BLG336E – Analysis of Algorithms

Recitation 7

Dynamic Programming: Weighted Interval Scheduling

19.04.2022

Overview of algorithm types

Greedy Algorithm ?

- Principle: Builds a solution in small steps, choosing a decision at each step locally to optimize some underlying criterion.
- Always optimum? NO
- Example: Dijsktra, kruskal, prim …

Divide & Conquer ?

- Principle: Break up problem into several parts, Solve each part recursively, Combine solutions to sub-problems into overall solution.
- Always optimum? Yes if correctly implemented
- Example: Merge sort, …

Greedy Algorithm ?

- Principle: builds a solution in small steps, choosing a decision at each step locally to optimize some underlying criterion.
- Always optimum? NO
- Example: Dijsktra, kruskal, prim …

Divide & Conquer ?

- Principle: Break up problem into several parts, Solve each part recursively, Combine solutions to sub-problems into overall solution.
- Always optimum? Yes if correctly implemented
- Example: Merge sort, …

Dynamic Programming ?

- Principle: Carefully decompose the problem into a series of sub-problems, and build up correct solutions to larger and larger sub-problems. Dangerously close to the edge of brute force search. Systematically works through the set of possible solutions to the problem, it does this without ever examining all of them explicitly (Stored solutions).
- Always optimum? Most likely
- Example: Weighted interval scheduling, Knapsack, Bellman Ford..

Dynamic programming

Dynamic Programming = Recursion + Memoization

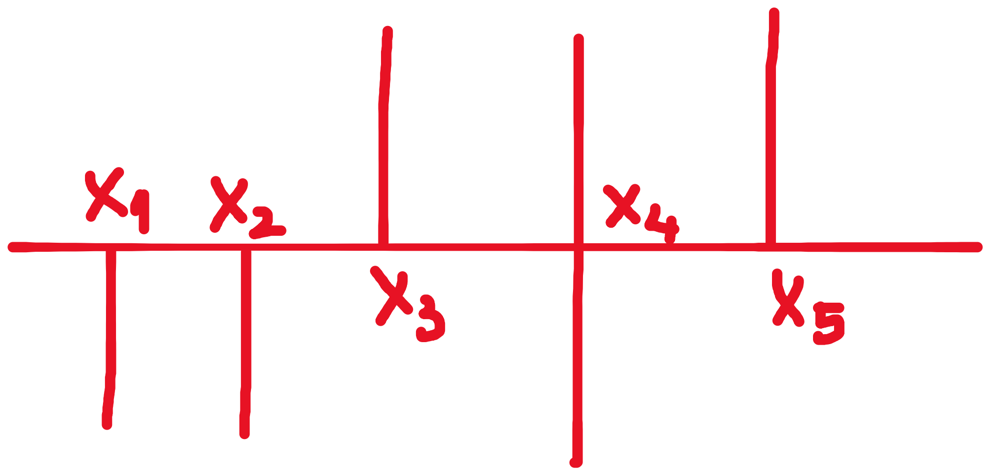
- Formulate problem recursively in terms of solutions to polynomially many sub-problems
- Solve sub-problems bottom-up, storing optimal solutions

An advertisement problem

- You are going to advertise your new internet startup company on İstiklal Caddesi.
- You are considering putting one advertisement at each of

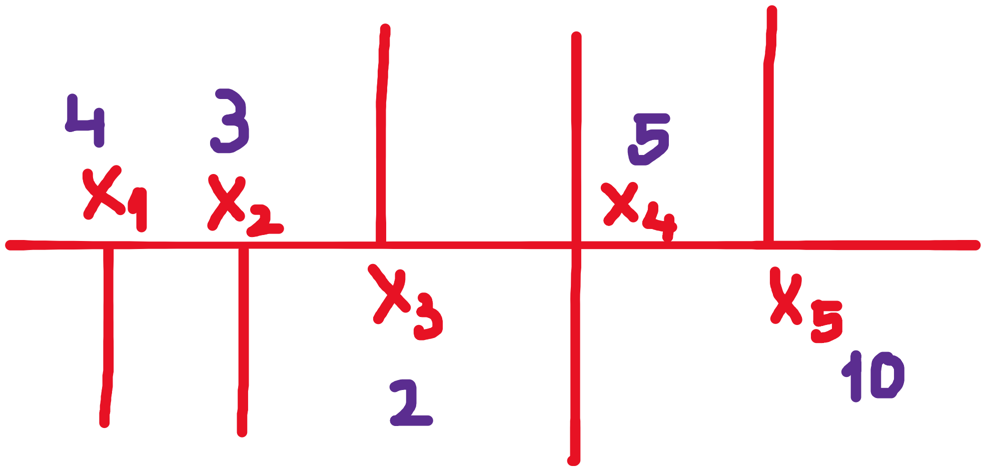
$$M = 5 \text{ intersections } \{x_1, x_2, x_3, x_4, x_5\}$$

given in map.



An advertisement problem

- These intersections allow you to reach the attention of
$$\{r_1, r_2, r_3, r_4, r_5\} = \{4, 3, 2, 5, 10\}$$
people per minute respectively.
- The municipality requires that if you put an advertisement in one intersection, you can not put an advertisement at the *two neighboring* intersections:
- If you pick x_1, you can not pick x_2.
- If you pick x_3,you can not pick x_2 or x_4.



An advertisement problem

- You need to find which intersections to put advertisements so that you can *maximize the number of people you reach every minute*.

- Use *dynamic programming* to provide a solution for this problem considering any $M > 0$ intersections.

How can we assert this problem?

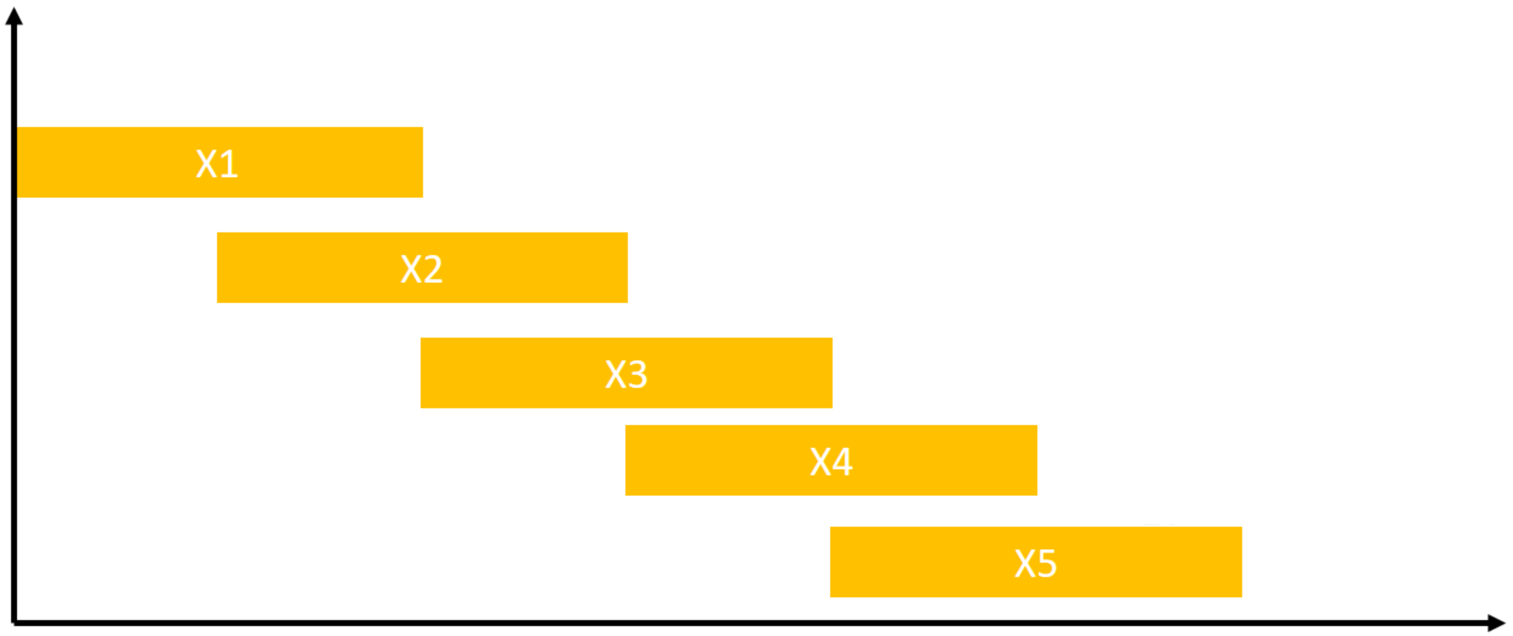
You can reach 4, 3, 2, 5, 10 people per minute

You can not put an advertisement at the two neighboring intersections

Aim: maximize the number of people you reach every minute

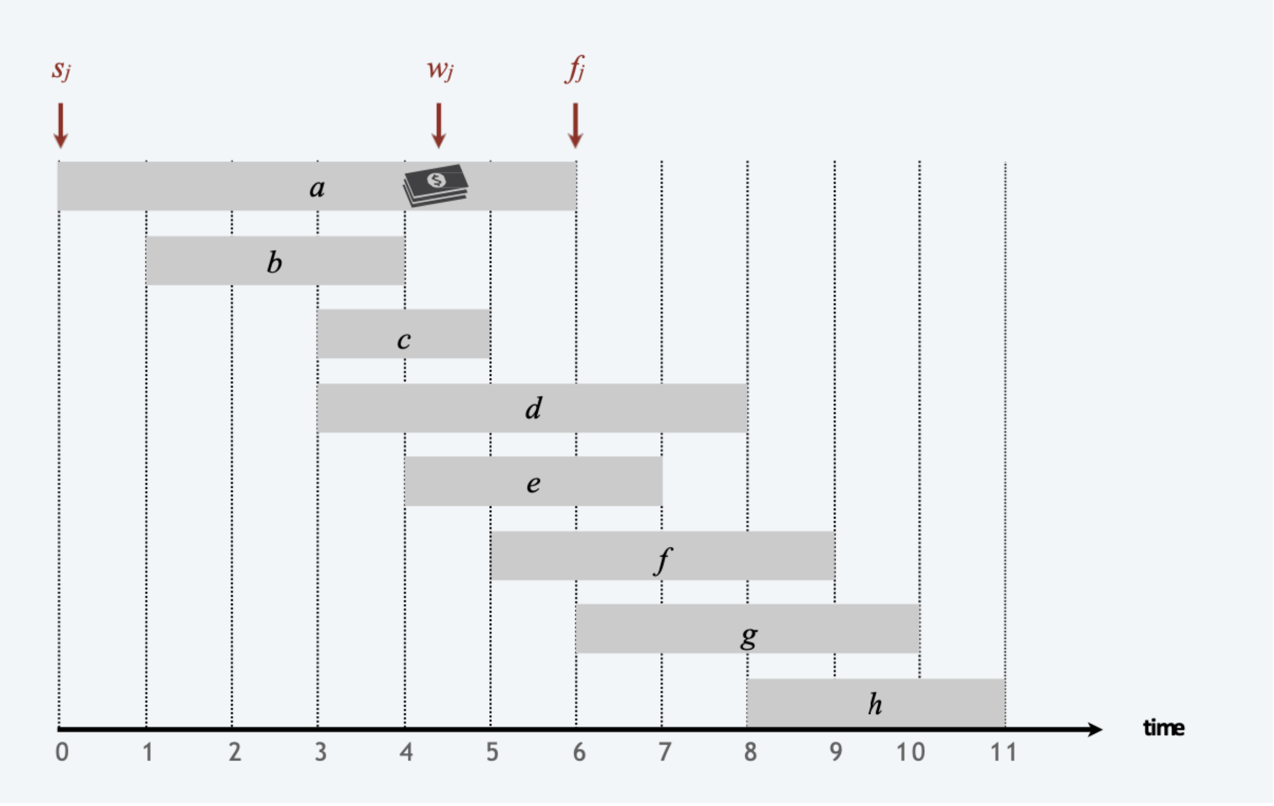
Solution to the advertisement problem

- Our problem can be asserted as a *weighted interval scheduling* where only jobs i and $i + 1$ intersect (two neighbor intersection).



Let’s remember the weighted interval scheduling problem

- Job j starts at s_j, finishes at f_j, and has weight $w_j > 0$.
- Two jobs are compatible if they don’t overlap.
- Goal: find max-weight subset of mutually compatible jobs.



Picture from course slides

Solve the problem with memoization (top-down dynamic programming)

- **Inputs:** n → Number of jobs

  $s_1, s_2, \dots, s_n$ → Start times

  $f_1, f_2, \dots, f_n$ → Finish times

  $w_1, \mathrm{w}_2, \dots, \mathrm{w}_\mathrm{n}$ → Weights

- **Sort** jobs by their finish times so that $f_1 \leq f_2 \leq \cdots \leq f_n$
- **Compute** $p(1), p(2), \dots, p(n)$ → Compatible jobs
- **Initialize the array M**

```
for j=1 to n
  M[j] = empty
M[0] = 0
```

Solve the problem with memoization (top-down dynamic programming)

M-Compute-Opt(j)

If $j = 0$ then

Return 0

Else if $M[j]$ is not empty then

Return $M[j]$

Else

Define $M[j] = \max(w_j +$ M-Compute-Opt($p(j)$), M-Compute-Opt($j - 1$))

Return $M[j]$

Endif

Cache results of subproblem j in M[j]

Solve the problem with memoization (top-down dynamic programming)

During finding the solution, use M[j] to avoid solving subproblem j more than once

Find-Solution(j)
 If $j = 0$ then
  Output nothing
 Else
  If $w_j + M[p(j)] \geq M[j-1]$ then
   Output j together with the result of Find-Solution($p(j)$)
  Else
   Output the result of Find-Solution($j - 1$)
  Endif
 Endif

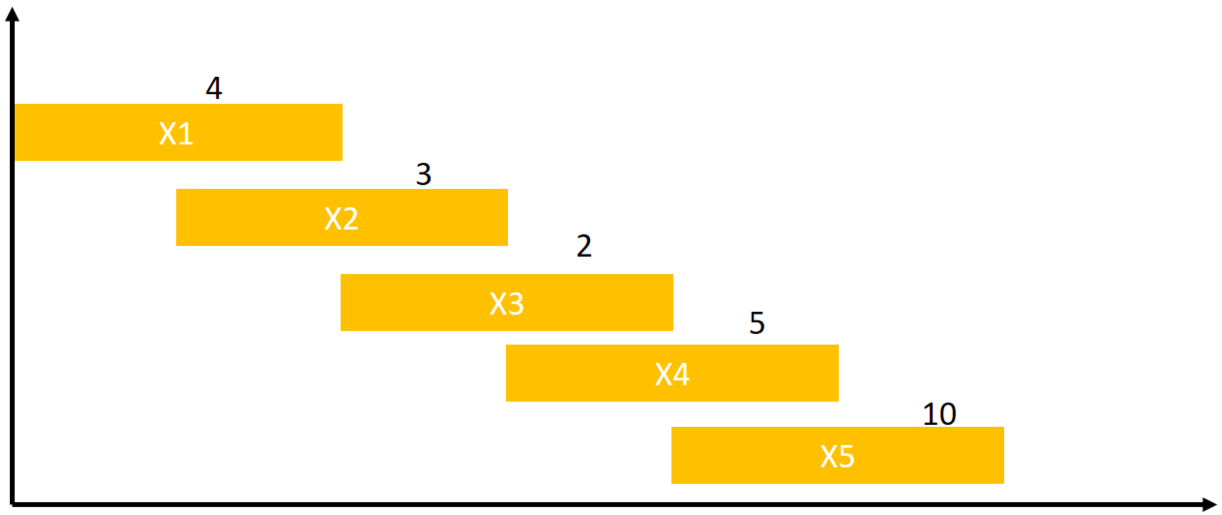
Solution to the advertisement problem

- **Step 1:** Weights w_i are replaced by r_i values (reached people per minute).

$$\{r_1, r_2, r_3, r_4, r_5\} = \{4, 3, 2, 5, 10\}$$

- **Step 2:** Label jobs by finishing time: $\mathrm{f}_1 \leq f_2 \leq \cdots \leq f_n$

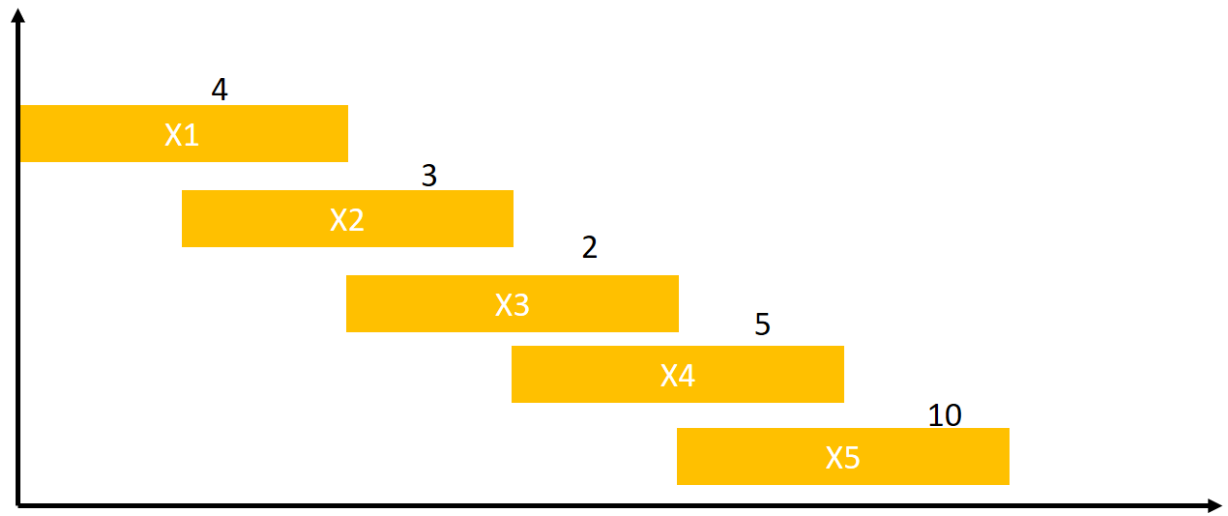
Jobs correspond to intersections $\{x_1, x_2, x_3, x_4, x_5\}$ and already ordered by their finish time



Solution to the advertisement problem

- **Step 3:** Compute

$$p(j) = largest\ index\ i < j\ such\ that\ job\ i\ is\ {\color{red}compatible}\ with\ j$$



p(1)=0, p(2)=0, p(3)=1, p(4)=2, p(5)=3

Solution to the advertisement problem

- **Step 4:** Objective function

$$OPT(j) = \begin{cases} 0, & if\ j = 0 \\ \max\{r_j + OPT\big(p(j)\big), OPT(j-1)\}, & otherwise \end{cases}$$

value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

Solve the problem with memoization

- **Step 5:** Run M-Compute-Opt(5)
- **Step 6:** Find-Solution(5)

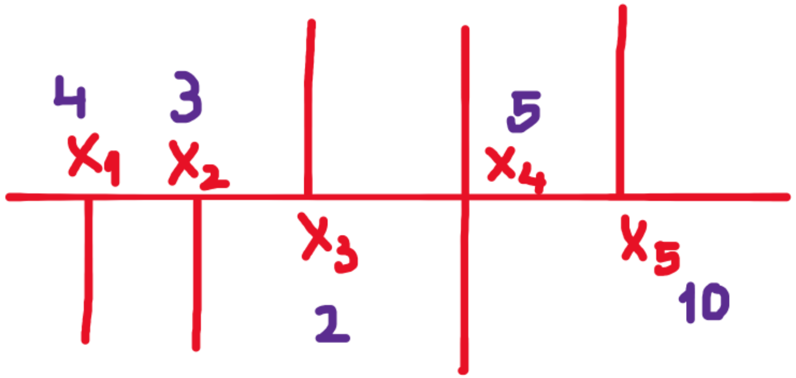
Compute optimal value array

M-Compute-Opt(j)
 If $j = 0$ then
  Return 0
 Else if $M[j]$ is not empty then
  Return $M[j]$
 Else
  Define $M[j] = \max(w_j +$ M-Compute-Opt$(p(j))$, M-Compute-Opt$(j - 1))$
  Return $M[j]$
 Endif

Find the solution

Find-Solution(j)
 If $j = 0$ then
  Output nothing
 Else
  If $w_j + M[p(j)] \geq M[j-1]$ then
   Output j together with the result of Find-Solution$(p(j))$
  Else
   Output the result of Find-Solution$(j - 1)$
  Endif
 Endif

Run M-Compute-Opt(5)



M-Compute-Opt(j)

If $j = 0$ then

  Return 0

Else if $M[j]$ is not empty then

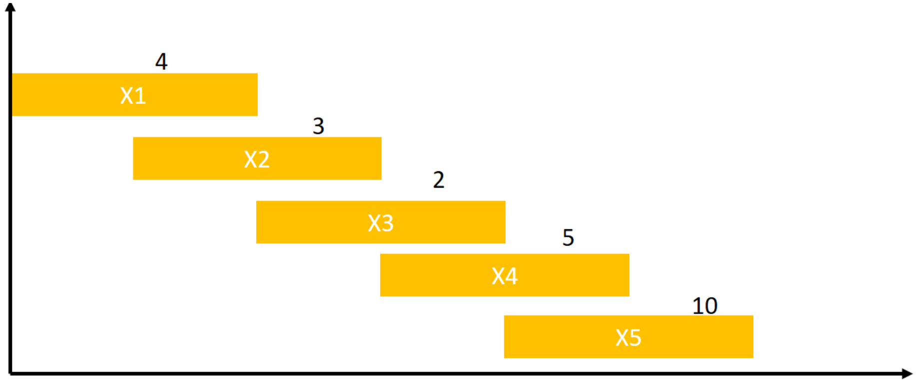
  Return $M[j]$

Else

  Define $M[j] = \max(w_j + \text{M-Compute-Opt}(p(j)), \text{M-Compute-Opt}(j-1))$

  Return $M[j]$

Endif



M:

0	1	2	3	4	5
0					

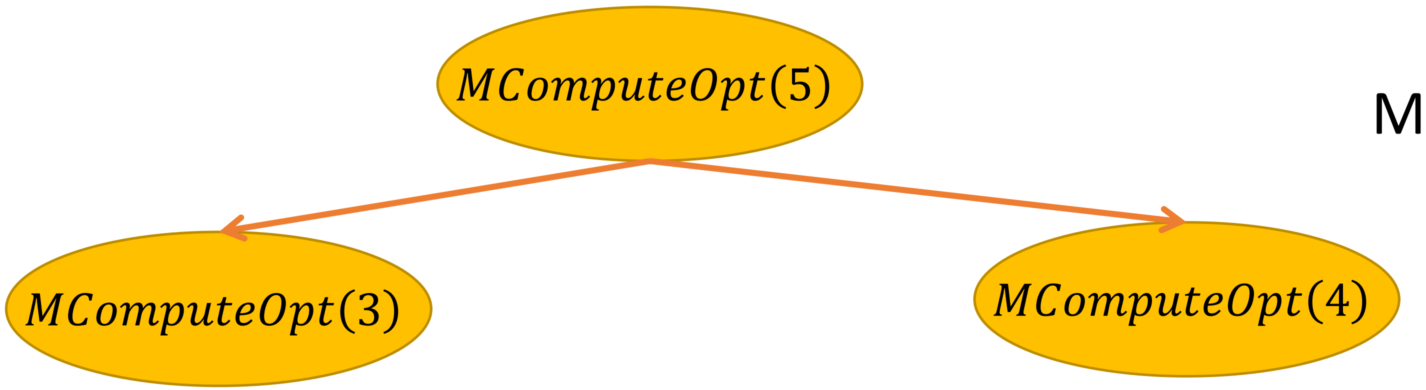
p(1)=0, p(2)=0, p(3)=1, p(4)=2, p(5)=3

$$M[5] = \max(r_5 + MComputeOpt\big(p(5)\big), MComputeOpt(5-1))$$

$$M[5] = \max(10 + MComputeOpt(3), MComputeOpt(4))$$

Idea behind the Dynamic programming

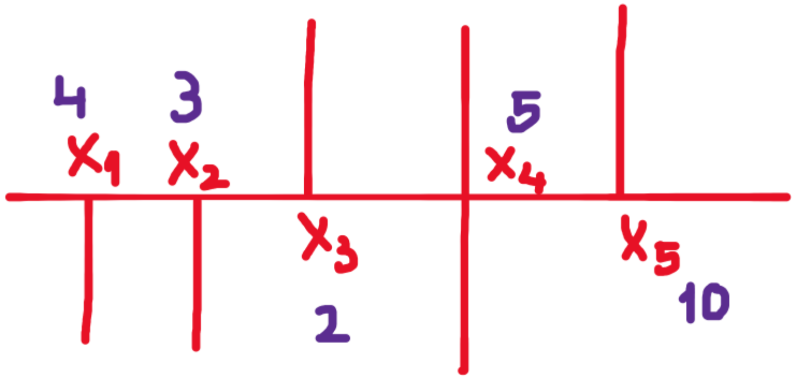
- Divide the the problem into subproblems
- Solve the subproblems and **store** the answers
- The subproblems are **dependent on each other**
- The subproblem may occur many times (repeats). No need to calculate/solve them again. <u>Use the stored answers when needed</u>!



	0	1	2	3	4	5
M:	**0**					

M-Compute-Opt(j)
 If $j = 0$ then
  Return 0
 Else if $M[j]$ is not empty then
  Return $M[j]$
 Else
  Define $M[j] = \max(w_j +$ M-Compute-Opt($p(j)$), M-Compute-Opt($j - 1$))
  Return $M[j]$
 Endif

Run M-Compute-Opt(3)



M-Compute-Opt(j)

If $j = 0$ then

Return 0

Else if $M[j]$ is not empty then

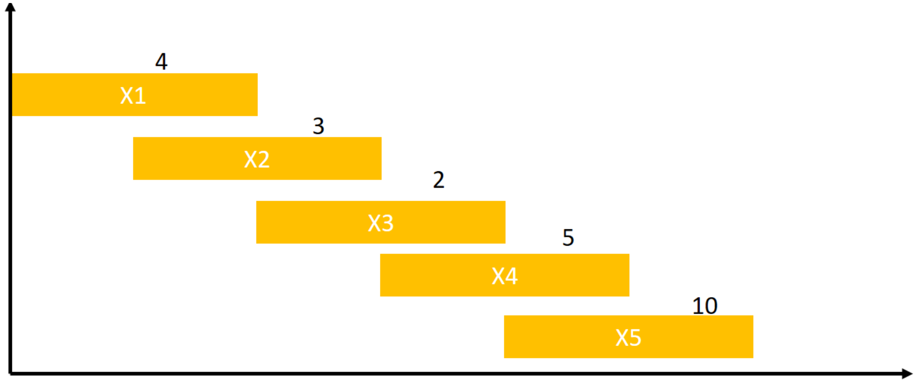
Return $M[j]$

Else

Define $M[j] = \max(w_j +$ M-Compute-Opt($p(j)$), M-Compute-Opt($j - 1$))

Return $M[j]$

Endif

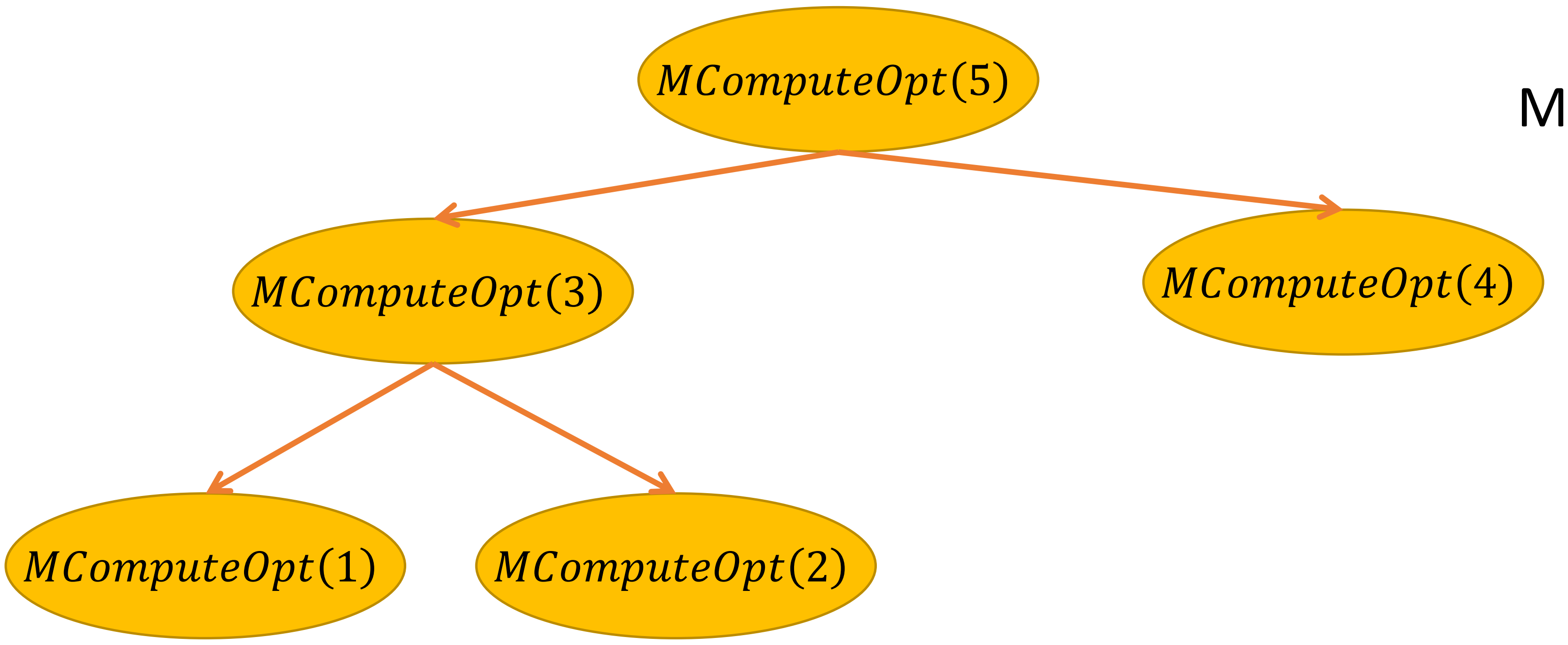


p(1)=0, p(2)=0, p(3)=1, p(4)=2, p(5)=3

	0	1	2	3	4	5
M:	0					

$$M[3] = \max(r_3 + MComputeOpt(p(3)), MComputeOpt(3-1))$$

$$M[3] = \max(2 + MComputeOpt(1), MComputeOpt(2))$$



M:

0	1	2	3	4	5
0					

M-Compute-Opt(j)

If $j = 0$ then

Return 0

Else if $M[j]$ is not empty then

Return $M[j]$

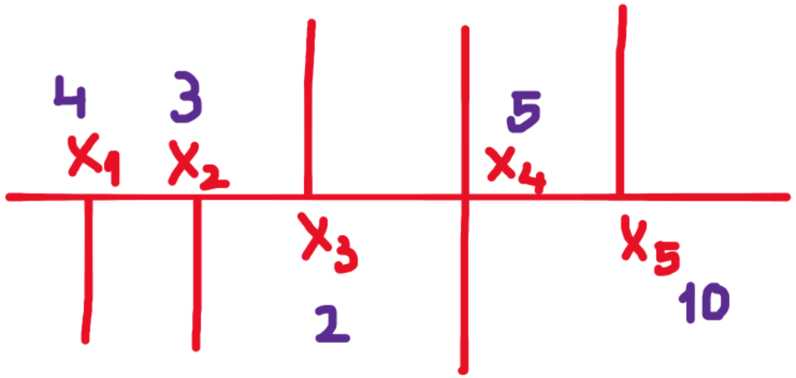
Else

Define $M[j] = \max(w_j +$ M-Compute-Opt($p(j)$), M-Compute-Opt($j - 1$))

Return $M[j]$

Endif

Run M-Compute-Opt(1)



M-Compute-Opt(j)

If $j = 0$ then

Return 0

Else if $M[j]$ is not empty then

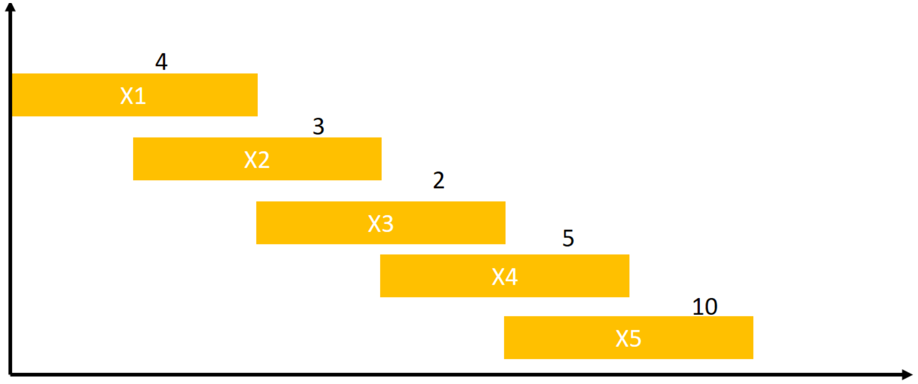
Return $M[j]$

Else

Define $M[j] = \max(w_j +$ M-Compute-Opt($p(j)$), M-Compute-Opt($j - 1$))

Return $M[j]$

Endif



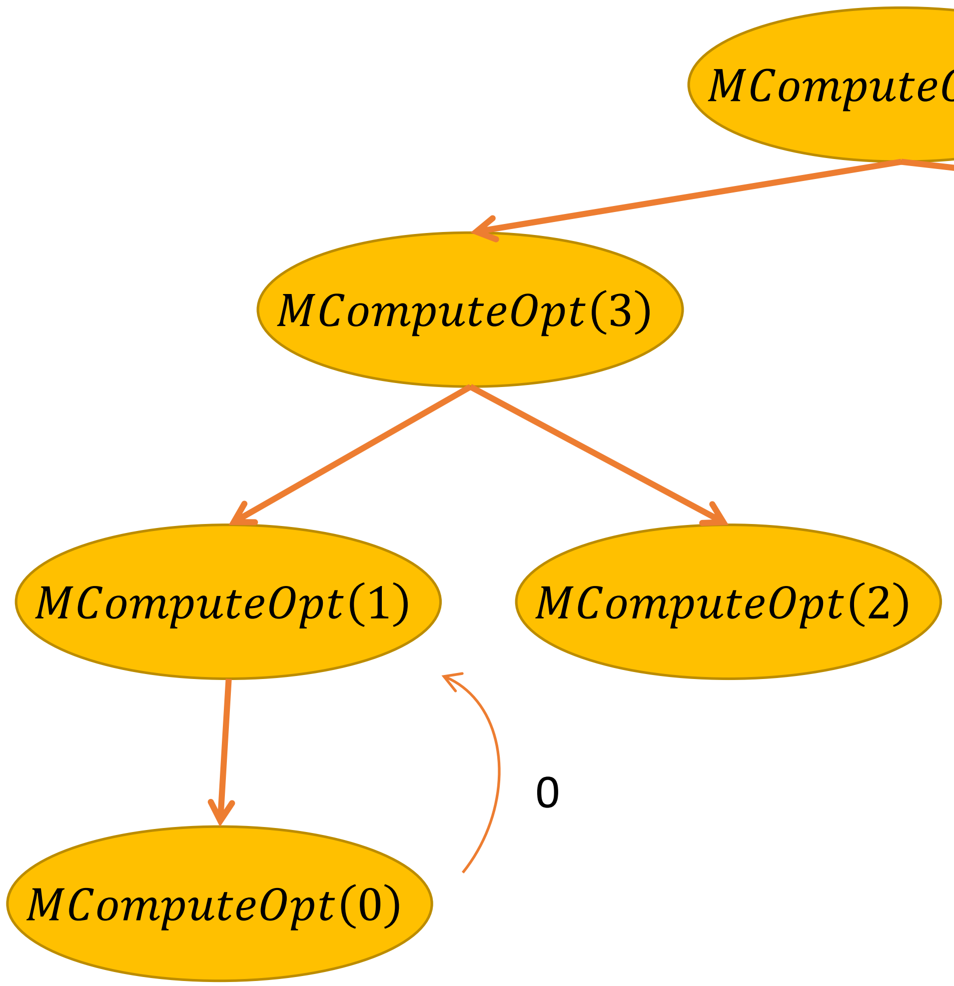
p(1)=0, p(2)=0, p(3)=1, p(4)=2, p(5)=3

M:

0	1	2	3	4	5
0					

$$M[1] = \max(r_1 + MComputeOpt\big(p(1)\big), MComputeOpt(1-1))$$

$$M[1] = \max(4 + MComputeOpt(0), MComputeOpt(0))$$



M:

0	1	2	3	4	5
0					

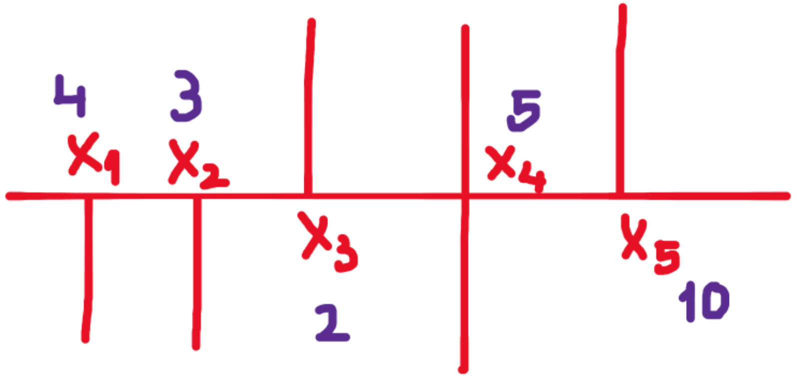
M-Compute-Opt(j)

$\boxed{\text{If } j = 0 \text{ then} \quad \text{Return } 0}$

$MComputeOpt(0)$

If $j = 0$ then
  Return 0
Else if $M[j]$ is not empty then
  Return $M[j]$
Else
  Define $M[j] = \max(w_j +$ M-Compute-Opt($p(j)$), M-Compute-Opt($j - 1$))
  Return $M[j]$
Endif

Back to M-Compute-Opt(1)



M-Compute-Opt(j)

If $j = 0$ then

Return 0

Else if $M[j]$ is not empty then

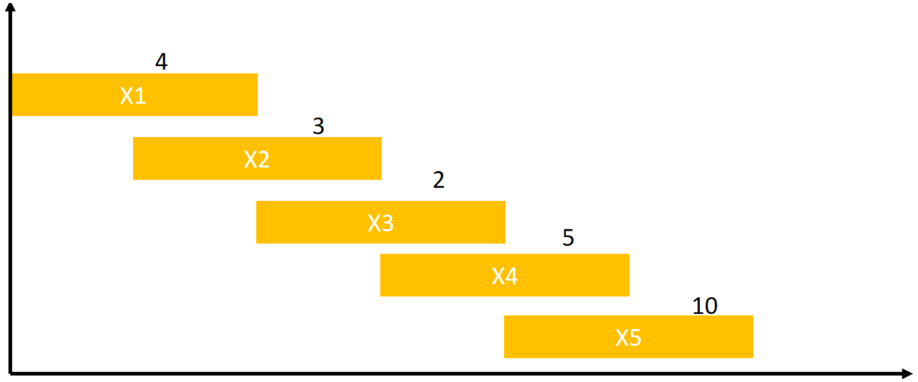
Return $M[j]$

Else

Define $M[j] = \max(w_j +$ M-Compute-Opt($p(j)$), M-Compute-Opt($j - 1$))

Return $M[j]$

Endif



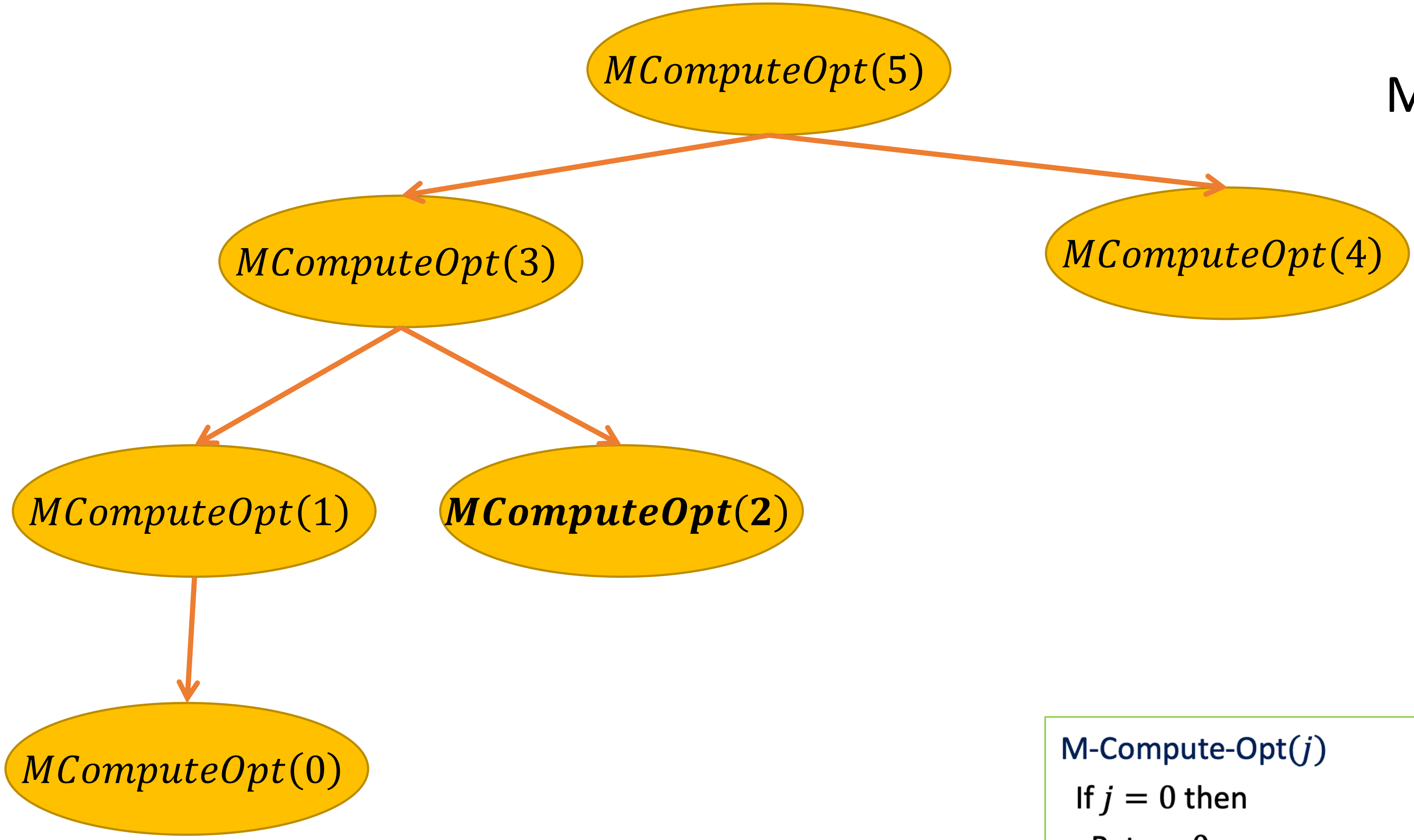
p(1)=0, p(2)=0, p(3)=1, p(4)=2, p(5)=3

	0	1	2	3	4	5
M:	0	4				

$$M[1] = \max(r_1 + MComputeOpt\big(p(1)\big), MComputeOpt(1-1))$$

$$M[1] = \max(4 + MComputeOpt(0), MComputeOpt(0))$$

$$M[1] = \max(4, 0) = 4$$

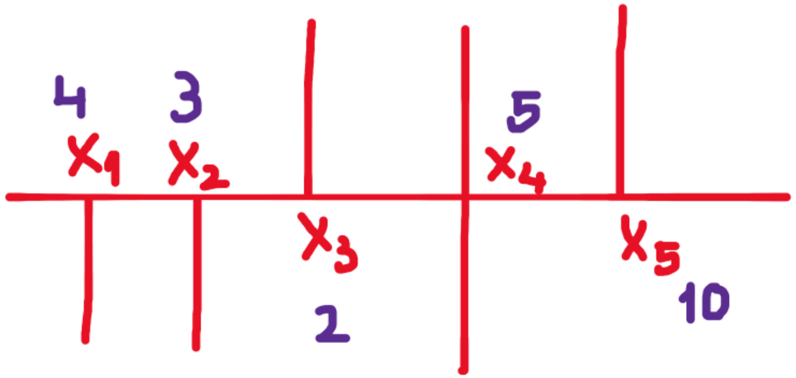


	0	1	2	3	4	5
M:	0	4				

M-Compute-Opt(j)
If $j = 0$ then
Return 0
Else if $M[j]$ is not empty then
Return $M[j]$
Else
Define $M[j] = \max(w_j +$ M-Compute-Opt($p(j)$), M-Compute-Opt($j - 1$))
Return $M[j]$
Endif

$MComputeOpt(2)$

Run M-Compute-Opt(2)



M-Compute-Opt(j)

If $j = 0$ then

  Return 0

Else if $M[j]$ is not empty then

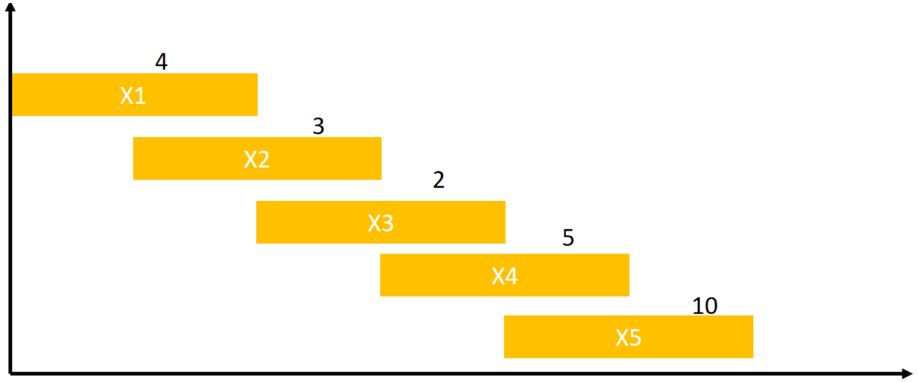
  Return $M[j]$

Else

  Define $M[j] = \max(w_j +$ M-Compute-Opt($p(j)$), M-Compute-Opt($j - 1$))

  Return $M[j]$

Endif

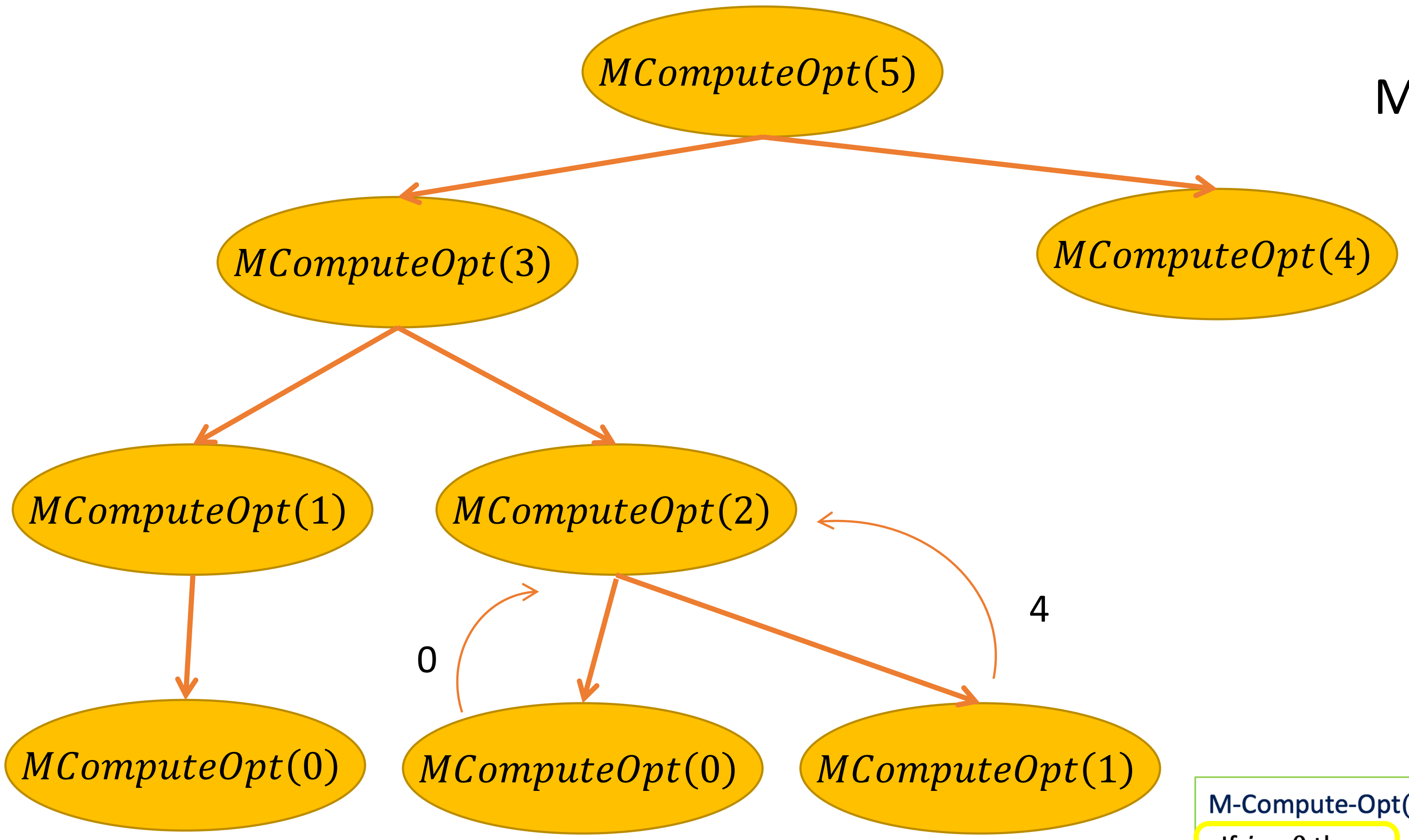


p(1)=0, p(2)=0, p(3)=1, p(4)=2, p(5)=3

	0	1	2	3	4	5
M:	0	4				

$$M[2] = \max(r_2 + MComputeOpt\big(p(2)\big), MComputeOpt(2-1))$$

$$M[2] = \max(3 + MComputeOpt(0), MComputeOpt(1))$$



M:

0	1	2	3	4	5
0	4				

M-Compute-Opt(j)

If $j = 0$ then
  Return 0

$MComputeOpt(0)$

Else if $M[j]$ is not empty then
  Return $M[j]$

$MComputeOpt(1)$

Else
  Define $M[j] = \max(w_j +$ M-Compute-Opt($p(j)$), M-Compute-Opt($j - 1$))
  Return $M[j]$
Endif

Back to M-Compute-Opt(2)

M-Compute-Opt(j)

If $j = 0$ then

  Return 0

Else if $M[j]$ is not empty then

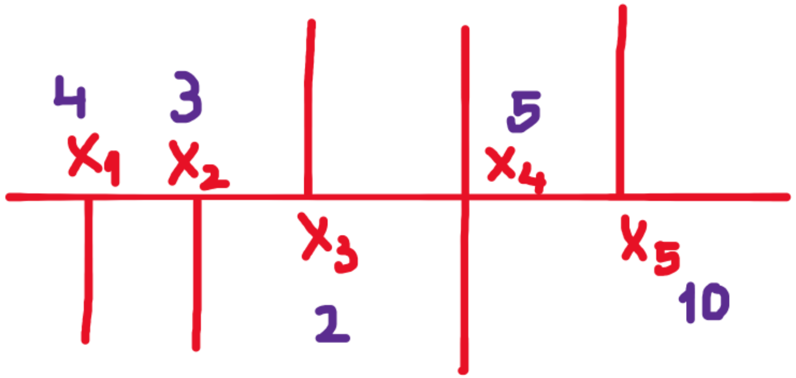
  Return $M[j]$

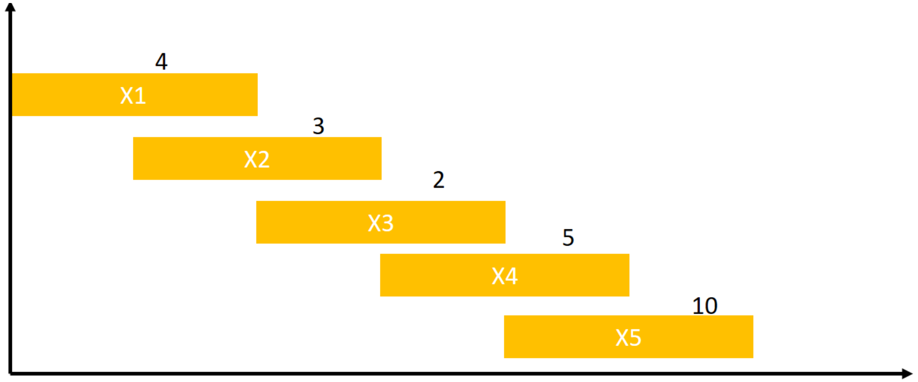
Else

  Define $M[j] = \max(w_j +$ M-Compute-Opt($p(j)$), M-Compute-Opt($j - 1$))

  Return $M[j]$

Endif





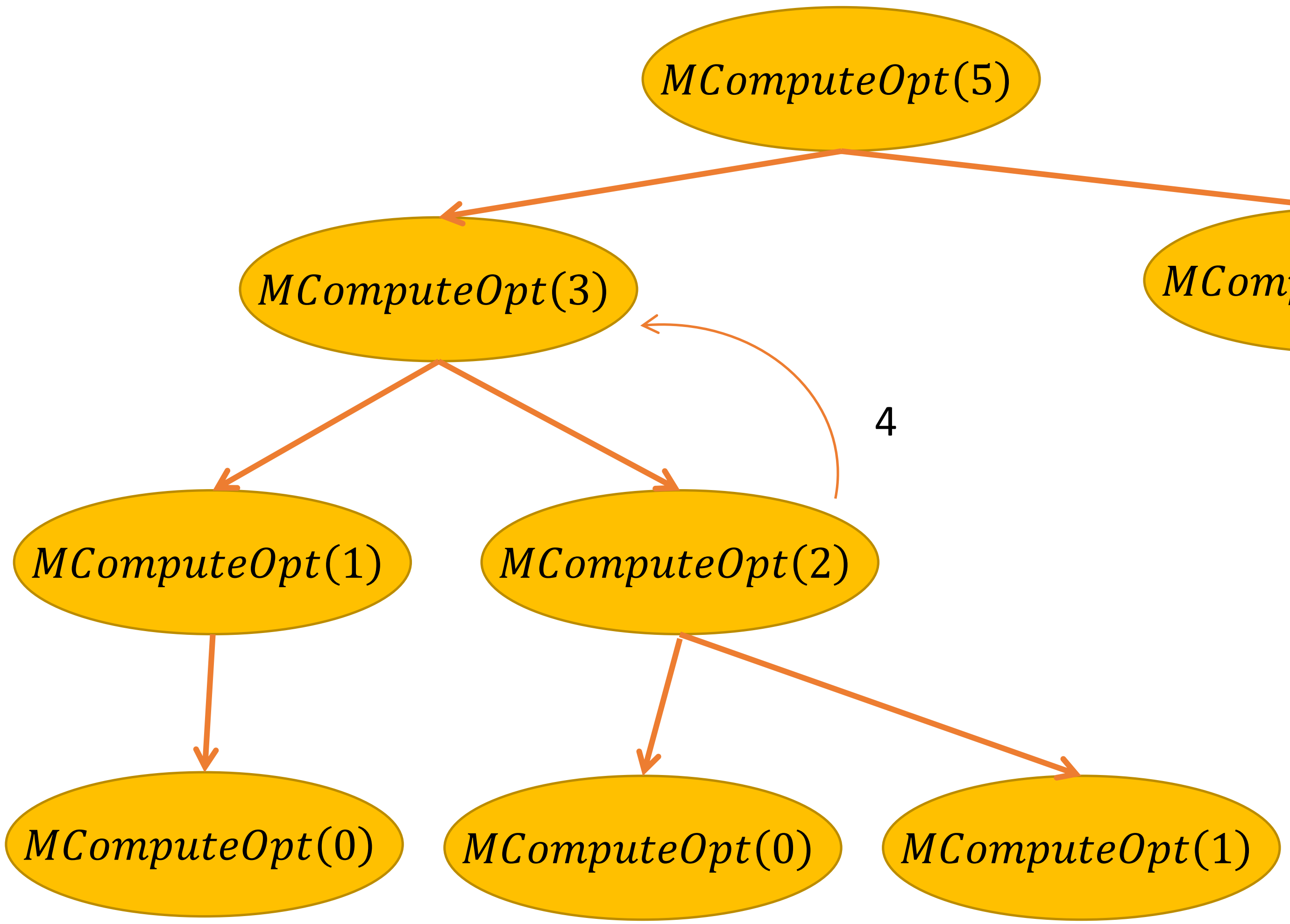
	0	1	2	3	4	5
M:	0	4	4			

p(1)=0, p(2)=0, p(3)=1, p(4)=2, p(5)=3

$$M[2] = \max(r_2 + MComputeOpt\big(p(2)\big), MComputeOpt(2-1))$$

$$M[2] = \max(3 + MComputeOpt(0), MComputeOpt(1))$$

$$M[2] = \max(3, 4) = 4$$



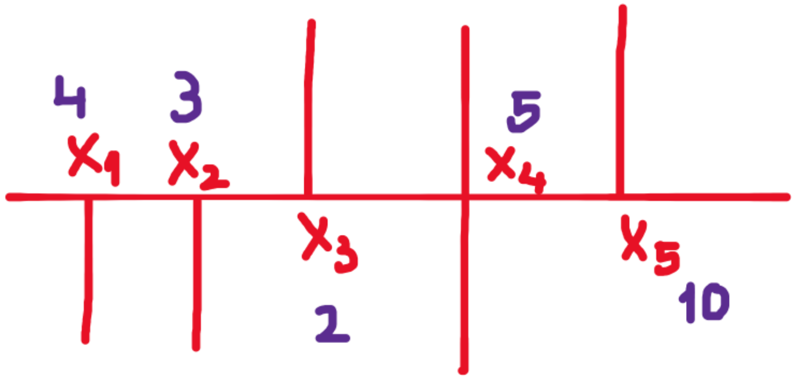
M:

0	1	2	3	4	5
0	4	4			

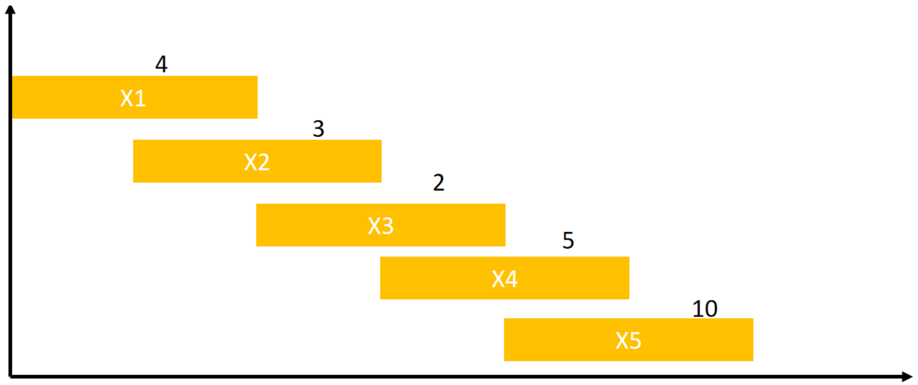
```
M-Compute-Opt(j)
 If j = 0 then
  Return 0
 Else if M[j] is not empty then
  Return M[j]
 Else
  Define M[j] = max(w_j + M-Compute-Opt(p(j)), M-Compute-Opt(j − 1))
  Return M[j]
 Endif
```

$MComputeOpt(2)$

Back to M-Compute-Opt(3)



M-Compute-Opt(j)
If $j = 0$ then
  Return 0
Else if $M[j]$ is not empty then
  Return $M[j]$
Else
  Define $M[j] = \max(w_j +$ M-Compute-Opt($p(j)$), M-Compute-Opt($j - 1$))
  Return $M[j]$
Endif



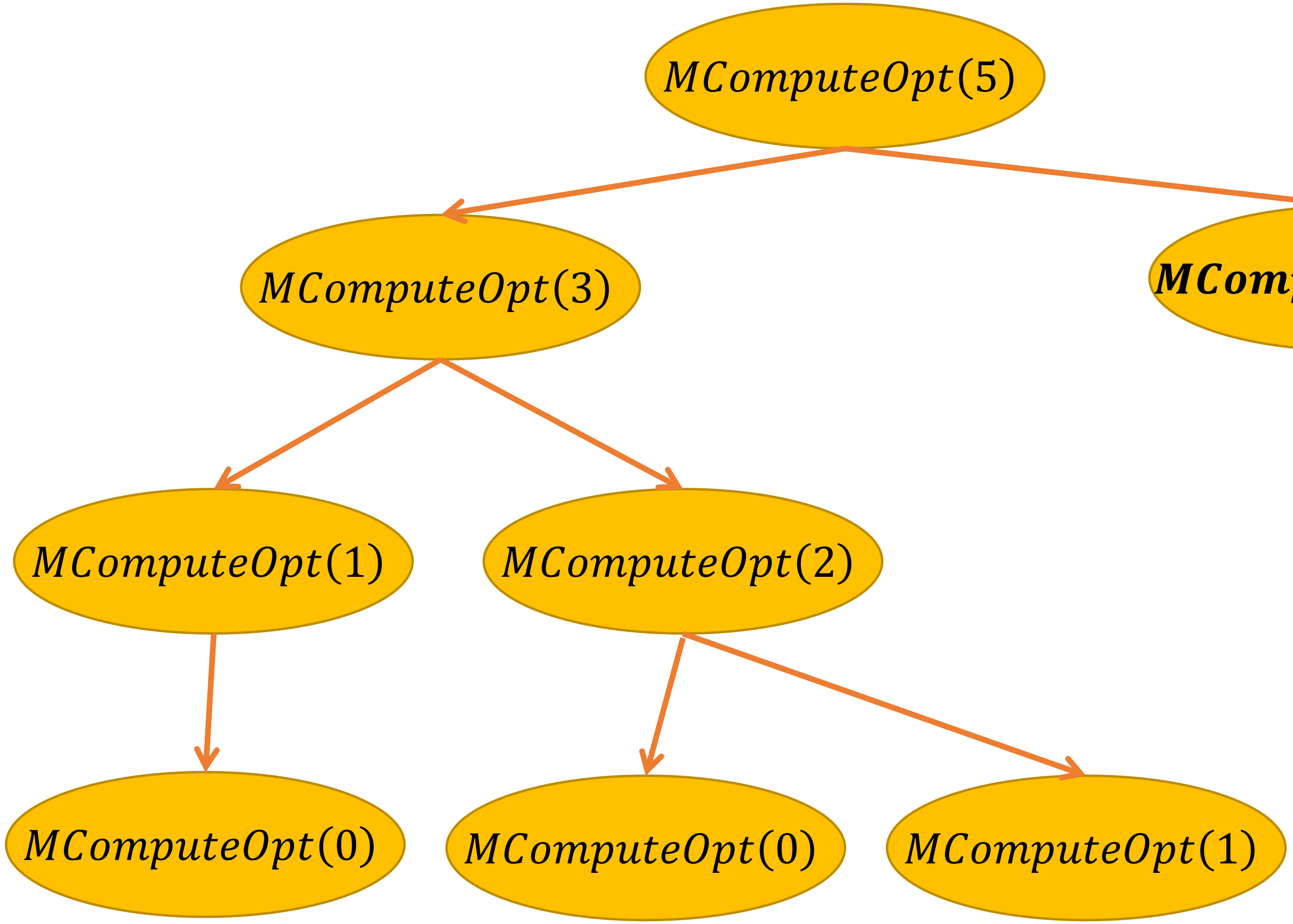
p(1)=0, p(2)=0, p(3)=1, p(4)=2, p(5)=3

	0	1	2	3	4	5
M:	0	4	4	6		

$$M[3] = \max(r_3 + MComputeOpt(p(3)), MComputeOpt(3-1))$$

$$M[3] = \max(2 + MComputeOpt(1), MComputeOpt(2))$$

$$M[3] = \max(2 + 4, 4) = \max(6, 4) = 6$$



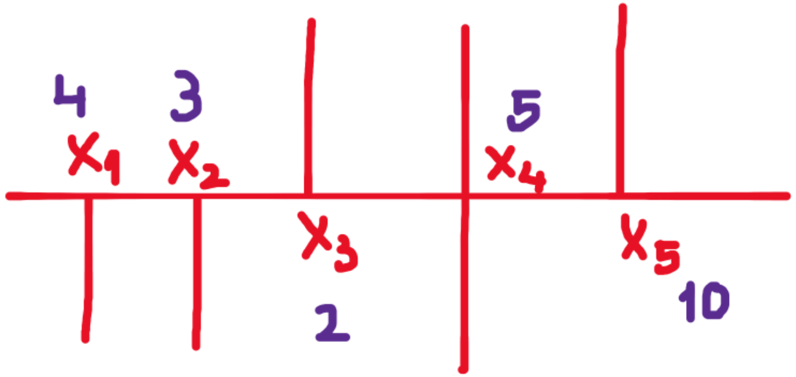
M:

0	1	2	3	4	5
0	4	4	6		

```
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(w_j + M-Compute-Opt(p(j)), M-Compute-Opt(j − 1))
  Return M[j]
 Endif
```

$MComputeOpt(4)$

Run M-Compute-Opt(4)



M-Compute-Opt(j)

If $j = 0$ then

  Return 0

Else if $M[j]$ is not empty then

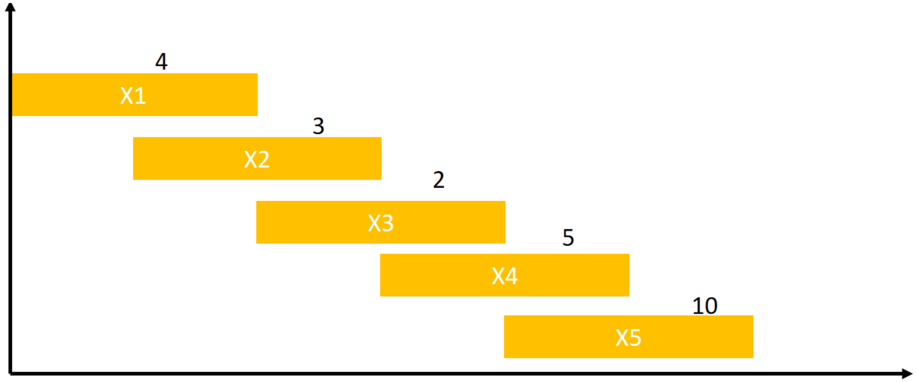
  Return $M[j]$

Else

  Define $M[j] = \max(w_j +$ M-Compute-Opt($p(j)$), M-Compute-Opt($j - 1$))

  Return $M[j]$

Endif



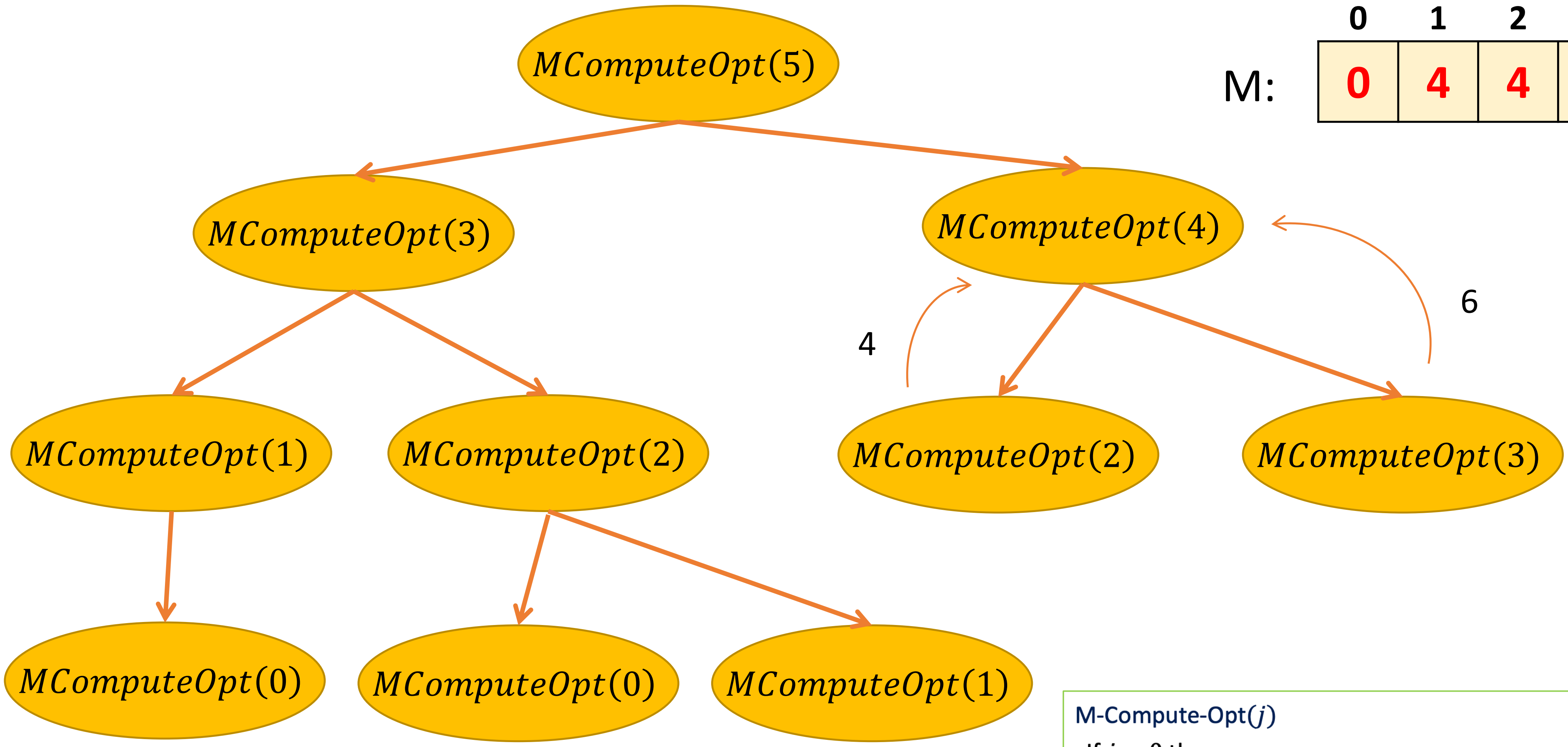
M:

0	1	2	3	4	5
0	4	4	6		

p(1)=0, p(2)=0, p(3)=1, p(4)=2, p(5)=3

$$M[4] = \max\left(r_4 + MComputeOpt\big(p(4)\big), MComputeOpt(4-1)\right)$$

$$M[4] = \max(5 + MComputeOpt(2), MComputeOpt(3))$$



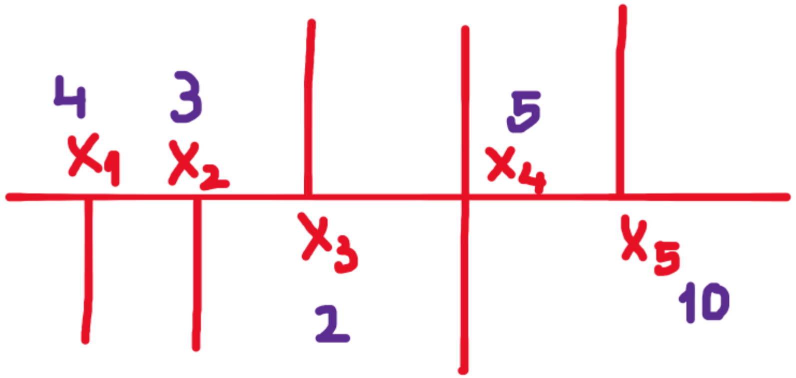
M:

0	1	2	3	4	5
0	4	4	6		

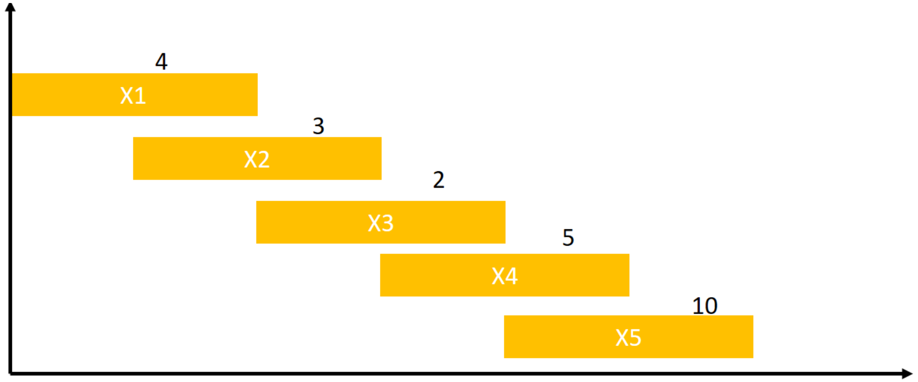
M-Compute-Opt(j)
  If $j = 0$ then
    Return 0
  Else if $M[j]$ is not empty then
    Return $M[j]$
  Else
    Define $M[j] = \max(w_j +$ M-Compute-Opt($p(j)$), M-Compute-Opt($j - 1$))
    Return $M[j]$
  Endif

$MComputeOpt(2)$ &
$MComputeOpt(3)$

Back to M-Compute-Opt(4)



M-Compute-Opt(j)
- If $j = 0$ then
  - Return 0
- Else if $M[j]$ is not empty then
  - Return $M[j]$
- Else
  - Define $M[j] = \max(w_j +$ M-Compute-Opt($p(j)$), M-Compute-Opt($j - 1$))
  - Return $M[j]$
- Endif



p(1)=0, p(2)=0, p(3)=1, p(4)=2, p(5)=3

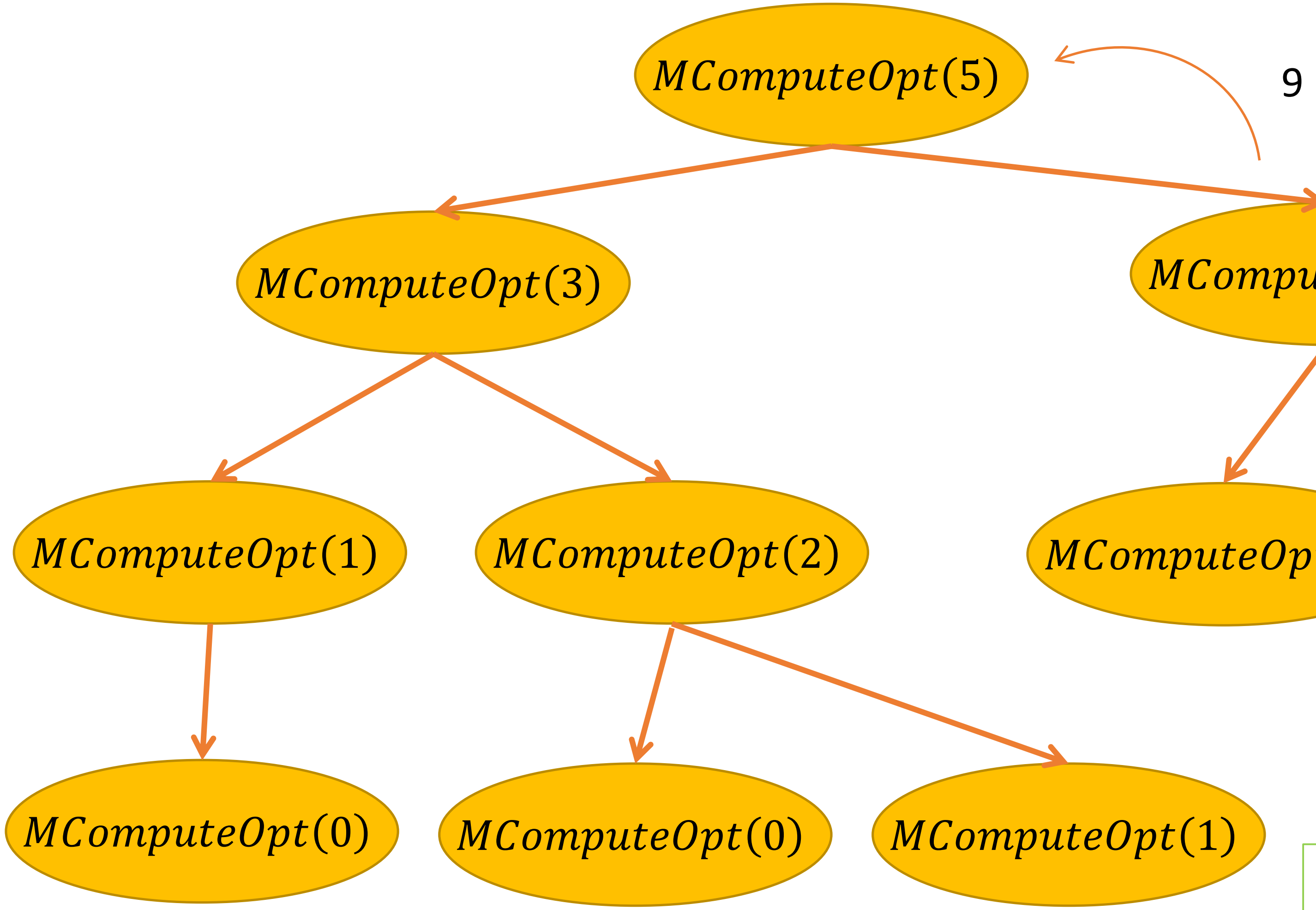
M:

0	1	2	3	4	5
0	4	4	6	9	

$$M[4] = \max(r_4 + MComputeOpt\big(p(4)\big), MComputeOpt(4-1))$$

$$M[4] = \max(5 + MComputeOpt(2), MComputeOpt(3))$$

$$M[4] = \max(5 + 4, 6) = \max(9, 6) = 9$$



M:

0	1	2	3	4	5
0	4	4	6	9	

M-Compute-Opt(j)

If $j = 0$ then

Return 0

Else if $M[j]$ is not empty then

Return $M[j]$

Else

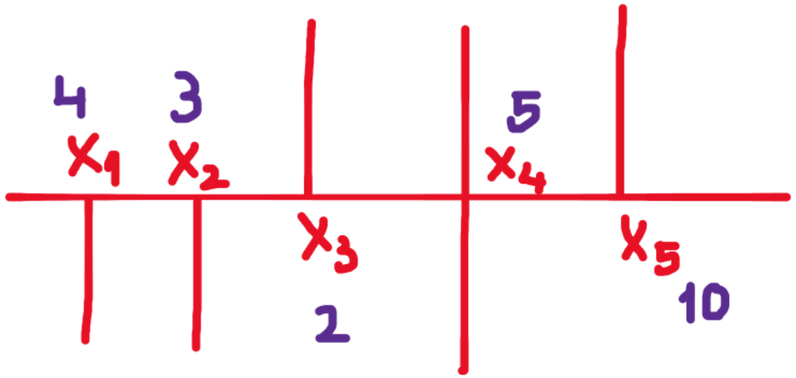
Define $M[j] = \max(w_j +$ M-Compute-Opt($p(j)$), M-Compute-Opt($j - 1$))

Return $M[j]$

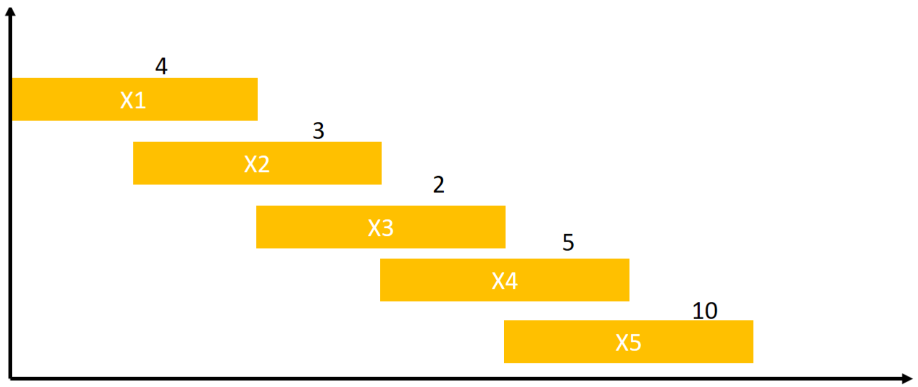
Endif

$MComputeOpt(4)$

Back to M-Compute-Opt(5)



M-Compute-Opt(j)
 If $j = 0$ then
  Return 0
 Else if $M[j]$ is not empty then
  Return $M[j]$
 Else
  Define $M[j] = \max(w_j +$ M-Compute-Opt($p(j)$), M-Compute-Opt($j - 1$))
  Return $M[j]$
 Endif



M:

0	1	2	3	4	5
0	4	4	6	9	16

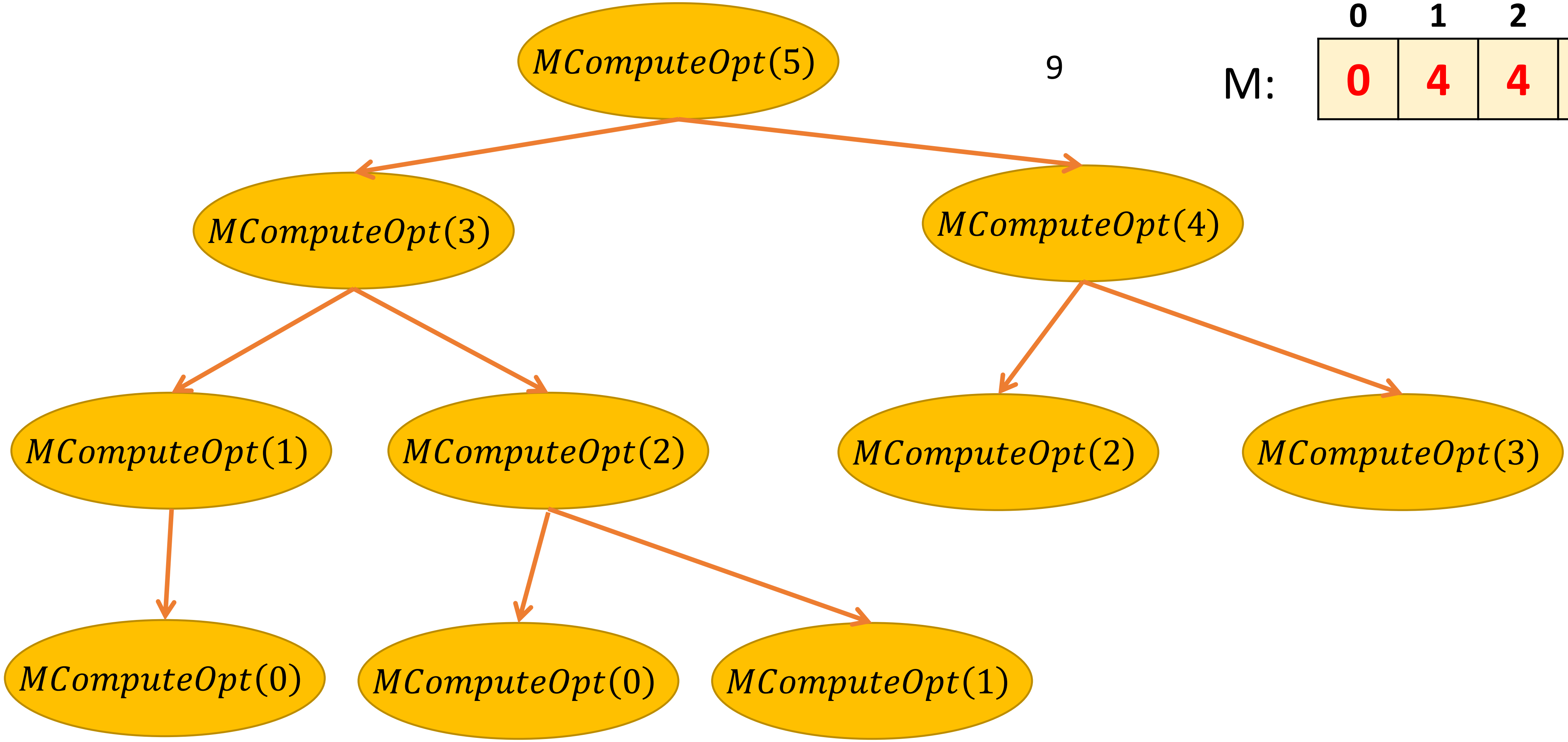
p(1)=0, p(2)=0, p(3)=1, p(4)=2, p(5)=3

$$M[5] = \max(r_5 + MComputeOpt(p(5)), MComputeOpt(5-1))$$

$$M[5] = \max(10 + MComputeOpt(3), MComputeOpt(4))$$

$$M[5] = \max(10 + 6, 9) = \max(16, 6) = 16$$

Total people reached per minute is 16.

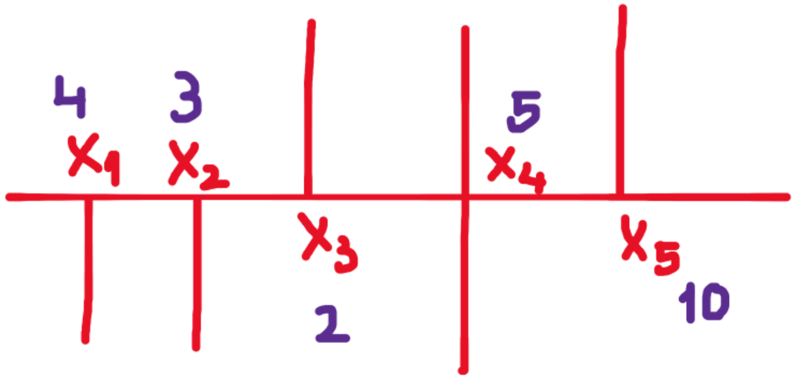


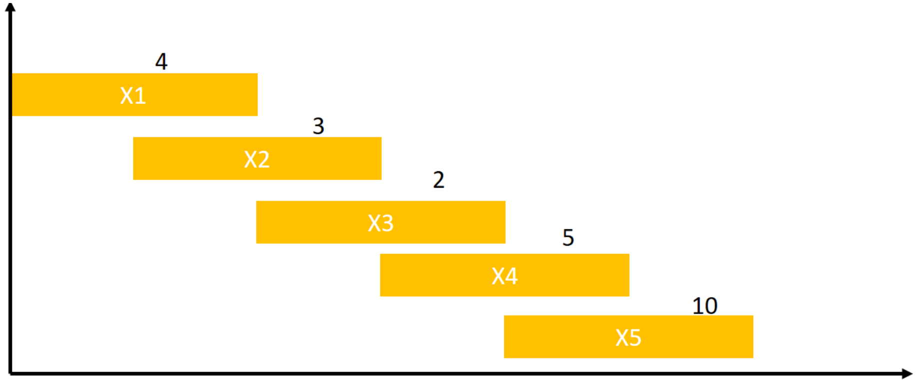
9

	0	1	2	3	4	5
M:	0	4	4	6	9	16

Run Find-Solution(5)

Find-Solution(j)
 If $j = 0$ then
  Output nothing
 Else
  If $w_j + M[p(j)] \geq M[j-1]$ then
   Output j together with the result of Find-Solution($p(j)$)
  Else
   Output the result of Find-Solution($j - 1$)
  Endif
 Endif





M:

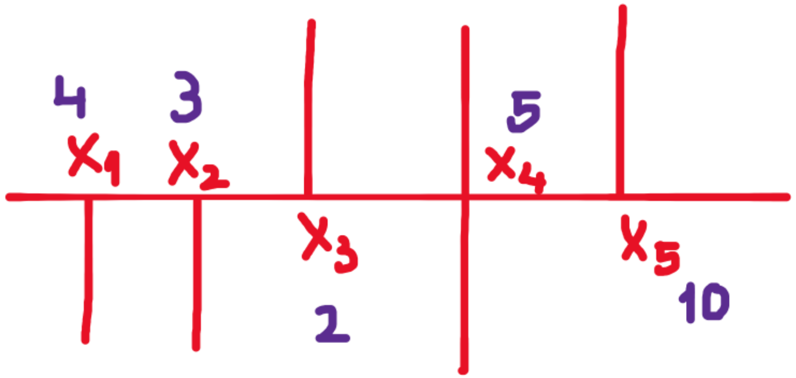
0	1	2	3	4	5
0	4	4	6	9	16

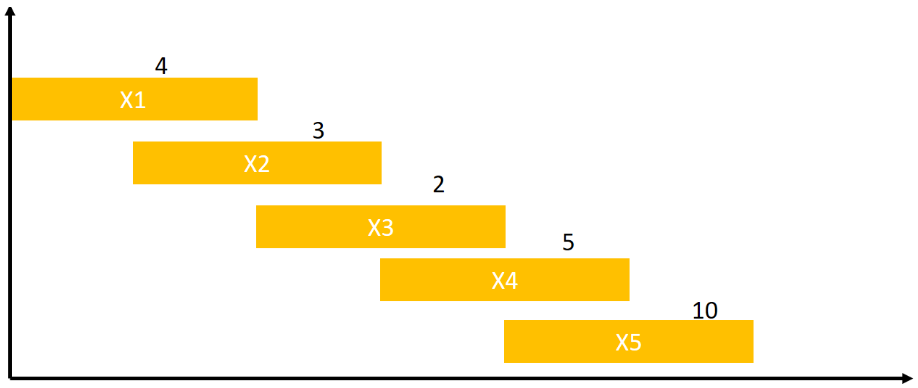
p(1)=0, p(2)=0, p(3)=1, p(4)=2, p(5)=3

$$10 + M[p(5)] \geq M[5-1]$$
$$10 + M[3] \geq M[4] \quad ? \quad \text{YES} \quad Print\text{: ``}5\text{''} + FindSolution(p(j))$$
$$10 + 6 \geq 9$$

Run Find-Solution(3)

Find-Solution(j)
  If $j = 0$ then
    Output nothing
  Else
    If $w_j + M[p(j)] \geq M[j-1]$ then
      Output j together with the result of Find-Solution($p(j)$)
    Else
      Output the result of Find-Solution($j - 1$)
    Endif
  Endif





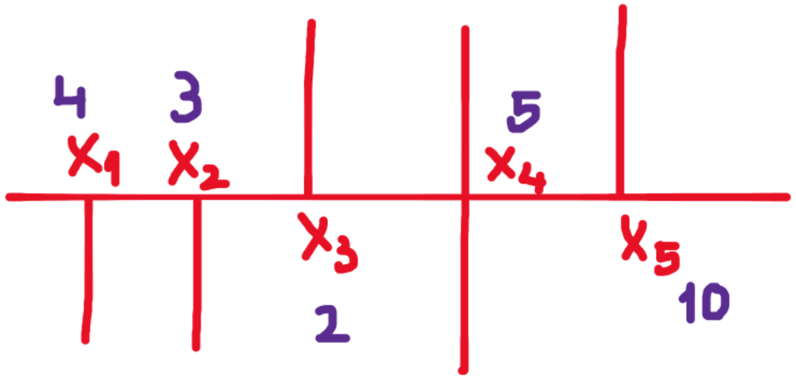
p(1)=0, p(2)=0, p(3)=1, p(4)=2, p(5)=3

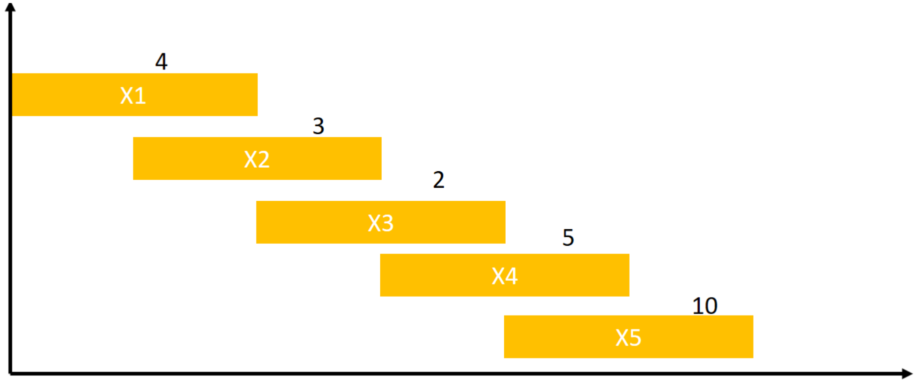
	0	1	2	3	4	5
M:	0	4	4	6	9	16

$$2 + M[p(3)] \geq M[3-1]$$
$$2 + M[1] \geq M[2] \quad ? \quad \text{YES} \quad Print: ``5'' + ``3'' + FindSolution(p(j))$$
$$2 + 4 \geq 4$$

Run Find-Solution(1)

Find-Solution(j)
If $j = 0$ then
Output nothing
Else
If $w_j + M[p(j)] \geq M[j-1]$ then
Output j together with the result of Find-Solution($p(j)$)
Else
Output the result of Find-Solution($j - 1$)
Endif
Endif





p(1)=0, p(2)=0, p(3)=1, p(4)=2, p(5)=3

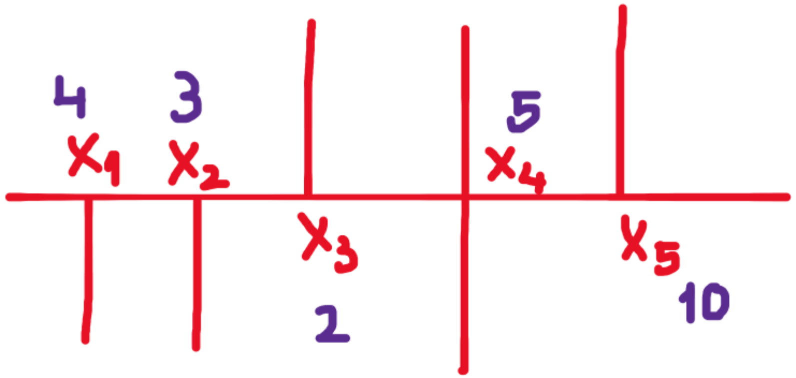
	0	1	2	3	4	5
M:	0	4	4	6	9	16

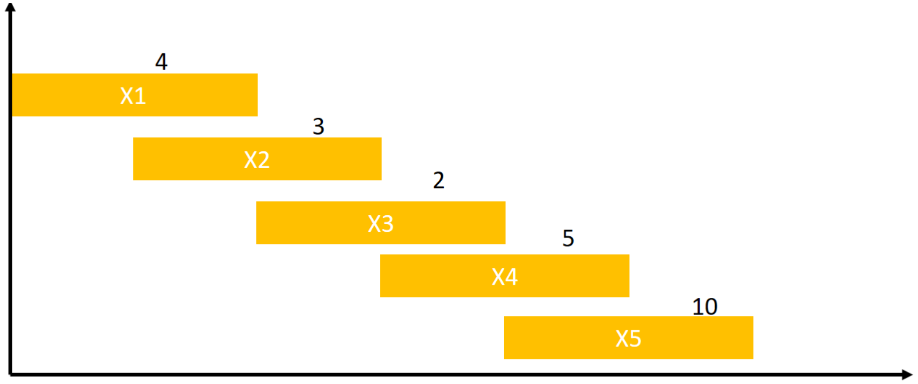
$$4 + M[p(1)] \geq M[1-1] \quad ? \text{ YES}$$
$$4 + M[0] \geq M[0]$$
$$4 + 0 \geq 0$$

$$Print: \text{``}5'' + \text{``}3'' + \text{``}1'' + FindSolution(p(0))$$

Run Find-Solution(0)

Find-Solution(j)
 If $j = 0$ then
  Output nothing
 Else
  If $w_j + M[p(j)] \geq M[j-1]$ then
   Output j together with the result of Find-Solution($p(j)$)
  Else
   Output the result of Find-Solution($j - 1$)
  Endif
 Endif





	0	1	2	3	4	5
M:	0	4	4	6	9	16

$p(1)=0,\ p(2)=0,\ p(3)=1,\ p(4)=2,\ p(5)=3$

$$Print: ``5" + ``3" + ``1" + Nothing$$

Solution: Intersections 1, 3 and 5 are selected.

Total people reached per minute is 16.

Weighted interval scheduling: bottom-up

- Question: Can this memoization be implemented without recursion?
- Yes: Bottom-up dynamic programming

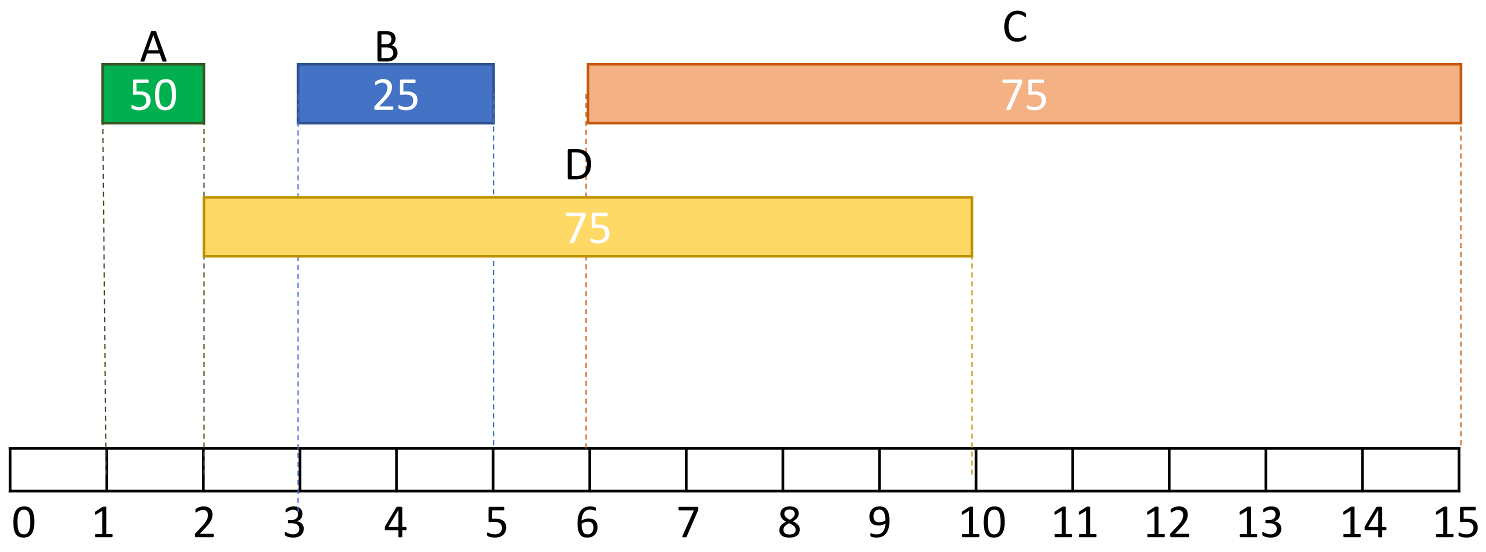
```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(v_j + M[p(j)], M[j-1])
}
```

Picture is taken from course slides

Let’s See Some Code

You have the following jobs with specific start-end times and the profit you would get.

What is the maximum profit you can get?



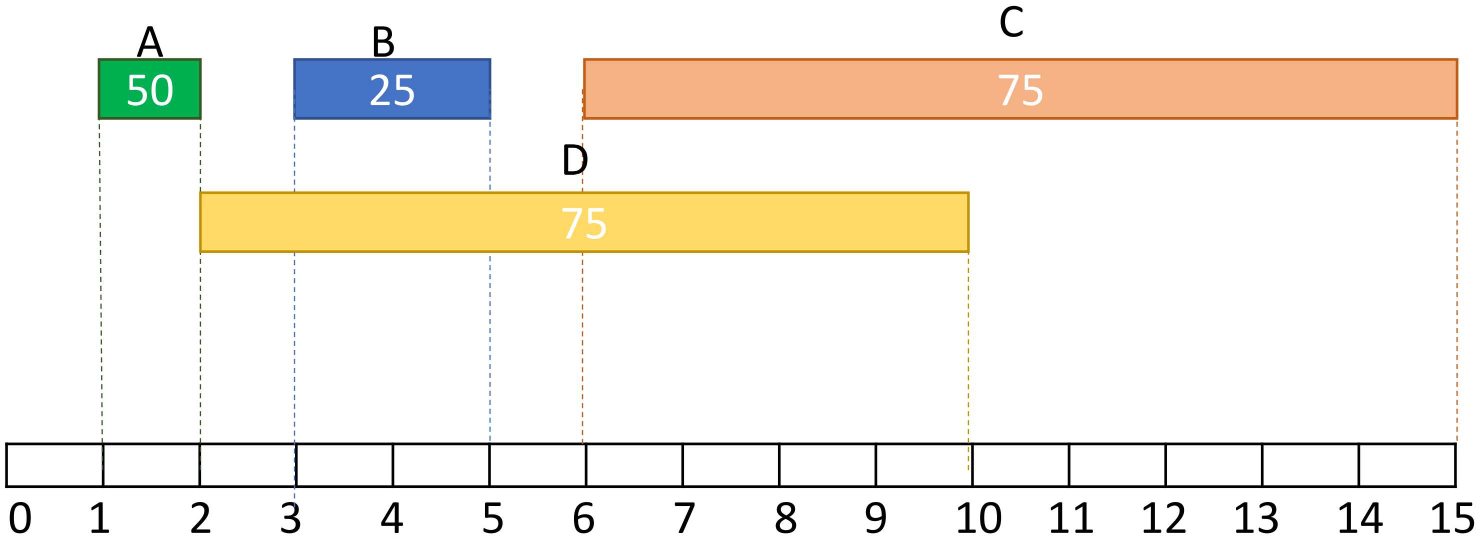
The code is adapted from the following online sources:
https://www.geeksforgeeks.org/weighted-job-scheduling/
https://www.techiedelight.com/weighted-interval-scheduling-problem/

```cpp
#include <iostream>
// We need this library to sort the jobs by the end time
#include <algorithm>
using namespace std;
```

```cpp
// We need to use a struct to easily sort the jobs by end time
struct Job {
    int start, end, profit;
};
```

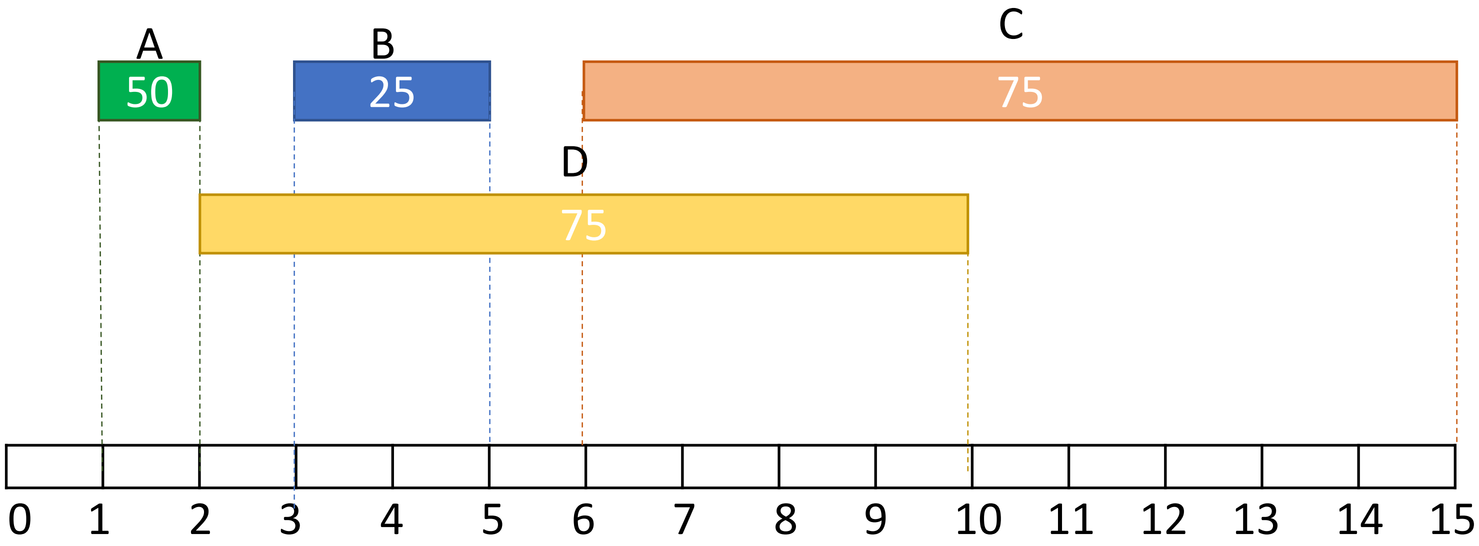
We need these to easily sort the jobs by their end time

```cpp
int main() {
	Job jobList[] = {
	{1, 2, 50},  // A
	{3, 5, 25},  // B
	{6, 15, 75}, // C
	{2, 10, 75}  // D
	};

  // Total number of jobs
	int n = sizeof(jobList)/sizeof(jobList[0]);
	cout << "\nThe maximum profit: " << findMaxProfit(jobList, n);
	return 0;
}
```



```cpp
int main() {
	Job jobList[] = {
	{1, 2, 50},  // A
	{3, 5, 25},  // B
	{6, 15, 75}, // C
	{2, 10, 75}  // D
	};

  // Total number of jobs
	int n = sizeof(jobList)/sizeof(jobList[0]);
	cout << "\nThe maximum profit: " << findMaxProfit(jobList, n);
	return 0;
}
```

This is where the magic happens



jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
C: [6, 15, 75]
D: [2, 10, 75]

```
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                 //clear table from memory
    return result;
}
```

"comp" is a specific function that we need to code.

We will sort by the end times of the jobs.

Beginning of the jobs list

End of the jobs list

```cpp
// Used as a sorting condition. Jobs will be sorted by their end time
bool comp(Job job1, Job job2) {
    return (job1.end < job2.end);
}
```

This function compares to jobs and returns if the first job finished before the second.

This Boolean information provides a specific condition for a comparison.

“Algorithm” library made this happen.

```
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                 //clear table from memory
    return result;
}
```

Before sorting:

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
C: [6, 15, 75]
D: [2, 10, 75]

After sorting:

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

```cpp
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                //clear table from memory
    return result;
}
```

This is the memory table.

n = 4

table:


```cpp
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50			

```cpp
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                  //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

At this point
we have A

table:

50			

```
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                    //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50			

i = 1

```
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50			

i = 1

addProfit = jobList[1].profit = 25

```
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;             //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50			

i = 1

addProfit = 25

```
// This is used to check if two jobs conflict.
// If one job's end time is later than other job's start time, they have a conflict!
int nonConflictJob(Job jobList[], int i) {
    // Among all jobs:
        for (int j=i-1; j>=0; j--) {
        // If there is a job that does not have a conflict:
        if (jobList[j].end <= jobList[i].start)
            // return the job number.
            return j;
        }
        // If there is no job that does not conflict (if all jobs cause a conflict)
        return -1;
}
```

j = 1 – 1 = 0

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

i = 1

```
// This is used to check if two jobs conflict.
// If one job's end time is later than other job's start time, they have a conflict!
int nonConflictJob(Job jobList[], int i) {
    // Among all jobs:
        for (int j=i-1; j>=0; j--) {
        // If there is a job that does not have a conflict:
        if (jobList[j].end <= jobList[i].start)
            // return the job number.
            return j;
        }
        // If there is no job that does not conflict (if all jobs cause a conflict)
        return -1;
}
```

jobList[0].end = 2
jobList[1].start = 3

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

i = 1

```
// This is used to check if two jobs conflict.
// If one job's end time is later than other job's start time, they have a conflict!
int nonConflictJob(Job jobList[], int i) {
    // Among all jobs:
        for (int j=i-1; j>=0; j--) {
        // If there is a job that does not have a conflict:
        if (jobList[j].end <= jobList[i].start)
            // return the job number.
            return j;
        }
        // If there is no job that does not conflict (if all jobs cause a conflict)
        return -1;
}
```

return j = 0.

This means job 1 and job 0 don't cause a conflict.

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

i = 1

```
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                    //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50			

i = 1

addProfit = 25

newJob = 0

```cpp
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                 //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50			

i = 1

addProfit = 25

newJob = 0

```cpp
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50			

i = 1

addProfit = 25

newJob = 0

addProfit = 25 + table[0] = 75

```
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                    //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50			

i = 1

addProfit = 25

newJob = 0

addProfit = 25 + table[0] = 75

addProfit = 75
table[0] = 50
max = 75

```cpp
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;              //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50	75		

i = 1

addProfit = 25

newJob = 0

addProfit = 25 + table[0] = 75

addProfit = 75
table[0] = 50
max = 75

```cpp
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

At this point we have A & B

table:

50	75		

i = 1

addProfit = 25

newJob = 0

addProfit = 25 + table[0] = 75

addProfit = 75
table[0] = 50
max = 75

```
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
       sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
       // Find profit of the current job
       int addProfit = jobList[i].profit;

       // Find the next non-conflicting job
       int newJob = nonConflictJob(jobList, i);

       // If there is a non-conflicting job:
       if (newJob != -1)
          addProfit += table[newJob];

       table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;              //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50	75		

i = 2

```cpp
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50	75		

i = 2

addProfit = jobList[2].profit = 75

```cpp
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;             //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50	75		

i = 2

addProfit = jobList[2].profit = 75

```
// This is used to check if two jobs conflict.
// If one job's end time is later than other job's start time, they have a conflict!
int nonConflictJob(Job jobList[], int i) {
    // Among all jobs:
        for (int j=i-1; j>=0; j--) {
        // If there is a job that does not have a conflict:
        if (jobList[j].end <= jobList[i].start)
            // return the job number.
            return j;
        }
        // If there is no job that does not conflict (if all jobs cause a conflict)
        return -1;
}
```

j = 2 – 1 = 1

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

i = 2

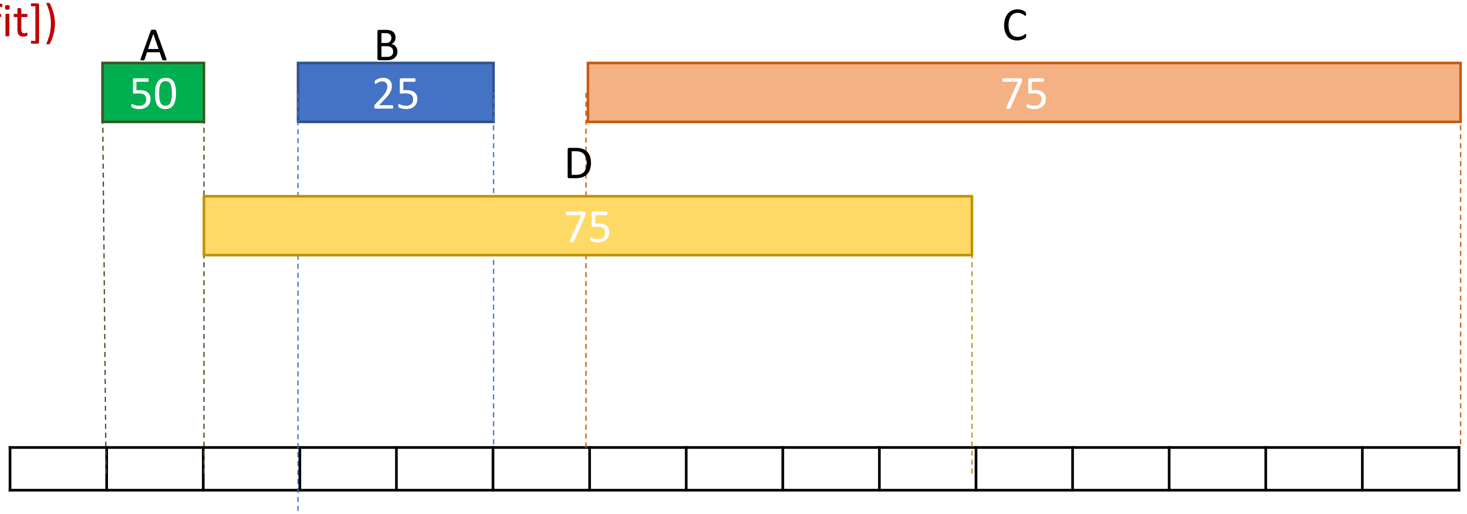
```
// This is used to check if two jobs conflict.
// If one job's end time is later than other job's start time, they have a conflict!
int nonConflictJob(Job jobList[], int i) {
    // Among all jobs:
        for (int j=i-1; j>=0; j--) {
        // If there is a job that does not have a conflict:
        if (jobList[j].end <= jobList[i].start)
            // return the job number.
            return j;
        }
        // If there is no job that does not conflict (if all jobs cause a conflict)
        return -1;
}
```

jobList[1].end = 5
jobList[2].start = 2

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

i = 2



```
// This is used to check if two jobs conflict.
// If one job's end time is later than other job's start time, they have a conflict!
int nonConflictJob(Job jobList[], int i) {
    // Among all jobs:
        for (int j=i-1; j>=0; j--) {
        // If there is a job that does not have a conflict:
        if (jobList[j].end <= jobList[i].start)
            // return the job number.
            return j;
        }
        // If there is no job that does not conflict (if all jobs cause a conflict)
        return -1;
}
```

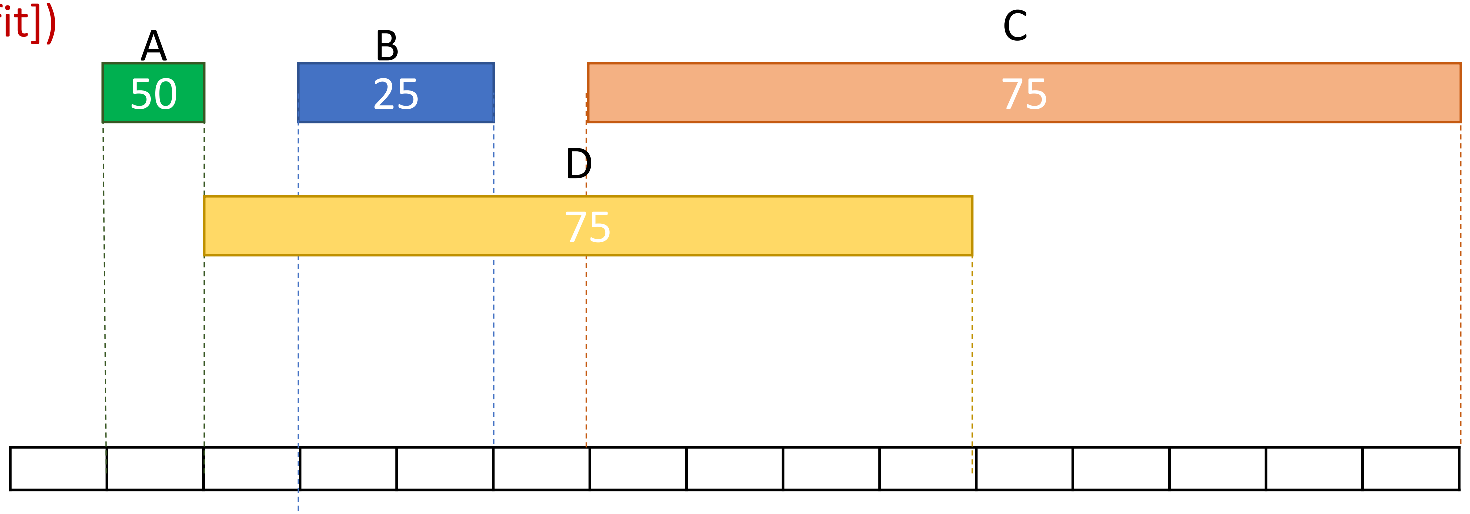
jobList[1].end = 5
jobList[2].start = 2

This means jobB and jobD have a conflict.

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

i = 2



```
// This is used to check if two jobs conflict.
// If one job's end time is later than other job's start time, they have a conflict!
int nonConflictJob(Job jobList[], int i) {
    // Among all jobs:
        for (int j=i-1; j>=0; j--) {
        // If there is a job that does not have a conflict:
        if (jobList[j].end <= jobList[i].start)
            // return the job number.
            return j;
        }
        // If there is no job that does not conflict (if all jobs cause a conflict)
        return -1;
}
```

j - -
j = 0

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

i = 2

```
// This is used to check if two jobs conflict.
// If one job's end time is later than other job's start time, they have a conflict!
int nonConflictJob(Job jobList[], int i) {
    // Among all jobs:
        for (int j=i-1; j>=0; j--) {
        // If there is a job that does not have a conflict:
        if (jobList[j].end <= jobList[i].start)
            // return the job number.
            return j;
        }
        // If there is no job that does not conflict (if all jobs cause a conflict)
        return -1;
}
```

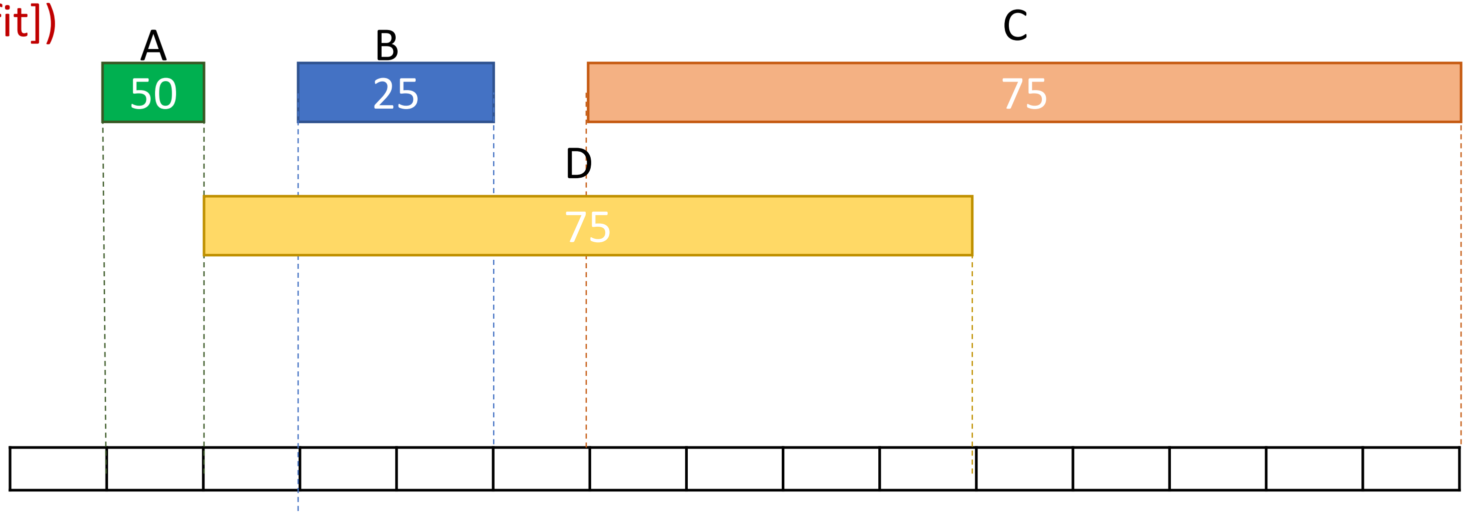
jobList[0].end = 2
jobList[2].start = 2

This means jobA and jobD don't have a conflict!

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

i = 2



```
// This is used to check if two jobs conflict.
// If one job's end time is later than other job's start time, they have a conflict!
int nonConflictJob(Job jobList[], int i) {
    // Among all jobs:
        for (int j=i-1; j>=0; j--) {
        // If there is a job that does not have a conflict:
        if (jobList[j].end <= jobList[i].start)
            // return the job number.
            return j;
        }
        // If there is no job that does not conflict (if all jobs cause a conflict)
        return -1;
}
```

return j = 0
(job 0 does not cause a conflict with job 2)

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

i = 2

```
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                    //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50	75		

i = 2

addProfit = 75

newJob = 0

```
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50	**75**		

i = 2

addProfit = 75

newJob = 0

```
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                 //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50	**75**		

i = 2

addProfit = 75

newJob = 0
addProfit = 75 + table[0] = 125

```cpp
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;              //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50	75	125	

i = 2

addProfit = 75

newJob = 0
addProfit = 75 + table[0] = 125

addProfit = 125
table[1] = 75
max = 125

```
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

At this point
we have A & D

table:

50	75	125	

i = 2

addProfit = 75

newJob = 0
addProfit = 75 + table[0] = 125

addProfit = 125
table[1] = 25
max = 125

```
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                 //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50	75	125	

i = 3

```cpp
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;              //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50	75	125	

i = 3

addProfit = jobList[3] = 75

```
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;               //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50	75	125	

i = 3

addProfit = 75

```
// This is used to check if two jobs conflict.
// If one job's end time is later than other job's start time, they have a conflict!
int nonConflictJob(Job jobList[], int i) {
    // Among all jobs:
        for (int j=i-1; j>=0; j--) {
        // If there is a job that does not have a conflict:
        if (jobList[j].end <= jobList[i].start)
            // return the job number.
            return j;
        }
        // If there is no job that does not conflict (if all jobs cause a conflict)
        return -1;
}
```

j = 3 – 1 = 2

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

i = 3

```
// This is used to check if two jobs conflict.
// If one job's end time is later than other job's start time, they have a conflict!
int nonConflictJob(Job jobList[], int i) {
    // Among all jobs:
        for (int j=i-1; j>=0; j--) {
        // If there is a job that does not have a conflict:
        if (jobList[j].end <= jobList[i].start)
            // return the job number.
            return j;
        }
        // If there is no job that does not conflict (if all jobs cause a conflict)
        return -1;
}
```

jobList[2].end = 10
jobList[3].start = 6

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

i = 3

```
// This is used to check if two jobs conflict.
// If one job's end time is later than other job's start time, they have a conflict!
int nonConflictJob(Job jobList[], int i) {
    // Among all jobs:
        for (int j=i-1; j>=0; j--) {
        // If there is a job that does not have a conflict:
        if (jobList[j].end <= jobList[i].start)
            // return the job number.
            return j;
        }
        // If there is no job that does not conflict (if all jobs cause a conflict)
        return -1;
}
```

j - -

j = 1

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

i = 3

```
// This is used to check if two jobs conflict.
// If one job's end time is later than other job's start time, they have a conflict!
int nonConflictJob(Job jobList[], int i) {
    // Among all jobs:
        for (int j=i-1; j>=0; j--) {
        // If there is a job that does not have a conflict:
        if (jobList[j].end <= jobList[i].start)
            // return the job number.
            return j;
        }
        // If there is no job that does not conflict (if all jobs cause a conflict)
        return -1;
}
```

jobList[1].end = 5
jobList[3].start = 6

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

i = 3

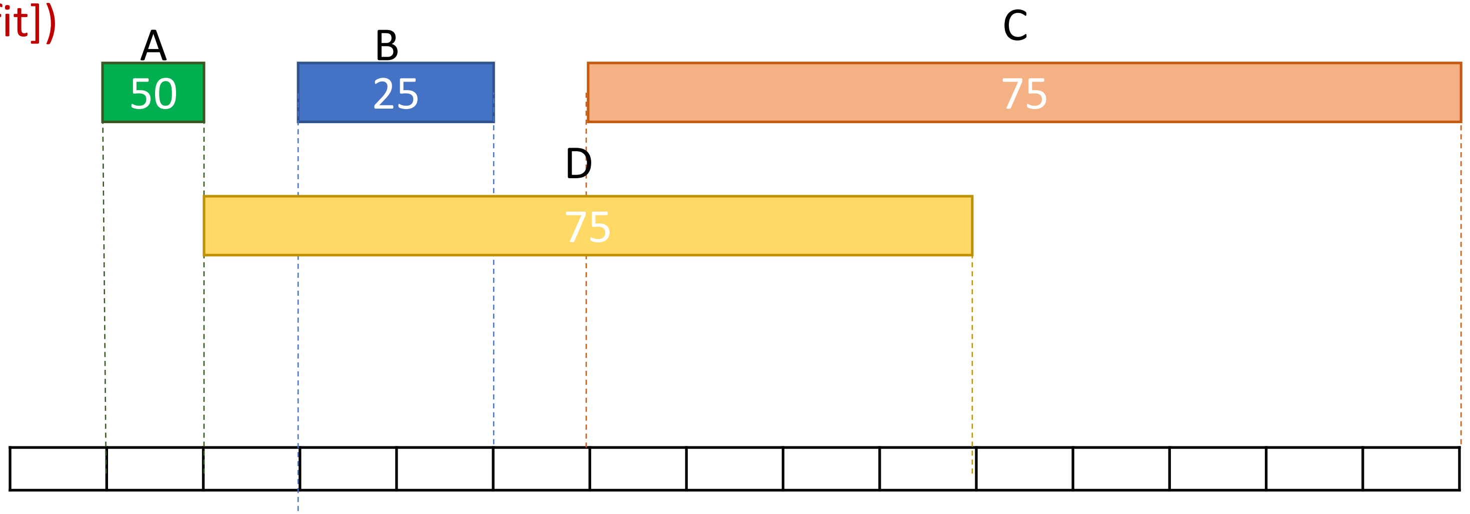
```
// This is used to check if two jobs conflict.
// If one job's end time is later than other job's start time, they have a conflict!
int nonConflictJob(Job jobList[], int i) {
    // Among all jobs:
        for (int j=i-1; j>=0; j--) {
        // If there is a job that does not have a conflict:
        if (jobList[j].end <= jobList[i].start)
            // return the job number.
            return j;
        }
        // If there is no job that does not conflict (if all jobs cause a conflict)
        return -1;
}
```

return j = 1
(job 1 does not cause a conflict with job 3)

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

i = 3



```
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;              //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50	75	125	

i = 3

addProfit = 75

newJob = 1

```
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                 //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50	75	125	

i = 3

addProfit = 75

newJob = 1

```
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50	75	125	

i = 3

addProfit = 75

newJob = 1

addProfit = 75 + table[1] = 150

```
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                 //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50	75	125	150

i = 3

addProfit = 75

newJob = 1
addProfit = 75 + table[1] = 150

addProfit = 150
table[2] = 125
max = 150

```cpp
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
       sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;              //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

At this point we have A, B & C

table:

50	75	125	150

i = 3

addProfit = 75

Because we have reached this value by adding current profit (C) and table entry of the non conflicting jobs (A&B).

newJob = 1

addProfit = 75 + table[1] = 150

addProfit = 150
table[2] = 125
max = 150

```cpp
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;                   //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50	75	125	150

i = 4

```cpp
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;              //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50	**75**	**125**	**150**

i = 4

Maximum profit = last entry of the memory table

```cpp
// n = number of jobs on the list
int findMaxProfit(Job jobList[], int n) {
    // The following line sorts the jobs in the order of their end times
        sort(jobList, jobList+n, comp);

    // Memory table
    int *table = new int[n];

    // First entry of the memory table is the profit of the first ending job.
    table[0] = jobList[0].profit;

    // For every job in the jobs list
    for (int i=1; i<n; i++) {
        // Find profit of the current job
        int addProfit = jobList[i].profit;

        // Find the next non-conflicting job
        int newJob = nonConflictJob(jobList, i);

        // If there is a non-conflicting job:
        if (newJob != -1)
            addProfit += table[newJob];

        table[i] = max(addProfit, table[i-1]) ;
    }

    int result = table[n-1];
    delete[] table;             //clear table from memory
    return result;
}
```

jobList:
(name: [start_time, end_time, profit])

A: [1, 2, 50]
B: [3, 5, 25]
D: [2, 10, 75]
C: [6, 15, 75]

table:

50	75	125	150

i = 4

Maximum profit = 150
Jobs for this profit: A,B,C

Maximum profit = last entry of the memory table

QUESTION: What is the maximum profit you can get?

ANSWER: With jobs A, B and C, you obtain a maximum profit of 150.