, t']}
        end if
    end for
end for
```

Step 4: Compute an optimal solution.
Let us assume we have correctly computed the values of the array A into the array B. It is now convenient to define a "helping" procedure PrintOpt(i,t) that will call itself recursively. Whenever we use a helping procedure, it is important that we specify the appropriate precondition/postcondition for it. In this case, we have:
Precondition: i and t are integers, $0 \leq i \leq n$ and $0 \leq t \leq d$.
Postcondition: A schedule is printed out that is an optimal way of scheduling only jobs from $\{1, \cdots, i\}$ so that all jobs end by time t.

We can now print out an optimal schedule by calling
PrintOpt(n,d)
Note that we have written a recursive program since a simple iterative version would print out the schedule in reverse order. It is easy to prove that PrintOpt(i,t) works, by induction on i. The full program (assuming we have already computed the correct values into B) is as follows:

```
procedure PrintOpt(i, t)
    if i = 0 then return end if
    if B[i, t] = B[i − 1, t] then
        PrintOpt(i − 1, t)
    else
        t′ ← min{t, d_i} − t_i
        PrintOpt(i − 1, t′)
        put "Schedule job", i, "at time", t′
    end if
end PrintOpt

PrintOpt(n, d)
```

Analysis of the Running Time

The initial sorting can be done in time $O(n \log n)$. The program in Step 3 clearly takes time $O(nd)$. Therefore we can compute the entire array A in total time $O(nd + n \log n)$. When d is large, this expression is dominated by the term nd. It would be nice if we could state a running time of simply $O(nd)$. Here is one way to do this. When $d \leq n$, instead of using an $n \log n$ sorting algorithm, we can so something faster by noting that we are sorting n numbers from the range 0 to n; this can easily be done (using only $O(n)$ extra storage) in time $O(n)$. Therefore, we can compute the entire array A within total time $O(nd)$. Step 4 runs in time $O(n)$. So our total time to compute an optimal schedule is in $O(nd)$. Keep in mind that we are assuming that each arithmetic operation can be done in constant time.

Should this be considered a polynomial-time algorithm? If we are guaranteed that on all inputs d will be less than, say, n^2, then the algorithm can be considered a polynomial-time algorithm.

More generally, however, the best way to address this question is to view the input as a sequence of bits rather than integers or real numbers. In this model, let us assume that all numbers are integers represented in binary notation. So if the $3n$ integers are represented with about k bits each, then the actual bit-size of the input is about nk bits. It is not hard to see that each arithmetic operation can be done in time polynomial in the bit-size of the input, but how many operations will the algorithm perform? Since d is a k bit number, d can be as large as 2^k. Since 2^k is not polynomial in nk, the algorithm is *not* a polynomial-time algorithm in this setting,.

Now consider a slightly different setting. As before, the profits are expressed as binary integers. The durations and deadlines, however, are expressed in *unary* notation. This means that the integer m is expressed as a string of m ones. Hence, d is now less than the bit-size of the input, and so the algorithm is polynomial-time in this setting.

Actually, in order to decide if this algorithm should be used in a specific application, all you really have to know is that it performs about nd arithmetic operations. You can then compare it with other algorithms you know. In this case, perhaps the only other algorithm you know is the "brute force" algorithm that tries all possible subsets of the jobs, seeing which ones can be (feasibly) scheduled. Using the previous lemma, we can test if a set of jobs can be feasibly scheduled in time $O(n)$, so the brute-force algorithm can be implemented to run in time about $n2^n$. Therefore, if d is much less than 2^n then the dynamic programming algorithm is better; if d is much bigger than 2^n then the brute-force algorithm is better; if d is comparable with 2^n, then probably one has to do some program testing to see which algorithm is better for a particular application.

The (General) Knapsack Problem

First, recall the Simple Knapsack Problem from the notes on Greedy algorithms. We are given a sequence of nonnegative integer weights $w_1, \cdots, w_n$ and a nonnegative integer capacity C, and we wish to find a subset of the weights that adds up to as large a number as possible without exceeding C. In effect, in the Simple Knapsack Problem we treat the weight of each item as its profit. In the (general) Knapsack Problem, we have a separate (nonnegative) profit for each job; we wish to find the most profitable knapsack possible among those whose weight does not exceed C. More formally, we define the problem as follows.

Let $w_1, \cdots, w_n \in \mathbb{N}$ be weights, let $g_1, \cdots, g_n \in \mathbb{R}^{\geq 0}$ be profits, and let $C \in \mathbb{N}$ be a weight. For each $S \subseteq \{1, \cdots, n\}$ let $K(S) = \sum_{i \in S} w_i$ and let $P(S) = \sum_{i \in S} g_i$. (Note that $K(\emptyset) = P(\emptyset) = 0$.) We call $S \subseteq \{1, \cdots, n\}$ *feasible* if $K(S) \leq C$.
The goal is to find a feasible S so that $P(S)$ is as large as possible.

We can view a Simple Knapsack Problem as a special case of a General Knapsack Problem, where for every i, $g_i = w_i$.

Furthermore, we can view a General Knapsack Problem as a special case of a Scheduling With Deadlines, Profits and Durations Problem, where all the deadlines are the same. To see this, say that we are given a General Knapsack Problem with capacity C and with n items, where the ith item has weight w_i and profit g_i. We then create a scheduling problem with n jobs, where the ith job has duration w_i, profit g_i, and deadline C. A solution to this scheduling problem yields a solution to the knapsack problem, and so we can solve the General Knapsack Problem with a dynamic programming algorithm that runs in time $O(nC)$.

All pairs Shortest Path Problem

We define a *directed graph* to be a pair $G = (V, E)$ where V is a set of vertices and $E \subseteq V \times V$ is a set of (directed) edges. Sometimes we will consider *weighted* graphs where associated with each edge (i, j) is a weight (or cost) $c(i, j)$. A (directed) path in G is a sequence of one or more vertices $v_1, \cdots, v_m$ such that $(v_i, v_{i+1}) \in E$ for every i, $1 \le i < m$; we say this is a path from v_1 to v_m. The cost of this path is defined to be the sum of the costs of the $m - 1$ edges in the path; if $m = 1$ (so that the path consists of a single node and no edges) then the cost of the path is 0.

Given a directed, weighted graph, we wish to find, for every pair of vertices u and v, the cost of a cheapest path from u to v. This should be called the "all pairs cheapest path problem", and that is how we will refer to it from now on, but traditionally it has been called the "all pairs shortest path problem". We will give the Floyd-Warshall dynamic programming algorithm for this problem.

Let us assume that $V = \{1, \cdots n\}$, and that *all* edges are present in the graph. We are given n^2 costs $c(i, j) \in \mathbb{R}^{\ge 0} \cup \{\infty\}$ for every $1 \le i, j \le n$. Note that our costs are either nonnegative real numbers or the symbol "∞". We don't allow negative costs, since then cheapest paths might not exist: there might be arbitrarily small negative-cost paths from one vertex to another. We allow ∞ as a cost in order to denote that we really view that edge as not existing. (We do arithmetic on ∞ in the obvious way: ∞ plus ∞ is ∞, and ∞ plus any real number is ∞.)

For $1 \le i, j \le n$, define $D(i, j)$ to be the cost of a cheapest path from i to j. Our goal is to compute *all* of the values $D(i, j)$.

Step 1: Describe an array of values we want to compute.
For $0 \le k \le n$ and $1 \le i, j \le n$, define
$A(k, i, j)$ = the cost of a cheapest path from i to j, from among those paths from i to j whose *intermediate* nodes are all in $\{1, \cdots, k\}$. (We define the intermediate nodes of the path $v_1, \cdots, v_m$ to be the set $\{v_2, \cdots, v_{m-1}\}$; note that if m is 1 or 2, then this set is empty.)

Note that the values we are ultimately interested in are the values $A(n, i, j)$ for all $1 \le i, j \le n$.

Step 2: Give a recurrence.

- If $k = 0$ and $i = j$, then $A(k,i,j) = 0$.
  If $k = 0$ and $i \neq j$, then $A(k,i,j) = c(i,j)$.

  This part of the recurrence is obvious, since a path with *no* intermediate nodes can only consist of 0 or 1 edge.

- If $k > 0$, then $A(k,i,j) = \min\{A(k-1,i,j), A(k-1,i,k) + A(k-1,k,j)\}$.

  The reason this equation holds is as follows. We are interested in cheapest paths from i to j whose intermediate vertices are all in $\{1, \cdots, k\}$. Consider a cheapest such path p. If p doesn't contain k as an intermediate node, then p has cost $A(k-1,i,j)$; if p does contain k as an intermediate node, then (since costs are nonnegative) we can assume that k occurs only once as an intermediate node on p. The subpath of p from i to k must have cost $A(k-1,i,k)$ and the subpath from k to j must have cost $A(k-1,k,j)$, so the cost of p is $A(k-1,i,k) + A(k-1,k,j)$.

We leave it as an exercise to give a more rigorous proof of this recurrence along the lines of the proof given for the problem of scheduling with deadlines, profits and durations.

Step 3: Give a high-level program.
We could compute the values we want using a 3-dimensional array $B[0..n, 1..n, 1..n]$ in a very straightforward way. However, it suffices to use a 2-dimensional array $B[1..n, 1..n]$; the idea is that after k executions of the body of the for-loop, $B[i,j]$ will equal $A(k,i,j)$. We will also use an array $B'[1..n, 1..n]$ that will be useful when we want to compute cheapest paths (rather than just costs of cheapest paths) in Step 4.

All_Pairs_CP

```
for i : 1..n do
    B[i,i] ← 0
    B'[i,i] ← 0
    for j : 1..n such that j ≠ i do
        B[i,j] ← C(i,j)
        B'[i,j] ← 0
    end for
end for

for k : 1..n do
    for i : 1..n do
        for j : 1..n do
            if B[i,k] + B[k,j] < B[i,j] then
                B[i,j] ← B[i,k] + B[k,j]
                B'[i,j] ← k
            end if
        end for
    end for
end for
```

We want to prove the following lemma about this program.

Lemma: For every k, i, j such that $0 \leq k \leq n$ and $1 \leq i, j \leq n$, after the kth execution of the body of the for-loop the following hold:

- $B[i,j] = A(k,i,j)$
- $B'[i,j]$ is the smallest number such that there exists a path p from i to j all of whose intermediate vertices are in $\{1, \cdots, B'[i,j]\}$, such that the cost of p is $A(k,i,j)$. (Note that this implies that $B'[i,j] \leq k$).

Proof: We prove this by induction on k. The base case is easy. To see why the induction step holds for the first part, we only have to worry about the fact that when we are computing the kth version of $B[i,j]$, some elements of B have already been updated. That is, $B[i,k]$ might be equal to $A(k-1,i,k)$, or it might have already been updated to be equal to $A(k,i,k)$ (and similarly for $B[k,j]$); however this doesn't matter, since $A(k-1,i,k) = A(k,i,k)$. The rest of the details, including the part of the induction step for the second part of the Lemma, are left as an exercise. $\Box$

This Lemma implies that when the program has finished running, $B'[i,j]$ is the smallest number such that there exists a path p from i to j all of whose intermediate vertices are in $\{1, \cdots, B'[i,j]\}$, such that the cost of p is $D(i,j)$.

Step 4: Compute an optimal solution.
For this problem, computing an optimal solution can mean one of two different things. One possibility is that we want to print out a cheapest path from i to j, for *every* pair of vertices (i,j). Another possibility is that after computing B and B', we will be given an arbitrary pair (i,j), and we will want to compute a cheapest path from i to j as quickly as possible; this is the situation we are interested in here.

Assume we have already computed the arrays B and B'; we are now given a pair of vertices i and j, and we want to print out the edges in some cheapest path from i to j. If $i = j$ then we don't print out anything; otherwise we will call PRINTOPT(i,j). The call PRINTOPT(i,j) will satisfy the following Precondition/Postcondition pair:
Precondition: $1 \le i, j \le n$ and $i \ne j$.
Postcondition The edges of a path p have been printed out such that p is a path from i to j, and such that all the intermediate vertices of p are in $\{1, \cdots, B'[i,j]\}$, and such that no vertex occurs more than once in p. (Note that this holds even if there are edges of 0 cost in the graph.)

The full program (assuming we have already computed the correct values into B') is as follows:

```
procedure PrintOpt(i, j)
    k ← B'[i, j]
    if k = 0 then
      put "edge from", i, "to", j
    else
      PrintOpt(i, k)
      PrintOpt(k, j)
    end if
end PrintOpt

if i ≠ j then PrintOpt(i, j) end if
```

Exercise:
Prove that the call PRINTOPT(i,j) satisfies the above Precondition/Postcondition pair.
Prove that if $i \ne j$, then PRINTOPT(i,j) runs in time linear in the number of edges printed out; conclude that the whole program in Step 4 runs in time $O(n)$.

Analysis of the Running Time
The program in Step 3 clearly runs in time $O(n^3)$, and the Exercise tells us that the program in Step 4 runs in time $O(n)$. So the total time is $O(n^3)$. We can view the size of the input as n – the number of vertices, or as n^2 – an upper bound on the number of edges. In any case, this is clearly a polynomial-time algorithm. (Note that if in Step 4 we want to print out cheapest paths for *all* pairs i, j, this would still take just time $O(n^3)$.)

Remark: The recurrence in Step 2 is actually not as obvious as it might at first appear. It is instructive to consider a slightly different problem, where we want to find the cost of a *longest* (that is, most expensive) path between every pair of vertices. Let us assume that there is an edge between every pair of vertices, with a cost that is a real number. The notion of longest path is still not well defined, since if there is a cycle with positive cost, then there will be arbitrarily costly paths between every pair of points. It does make sense, however, to ask for the length of a longest *simple* path between every pair of points. (A simple path is one on which no vertex repeats.) So define $D(i, j)$ to be the cost of a most expensive simple path from i to j. Define $A(k, i, j)$ to be the cost of a most expensive path from i to j from among those whose intermediate vertices are in $\{1, 2, \ldots, k\}$. Then it is *not* necessarily true that $A(k, i, j) = \max\{A(k-1, i, j), A(k-1, i, k) + A(k-1, k, j)\}$. Do you see why?

Activity Selection with Profits

The book CLR, as well as the Notes on Greedy Algorithms, considers the problem of Activity Selection, and gives a greedy algorithm that works for this problem. We will now consider the more difficult problem of of Activity Selection with Profits, and give a dynamic programming algorithm for this problem.

The input is information about n activities; for the ith activity we are given three non-negative real numbers (s_i, f_i, g_i) where s_i is the start time of activity i, f_i is the finish time of activity i (and $s_i < f_i$), and $g_i \geq 0$ is the profit we get if we schedule activity i. For $i, j \in \{1, 2, \cdots, n\}$, $i \neq j$, we say i and j are *compatible* if the corresponding courses don't overlap, that is, if $f_i \leq s_j$ or $f_j \leq s_i$. A (feasible) *schedule* is a set $S \subseteq \{1, 2, \cdots, n\}$ such that every two distinct numbers in S are compatible. The *profit* of a schedule S is $P(S) = \sum_{i \in S} g_i$. We want an algorithm for finding a schedule that maximizes profit. (Note that we are assuming, for convenience, that $s_i < f_i$ for every activity i, rather than merely $s_i \leq f_i$. This is no great loss: if trivial activities – that is, activities of 0 duration – exist, we can always remove them, find an optimal schedule S, and then add the trivial activities to S.)

Before starting the dynamic programming algorithm itself, we do some precomputation, as follows. We first sort the activities according to finish time, so that $f_1 \leq f_2 \leq \ldots \leq f_n$. We also compute, for every activity i, a number $H(i)$ defined as $H(i) = \max\{l \in \{1, 2, \ldots, i-1\} \mid f_l \leq s_i\}$; the maximum value of the empty set is 0. We can compute each value $H(i)$ in time $O(\log n)$ using binary search, so all of the precomputation can be done in time $O(n \log n)$.

We now perform the four steps of the dynamic programming method.

Step 1: Describe an array of values we want to compute.
For every integer i, $0 \leq i \leq n$, define
$A(i)$ = the largest profit we can get by (feasibly) scheduling activities from $\{1, 2, \ldots, i\}$.

The value we are ultimately interested in is $A(n)$.

Step 2: Give a recurrence.

- $A(0) = 0$.
  This is true because if we are not allowed to schedule any activities at all, then the highest profit we can make is 0.

- Let $1 \leq i \leq n$. Then
  $A(i) = \max\{A(i-1), g_i + A(H(i))\}$.

  To see why this equality holds, consider a best possible schedule S among those containing activities from $\{1, 2, \ldots, i\}$. If $i \notin S$, then $P(S) = A(i-1)$. Otherwise $i \in S$; because the activities were sorted by finish time, the other activities in S have finish times $\leq f_i$, and therefore must have finish times $\leq s_i$; so the rest of S must be an optimal schedule among those that contain activities from $\{1, 2, \ldots, H(i)\}$.

Step 3: Give a high-level program.
We leave it as an exercise to give a program for computing the values of A in time $O(n)$.

Step 4: Compute an optimal solution.
This is left as an exercise. If one is careful, then using the array A, one can compute an optimal schedule in time $O(n)$.

The total time used by this algorithm is $O(n \log n)$. It is a polynomial-time algorithm.

Example

Activity i:	1	2	3	4
Start s_i:	0	2	3	2
Finish f_i:	3	6	6	10
Profit g_i:	20	30	20	30
$H(i)$:	0	0	1	0

$$
\begin{aligned}
&A(0) = 0 \\
&A(1) = \max\{0, 20 + A(H(1))\} = 20 \\
&A(2) = \max\{20, 30 + A(H(2))\} = 30 \\
&A(3) = \max\{30, 20 + A(H(3))\} = 40 \\
&A(4) = \max\{40, 30 + A(H(4))\} = 40
\end{aligned}
$$

The reader should note that the problem of scheduling activities with profits can be generalized to multiple processors, and an appropriate generalization of the above dynamic programming algorithm can be used to solve this more general problem. This algorithm will run in polynomial time, but only if the number of prcessors is fixed. (That is, if the running time is $O(n^c)$, then the exponent c may depend on the number of processors.)

Finding a Longest Increasing Subsequence

Our input is a sequence of *distinct* integers $\vec{a} = a_1, a_2, \ldots, a_n$. We wish to find a longest increasing subsequence. More specifically, if $1 \leq i_1 < i_2 < \ldots < i_k \leq n$ is a sequence of integers such that $a_{i_1} < a_{i_2} < \ldots < a_{i_k}$, we say $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ is an increasing subsequence of $\vec{a}$; our goal is to find such an increasing subsequence where k is as large as possible. We will do a dynamic programming algorithm.

Step 1: Describe an array of values we want to compute.
For $1 \leq i \leq n$, let $A(i)$ be the length of a longest increasing sequence of $\vec{a}$ that end with a_i. Note that the length we are ultimately interested in is $\max\{A(i) \mid 1 \leq i \leq n\}$.

Step 2: Give a recurrence.
For $1 \leq i \leq n$,
$A(i) = 1 + \max\{A(j) \mid 1 \leq j < i \text{ and } a_j < a_i\}$.
(We assume $\max \emptyset = 0$.)

We leave it as an exercise to explain why, or to prove that, this recurrence is true.

Step 3: Give a high-level program to compute the values of A.
This is left as an exercise. It is not hard to design this program so that it runs in time $O(n^2)$. (In fact, using a more fancy data structure, it is possible to do this in time $O(n \log n)$.)

Step 4: Compute an optimal solution.
The following program uses A to compute an optimal solution. The first part computes a value m such that $A(m)$ is the length of an optimal increasing subsequence of $\vec{a}$. The second part computes an optimal increasing subsequence, but for convenience we print it out in reverse order.

```
m ← 1
for i : 2..n
    if A(i) > A(m) then
        m ← i
    end if
end for

put a_m
while A(m) > 1 do
    i ← m − 1
    while not(a_i < a_m and A(i) = A(m) − 1) do
        i ← i − 1
    end while
    m ← i
    put a_m
end while
```

The program in Step 4 runs in time $O(n)$, so the entire algorithm runs in time $O(n^2)$.