Dynamic Programming

Dynamic Programming is a general technique for constructing algorithms. When the method works it almost always gives an efficient solution to a problem. In order to apply the method you should go trough the following steps:

1. Find a way if splitting your problem into subproblems. The solutions to the subproblems will usually be recorded in an array.

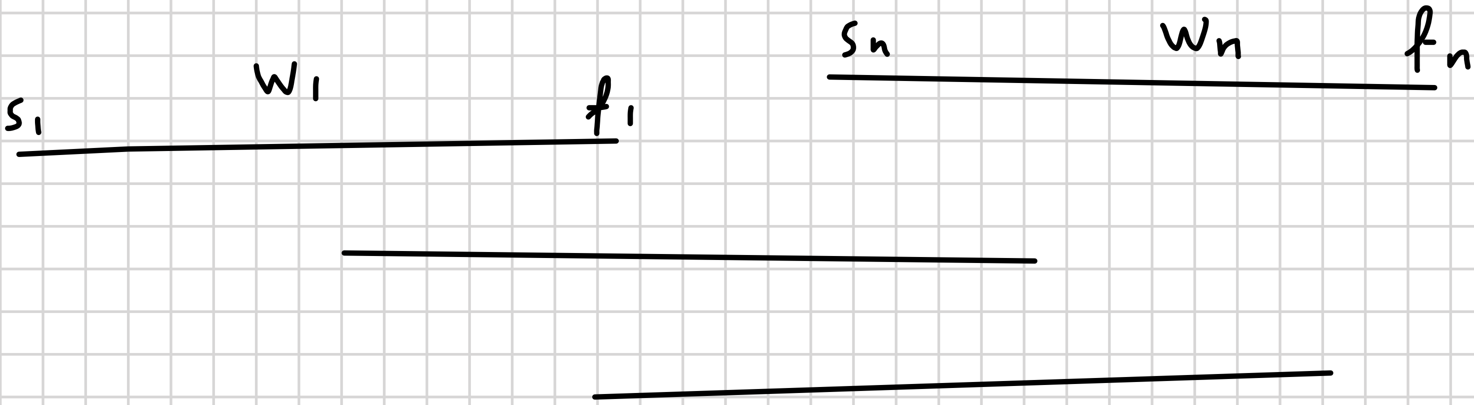
2. Find a recursion formula that relates the values of subproblems to the values of simpler subproblems.

3. Find a natural ordering of the subproblems and then compute the values of all subproblems in that order, using the recursion formula.

Selection of weighted intervals

As in lecture 2 we have a set of activities given by time intervals [s[i], f[i]). We assume that the intervals are sort by increasing finishing time. In this problem we have weights w[i] on the intervals. The problem is this:

Input: n Intervals [s[i], f[i]) with weights w[i].
Goal: Find a selection of non-overlapping intervals with maximal weight sum.



1. How can we find natural subproblem? Why not index problems after the numbers of intervals?

Def: Let M[k] be the maximal weigth sum you can get if you only are allowed to use the first k intervals.

2. How do we find a recursion formula? It is obvious that M[1] = w[1]. If we want to use n intervals, how do we do? Do we include interval n in the solution or not?

If we don't then obviously we get $M[n] = M[n-1]$.

If we do, then there is a largest k such that interval k does not overlap interval n. Then we must have $M[n] = M[k] + w[n]$.

But now we can compare these two possible values of $M[n]$ and see which value is largest. From this we can tell what the best choice is.

$M[1] = w[1]$

$M[n] = \max(M[n-1], M[k] + w[n])$ where k is the largest number such that $f[k] \leq s[n]$

3. We now compute the values $M[1], M[2], \ldots, M[n]$ in the natural order. We use an array choose[k] that indicates if interval k should be a part of the optimal choice for $M[k]$ and an array p[k] that indicates what the previous choice in the selection corresponding to Mk] is.

```
M[1] ← 1
p[1] ← NULL
choose[1] ← TRUE
For i ← 2 to n
    If f[1] > s[i]
        If w[i] > M[i-1]
            M[i] ← w[i]
            choose[i] ← TRUE
            p[i] ← NULL
        Else
            M[i] ← M[i-1]
            choose[i] ← FALSE
            p[i] ← i-1
    Else
        k ← 1
        While f[k] ≤ s[i]
            k ← k+1
        k ← k-1
        If M[i-1] ≥ M[k] + w[i]
            M[i] ← M[i-1]
            choose[i] ← FALSE
            p[i] ← i-1
        Else
            M[i] ← M[k] + w[i]
            choose[i] ← TRUE
            p[i] ← k
```

Since the algorithm has two loops of size n (in the worst case) we get complexity $O(n^2)$.

When the algorithm has stopped we get the solution from M[n]. If we want to know which intervals we should choose we just check the sequence.
choose[n], choose[p[n]], choose[p[p[n]]], ... and so on for the value TRUE.

In some of the problems we study we will just be interested in finding the optimal values of the choices rather than the actual choices.

The recursion formula normally involves the values of the optimal choices and it's easiest to first write a program for solving the equation. Then a slight modification of the program will give us the actual choices.

Increasing sequence of numbers

Problem: Given a sequence of numbers

$x_1, x_2, ..., x_n$ we want to compute the longest sequence of increasing consecutive numbers.

Let $v(i) =$ be the length of the longest sequence ending in x_i.

Algoritm:

(1) $v(1) \leftarrow 1$
(2) **for** $i = 2$ **to** n
(3)   **if** $x(i-1) \leq x(i)$
(4) $v(i) \leftarrow v(i-1) + 1$
(5)   **else**
(6) $v(i) \leftarrow 1$
(7) **return** v

Then we compute $\max_i v(i)$.

Longest subsequences (case 2)

We have a sequence of n numbers. We want to find a longest increasing subsequence. In this case the numbers don't have to be consecutive.

(Strictly speaking, by increasing we will rather mean non-decreasing, i.e. $\leq$)

Let the numbers be $x[1], x[2], \ldots, x[n]$.

Set $v[i]$ = Length of the longest increasing sequence ending in $x[i]$

Then $v[1] = 1$. For larger i we set $v[i] = \max(v[k] + 1)$ where the max runs over all k such that $x[k] \leq x[i]$.

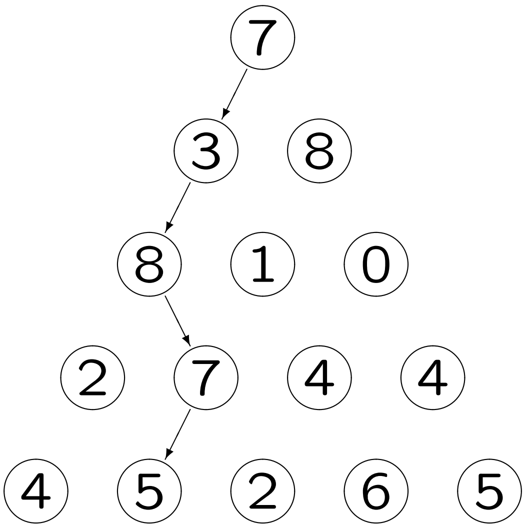
We can implement this with the algorithm:

```
v[1] ← 1
For i ← 2 to n
    max ← 1
    For k ← 1 to i-1
        If x[k] < x[i] and v[k] + 1 > max
            max ← v[k]+1
    v[i] ← max
max ← 0
For j ← 1 to n
    If v[j] > max
    max ← v[j]
Return max
```

The complexity is $O(n^2)$. The algorithm just gives us the length of the sequences but we can modify it to give us the actual sequences.

One more problem

Problem: Find the path from top to bottom that maximizes the sum of the numbers.



Let a_{ij} be the number in row i, column j.

Let $V[i,j]$ be the value of the best path from (i,j) down to bottom row n. Then

$$V[i,j] = \begin{cases} a_{ij} & i = n, \\ a_{ij} + \max\{V[i+1,j], V[i+1,j+1]\} & \text{otherwise.} \end{cases}$$

Compute all $V[i,j]$:

```
for j = 1 to n
    V[n, j] ← a_nj
for i = n − 1 to 1
    for j = 1 to i
        V[i, j] ← a_ij +
        max{V[i + 1, j], V[i + 1, j + 1]}
```

The runtime for finding $V[1,1]$ is $\Theta(n^2)$.

Subset Sum

We assume that we have n positive integers $a[1], a[2], \dots, a[n]$. We are given an integer M. We want to know if there is a subset of the integers with sum M.

What are the natural subproblems here? We can try to get the sum M by using fewer than n integers. Or we can try to get a smaller sum than M. In fact, we will combine these two ideas.

Set $v[i,m] = 1$ if there is a subset of $a[1], a[2], \dots, a[i]$ with sum m and $v[i,m] = 0$ otherwise.

If $v[i,m] = 1$ it must be either because we can get m just by using the numbers $a[1], a[2], \dots, a[i-1]$ or because we can get the sum $m - a[i]$ by using the same numbers. We get the recursion

$v[1, 0] = 1$
For all i such that $2 \le i \le n$ and all m M such that $a[i] \le m$
$v[i,m] = \max(v[i-1,m], v[i-1,m-a[i]])$

We now try to construct an algorithm. We have to order the subproblems. We compute all $v[i,m]$ by running an outer loop over $1 \le i \le n$ and an inner loop over $1 \le m \le M$.

```
Set all v[i,j] = 0
For i ← 1 to n
    v[i,0] ← 1
For i ← 2 to n
    For m ← 1 to M
        If v[i-1, m] = 1
            v[i,m] ← 1
        Else If m ≥ a[i]  and  v[i-1, m-a[i]] = 1
            v[i, m] ← 1
```

When the algorithm stops, the value of v[n,M] tells us the solution to the problem. (1 = " It's possible" , 0 = " It's not possible".) The complexity is $O(n\,M)$.

Shortest paths in graphs

In lecture 3 we discussed the problem of finding shortest paths in graphs with negative weights. Floyd-Warshall's algorithm is a dynamic programming algorithm for solving the problem. Actually it find the shortest distances between all pairs of nodes. We assume that we have no negative cycles.

Subproblems:

We set $d[i,j,k]$ = length of shortest path using just nodes i,j and nodes $1,2,\ldots,k$.

Recursion:

$d[i,j,0] = w[i,j]$ for all i,j ($w[i,j] = \infty$ if there is no edge (i,j))

$d[i,j,k] = \max (d[i,j,k-1], d[i,k,k-1] + d[k,j,k-1])$ $1 \leq k \leq n$

Algorithm:

For $i \leftarrow 1$ to n

    For $j \leftarrow 1$ to n

        $d[i,j,0] \leftarrow w[i,j]$

        $p[i,j,0] \leftarrow i$

For $k \leftarrow 1$ to n

    If $d[i,j,k-1] \leq d[i,k,k-1] + d[k,j,k-1]$

        $d[i,j,k] \leftarrow d[i,j,k-1]$

        $p[i,j,k] \leftarrow p[i,j,k-1]$

    Else

        $d[i,j,k] \leftarrow d[i,k,k-1] + d[k,j,k-1]$

        $p[i,j,k] \leftarrow p[k,j,k-1]$

For $i \leftarrow 1$ to n

    For $j \leftarrow 1$ to n

        $d[i,j] \leftarrow d[i,j,n]$

        $p[i,j] \leftarrow p[i,j,n]$

The algorithm has complexity $O(n^3)$.