Dynamic Programming

Dynamic programming is another important algorithmic technique. The main idea behind it is to solve a big problem by combining the solutions to smaller subproblems.
We usually want to maximize profit or minimize cost. A dynamic programming algorithm works by creating an array of related but simpler subproblems, and then, it computes the solution to the big complicated problem by using the solutions to the easier subproblems which are stored in this array.

Longest Monotone Subsequence Problem

Input: $d, a_1, a_2, \ldots, a_d \in \mathbb{N}$
Problem: Find L = length of the longest monotone non-decreasing subsequence.
Note: the subsequence need not be consecutive. Thus $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ is a monotone subsequence provided $1 \leq i_1 < i_2 < \ldots < i_k \leq d$ and $a_{i_1} \leq a_{i_2} \leq \ldots \leq a_{i_k}$.

Example 1 The length of the longest monotone subsequence of $\{4, 6, 5, 9, 1\}$ is 3.

We want to give a dynamic programming solution to the LMS Problem:

1 Define an array of subproblems: $R(j)$ = length of the longest monotone subsequence which ends in a_j. Then the answer would be $L = \max_{1 \leq j \leq n} R(j)$.

2 Find a recurrence: let $R(1) = 1$, and for $j > 1$,

$$R(j) = \begin{cases} 1 & \text{if } a_i > a_j \text{ for all } 1 \leq i < j \\ 1 + \max_{1 \leq i < j}\{R(i) | a_i \leq a_j\} & \text{otherwise} \end{cases}$$

3 Write an algorithm that computes R:

```
R(1) ← 1
for j : 2..d
    max ← 0
    for i : 1..j − 1
        if R(i) > max and a_i ≤ a_j then
            max ← R(i)
        end if
    end for
    R(j) ← max +1
end for
```

What are the pre/post-conditions? What is an appropriate loop invariant?

All Pairs Shortest Path Problem

Input: Directed graph $G = (V, E)$, $V = \{1, 2, \ldots, n\}$. Cost $C(i, j) \in \mathbb{R}^+ \cup \{\infty\}$ (think of the costs as distances), $1 \leq i, j \leq n$, $C(i, j) = \infty$ if (i, j) is not an edge.
Problem: Find $D(i, j)$, the length of the shortest directed path from i to j.
Note that for a general graph on n nodes there are exponentially many (in n) paths, so exhaustive search is not feasible.

1 Define an array of subproblems: Let $A(k, i, j)$ be the length of the shortest path from i to j s.t. all *intermediate* nodes on the path are in $\{1, 2, \ldots, k\}$. Then $A(n, i, j) = D(i, j)$ is the solution. Notational convention: if $k = 0$ then $\{1, 2, \ldots, k\} = \emptyset$.

2 Define a recurrence: Initialization: for $k = 0$, $A(0, i, j) = C(i, j)$. Now for $k > 0$. To come up with a recurrence, notice that the shortest path between i and j either includes k or does not. Assume we know $A(k-1, r, s)$ for all r, s. Suppose node k is not included. Then, obviously, $A(k, i, j) = A(k-1, i, j)$. If node k occurs on a shortest path, then it occurs exactly once, so $A(k, i, j) = A(k-1, i, k) + A(k-1, k, j)$. The shortest path length is obtained by taking the minimum of these two cases:

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

3 Write an algorithm:

It turns out that we only need space for a two-dimensional array $B(i, j) = A(k, i, j)$. To compute $A(k, *, *)$ from $A(k-1, *, *)$, we can overwrite $A(k-1, *, *)$. The following algorithm is known as *Floyd's algorithm*:

```
for i : 1..n do
    for j : 1..n do
        B(i,j) ⟵ C(i,j)          (Initialize array B, i.e. copy C there)
    end for
end for

for k : 1..n do
(at this point B(i,j) = A(k-1,i,j) for all i,j)
    for i : 1..n do
        for j : 1..n do
            B(i,j) ⟵ min{B(i,j), B(i,k) + B(k,j)}
        end for
    end for
end for
D ⟵ B
return D
```

Why does it work? Worry: $B(i, k)$ or $B(k, j)$ may have already been updated, e.g. $k < j$ or $k < i$. But $B(i, k)$ and $B(k, j)$ are unchanged because the shortest path from i to k does not contain k as an intermediate node.
What are the pre/post-conditions? What is an appropriate loop invariant?

Simple Knapsack Problem (SKS)

Input: $d, w_1, \ldots, w_d, C \in \mathbb{N}$ (where C is the capacity)
Output: $\max_S\{K(S)|K(S) \leq C\}$, where $S \subseteq \{1, \ldots, d\}$, $K(S) = \sum_{i \in S} w_i$.
This is an NP-hard problem, which means that we cannot expect to find a polynomial time algorithm that works in general. We'll give a Dynamic Programming solution that works for relatively small C. It is important that the inputs $w_1, \ldots, w_d, C$ are integers (and in fact positive integers).

1 Define an array of subproblems: Only consider first i objects $1..i$ summing up to an *intermediate* weight limit j.

$$R(i,j) = \begin{cases} True & \text{if } \exists S \subseteq \{1, \ldots, i\} \text{ s.t. } K(S) = j, \text{ i.e., the sum of weights is exactly } j. \\ False & \text{otherwise} \end{cases}$$

$0 \leq i \leq d$, $0 \leq j \leq C$. We can compute the solution from R as follows:

$$M = \max_{j \leq C}\{j|R(d,j) = True\}.$$

2 Define a recurrence: Initialize: $R(0,j) = False$, $j > 0$; $R(i,0) = True$, $i = 0, 1, \ldots, d$. Before writing a recurrence for $i, j \geq 1$, consider the following: Suppose we do not include object i. Then, obviously, $R(i,j) = True$ iff $R(i-1,j) = True$. Suppose object i is included. Then it must be the case that $R(i,j) = True$ iff $R(i-1, j-w_i) = True$, $j - w_i \geq 0$ (i.e., there is a subset S of objects $\{1, \ldots, i-1\}$ s.t. the sum of weights $K(S)$ is exactly $j - w_i$, $j \geq w_i$).

We obtain the following recurrence for $i, j \geq 1$: $R(i,j) = True$ iff $R(i-1,j) = True$ or ($j \geq w_i$ and $R(i-1, j-w_i) = True$).

3 Write an algorithm: We will use the same space saving trick as before by using one-dimensional array $S(j)$ for $R(i,j)$.

$S(0) \longleftarrow True$ (*Initialize array for* $i = j = 0$)
for $j : 1..C$ do
  $S(j) \longleftarrow False$ (*Initialize array for* $i = 0$ *and* $j = 1..C$)
end for
for $i : 1..d$
  for decreasing $j : C$ downto 1
    if ($j \geq w_i$ and $S(j - w_i) = True$) then
      $S(j) \longleftarrow True$
    end if
  end for
  (*at this point* $S(j) = R(i,j)$ *for all* j)
end for

Note that overwriting does not create a problem here, because the values of j are considered in decreasing order $C, C-1, \ldots, 1$. Thus the array position $S(j-w_i)$ has not yet been updated when the reference is made.
Exercise: Find an input for which the program would make an error if the inner loop "for decreasing $j : C..1$" were changed to "for $j : 1..C$".
The assertion $S(j) = R(i,j)$ can be proved by induction on the number of times the inner loop body is executed. This assertion implies that upon termination of the algorithm, $S(j) = R(d,j)$ for all j. From this, the solution M can be computed as indicated above.

The (General) Knapsack Problem

We generalize the Simple Knapsack Problem (SKS) to the Knapsack Problem by assigning a positive integer value v_i to each weight w_i.
Input: $d, w_1, \ldots, w_d, v_1, \ldots, v_d, C \in \mathbb{N}$ (where C is the capacity)
Problem: Find $\max_S\{V(S)|K(S) \leq C\}$, where $S \subseteq \{1, \ldots, d\}$, $K(S) = \sum_{i \in S} w_i$, and $V(S) = \sum_{i \in S} v_i$.
Thus $V(S)$ is the total value of the set S of weights. The goal is to maximize $V(S)$, subject to the contraint that $K(S)$ (the sum of the weights in S) is at most C.
The Simple Knapsack Problem is the special case in which $v_i = w_i, 1 \leq i \leq d$.
To solve the general Knapsack problem, we start by computing the same Boolean array $R(i, j)$ that was used to solve SKS. Thus $R(i, j)$ ignores the values v_i, and only depends on the weights w_i. Next we define another array $V(i, j)$ that depends on the values v_i as follows: For $0 \leq i \leq d$, $0 \leq j \leq C$

$$V(i, j) = \max\{V(S)|S \subseteq \{1, \ldots, i\} \text{ and } K(S) = j\}$$

Exercise: Give a recurrence for computing the array $V(i, j)$, using the Boolean array $R(i, j)$. (You may assume that the array $R(i, j)$ has already been computed.) Also, give an algorithm for computing $V(i, j)$.
Remark: It turns out that if the above definition of $V(i, j)$ is changed so that we only require $K(S) \leq j$ instead of $K(S) = j$, then the Boolean array $R(i, j)$ is not needed in the recurrence.

The Activity Selection Problem

An *activity* i has a fixed start time s_i, finish time f_i, and profit p_i. Given a set of activities, we want to select a subset of non-overlapping activities with maximum total profit. (An example is a set of lectures with fixed start and finish times that need to be scheduled in a single class room.
Input: A list of activities $(s_1, f_1, p_1), \ldots, (s_n, f_n, p_n)$. Assume $p_i > 0$, $s_i < f_i$, and $s_i, f_i, p_i \in \mathbb{R}$ where $1 \leq i \leq n$.
Problem: Find a set $S \subseteq \{1, \ldots, n\}$ of selected activities such that no two selected activities overlap, and the profit $P(S) = \sum_{i \in S} p_i$ is as large as possible.
Greedy methods do not work for this problem. We will describe a Dynamic Programming solution.
1.) Define an array (of subproblems):
Sort the activities by their finish time: $f_1 \leq f_2 \leq \ldots \leq f_n$. It is possible that two or more activities have the same finish time. Partition the activities according to their finish times. (Remark: This partitioning is not strictly necessary, although it may lead to a more efficient algorithm in some cases.) Denote these *distinct* finish times by $u_1 < u_2 < \ldots < u_k$. For instance, if we have activities finishing at times 1.24, 4, 3.77, 1.24, 5 and 3.77, then we partition them into four groups: activities finishing at 1.24, 3.77, 4 and 5 respectively. In this case $u_1 = 1.24$, $u_2 = 3.77$, $u_3 = 4$, $u_4 = 5$. Let u_0 be $\min_{1 \leq i \leq n} s_i$, i.e., the earliest start time. Thus, $u_0 < u_1 < u_2 < \ldots < u_k$. Define an array $A(0..k)$ as follows.

$$A(j) = \max_{S \subseteq \{1, \ldots, n\}} \{P(S)|S \text{ is feasible and } f_i \leq u_j \text{ for each } i \in S\}.$$

where S is *feasible* if no two activities in S overlap.
Note that $A(k)$ is the maximum possible profit for all feasible schedules S.
In order to give a recurrence for computing $A(0 \ldots k)$ we define an array $H(1..n)$ such that $H(i)$ is the index of the largest distinct finish time no greater than the start time of activity i. Formally,

$H(i) = l$ if l is the largest number such that $u_l \leq s_i$. To compute $H(i)$, we need to search the list of distinct finish times. To do it efficiently, for each i, apply the binary search procedure that runs in logarithmic time in the length of the list of distinct finish times (try $l = \lfloor \frac{k}{2} \rfloor$ first). Since the length k of the list of distinct finish times is at most n, and we need to apply binary search for each element of the array $H(1..n)$, the time required to compute all entries of the array is $O(n \log n)$.
2.) Define a recurrence for $A(0..k)$.
Initialization: $A(0) = 0$.
Want $A(j)$ given $A(0), \ldots, A(j-1)$. Consider $u_0 < u_1 < u_2 < \ldots < u_{j-1} < u_j$. Can we beat profit $A(j-1)$ by scheduling some activity that finishes at time u_j? Try all activities that finish at this time and compute maximum profit in each case. We obtain the following recurrence:

$$A(j) = \max\{A(j-1), \max_{1 \leq i \leq n} \{p_i + A(H(i)) \mid f_i = u_j\}\}$$

where $H(i)$ is the last finish time before the start of activity i.
Example:

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

j:	0	1	2	3
u_j:	0	3	6	10
$A(j)$:	0	20	40	40

$$A(2) = \max\{20, 30 + A(0), 20 + A(1)\} = 40$$
$$A(3) = \max\{40, 30 + A(0)\} = 40$$

The answer is $\max P = A(k) = 40$.
Exercise: Given that A has been computed, how do you find a set of activities S such that $P(S) = A(k)$?
Big hint: If $A(k) = A(k-1)$, then we know that no selected activity finishes at time u_k, so we go on to consider $A(k-1)$. If $A(k) > A(k-1)$, then some selected activity finishes at time u_k. How do we find this activity? We must find an i_0 such that $p_{i_0} + A(H(i_0)) = A(k)$ and $f_{i_0} = u_k$, output i_0, and go on to consider $A(H(i_0))$.
3.) Exercise: write an algorithm.

Job Scheduling with Profits

A job differs from an activity in that a job can be scheduled any time as long as it finishes by its deadline; an activity has a fixed start time and finish time. Because of the flexibility in scheduling jobs, it is "harder" to find an optimal schedule for jobs than to select an optimal subset for activities, and we need a 2 dimensional array, rather than a 1 dimensional one.
Before we considered job scheduling problems for the case in which each job takes unit time. We now generalize this to the case in which each of n jobs i has duration d_i, deadline t_i, and profit p_i. We assume that d_i and t_i are positive integers, but the profit p_i can be a positive real number.
The goal is to find a feasible schedule $C(1 \ldots n)$ for the n jobs for which the profit $P(C)$ is maximized.

Our convention is that job i is scheduled to begin at time $C(i) \geq 0$, except that $C(i) = -1$ if job i is not scheduled by C.
The profit is:

$$P(C) = \sum_{C(i) \geq 0} p_i$$

We say that the schedule $C(1 \ldots n)$ is *feasible* if the following two conditions hold:

1. If $C(i) \geq 0$, then $C(i) + d_i \leq t_i$ (each scheduled job finishes by its deadline) and,
2. If $i \neq j$ and $C(i) \geq 0$ and $C(j) \geq 0$ then either $C(i) + d_i \leq C(j)$ or $C(j) + d_j \leq C(i)$ (no two scheduled jobs overlap).

Now we can define the problem Job Scheduling with Profits:
Input: A list of jobs (d_1, t_1, p_1),..., (d_n, t_n, p_n)
Problem: A feasible schedule $C(1), \ldots, C(n)$ such that the profit $P(C)$ is the maximum possible among feasible schedules.
Aside: Notice that this problem is at least as hard as SKS. In fact an SKS instance $w_1, \ldots, w_n, C$ can be viewed as a job scheduling problem in which each duration $d_i = w_i$, each deadline $t_i = C$, and each profit $p_i = w_i$. Then the maximum profit of any schedule is the same as the maximum weight that can be put into the knapsack.
To give a Dynamic Programming solution to the job scheduling problem, we start by sorting the jobs according to deadline. Thus we assume that

$$t_1 \leq t_2 \leq \ldots \leq t_n$$

It turns out that to define a suitable array A for solving the problem, we must consider all possible integer times t, $0 \leq t \leq t_n$ as a deadline for the first i jobs. It is not enough to only consider the specified deadline t_i given in the problem input.
Thus define the array $A(i, t)$ for $0 \leq i \leq n, 0 \leq t \leq t_n$ by:

$$A(i,t) = \max\{P(C) | \begin{cases} C \text{ is a feasible schedule,} \\ \text{only jobs in } \{1, \ldots, i\} \text{ are scheduled,} \\ \text{all scheduled jobs finish by time } t. \end{cases} \}$$

Exercise: Find a recurrence for $A(i, t)$.
Hint: Consider the two cases that either job i occurs or does not occur in the optimal schedule (and note that job i will not occur in the optimal schedule if $d_i > \min\{t_i, t\}$). If job i does not occur, we already know the optimal profit. If job i occurs in an optimal schedule, then we may as well assume that it is the last job (among jobs $\{1, \ldots, i\}$) to be scheduled, because it has the lastest deadline. Hence we assume that job i is scheduled as late as possible, so that it finishes either at time t, or at time t_i, which ever is smaller, i.e. it finishes at time $t_{\min} = \min\{t_i, t\}$.