DYNAMIC PROGRAMMING

INTRODUCTION

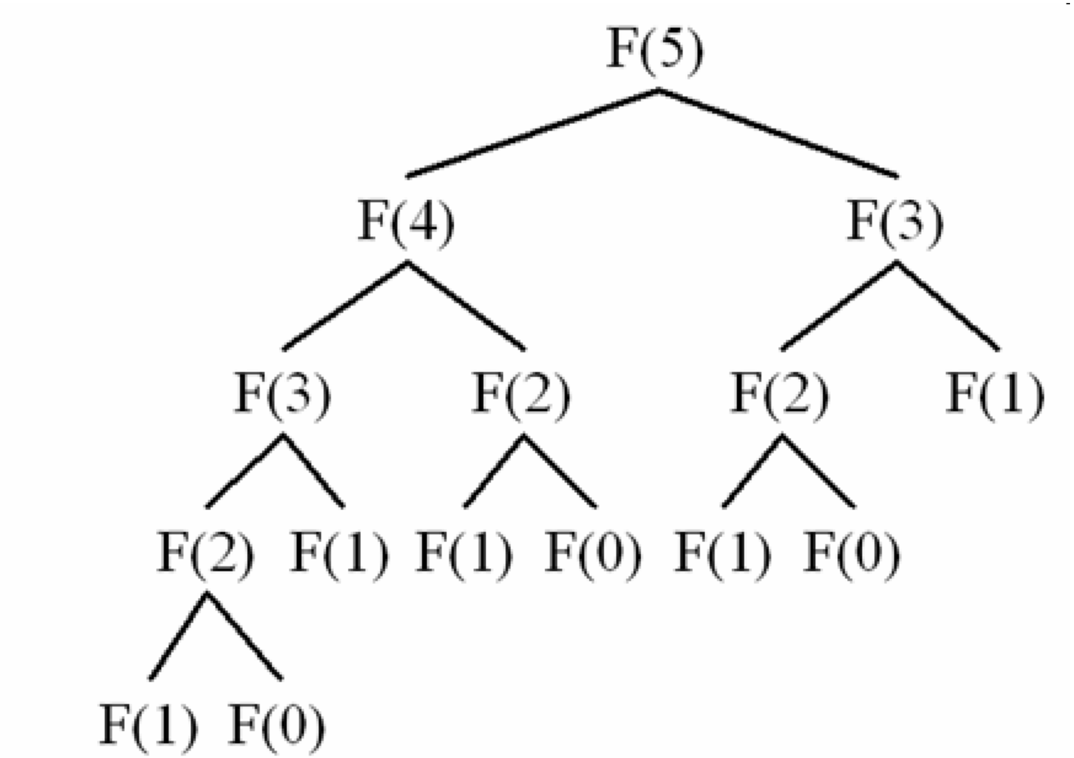
- FIBONACCI SERIES

$$F(n) = F(n-1) + F(n-2) \quad \text{for } n > 1$$

$$F(0) = 0, \qquad F(1) = 1.$$

INTRODUCTION

- Recursive Tree of Fibonacci Series



DYNAMIC PROGRAMMING

- Dynamic programming is a technique for solving problems with overlapping sub problems.

- Typically, these sub problems arise from a recurrence relating a given problem’s solution to solutions of its smaller sub problems.

DYNAMIC PROGRAMMING

- Rather than solving overlapping sub problems again and again, dynamic programming suggests solving each of the smaller sub problems only once and recording the results in a table from which a solution to the original problem can then be obtained.

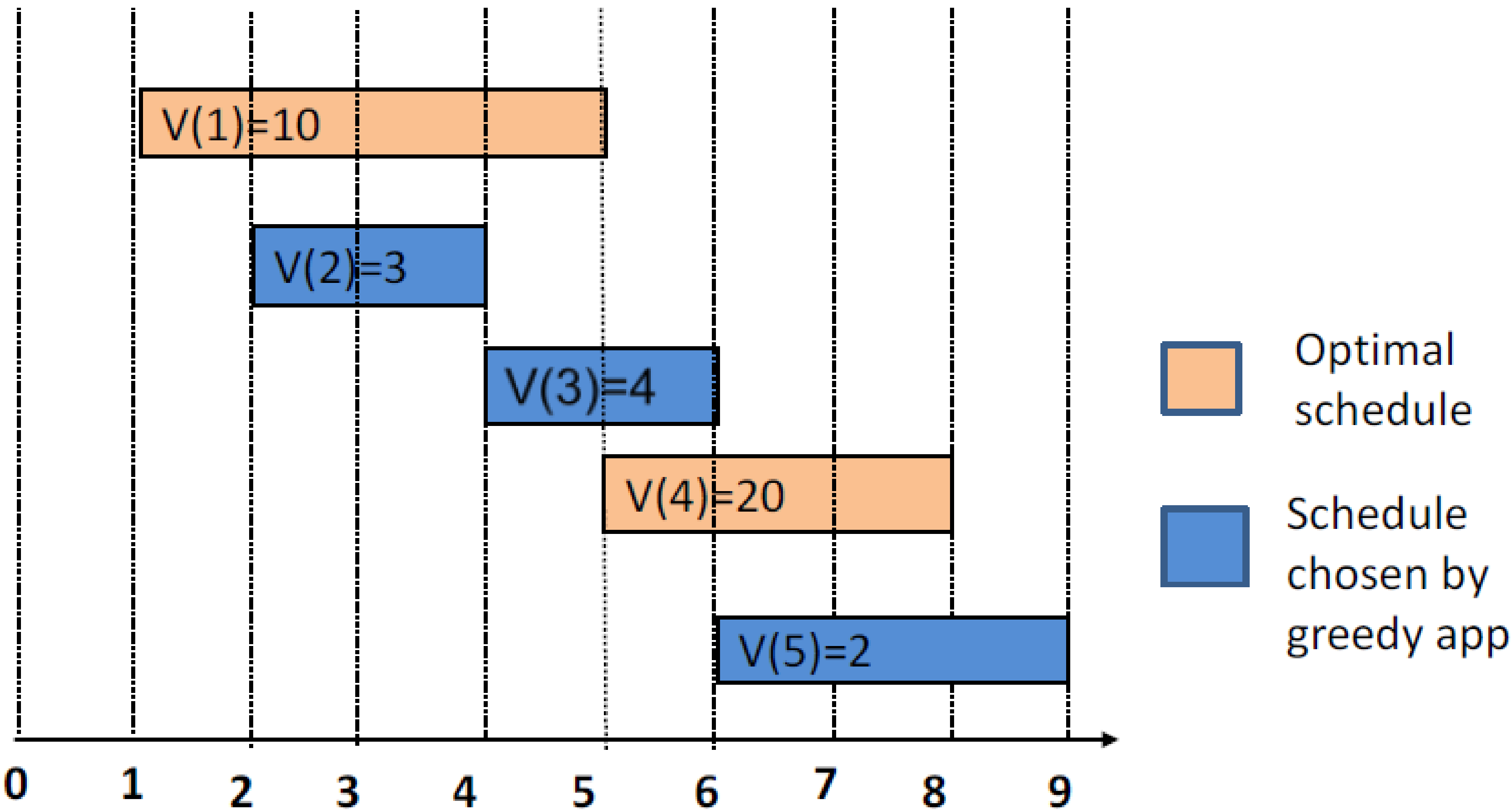
WEIGHTED INTERVAL SCHEDULING

- **Problem and Goal**
- We have n requests labeled **1, . . . , n**, with each request **i** specifying a start time **si** and a finish time **fi**.
- Each interval **i** now also has a value, or weight **vi**.
- Two intervals are compatible if they do not overlap.

WEIGHTED INTERVAL SCHEDULING

- **Problem and Goal**
- The goal of our current problem is to select a subset S ⊆ {1, . . . , n} of mutually compatible intervals, so as to maximize the sum of the values of the selected intervals, $\sum_{i \in S} \boldsymbol{Vi}$

GREEDY DOES NOT WORK



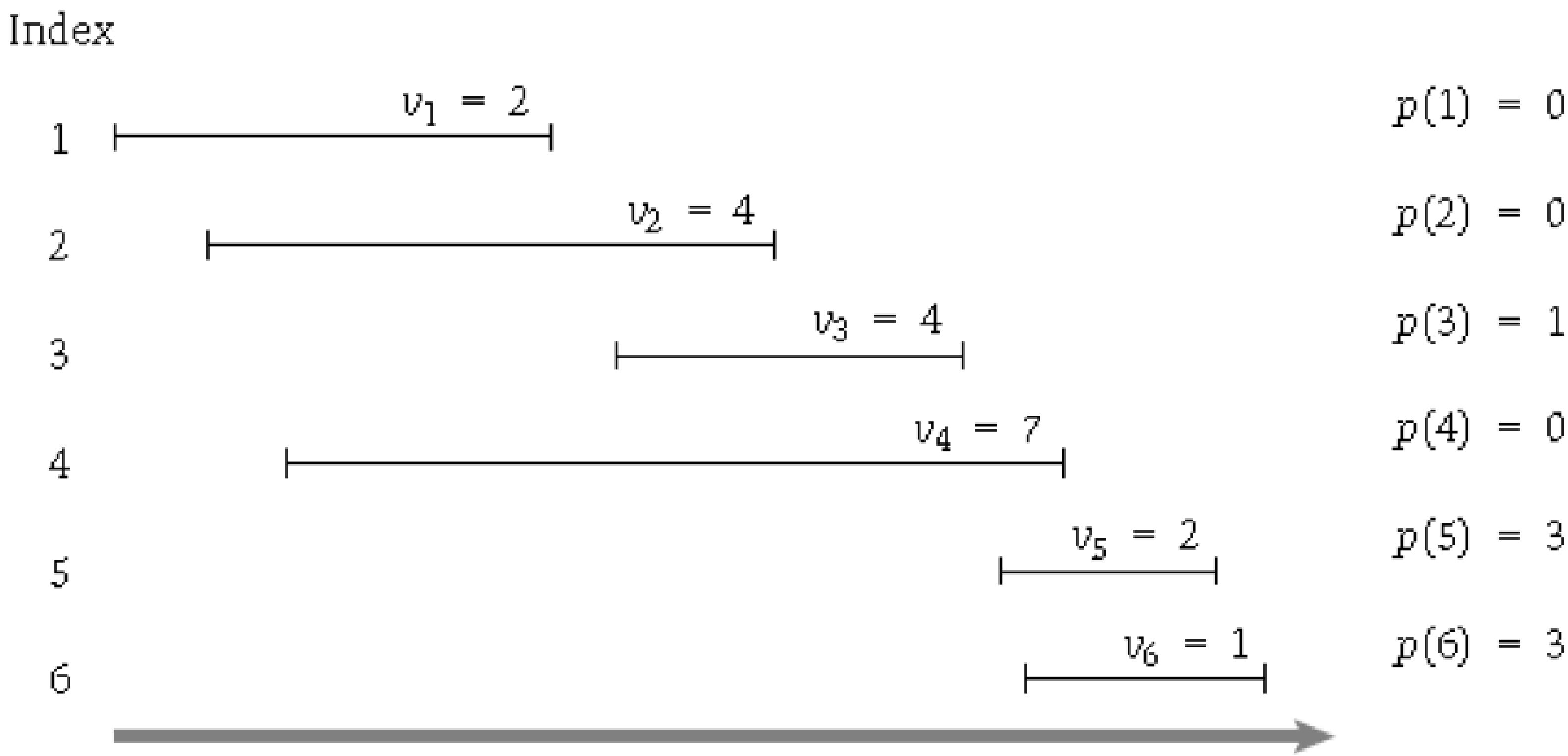
WEIGHTED INTERVAL SCHEDULING

- **Compute predecessor of a request**

- We define **p(j)**, for an interval **j**, to be the largest index **i < j** such that intervals **i** and **j** are disjoint i.e. interval **i** doesn't overlap with **j**.

WEIGHTED INTERVAL SCHEDULING

- **Compute predecessor of a request**

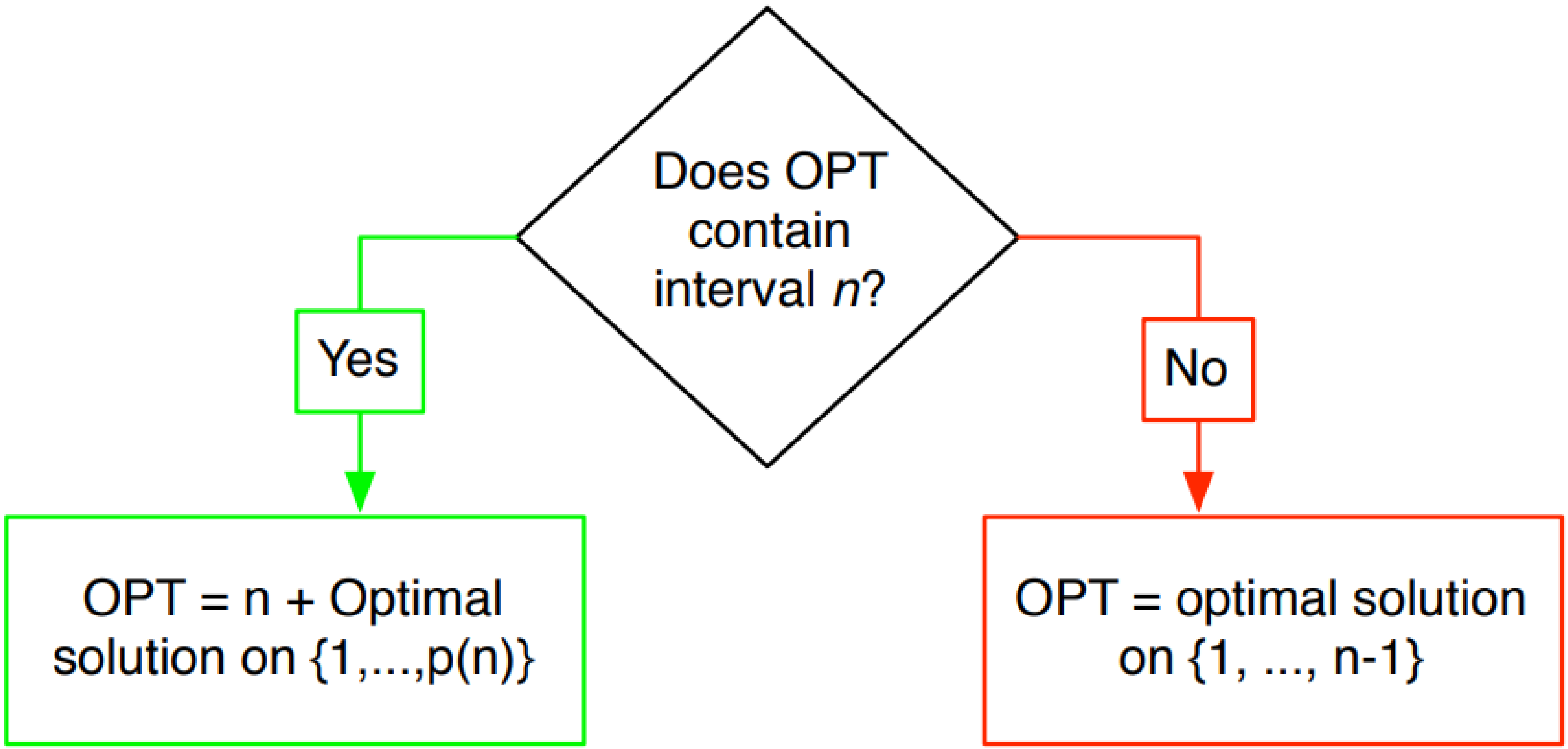


WEIGHTED INTERVAL SCHEDULING

- **Dynamic Programming Approach**
- Let us consider there is an optimal schedule OPT for the given set of requests and their values.
- Let "n" be the last interval in the given set of requests.
- If we want to find whether an interval "n" belongs to OPT there are 2 choices left. They are:
  1. Interval "n" belongs to the OPT. If it belongs then we have to continue for further intervals that are compatible with interval "n" using p(n) calculated for the nth interval.
  2. If it does not belong then, consider the set of intervals from set of {1,…,n-1}.

WEIGHTED INTERVAL SCHEDULING

- **Dynamic Programming Approach**



WEIGHTED INTERVAL SCHEDULING

- **Dynamic Programming Approach**

- The **recurrence relation** of a request "j" in the optimal set of requests can be denoted as OPT(j) given by

$$OPT(j) = \max \begin{cases} v_j + OPT(p(j)) & j \text{ in OPT solution} \\ OPT(j-1) & j \text{ not in solution} \\ 0 & j = 0 \end{cases}$$

Algorithm implementing recurrence relation

//**Purpose**: To find the optimal weights for weighted interval scheduling problem.

//**Input**: 1….n requests each having start time “s” and finish time “f” and weight “v”

//**Output**: Optimal weight of the given set of “n” intervals.

Sort the intervals according to their finish times $f1 \leq f2 \leq f3 \ldots.. \leq fn$

Compute p(1), p(2)……., p(n)

j = n_{th} interval

Compute-Opt (j):

    If j = 0

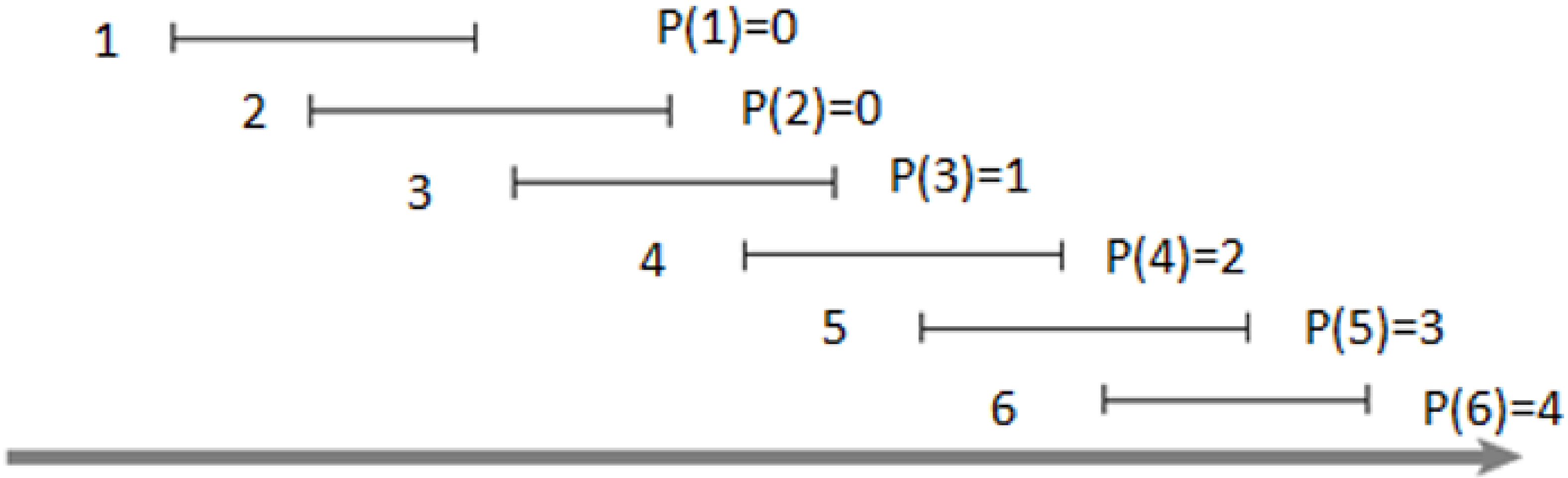
        Return 0

    Else

        Return max(v[j] + Compute-Opt (p[j]),Compute-Opt (j-1))

WEIGHTED INTERVAL SCHEDULING

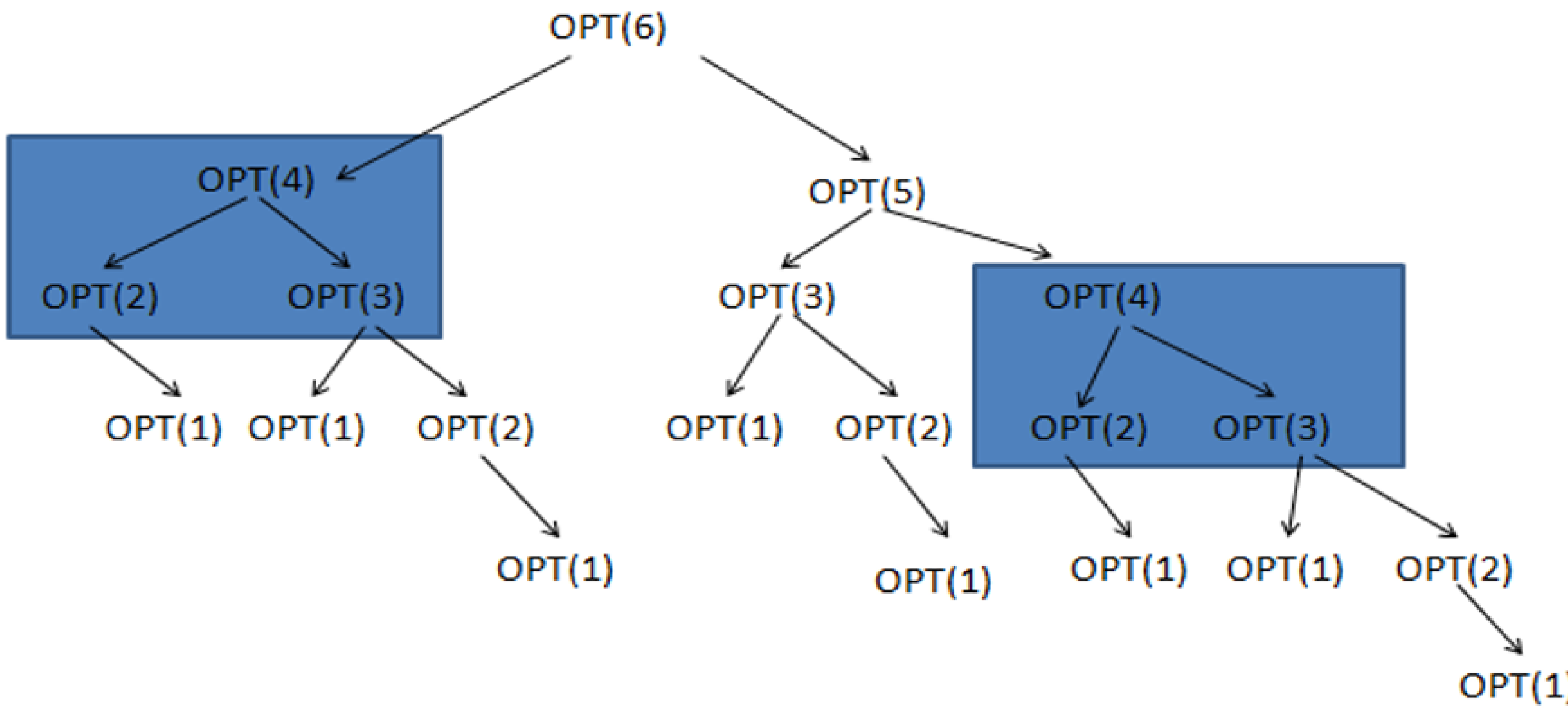
- **Example**



WEIGHTED INTERVAL SCHEDULING

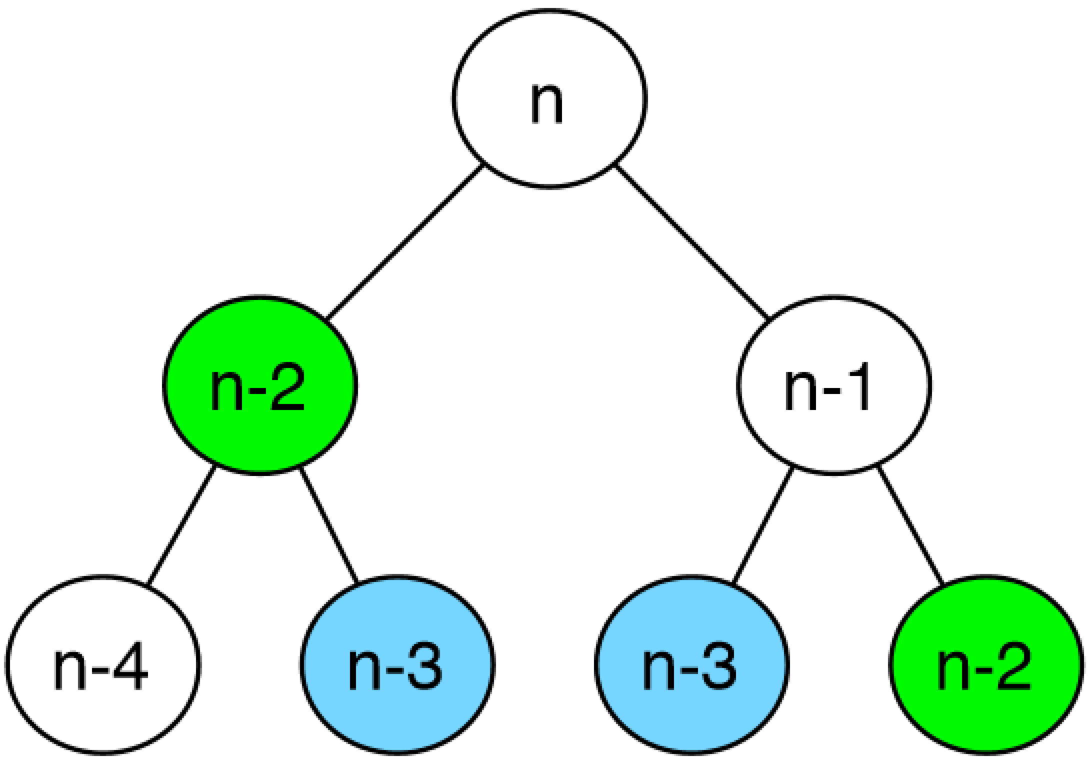
- **Recursive Tree**

MAX ([v[j] + Compute-Opt (p[j])] , Compute-Opt (j-1))



WEIGHTED INTERVAL SCHEDULING

- **This Takes Exponential Time**



MEMOIZATION

- The exponential time of the recursive procedure for weighted interval scheduling can be reduced using Memoization technique.
- We can store the value of Compute-Opt in a globally accessible place the first time we compute it and then simply use this precomputed value in place of all future recursive calls.
- This technique of saving values that have already been computed is referred to as memoization.

Memoization Algorithm

//Purpose: To find the optimal weights for weighted interval scheduling problem.

//Input: 1….n requests each having start time "s" and finish time "f" and weight "v"

//Output: Optimal weight of the given set of "n" intervals.

M-Compute-Opt(j)

If j = 0 then

Return 0

Else if M[j] is not empty then

Return M[j]

Else

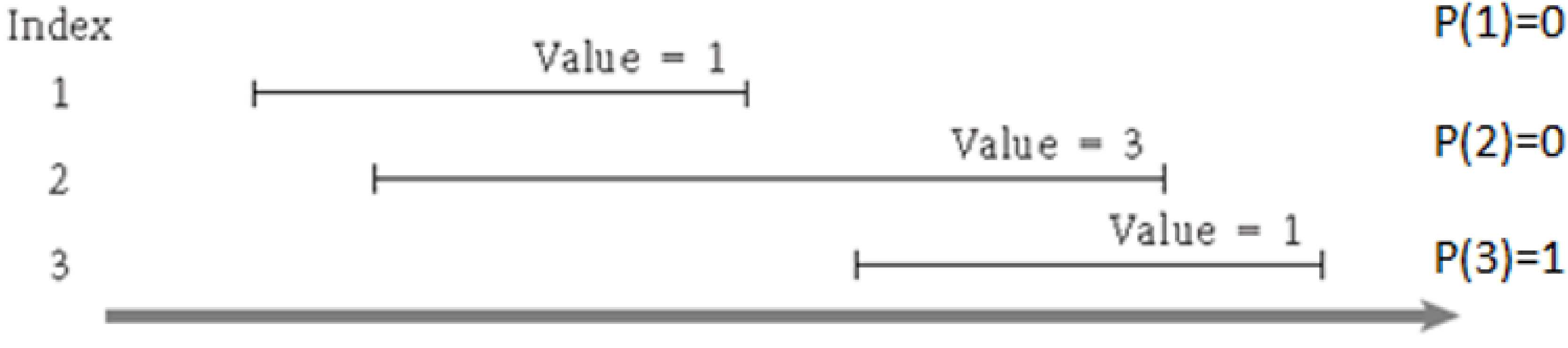
Define M[j] = max(**vj + M-Compute-Opt(p(j))**, **M-Compute-Opt(j − 1)**)

Return M[j]

Endif

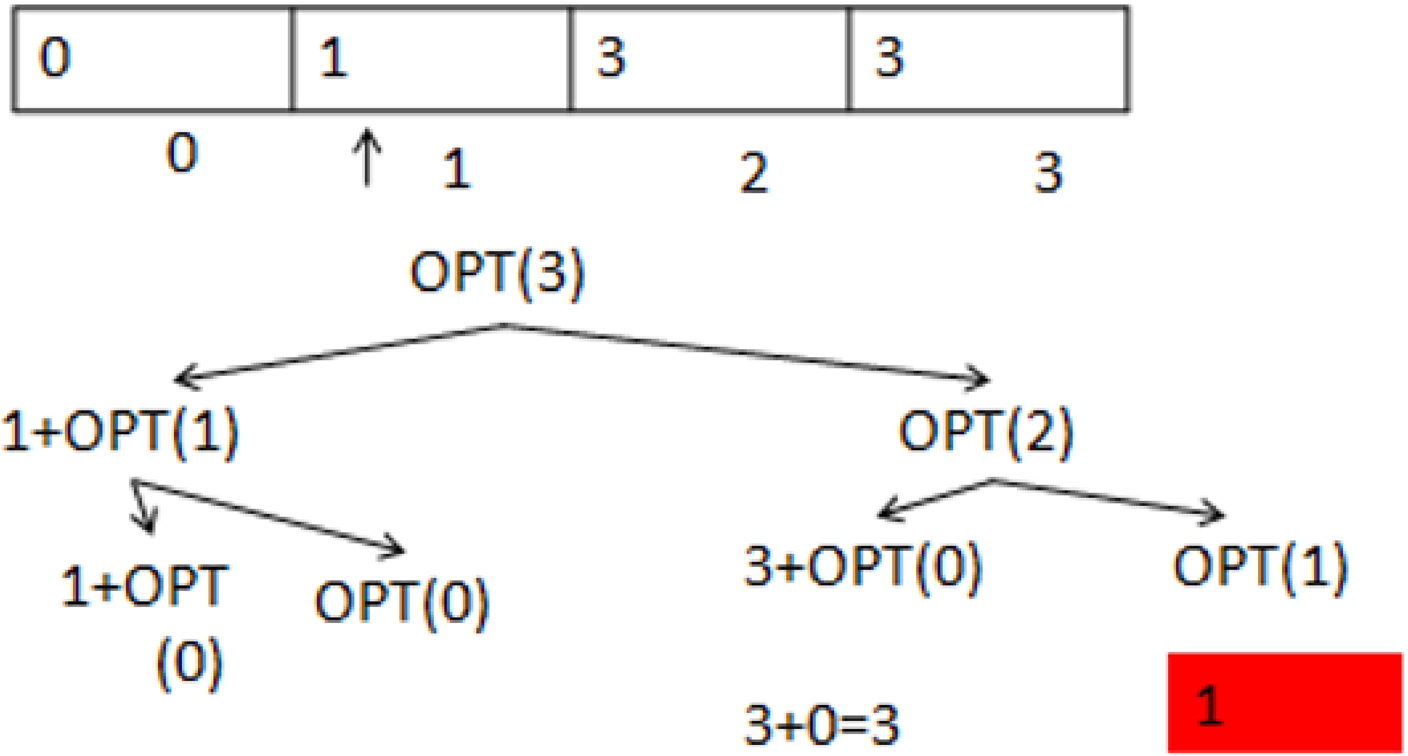
MEMOIZATION

- EXAMPLE



MEMOIZATION

M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j – 1))

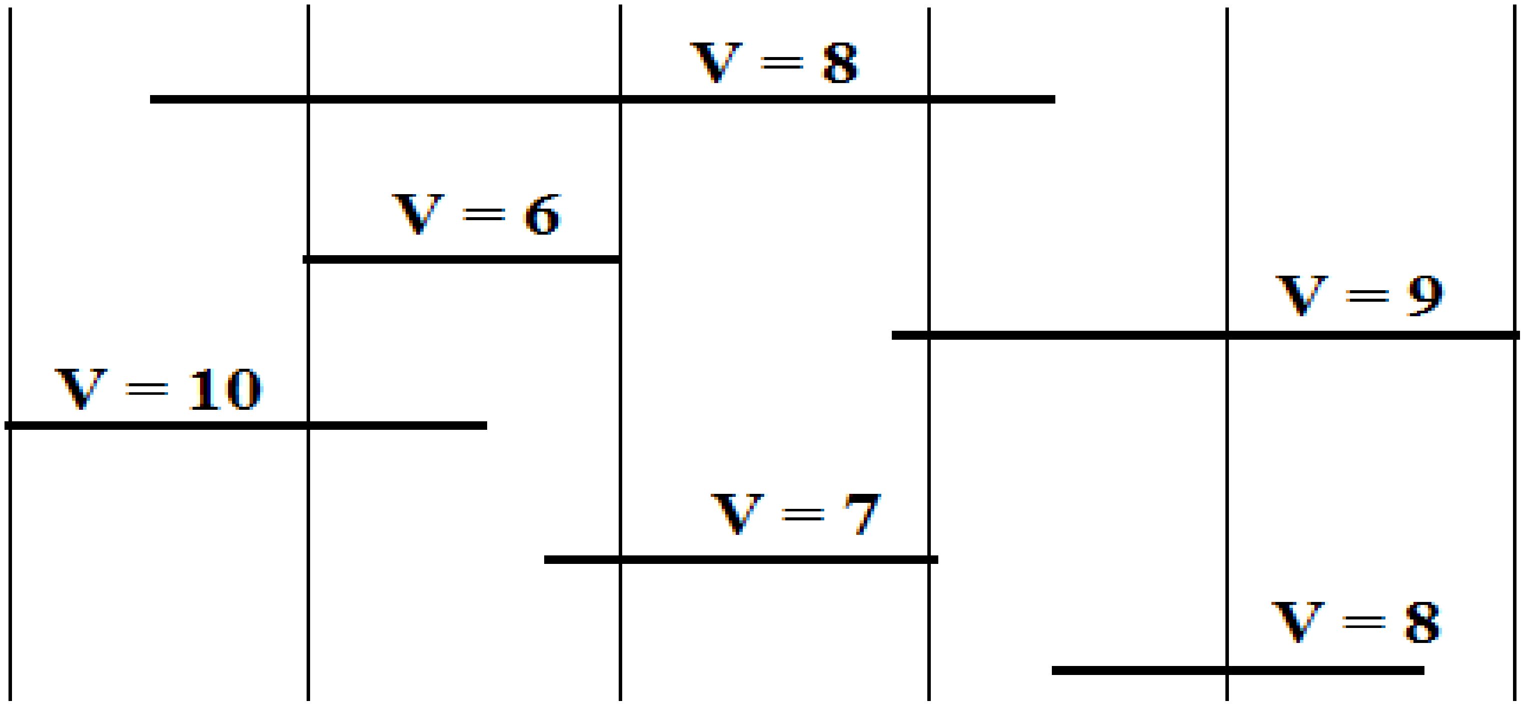


MEMOIZATION

- Find the intervals that belong to the output set.
- Given the global array M we can find the intervals that belong to the output set using the following relation and algorithm.
- Relation: $\mathbf{vj + OPT(p(j)) \geq OPT(j - 1)}$.

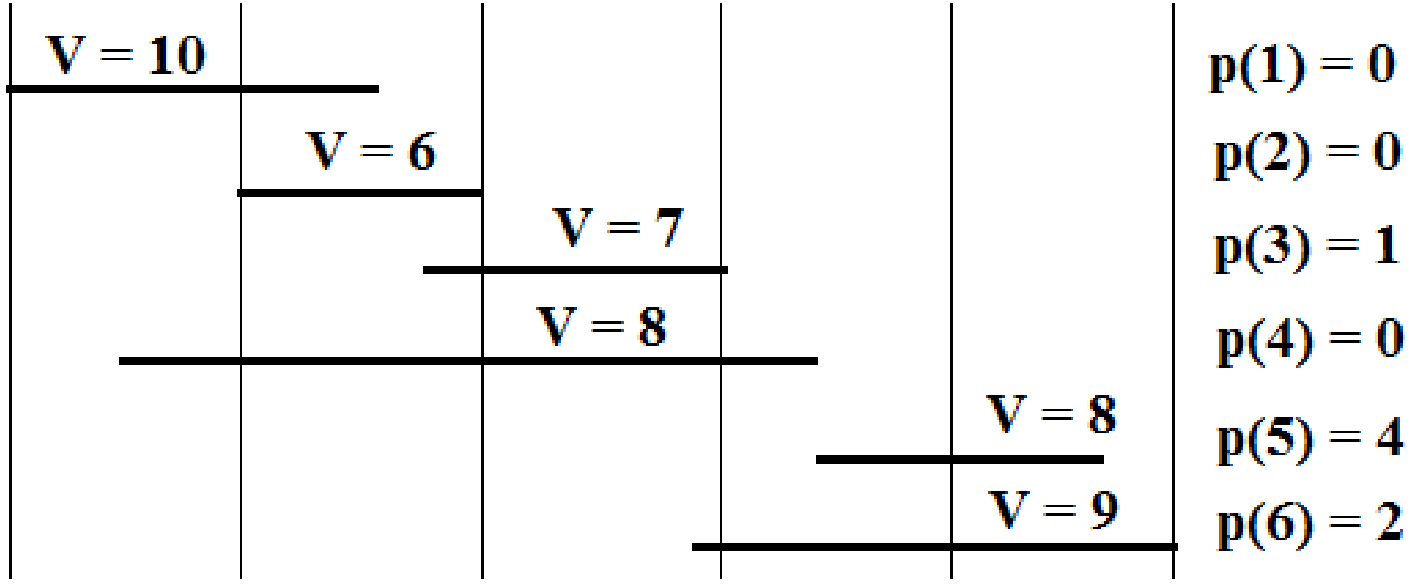
MEMOIZATION

- Apply the Memoization algorithm for the following problem.



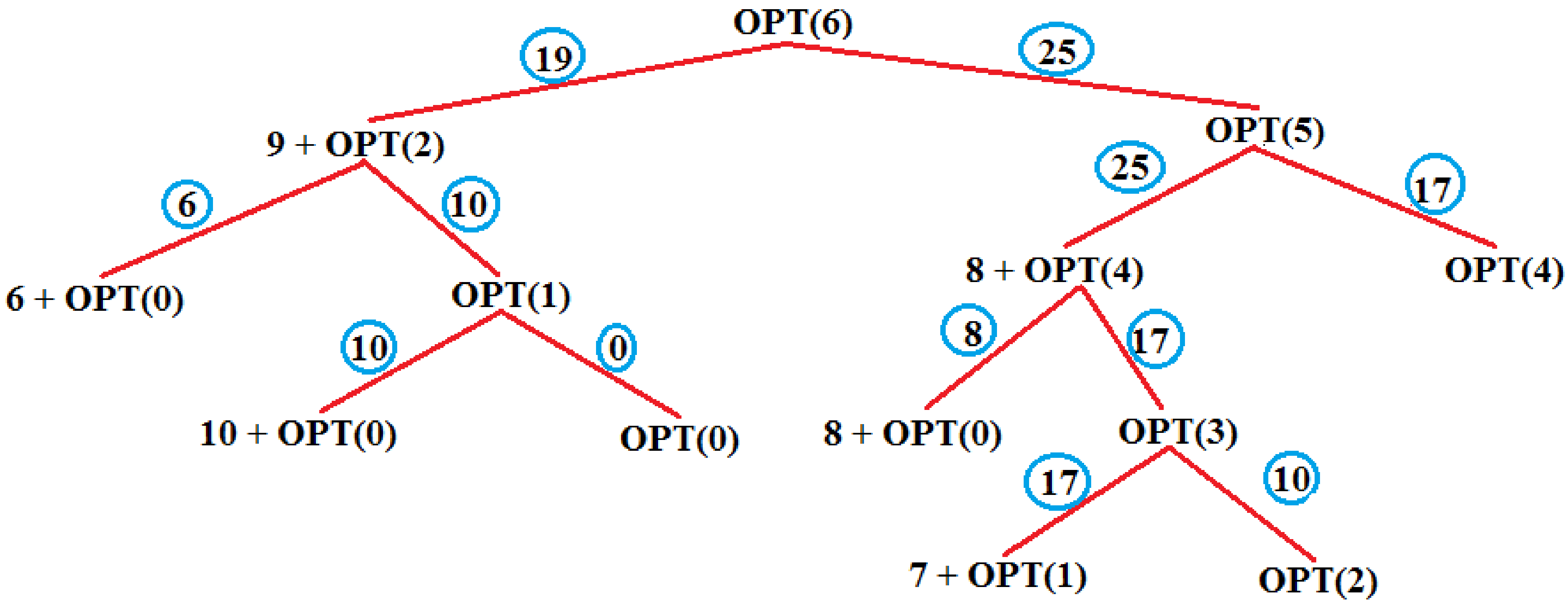
MEMOIZATION

- **Solution:**
- 1. Sort the Intervals according to the finish time. (f1 < f2 < f3 . . .< f6)
- 2. Compute Predecessors.
- 3. Draw Recursive Call Tree.



MEMOIZATION

M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]
0	10	10	17	17	25	25



Algorithm Find-Solution(j)

//Purpose: To find the intervals belonging to the output set for weighted interval scheduling problem.

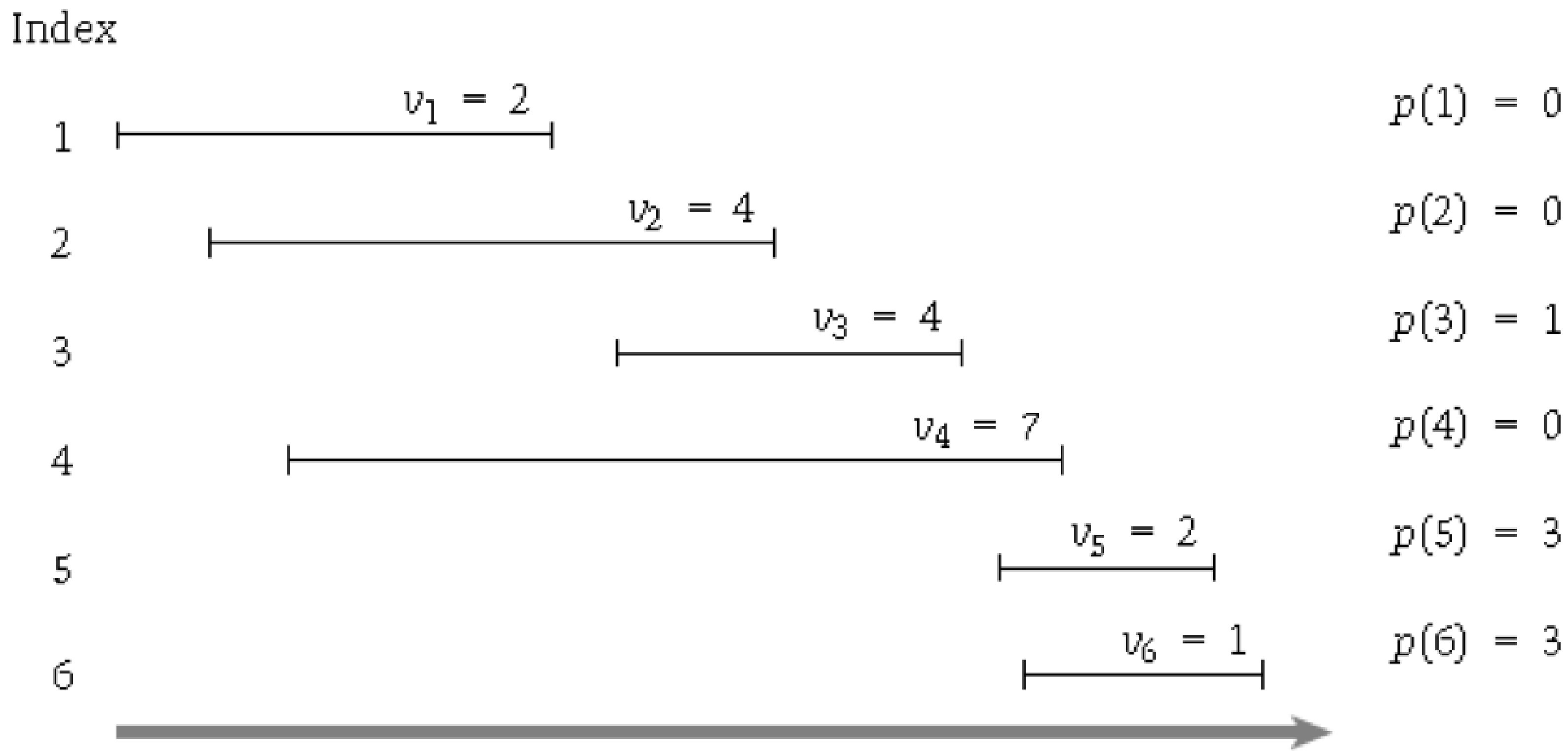
//Input: 1….n requests each having start time "s" and finish time "f" and weight "v"

//Output: The intervals belonging to the output set for weighted interval scheduling problem.

```
Find-Solution(j)
{
if (j > 0)
   if (vj + M[p(j)] > M[j-1])
        print j
        Find-Solution(p(j))
   else
        Find-Solution(j-1)
}
```

MEMOIZATION

- Example:



MEMOIZATION

- Example:

N	Global Array M							Vj + M[p(j)]	M[j-1]	Output Set
6	**0**	**2**	**4**	**6**	**7**	**8**	**8**	1 + 6 = 7	8	**{ }** Find_solution(5)
5	**0**	**2**	**4**	**6**	**7**	**8**	**8**	2 + 6 = 8	7	**{5}** Find_solution(3)
3	**0**	**2**	**4**	**6**	**7**	**8**	**8**	4 + 2 = 6	4	**{5, 3}** Find_solution(1)
1	**0**	**2**	**4**	**6**	**7**	**8**	**8**	2 + 0 = 2	0	**{5, 3, 1}** Find_solution(0)

ITERATIVE PROCEDURE

- We have **Iterative Procedure** to calculate the global array “M” and use the find solution method to find the max weight possible and intervals in the output set respectively.

Iterative Algorithm

//**Purpose**: To find the optimal weights for weighted interval scheduling problem.

//**Input**: 1….n requests each having start time “s” and finish time “f” and weight “v”

//**Output**: Optimal weight of the given set of “n” intervals.

Iterative-Compute-Opt

M[0]= 0

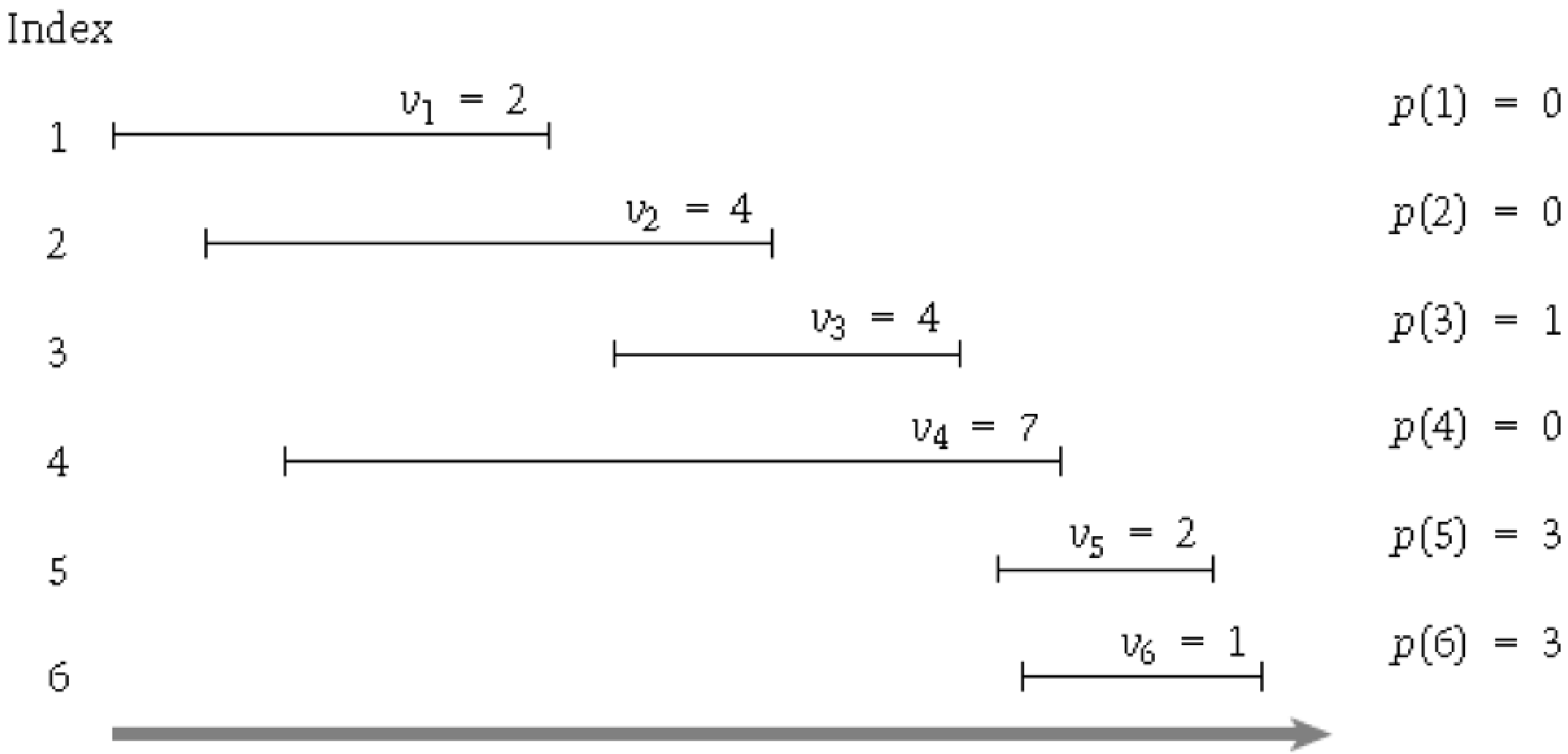
For j = 1, 2, . . . , n

M[j] = max (vj + M[p(j)], M[j − 1])

Endfor

ITERATIVE PROCEDURE

- Example:



ITERATIVE PROCEDURE

- Example:

- M

0	1	2	3	4	5	6
0	2					

- j = 1
- M[1] = max (vj + M[p(j)], M[j – 1])

  M[1] = max (2 + M[0], M[0])

  M[1] = max (**2 + 0**, 0)

ITERATIVE PROCEDURE

- Example:

- M

0	1	2	3	4	5	6
0	2	4				

- j = 2
- $M[2] = \max (vj + M[p(j)],\ M[j-1])$

  $M[2] = \max (4 + M[0],\ M[1])$

  $M[2] = \max (\mathbf{4 + 0},\ 2)$

ITERATIVE PROCEDURE

- Example:

- M

0	1	2	3	4	5	6
0	2	4	6			

- j = 3
- M[3] = max (vj + M[p(j)], M[j – 1])

  M[3] = max (4 + M[1], M[2])

  M[3] = max (**4 + 2**, 4)

ITERATIVE PROCEDURE

- Example:

- M

0	1	2	3	4	5	6
0	2	4	6	7		

- j = 4
- M[4] = max (vj + M[p(j)], M[j – 1])

  M[4] = max (7 + M[0], M[3])

  M[4] = max (**7 + 0**, 6)

ITERATIVE PROCEDURE

- Example:

- M

0	1	2	3	4	5	6
0	2	4	6	7	8	

- j = 5
- M[5] = max (vj + M[p(j)], M[j – 1])

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

  M[5] = max (**2 + 6**, 7)

ITERATIVE PROCEDURE

- Example:

- M

0	1	2	3	4	5	6
0	2	4	6	7	8	8

- j = 6
- M[6] = max (vj + M[p(j)], M[j – 1])

  M[6] = max (1 + M[3], M[5])

  M[6] = max (1 + 6, **8**)

THANK YOU