Dynamic Programming

24-01-2022

Sakeena Shahid

Dynamic Programming - Introduction

Writes down "1+1+1+1+1+1+1+1 =" on a sheet of paper.

"What's that equal to?"

Counting "Eight!"

Writes down another "+1" on the left.

"What about that?"

"Nine!" " How'd you know it was nine so fast?"

"You just added one more!"

"So you didn't need to recount because you remembered there were eight! Dynamic Programming is just a fancy way to say remembering stuff to save time later!"

Weighted Interval Scheduling

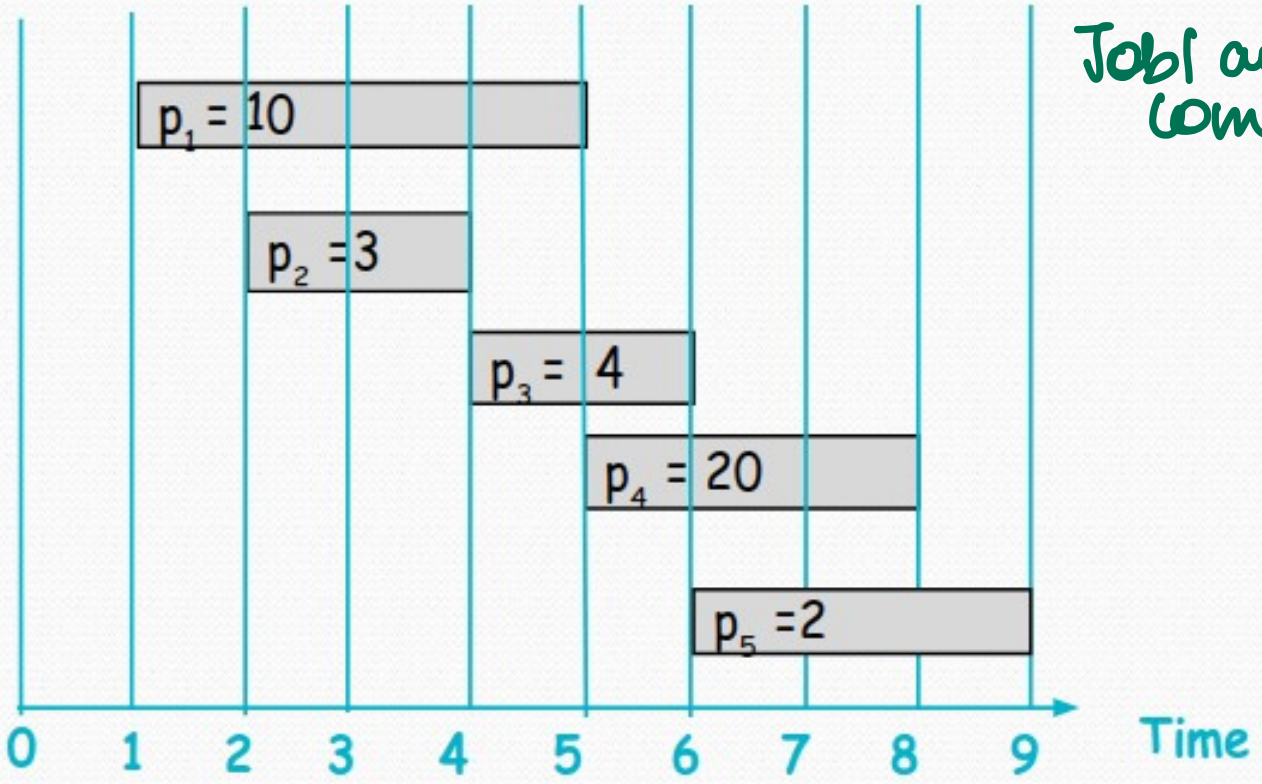
- **Problem:** Given a set of jobs described by (s_i , f_i , p_i) where, s_i and f_i are the starting and the finishing times and p_i is the profit of scheduling the i-th job.
- **Aim:** Find an **optimal** schedule of **compatible** jobs that earns the maximum profit for the company.
- Two jobs are said to be **compatible** if they don't overlap in time i.e. one finishes before the other one starts.
- **Greedy approach:** Choose the job which finishes first.....does it work?

  ↳ shortest job first

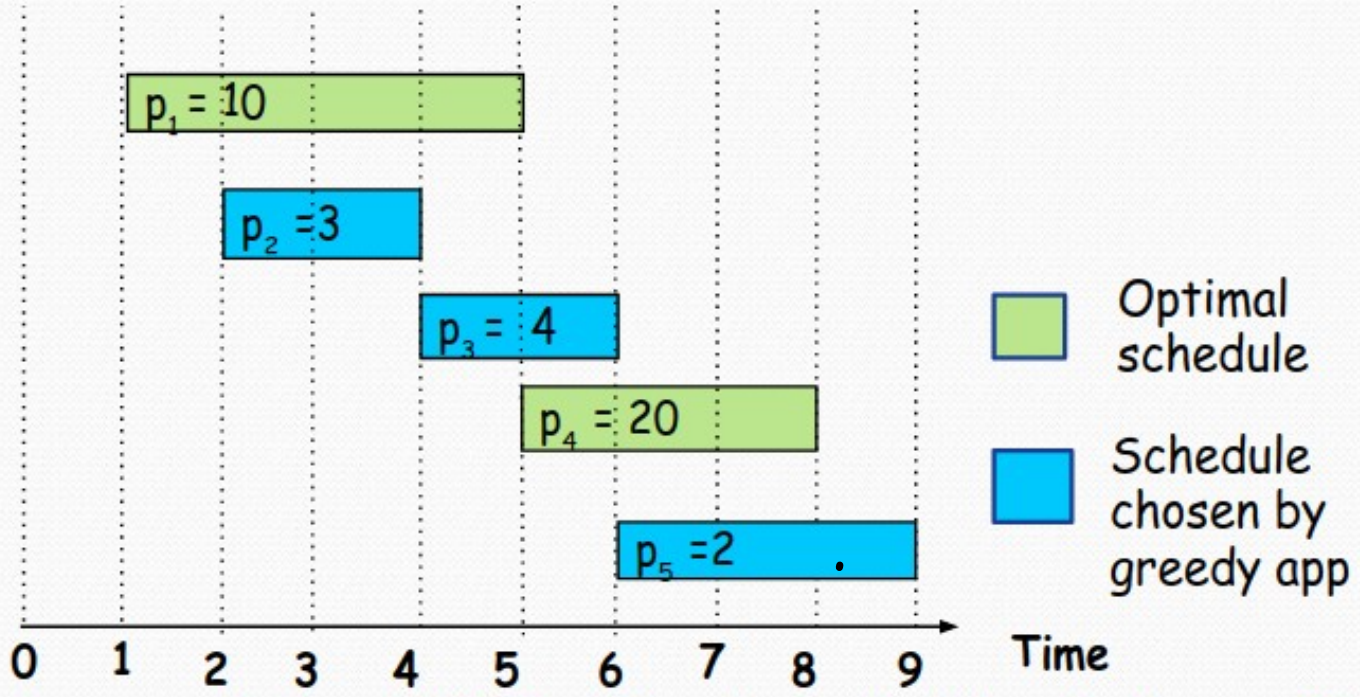
Does Greedy work?

Note: Job1 and Job2 are not compatible.

Job1 and Job4 are compatible.



Greedy does not work

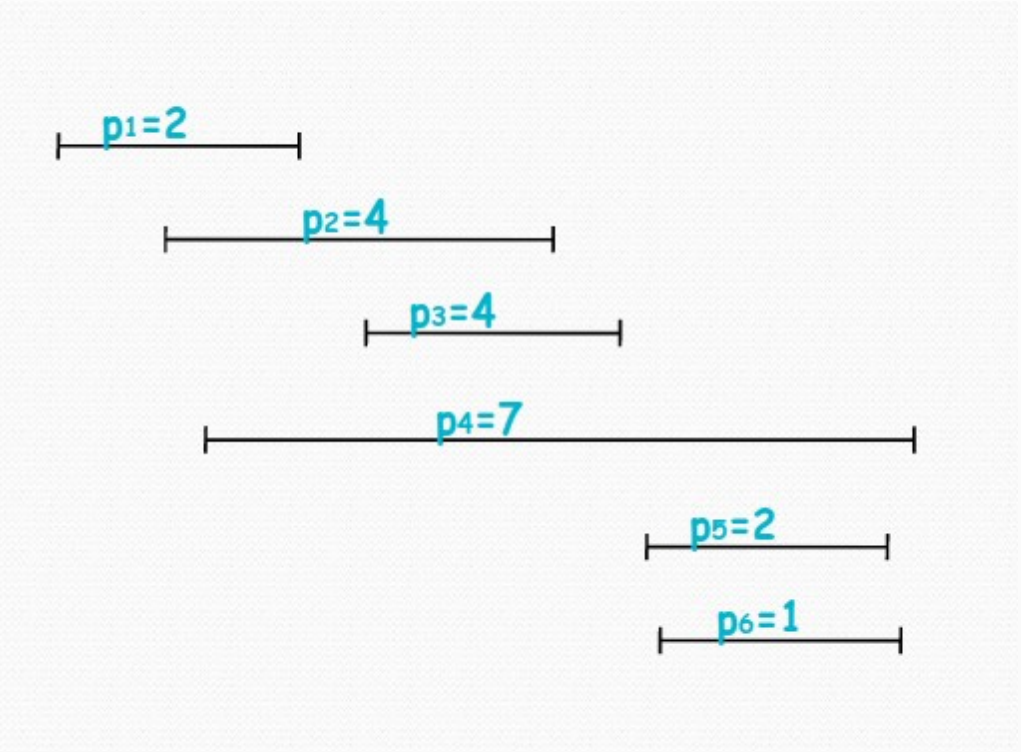


Greedy approach takes job 2, 3 and 5 as best schedule and makes profit of 7. While optimal schedule is job 1 and job4 making profit of 30 (10+20). Hence greedy will not work

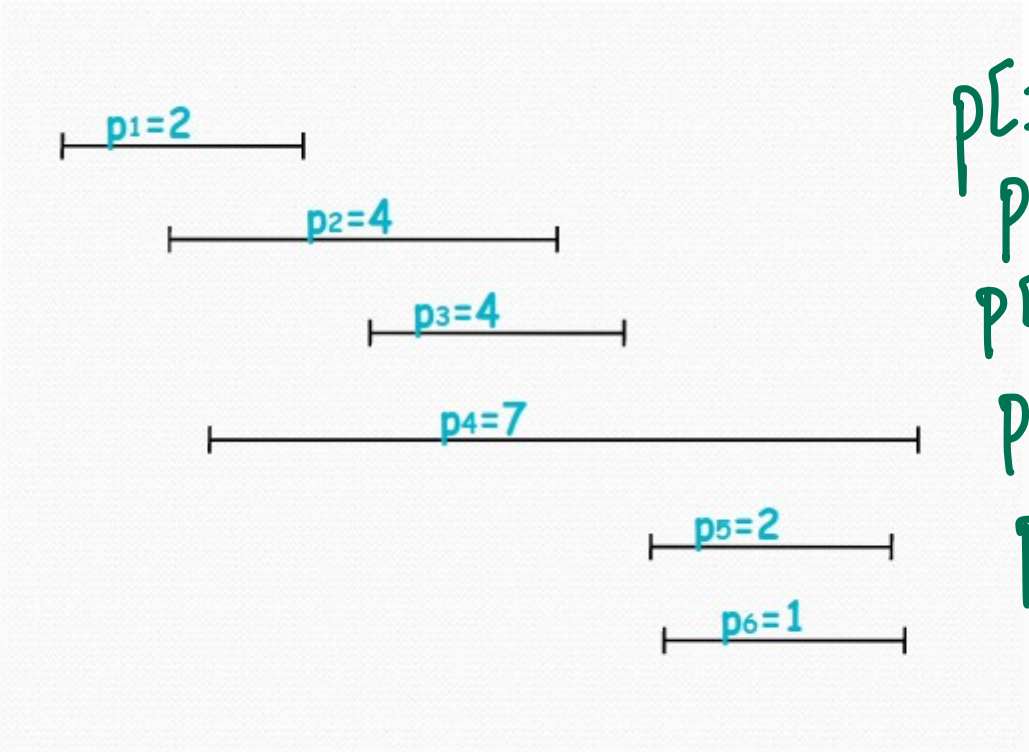
Solution

- Assume: Jobs are sorted in non-decreasing order: $f_1 \leq f_2 \leq f_3 \leq \ \ldots \leq f_n$
- Job/Interval i comes before job j if i < j
- Define p(j): defined for interval j; it is the largest index i < j such that i and j are **compatible.**
- Example (see next slide)

Example



Example



p[1] = 0
p[2] = 0
p[3] = 1
p[4] = 0
p[5] = 3
p[6] = 3

Recursive Solution

- Let us consider the optimal solution to WIS is ***O***
- We do not know what ***O*** is!
- But we know, either the last interval (interval n) belongs to ***O*** or it does not.
- Let us explore both possibilities:
  - Case 1: n $\in$ ***O***
  - Case 2: n $\notin$ ***O***

Case 1: n ∈ *O*

- If n ∈ ***O***, then no interval between p(n) and n can be part of ***O***
- Intervals p(n)+1, p(n)+2, …. , n-1 all overlap with n

intervals b/w p(n) and n.

- Also, if n ∈ ***O***, then ***O*** must include an optimal solution to the problem consisting of intervals 1 to p(n)

Case 2: $n \notin O$

- Simple
- If $n \notin O$, then O is simply the optimal solution to problem consisting of intervals 1 to n-1.

What do we observe?

- We observe that finding the optimal solution O involves looking at optimal solutions of smaller problems of the form {1, 2, ..., j}
- Therefore, for any value of j between 1 and n, let O_j be the optimal solution to the problem consisting of intervals {1, 2, ..., j}
- OPT(j): denote the value of optimal solution O_j
- OPT(0) = 0 (optimal value of empty set)
- **Aim:** to find solution O_n with value **OPT(n)**

optimal solution ⟵

⟶ optimal value

What is ***OPT(j)***?

- As before, either $j \in O_j$ ***or*** $j \notin O_j$
- Case 1: $j \in O_j$
  - $OPT(j) = p_j + OPT(p(j))$
- Case 2: $j \notin O_j$
  - $OPT(j) = OPT(j-1)$
- As we are looking for maximizing the company's profit,
  - $OPT(j) = \max(p_j + OPT(p(j)), OPT(j-1))$

How do we decide if $j \in O_j$?

- Simple:
- If and only if : $\mathbf{p_j + OPT(p(j)) \geq OPT(j-1)}$

Steps for a recursive solution for WIS

- Take decision about the nth job.
- What choices optimal has?
  - schedule the nth job or,
  - not to schedule the nth job
- What is the profit in either case?
- Let OPT[n]= profit from the optimal schedule from the first n jobs,
- and let p(j) be the rightmost interval before j that is compatible with j.

Steps for a recursive solution for WIS

- Either nth job is scheduled by the OPTimal solution or it is not.
  - If it is not, then OPT(n) = OPT(n-1)
  - If it is, then OPT(n) = p_n + OPT(p(n))
- Thus, OPT(n)= max(p_n + OPT(C(n)), OPT(n-1))
- We haven't defined OPT(p(n)) and OPT(n-1). So we need to generalize the definition of OPT.

Generalizing

- Let Opt[~~n~~ j]= profit from the optimal schedule from the first ~~n~~ j jobs,
- Opt[~~n~~ j]= max($p_{~~n~~\,j}$ + Opt[C(~~n~~ j)], Opt[~~n~~ j-1])

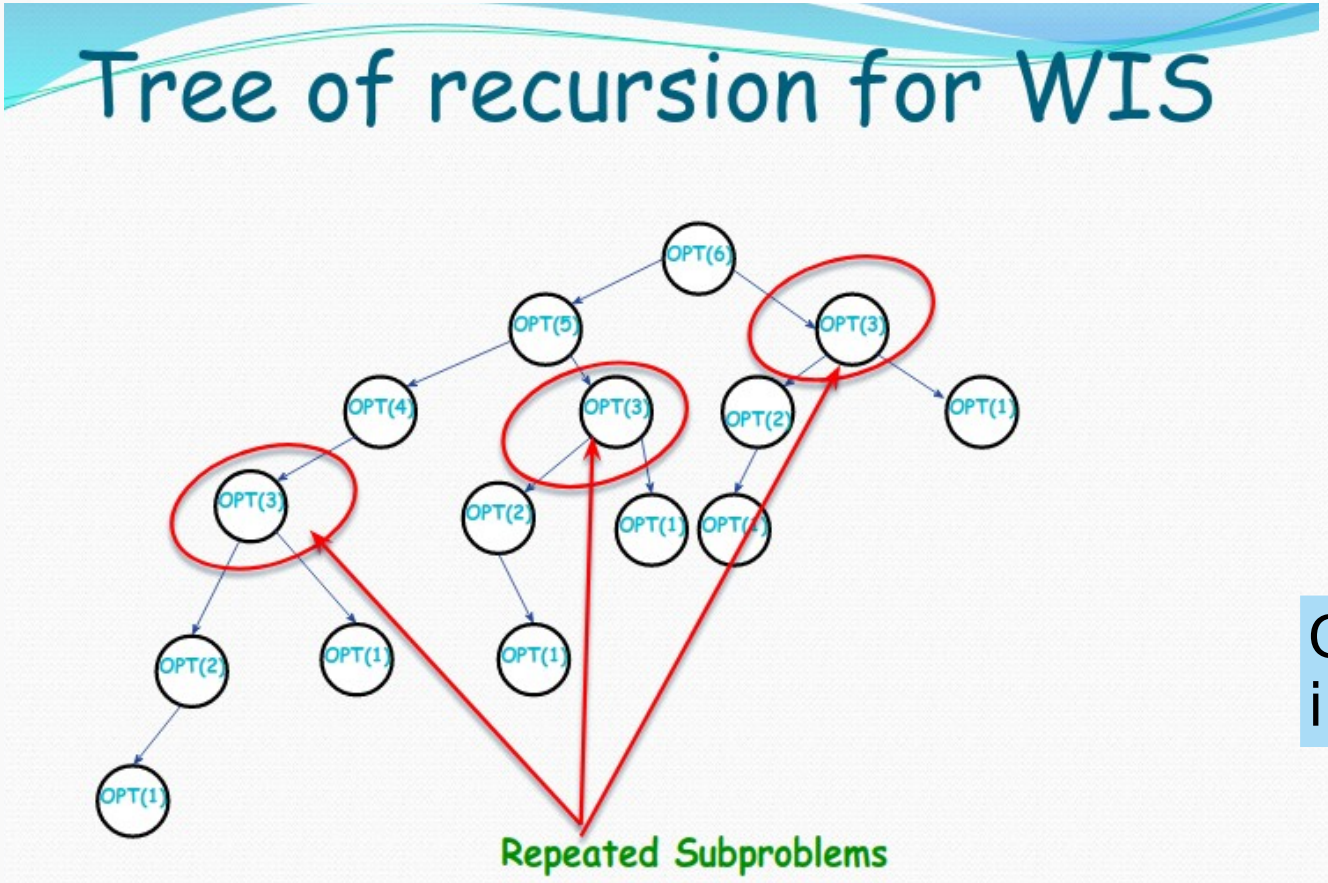
→ for the n-th job, $j \in [1, p(n)]$

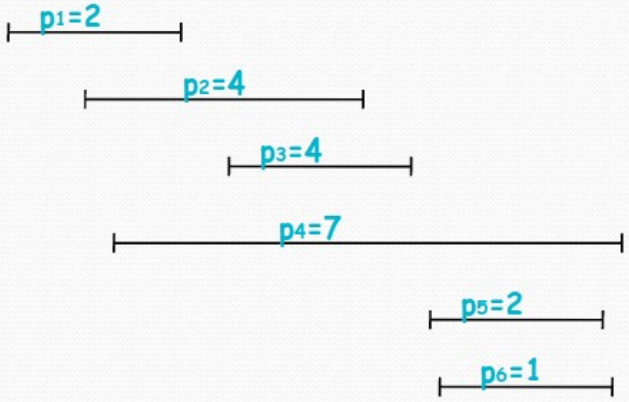
Algorithm

Recursive algorithm

```
Compute_Opt(j)
If j=0 then
  Return 0
Else
     m1 = Compute_Opt(j-1)
     m2 = Compute_Opt(p(j))
     If m2 + p_j  >  m1 then add j to the solution
     return m2 + p_j  else return m1.
Endif
```

Oh no!!





Can be exponential in the worst-case!

Solution: Memoizing the solution

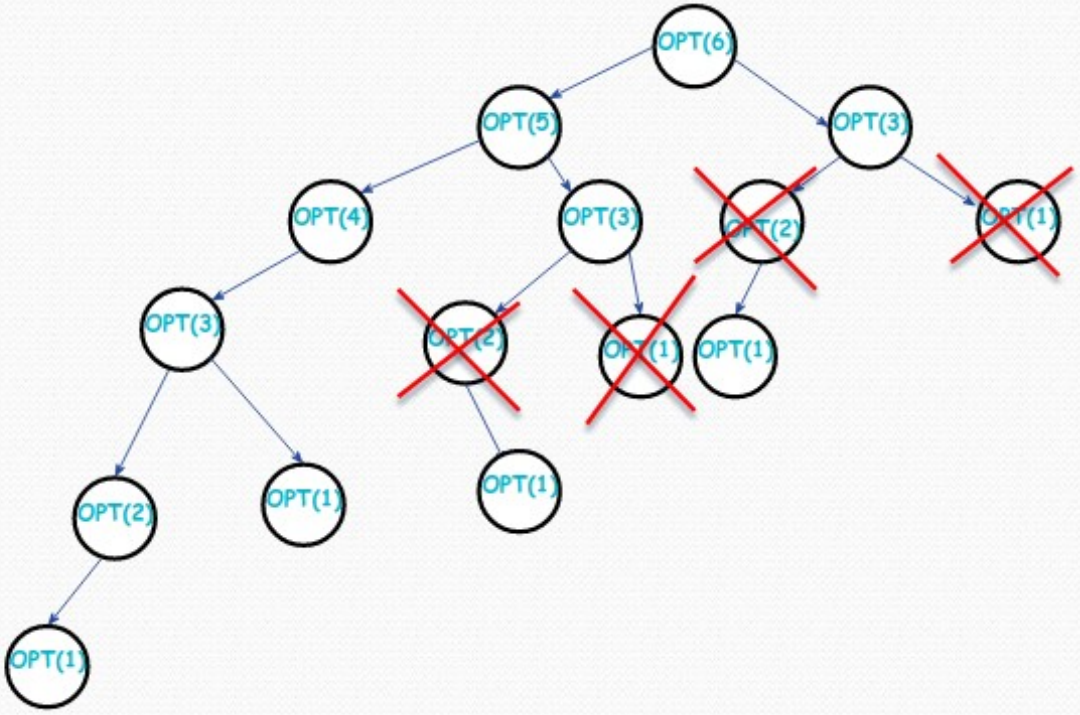
- Store and re-use

```
M-Compute-Opt(j)
  If j=0 then
     return 0
  Else if Opt[j] is already computed then
     return Opt[j]
  Else
 Opt[j]= max( p_j + M-compute-Opt(C(j)), M-Compute-Opt(j-1))
     return Opt[j]
  Endif
```

Reuse

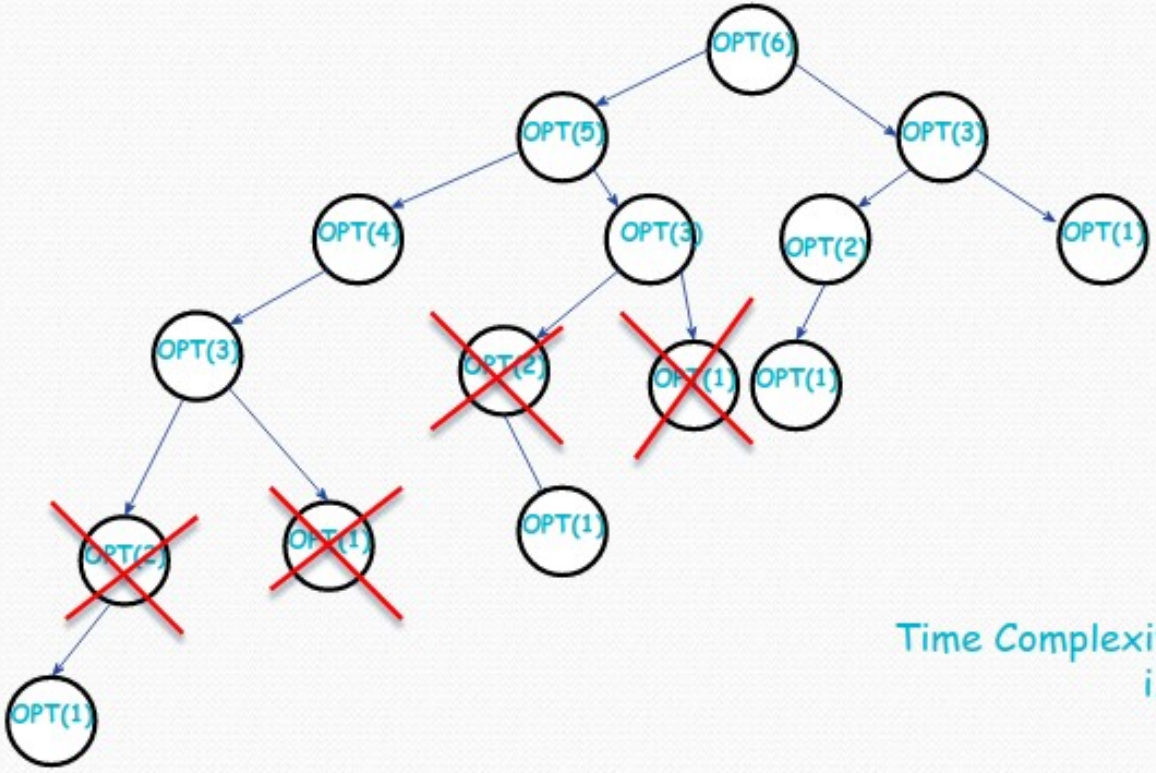
Store

Tree of recursion with Memoization (If recursive call to j-1 is executed before C(j))



It is easy to see that time spent on each call on C(j) is constant as it has been computed earlier and so it simply returns the pre-computed stored value. Clearly the time complexity is O(n)

Tree of recursion with Memoization (If recursive call to C(j) is executed before j-1)



Time Complexity Remains the Same
i.e. O(n)

Next Lecture

29-01-2022

- **Iterative solution to WIS**
- **Printing the optimal solution**
- **Other problems**

Quick Recap

- **Problem:** Weighted Interval Sceduling
- **Aim:** Find an **optimal** schedule of **compatible** jobs that earns the maximum profit for the company. => Greedy approach didn't work.
- **Recurrence solution:**
  - Take decision about the nth job.
  - What choices optimal has?
    - schedule the nth job or,
    - not to schedule the nth job
  - What is the profit in either case? Consider max and write the relation.
  - Generalize
  - Memoization

Iterative Solution: DP solution

DP-Compute-Opt

Order the intervals in the increasing order of finish times (assumed in the previous approach as well)

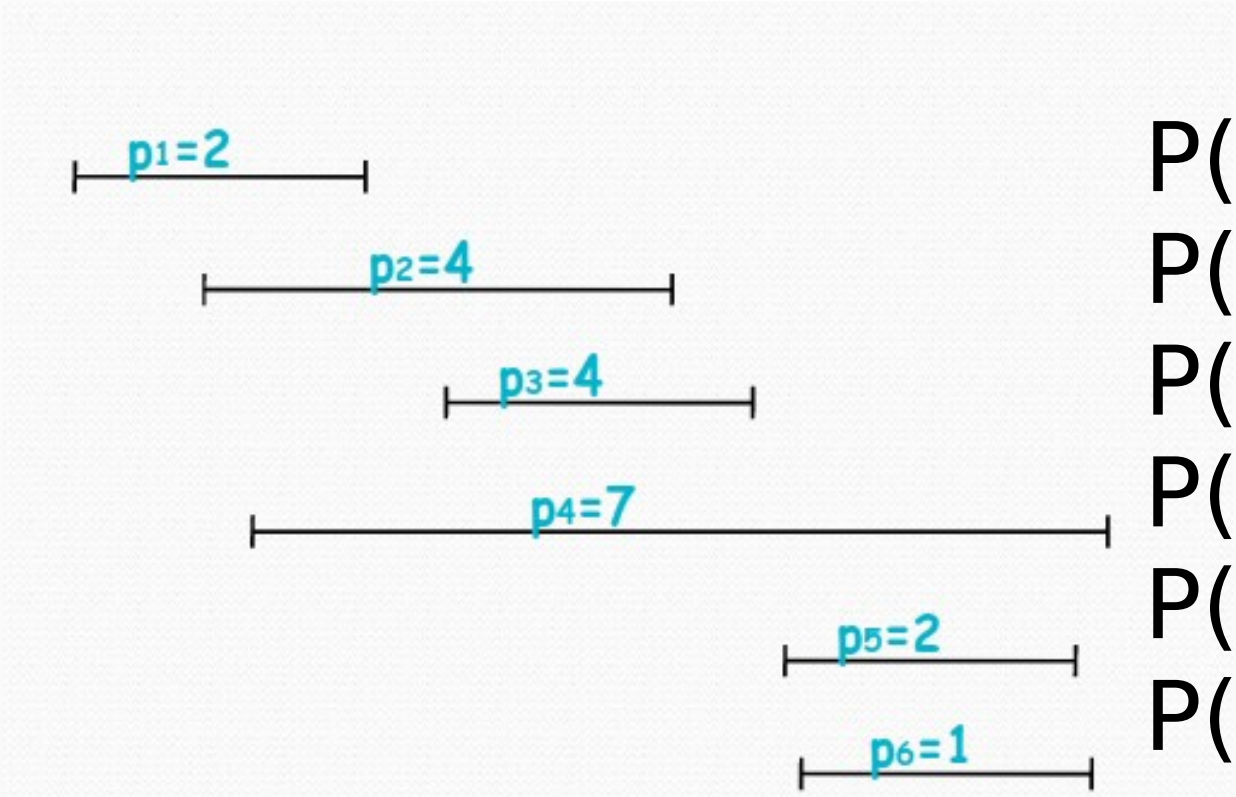
m[0]=0

For j=1,2,....,n

m[j]=max (**p_j** + **m[p(j)]** , **m[j-1]**)

End for

Let us consider an example

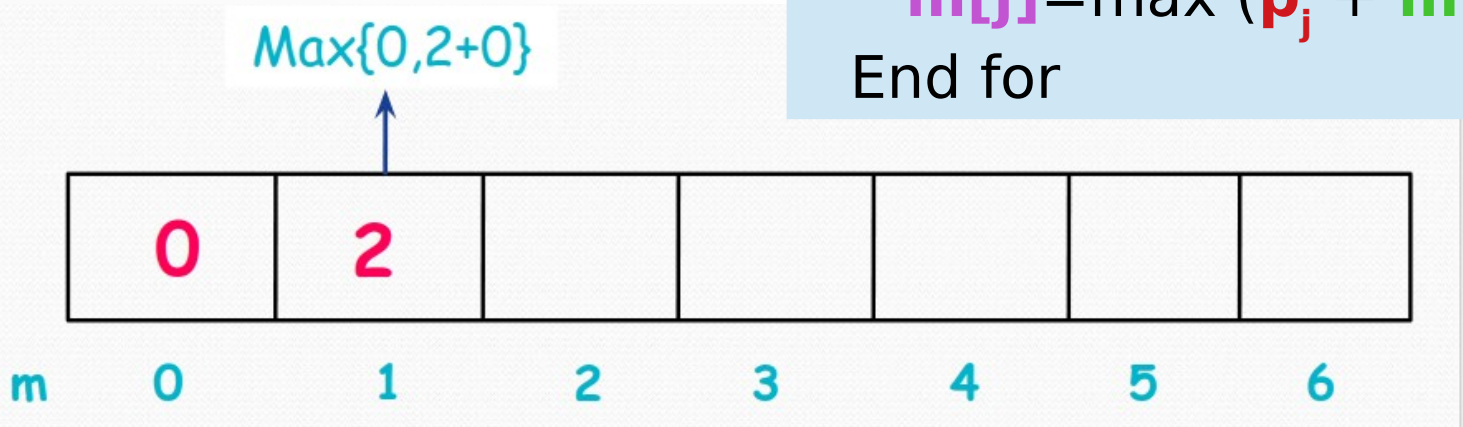


P(1) = 0
P(2) = 0
P(3) = 1
P(4) = 0
P(5) = 3
P(6) = 3

```
m[0]=0
  For j=1,2,....,n
    m[j]=max (p_j + m[p(j)] ,  m[j-1])
  End for
```

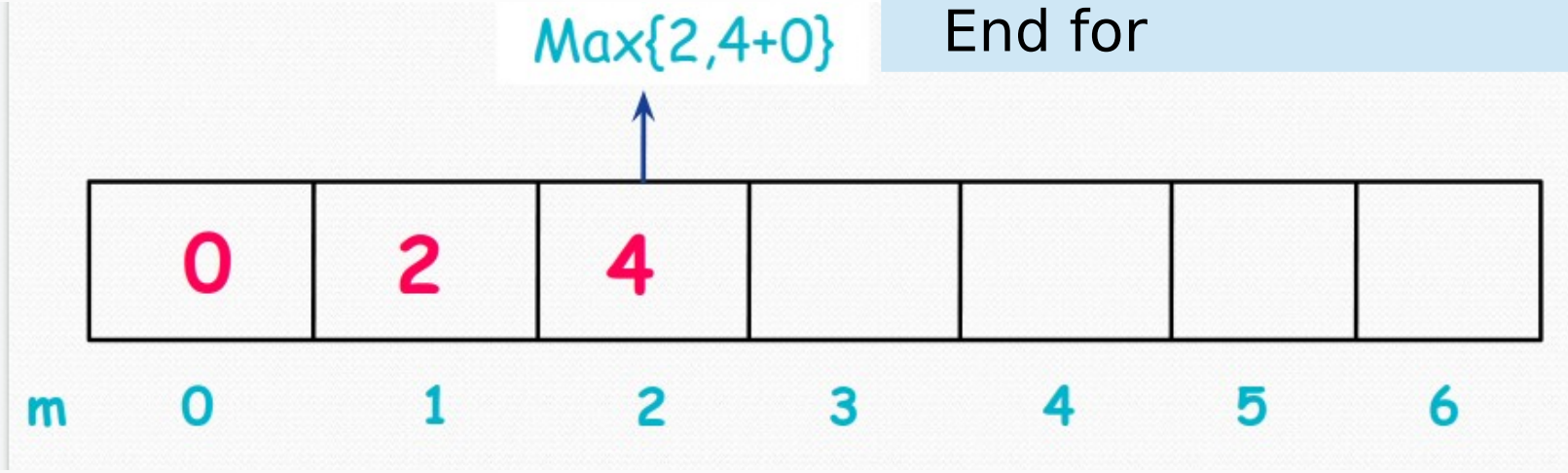
Initially, no job belongs to optimal solution.

m[0]=0
  For j=1,2,.....,n
    m[j]=max (p_j + m[p(j)] , m[j-1])
  End for



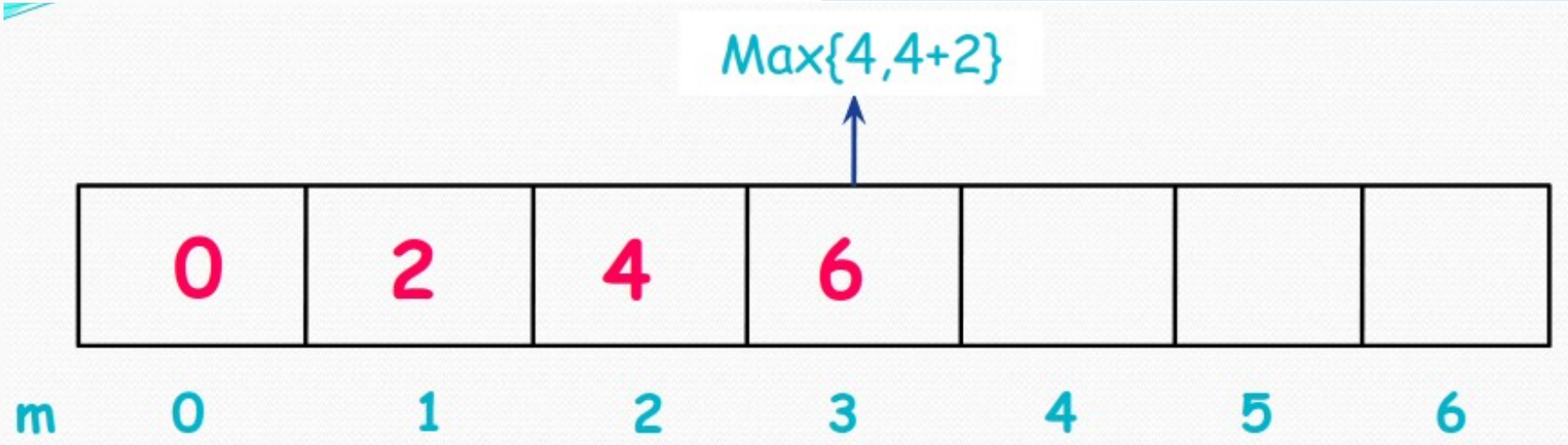
Pj = 2
If job 1 selected => m[1] = m[p(1)]+2 = 0+2 = 2
If job 1 not selected => m[1] = m[0] = 0

m[0]=0
  For j=1,2,....,n
    m[j]=max (p_j + m[p(j)] , m[j-1])
  End for



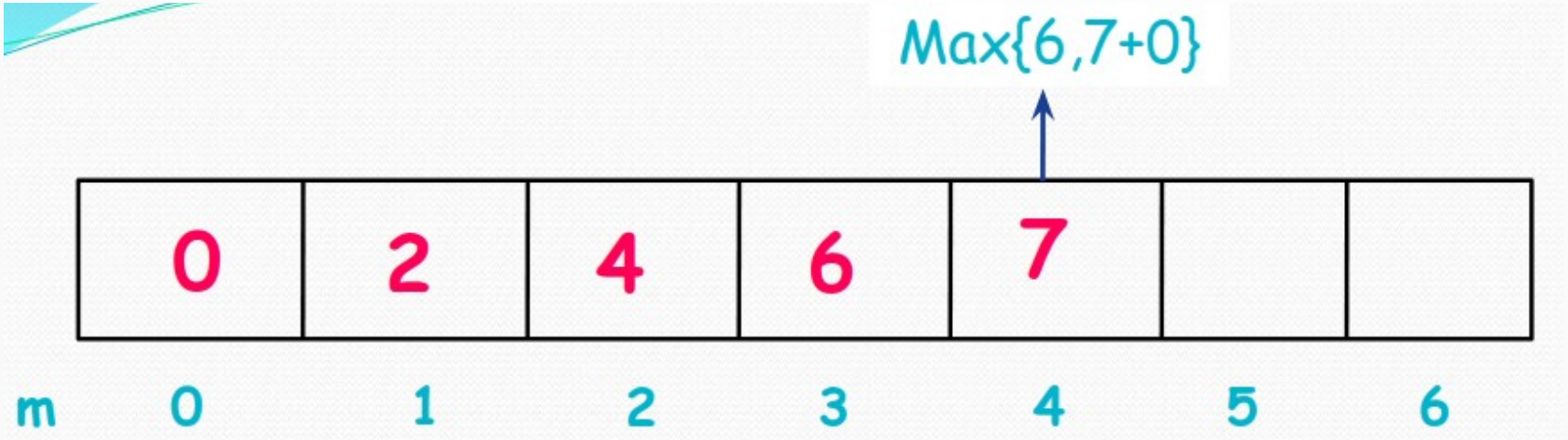
Pj=4
If job 2 is selected => m[2] = m[p(2)] + 4 = 0 + 4 = 4
If job 2 is not selected => m[2] = m[1] = 2

m[0]=0
  For j=1,2,....,n
    m[j]=max (p_j + m[p(j)] , m[j-1])
  End for



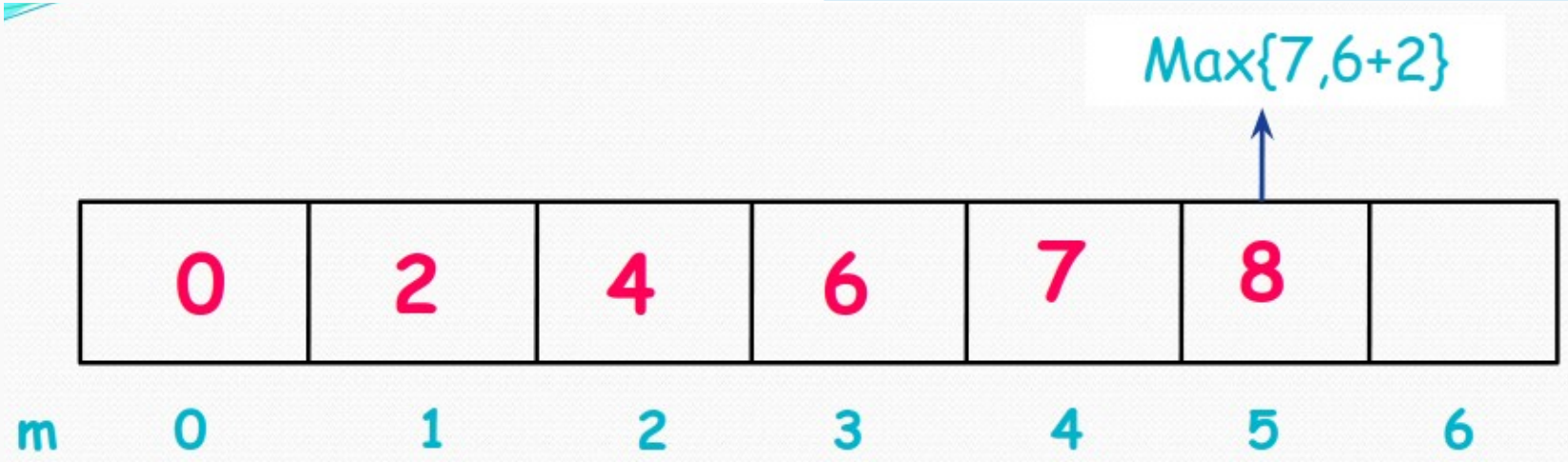
Pj= 4
If job 3 is selected => m[3] = m[p(3)] + 4 = 2 + 4 = 6
If job 3 is not selected => m[3] = m[2] = 4

```
m[0]=0
  For j=1,2,....,n
    m[j]=max (p_j + m[p(j)] ,  m[j-1])
  End for
```



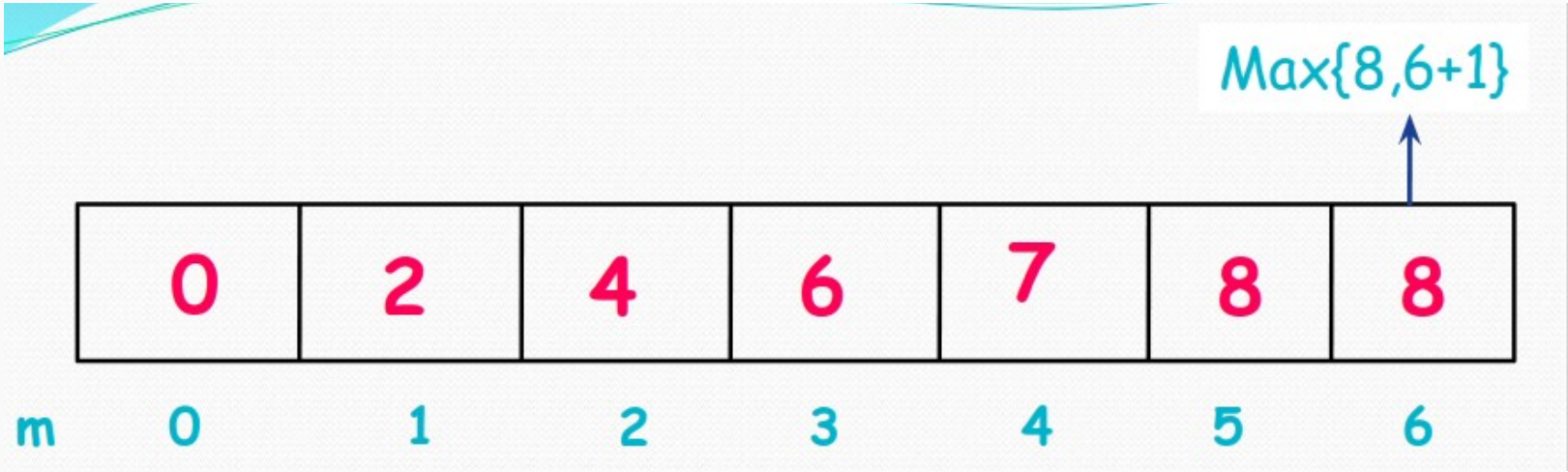
Pj= 7
If job 4 is selected => m[4] = m[p(4)] + 7 = 0 + 7 = 7
If job 4 is not selected => m[4] = m[3] = 6

```
m[0]=0
   For j=1,2,....,n
      m[j]=max (p_j + m[p(j)] ,  m[j-1])
   End for
```



Pj= 2
If job 5 is selected => m[5] = m[p(5)] + 2 = 6 + 2 = 8
If job 5 is not selected => m[5] = m[4] = 7

```
m[0]=0
  For j=1,2,....,n
    m[j]=max (p_j + m[p(j)] ,  m[j-1])
  End for
```



n.

Pj= 1
If job 6 is selected => m[6] = m[p(6)] + 1 = 6 + 1 = 7
If job 6 is not selected => m[6] = m[5] = 8

Computing a solution in addition to its value

- Use the optimal solution array M you just computed to find the optimal solution.
- Include job j only if $p_j + opt(p(j)) \geq opt(j-1)$
- That is, you **TRACE BACK** through M.
- See next slide.

Jobs

Input Weights

0 2 4 4 7 2 1

Please run this algorithm on the input above and verify it works okay.

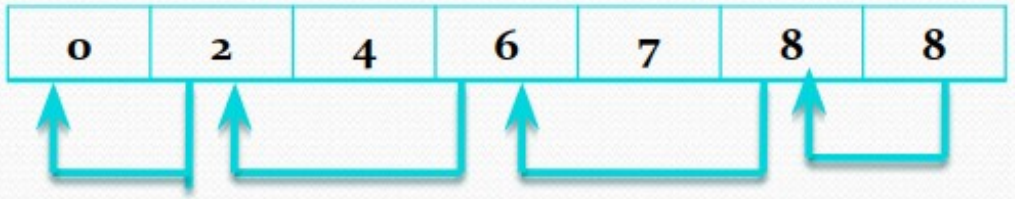
```
Find-Solution(j)
  If j = 0 then
    Output nothing
  Else
    If v_j + M[p(j)] ≥ M[j − 1] then
      Output j together with the result of Find-Solution(p(j))
    Else
      Output the result of Find-Solution(j − 1)
    Endif
  Endif
```

Jobs

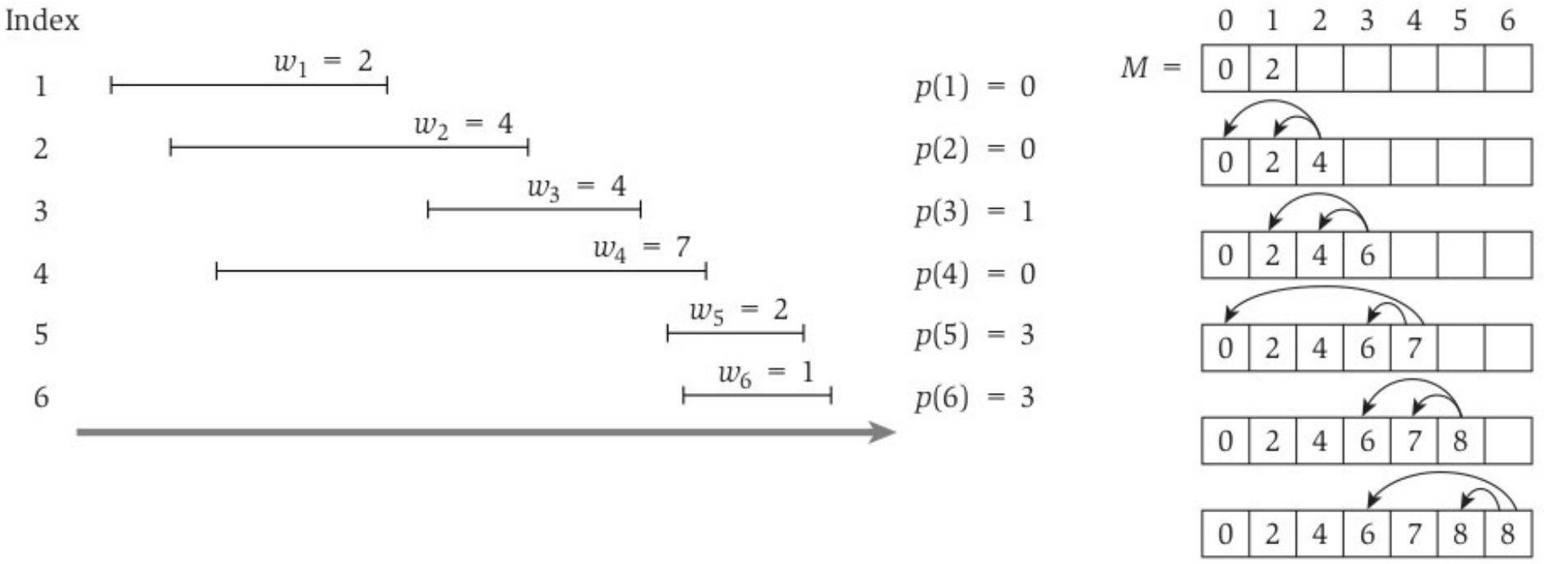
Input Weights

0 2 4 4 7 2 1

0	2	4	6	7	8	8




```
Find-Solution(j)
  If j = 0 then
    Output nothing
  Else
    If v_j + M[p(j)] ≥ M[j − 1] then
      Output j together with the result of Find-Solution(p(j))
    Else
      Output the result of Find-Solution(j − 1)
    Endif
  Endif
```


Index
1
2
3
4
5
6
$w_1 = 2$
$w_2 = 4$
$w_3 = 4$
$w_4 = 7$
$w_5 = 2$
$w_6 = 1$
$p(1) = 0$
$p(2) = 0$
$p(3) = 1$
$p(4) = 0$
$p(5) = 3$
$p(6) = 3$
$M =$
0 1 2 3 4 5 6
0 2
0 2 4
0 2 4 6
0 2 4 6 7
0 2 4 6 7 8
0 2 4 6 7 8 8

Principles of Dynamic Programming

General template

- First, we designed an exponential recursive algorithm for WIS.
- The we applied memoization to turn it into a polynomial solution => Create the gloabal array M
- Then we designed an iterative algorithm => create M
- As soon as you have M, you have the solution.
- This is, we can compute the entries in M by an iterative algorithm (increment j, repeat the procedure), rather than memoized recursion.

Time complexity

- What is the time complexity?

```
m[0]=0
   For j=1,2,....,n
      m[j]=max (p_j + m[p(j)] ,  m[j-1])
   End for
```

So...

- So from now on,

  DP solutions will be written in iterative fashion instead of memoized recursions, simply because iterative code is easier to write and understand.

To ponder

- Both approaches (recursive or iterative), require that the solution of the subproblem is supplied so that the new problem can be solved.
- DP algorithms needs a collection of sub-problems derived from the original problem s.t.
  - There are only a polynomial no. of problems
  - The solution to the original problem can be computed easily for the solutions of the subproblems
  - There is a notion of ordering on subproblems, we move from smaller subproblems to larger ones.

Principles of DP

- Recursive Solution (Optimal Substructure Property)
- Overlapping Subproblems
- Total Number of Subproblems is polynomial
- A major ingredient of a DP solution is "ordering".

Some definitions

- **Optimal Substructure:** A given problem has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.
- **Overlapping Subproblems:** Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don’t have to recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again.

Programming task

- To be discussed on Wednesday.
- Implement WIS.
- Create a file with information about n intervals in the form of (start_time, finish_time, profit), Can use random.
- Take n = any value for which interpreting results is easy. Can use random.
- Goal: Find the optimal solution and the optimal value