General Approach to Solving optimization problems using Dynamic Programming

1- characterize the structure of an opt. solution

2- Recursively define the value of an opt. solution

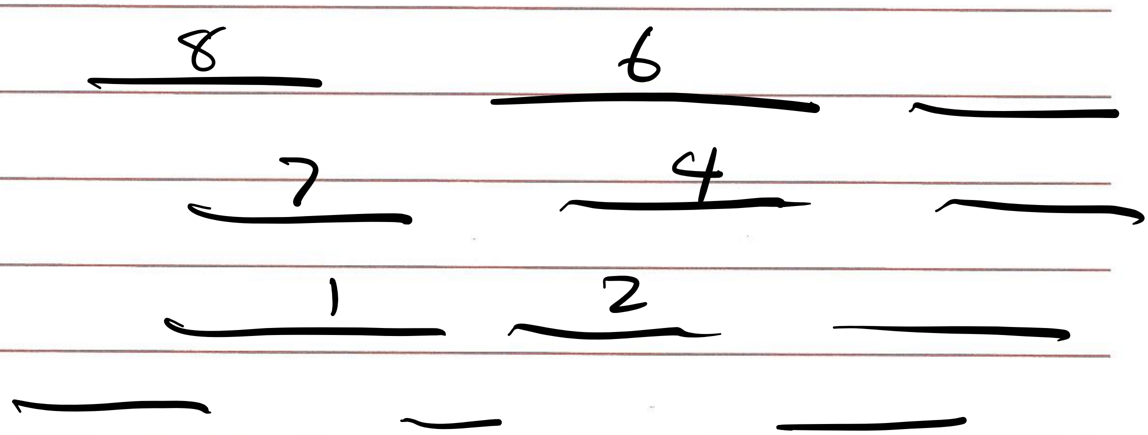
3- Compute the value of an opt. solution in a bottom up fashion

4- Construct an opt. sol. from computed information

Problem Statement

- We have 1 resource
- " " n requests labeled 1 to n
- Each request has start time s_i, finish time f_i, and weight w_i

Goal: Select a subset $S \subseteq \{1..n\}$ of mutually compatible intervals so as to Maximize $\sum_{i \in S} w_i$



observation: Either job i is part of the opt. sol. or it isn't

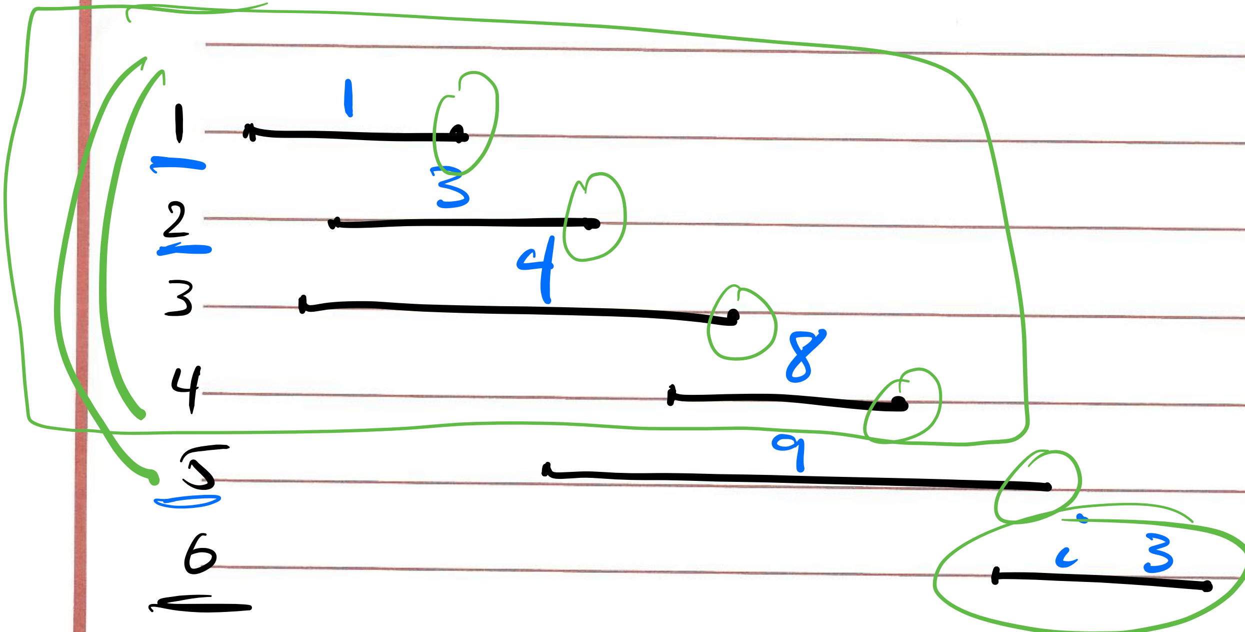
Case 1 - if it is, value of the opt. sol. = w_i + value of the opt. sol. for the subproblem that consists only of compatible requests with i

Case 2 - if it isn't, value of the opt. sol. = value of the opt. sol. without job i

Sort requests in order of non-decreasing finish time.

$$f_1 \leq f_2 \leq \cdots \leq f_n$$

Define $P(j)$ for an interval j to be the largest index $i < j$ such that interval i & j are disjoint.



$P(\,)$

$P(1)=0$

$P(2)=0$

$P(3)=0$

$P(4)=2$

$P(5)=1$

$P(6)=4$

Total cost of building $P(\,)$ is $O(n \lg n)$

Def. Let O_j denote the opt. solution to the problem consisting of requests $\{1..j\}$.
Let $OPT(j)$ denote the value of O_j

$O_4 = \{2, 4\}$, $OPT(4) = 11$

Case#1: $j \in O_j \Rightarrow OPT(j) = w_j + OPT(P(j))$

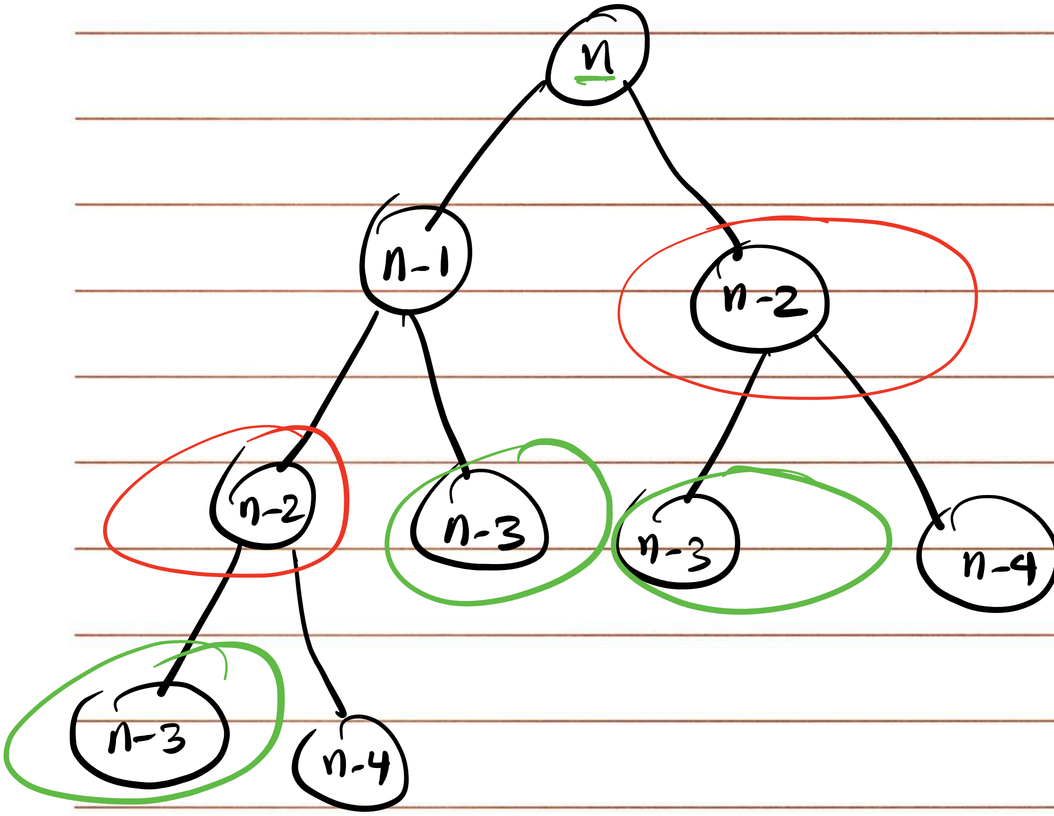
Case#2: $j \notin O_j \Rightarrow OPT(j) = OPT(j-1)$

↳ recurrence formula

Solution:

```
Compute-opt (j)
    if j = 0 then
        return 0
    else
        return Max (
        wj + Compute_opt( p(j) ),
        Compute_opt (j-1) )
    endif
```

$$T(n) = T(n-1) + T(n-2)$$


n
n-1
n-2
n-2
n-3
n-3
n-4
n-3
n-4

Memoization

Store the value of compute-opt. in a globally accessible place the first time we compute it. Then simply use this precomputed value in place of all future recursive calls.

```
M_compute-opt(j)
  if j=0 then
      return 0
  else if M[j] is not empty then
      return M[j]
  else define M[j] = Max(
        wj + M_compute-opt(p(j)),
          M_compute_opt(j-1))
      return M[j]
  endif
```

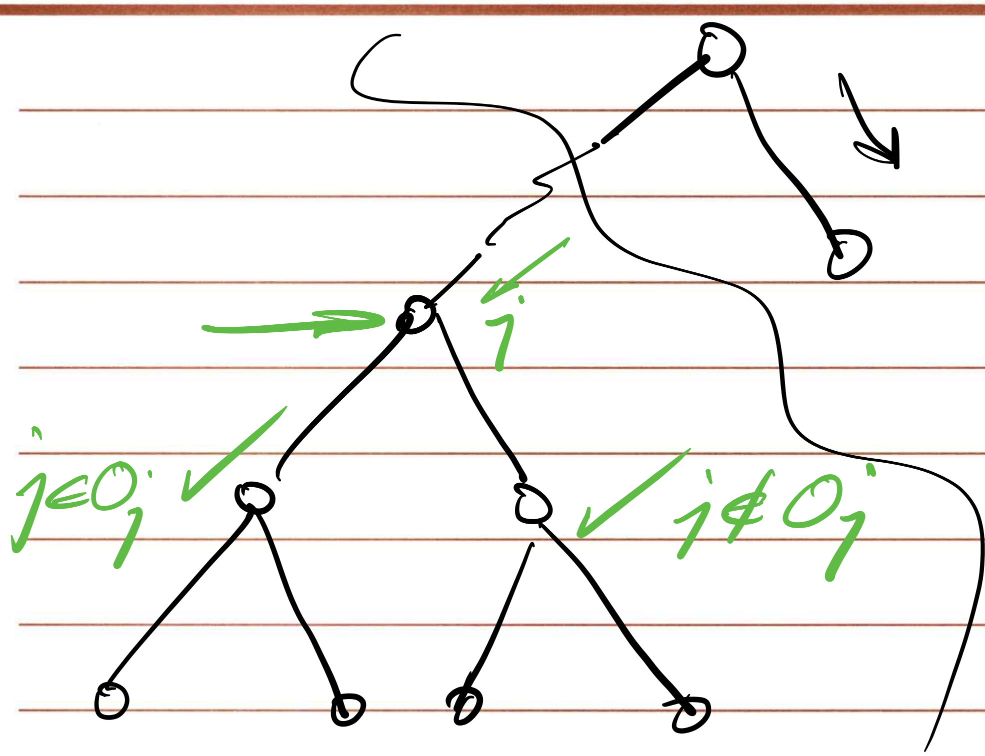
Total cost of this function is $\Theta(n)$

initial sort : $\Theta(n \lg n)$

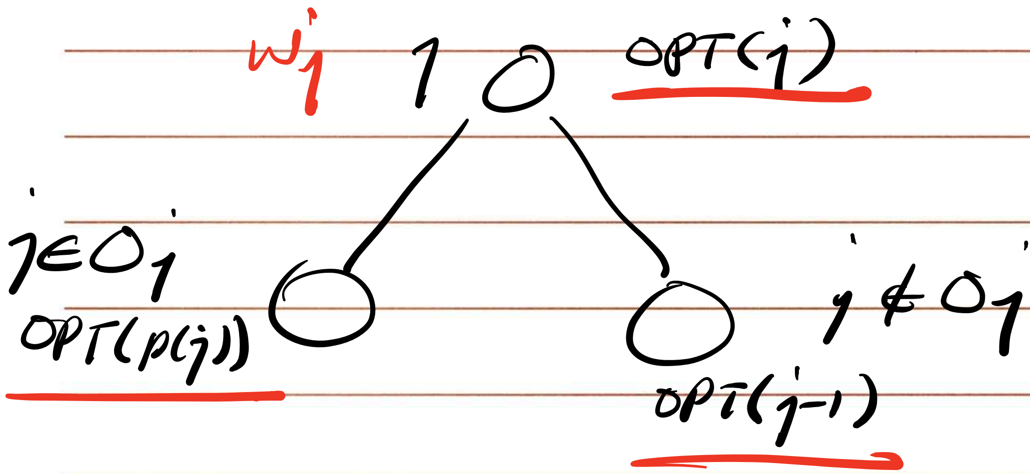
Build $P(\,)$: $\Theta(n \lg n)$

M_compute-opt : $\Theta(n)$

overall cost : $\Theta(n \lg n)$



Compute an opt. sol.



j belongs to O_j iff

$$w_j + OPT(p(j)) \geq OPT(j-1)$$

Find-Solution

if $j > 0$ then

if $w_j + M[p(j)] \geq M[j-1]$ then

output j together w/ the results of Find-Solution $(p(j))$

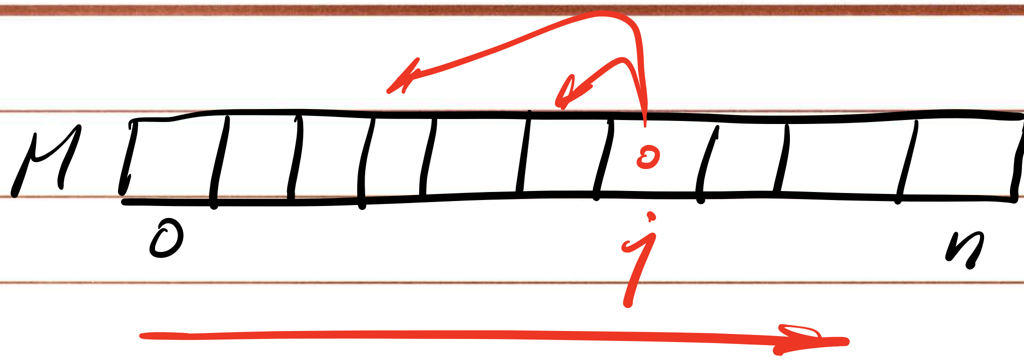
else

output the results of Find-Solution $(j-1)$

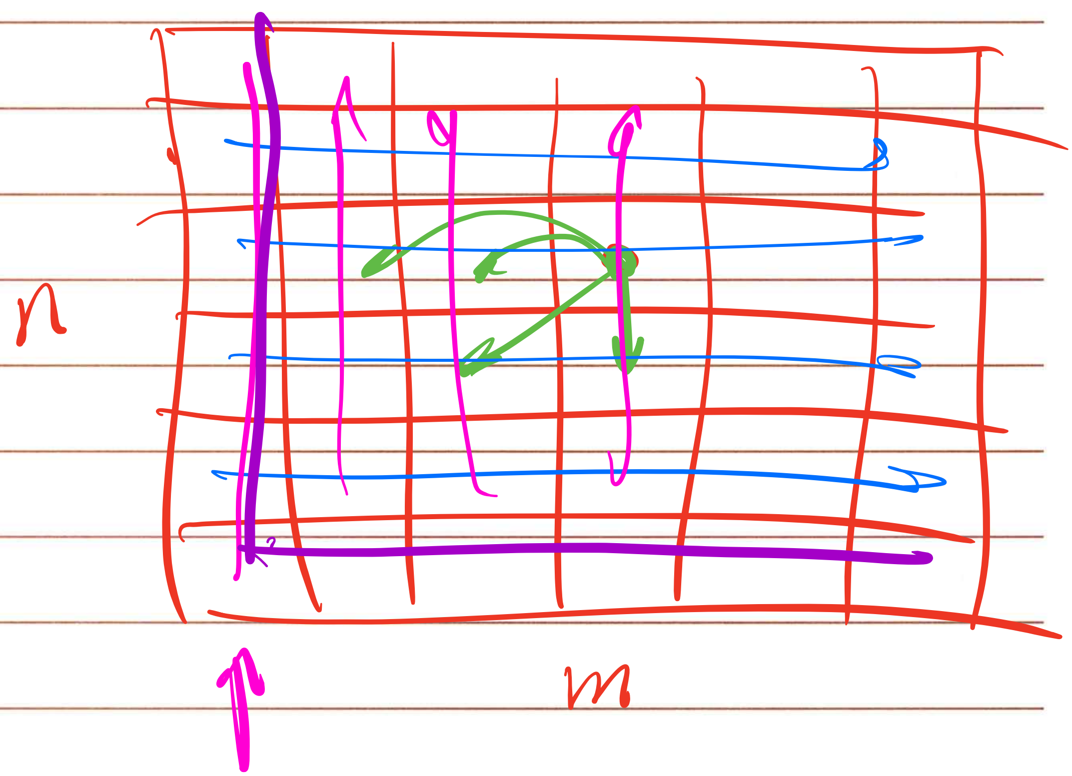
endif

endif

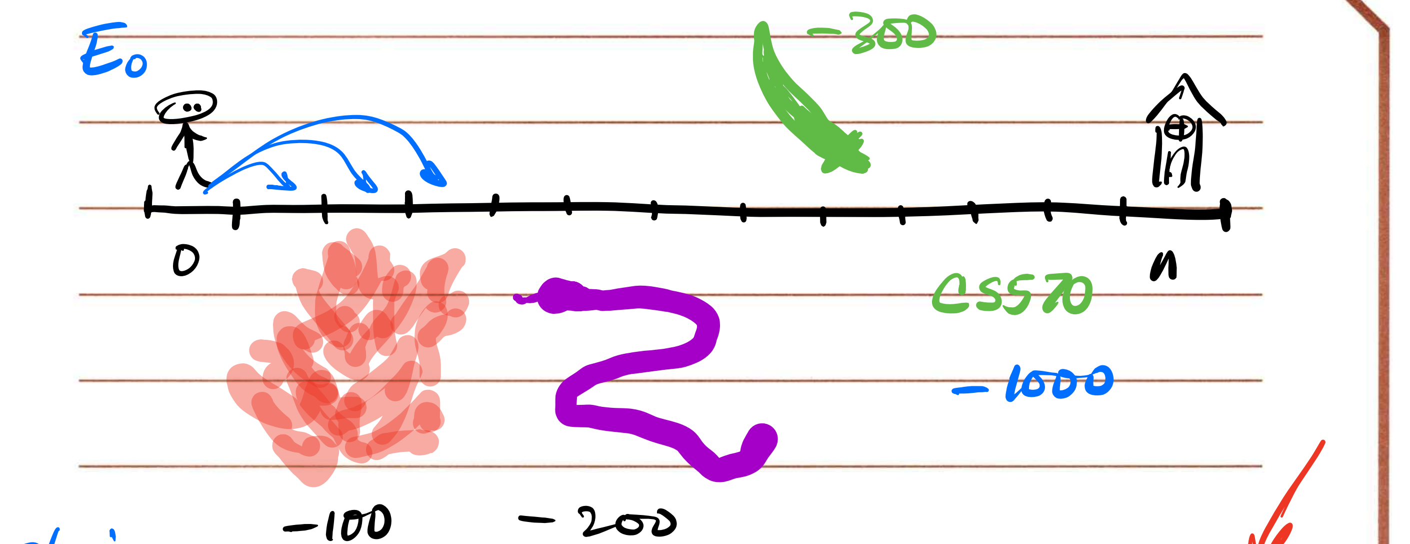
Takes $O(n)$ time




```
M[0] = 0
for i=1 to n
    M[i] = Max( M[i-1],
                w_i + M[p(i)] )
endfor
```



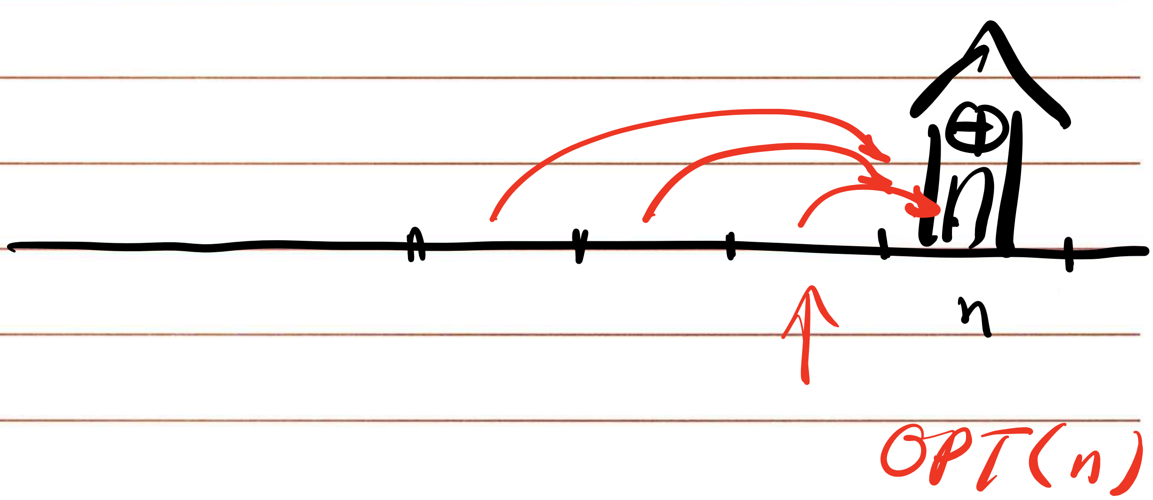
Videogame Problem



Choices:

1- walk to next stage — costs $50 units
2- jump over one stage — " $150 "
3- " " two stages — " $350 "

in general, we lose e_i units of energy when landing on stage $\underline{i}$



$$OPT(n) = Max(OPT(n-1) - 50 - e_n,\ OPT(n-2) - 150 - e_n,\ OPT(n-3) - 350 - e_n)$$

rec. formula (1)

$OPT(j)$ = Value of the opt. sol. when we reach stage j

= highest level of energy possible when we reach stage j.

$OPT(0) = E_0$, $OPT(1) = E_0 - 50 - e_1$, $OPT(2) = Max(\ldots, \ldots)$

for $i = \cancel{1}\ 3$ to n

  use recurrence formula (1)

endfor