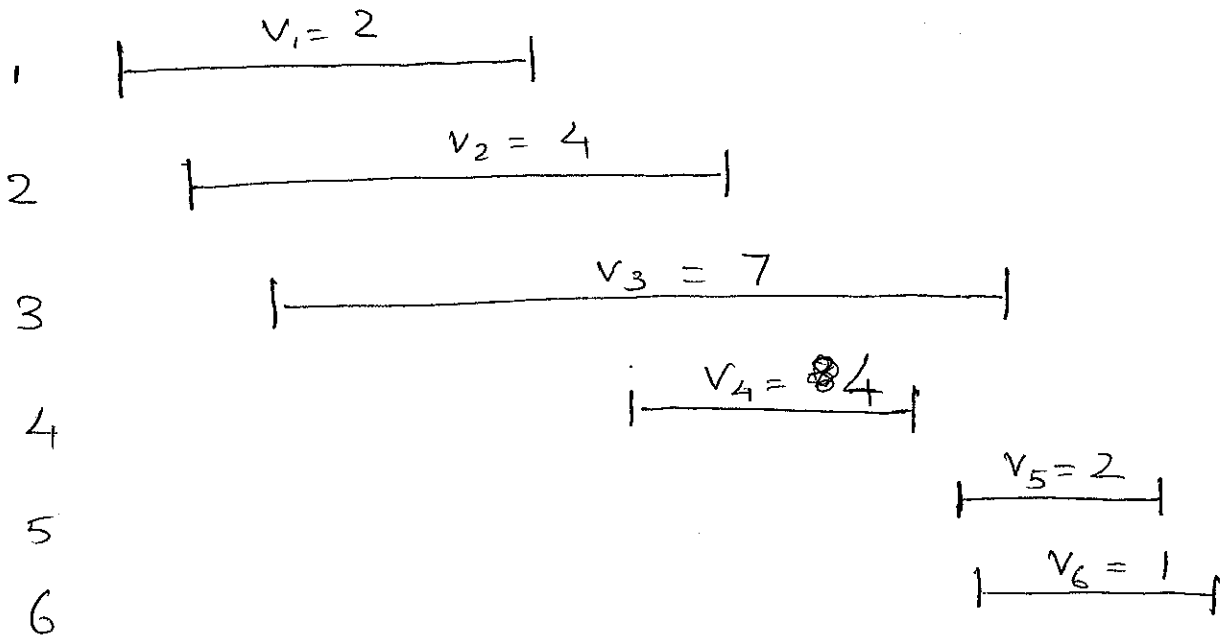
Dynamic Programming - Weighted Interval Scheduling.

We are given a set of requests - $\{1, 2, \ldots n\}$ each request i is an interval with a start time s_i and finish time f_i $(s_i < f_i)$ and value $v_i > 0$. We want to select a set S of mutually compatible intervals that maximizes the <u>value</u>. $\text{value}(S) = \sum_{i \in S} v_i$

Because we'll be going through the intervals beginning with the one that starts first, we'll assume that the intervals are ordered by increasing start time with ties broken arbitrarily.

That is, $i < j \Rightarrow s_i \leq s_j$



In the above example, an optimal solution is $\{1, 4, 5\}$ with value 8.

Since several of the natural greedy algorithms fail for this problem, we'll try a more systematic search for an optimal solution.

So let us consider an optimal solution O for the given instance.

Now either $1 \in O$ or $1 \notin O$.

If $1 \notin O$, O must be optimal for the instance consisting of requests $\{2, \ldots n\}$.

If $1 \in O$, $O \setminus \{1\}$ must be optimal for the instance consisting of all the requests compatible with 1. (This requires some thought.) Notice that the set of requests compatible with 1 is either empty or the set $\{i \ldots n\}$ for some i. This motivates the definition: For $1 \leq j \leq n$, define $p(j)$ to be the first interval that starts after j ends. ~~Otherwise~~ If no such interval exists, define $p(j) = n+1$

Let O_j denote an optimal solution for the set $\{j \ldots n\}$, and $opt(j)$ the cost of such an optimal solution.

Thus O_1 is either an optimal solution for $\{2, \ldots, n\}$ or $\{1\}$ plus an optimal solution for $\{p(1) \ldots n\}$. We can recursively compute both optima and take the better of the two solutions.

Also,

$$Opt(1) = \max\{v_1 + opt(p(1)),\ opt(2)\}.$$

($Opt(n+1)$ is defined to be zero.)

In general,

$$Opt(j) = \max\{v_j + opt(p(j)),\ opt(j+1)\}.$$

So here is a recursive algorithm that computes $opt(j)$.

Compute-opt(j)

If $j = n+1$, then return 0.

Otherwise, Return

$$\max\{v_j + \text{Compute-opt}(p(j)), \text{Compute-opt}(j+1)\}$$

Correctness: For each $1 \le j \le n+1$, Compute-opt(j) returns $opt(j)$.

<u>Proof</u>: A simple inductive argument. For the base case $(j = n+1)$, we know Compute-opt$(n+1) = opt(n+1)$. Fix some j between 1 and n and assume inductively that Compute-opt$(i) = opt(i)$ for all $i > j$. Now

$$\cancel{Opt(j) = \max\{v_j + \text{Compute-opt}(p(j)), \text{Compute-opt}(j+1)\}}$$

$=$

$$\mathrm{Opt}(j) = \max\{v_j + \mathrm{Opt}(p(j)),\ \mathrm{Opt}(j+1)\}$$

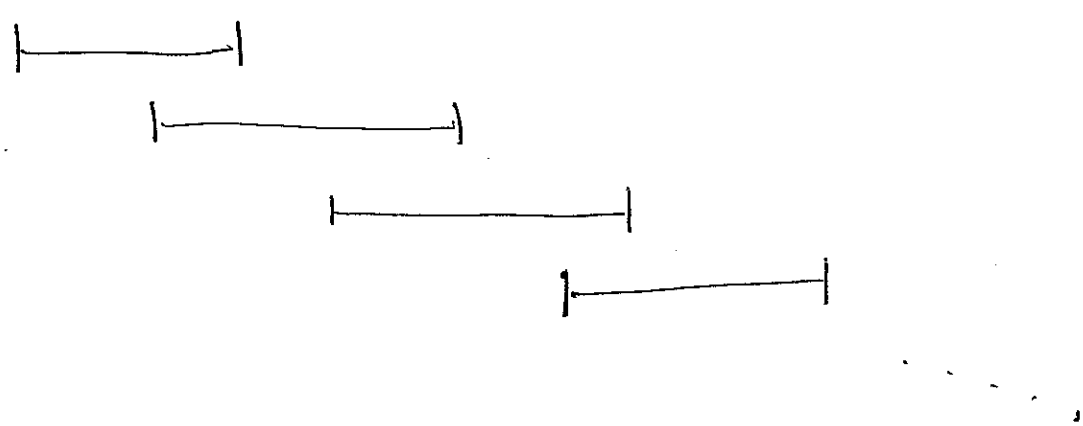
$$= \max\{v_j + \text{Compute-Opt}(p(j)),\ \text{Compute-Opt}(j+1)\}$$

(Induction Hypothesis)

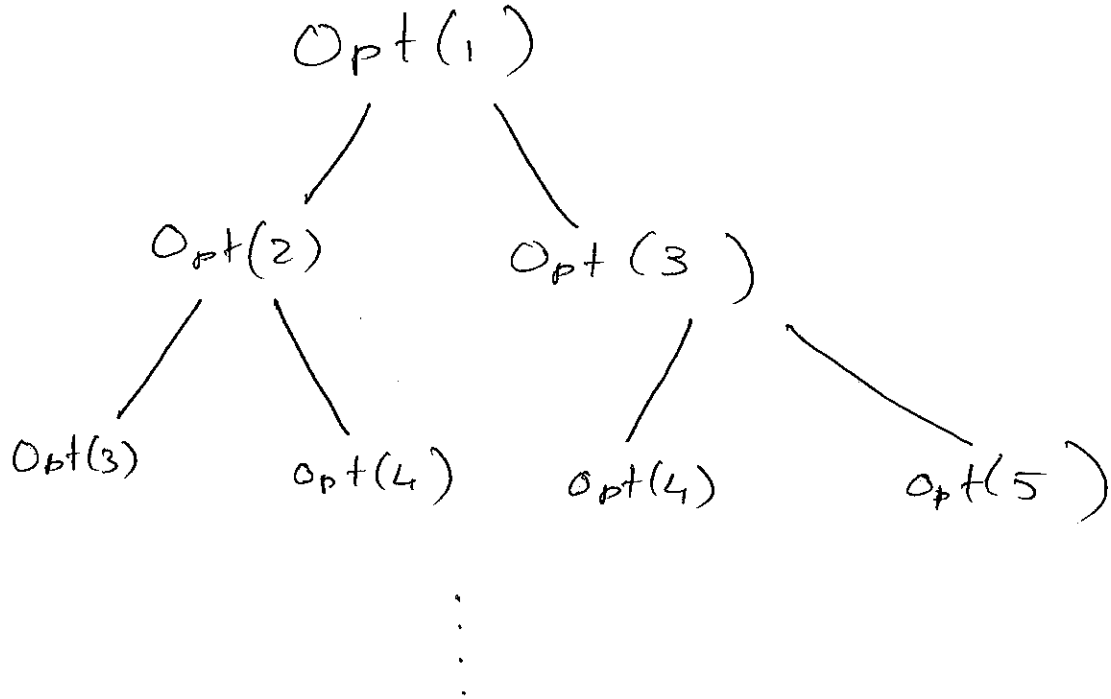
$$= \text{Compute-Opt}(j),$$

finishing the inductive step. $\boxtimes$

We have a correct algorithm, but what is its running time? Consider the following instance:



The tree of subproblems when $Opt(1)$ is called looks like:



The number of subproblems is exponential. A shortest root to leaf path has length at least $\frac{n}{2}$, so no. of subproblems is at least $2^{n/2}$. This is a crude lower bound, but good enough to tell us we are doing badly.

Why are we doing badly? It is not because (and this is important) the number of different subproblems that need to be solved ~~are~~ is large. In fact, each ~~recu~~ subproblem is of the form Compute-Opt(j) for some j between 1 and n+1, so there are only n+1 distinct subproblems.

We are doing badly because each sub-problem is being solved repeatedly.

One fix is obvious. We'll use an array M[1..n+1], initialized to infinity. The first call to Compute-Opt(j) ~~makes~~ records Opt(j) in M[j]. Subsequent calls to

Compute-Opt(j) just return $M[j]$.

M-Compute-Opt(j)

~~if j = n+1, ret~~

If $M[j]$ is not ∞, return $M[j]$.

Otherwise, if $j = n+1$

Set $M[j] \leftarrow 0$

Return $M[j]$,

else // $j < n+1$

Set $M[j] \leftarrow$

$\max\{v_j + \text{M-Compute-Opt}(p(j)),$
$\text{M-Compute-Opt}(j+1)\}$

Return $M[j]$.

Let's analyze the running time.

Clearly every ~~subpro~~ recursive subproblem takes constant time not counting the time taken by the recursive calls that it makes. So it is enough

to count the number of recursive subproblems solved. There are two kinds.

(1) The subproblems that make an entry into $\mathcal{M}$: There are only $n+1$ of these.

(2) The subproblems ~~not~~ that don't make an entry into $\mathcal{M}$. Observe that any such subproblem is a <u>direct</u> <u>descendant</u> of a type-(1) subproblem. But each type-(1) subproblem makes at most two recursive calls. We conclude that number of type-(2) subproblems $\leq 2(n+1)$.

Thus running time is $O(n)$.