Algorithms

DATE: 26/4/22

(LOCAL SEARCH ALGORITHMS)

- GREEDY ALGORITHMS
- DIVIDE - ET- IMPERA "

WEIGHTED INTERNAL SCHEDULING

We are given a set of intervals:

$$I = \{i_1, \dots, i_n\} = \{(s_1, f_1), (s_2, f_2), \dots, (s_n, f_n)\}$$

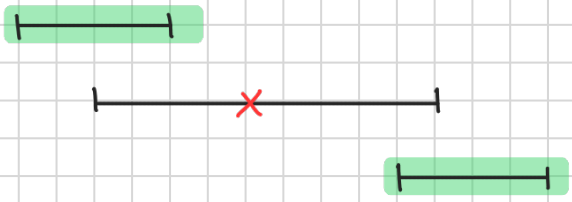
where s_j is the starting time of interval i_j, and f_j is the ending time of interval i_j, as well as a weighting function w that assigns a weight/value to each interval where $w(i_j)$ is the weight/value of i_j.

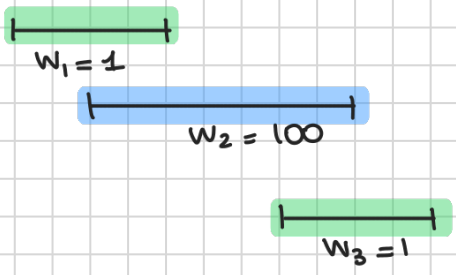
GOAL: Select a subset $S \subseteq I$ of non-overlapping intervals of maximum value, that is, that maximises $\sum_{i_j \in S} w(i_j)$

Intervals scheduling is a special case of weighted interval scheduling (just set $w(i_j) = 1 \quad \forall i_j \in I$).

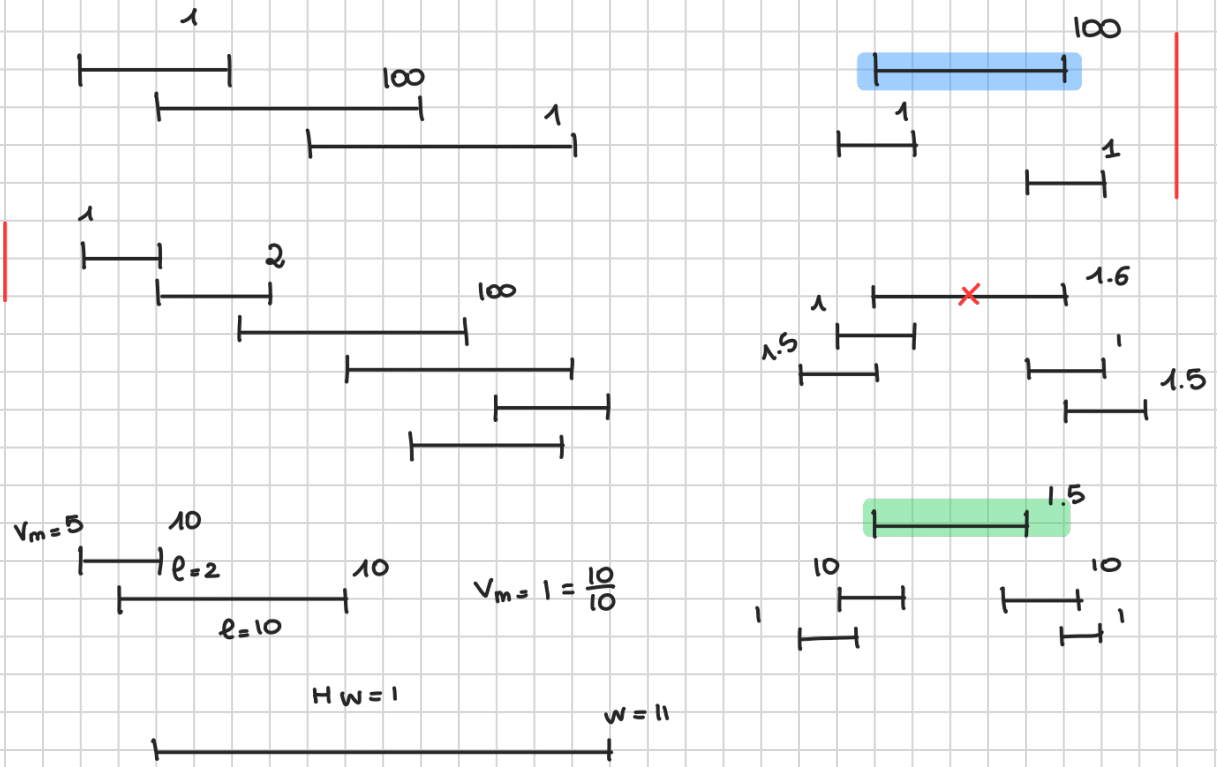
(Weighted interval sch. is a generalization of int. sch.)

We solved Interval scheduling with the "earliest-finishing-time" greedy algorithm.





with weights, the greedy algorithm can do a very bad job.



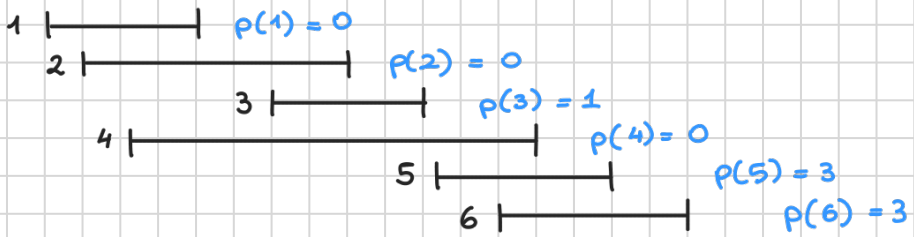
DINAMYC PROGRAMMING

An algorithmic technique that keeps track of all solutions at once, in "little" time.
(Greedy algorithms, instead, consider one partial solution at a time).

First of all, let us assume that the intervals are sorted by finishing time : $f_1 \leq f_2 \leq \dots \leq f_n$.

DEF: INTERVAL i "comes before" interval j IFF $f_i \leq f_j$.

DEF: Let $p(j)$, for an interval j, be the largest index $i < j$ S.T. intervals i and j are disjoint (compatible), or, if no such i exists, let $p(j) = 0$.



L: The generic interval i is disjoint (compatible) with each of the intervals $1, 2, \ldots, p(i)$.

P: i is disjoint from $p(i)$ intervals are sorted increasingly by their finishing time.
i being compatible with $p(i) < i$ then means that $s_i > f_{p(i)}$. Also, $f_{p(i)} \geq f_{p(i)-2} \geq \cdots \geq f_1$. Thus, $s_i > f_j$ for each $j \in \{1, 2, \ldots, p(i)\}$. ■

Suppose that O_i is an optimal solution of the problem restricted to the first i intervals ($\{(s_1, f_1), (s_2, f_2), \ldots, (s_i, f_i)\}$)

Let OPT_i be the value of O_i

For each $j \geq 1$:

- either interval j is in O_j, $j \in O_j$, then

$$OPT_j = w_j + OPT_{p(j)}.$$

- or $j \notin O_j$, then

$$OPT_j = OPT_{j-1}$$

OBS: ① Interval j is in an optimal solution O_j if

$$w_j + OPT_{p(j)} \geq OPT_{j-1}.$$

② Interval j is <u>not</u> in an optimal solution O_j if

$$OPT_{j-1} \geq w_j + OPT_{p(j)}.$$

Dynamic Programming, in general, expresses the value of optimal solutions in terms of the value of optimal solutions to <u>smaller</u> problems.

```
DEF   Compute-OPT(j):        // compute the value of OPT_j for j=0,1,..,n
      IF  j==0:
          RETURN 0
```

ELSE:

RETURN $\max(w_j + \text{COMPUTE-OPT}(p(j)), \text{COMPUTE-OPT}(j-1))$

L: COMPUTE-OPT(j) returns OPT_j for $j = 0, 1, \ldots, n$

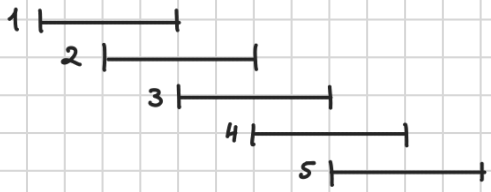
P: BASE CASE: $j = 0$ and $OPT(0) = 0 = \text{COMPUTE-OPT}(0)$. ✓

IND. STEP: Suppose that $\text{COMPUTE-OPT}(i) = OPT(i)$ $\forall\, i \leq j$.

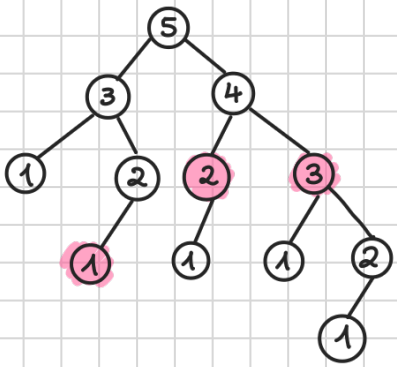
Then, by OBS. $OPT(j+1) = \max(w_{j+1} + OPT(p(j+1)), OPT(j))$.

Then, $\text{COMPUTE-OPT}(j+1) = OPT(j+1)$. ■

The issue with the algorithm is that it takes exponential time



COMPUTE-OPT(5)



Then, COMPUTE-OPT(n) takes time $\geq 2^{n/2}$

M-COMPUTE-OPT(j):

GLOBAL M memoization

IF j IS A KEY OF M:

return M[j]

We assume that the intervals are sorted by finishing time, otherways, we should sort them, in order to compute the p(J) quick.

ELSE:

IF $j == 0$: # Base case

$M[j] = 0$

ELSE:

$M[j] = \max(w_j + \text{M-COMPUTE-OPT}(p(j)) \text{ AND } \text{M-COMPUTE-OPT}(j-1))$

RETURN $M[j]$

This algo fills-up the dictionary M, so that M[J] contains the value of an optimal solution that uses only the first J intervals.

L : M-COMPUTE-OPT(j) RETURNS OPT_j = COMPUTE-OPT(j)

L : M-COMPUTE-OPT(j) TAKES O(j) TIME.

EXERCISE :

Write down a version of M-COMPUTE-COST that is not recursive.