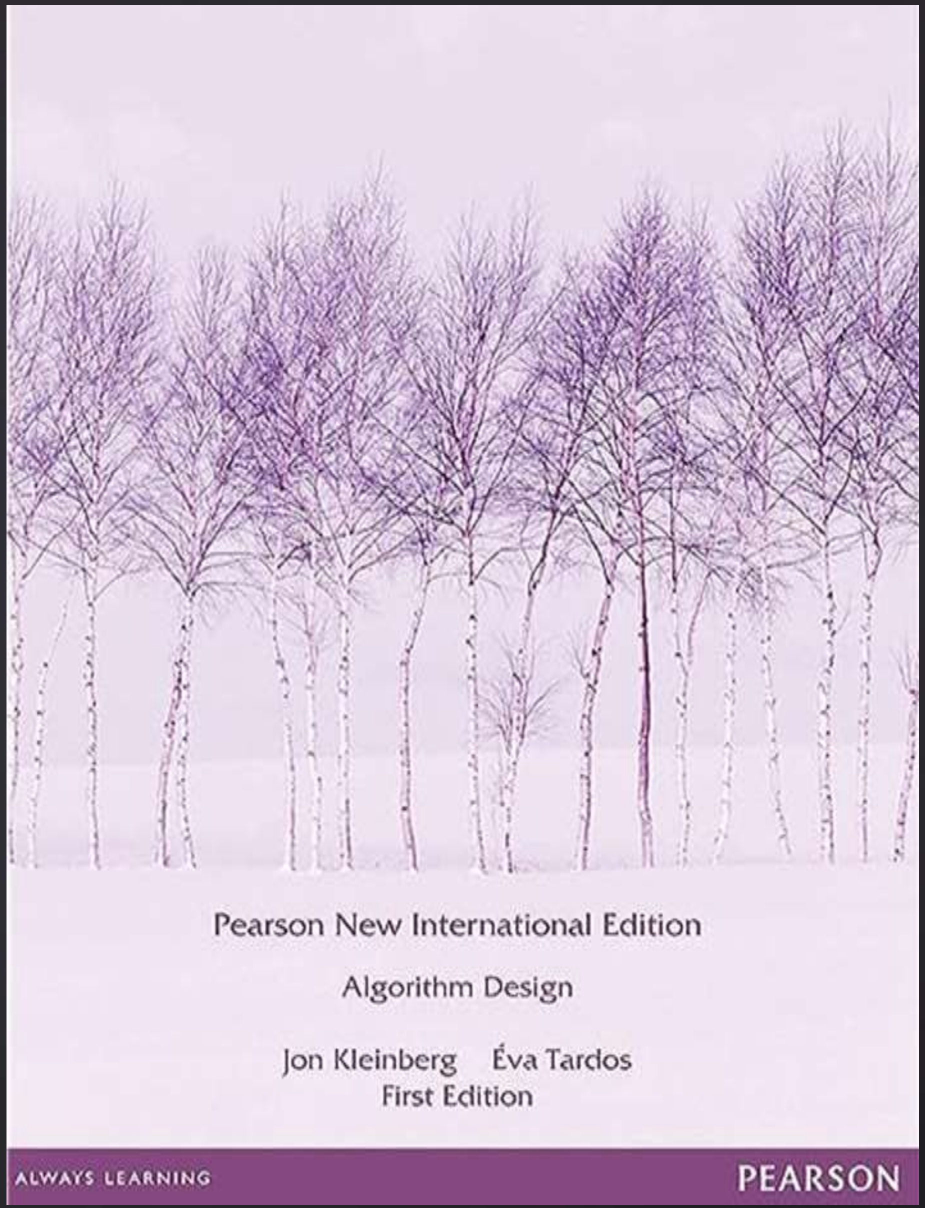
CSC 212

Algorithms & Complexity

Dynamic Programming

Activity

Read Chapter 6 and look through the exercises



Contents

Overview of Problem-solving approaches

Dynamic Programming

Weighted interval Scheduling Problem

Problem-solving approaches

Brute-force

Divide and Conquer

Greedy

Dynamic Programming

focus on finding the CORRECT solution

focus on finding an OPTIMAL solution

Optimization is a process to find the best solution among alternatives

Brute-force

Makes an exhaustive search until the solution to a problem is found

used when you first encounter a problem

worst in terms of time and space complexity

Example:

Find the maximum element in an array

Brute-force Solution:

Iterate through all the elements in an array

```
for (i = 1; i < arr.length; i++)
        if (arr[i] > max)
            max = arr[i];
```

Inefficient because the maximum number could be the first element of the array

Brute-force

in some cases, the brute-force approach might be the only way to solve a problem

Example:

Crack a password which has the letters a,b,c,d or e, assuming none of the letters are repeated

Brute-force Solution:

enumerate all possible strings using these letters and see if any of the strings work

Divide and Conquer

Which is easier? To break 100 sticks at once or to break them one at a time?

Break a problem into smaller sub-problems,
solve sub-problems and combine to form final solution

usually solves a problem using recursion

Example:

Sort an array

Divide and Conquer Solution:

Merge sort or Quick sort

Greedy

Assume that you have an objective function that needs to be optimized (either maximized or minimized) at a given point.

A Greedy algorithm makes short-sighted choices at each step to ensure that the objective function is optimized.

Optimization is a process to find the best solution among alternatives

The Greedy algorithm has only one shot to compute the optimal solution, so it has to make a decision in the moment AND that it never goes back to change the decision.

Greedy

PROPERTIES OF PROBLEMS

Greedy choice

Optimal substructure

(*aka* Principle of optimality)

Greedy

PROPERTIES OF PROBLEMS

Greedy choice: the globally optimal solution can be found by selecting a locally optimal choice.

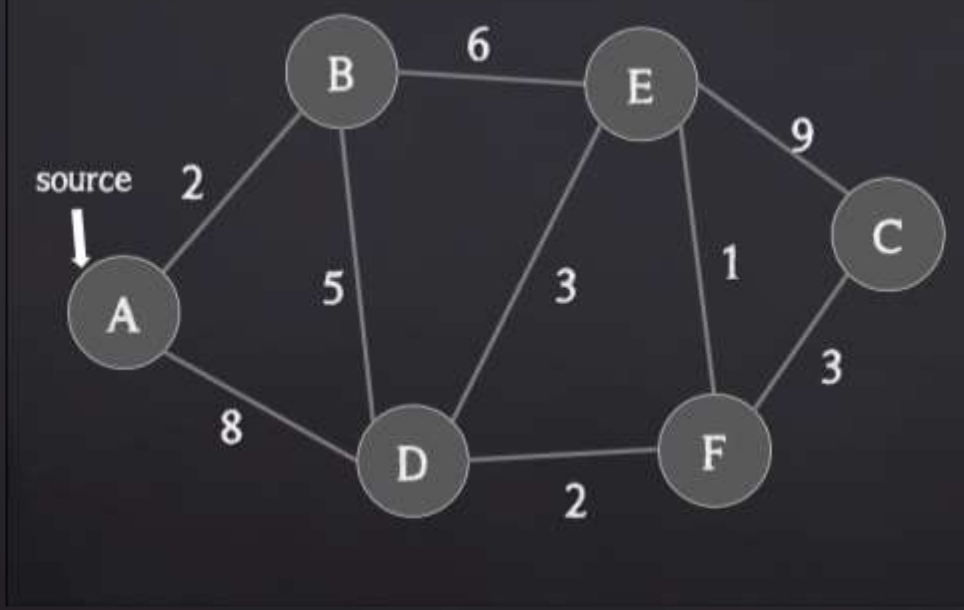
Greedy Algorithm works in an iterative manner, making one locally optimal (greedy) choice after another. The choice made by the algorithm may depend on earlier choices but not on the future.

Greedy

PROPERTIES OF PROBLEMS

Optimal substructure: A problem exhibits optimal substructure if the optimal solution to the problem also contains optimal solutions to the subproblems.

also means that for the global problem to be solved optimally, each subproblem should be solved optimally.



	Shortest distance from A	Previous node
A	0	
B	2	A
C	12	F
D	7	B
E	8	B
F	9	D

(also known as principle of optimality)

The solution to finding the shortest path from A to C also allows us to find the shortest path to any two vertices

Greedy Algorithm: Dijkstra
Finding shortest path

Greedy

PROS

simple and fast

CONS

Doesn't always find a global optimal solution

Usually hard to prove its correctness

Similarities and Differences between Dynamic programming and the others

Brute-Force	like	Explores all possible solutions
	UN-like	Has a condition such that it doesn't always need to check all the solutions
Divide and Conquer	like	Solutions involve sub-problems
	UN-like	there is no overlap between sub-problems

Similarities and Differences between Dynamic programming and the others

Greedy

like

- Solutions involve sub-problems
- Solve optimal substructure problems

UN-like

- Dynamic programming has several attempts to make the right decision
- Solves overlapping sub-problems

Dynamic programming

CORE IDEA

Those who cannot remember
the past are doomed to repeat it.

avoid repeated work

store intermediate (partial) result

Dynamic Programming

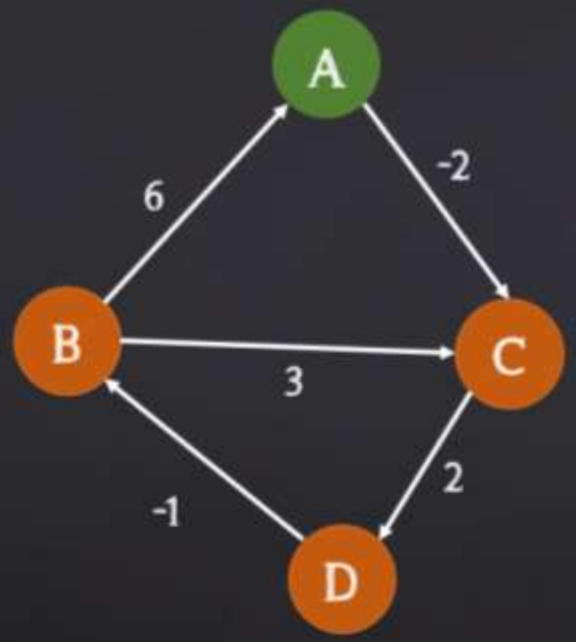
PROPERTIES OF PROBLEMS

Optimal substructure

Overlapping Subproblems

Dynamic Programming

Optimal substructure: A problem exhibits optimal substructure if the optimal solution to the problem also contains optimal solutions to the subproblems.



	A	B	C	D
1st	0	−1	−2	0
2nd	0	−1	−2	0
3rd				
4th				
Previous node		D	A	C

Dynamic Program Algorithm: Bellman-Ford
Finding shortest path

also means that for the global problem to be solved optimally, each subproblem should be solved optimally.

(also known as principle of optimality)

The solution to finding the shortest path from A to C also allows us to find the shortest path to any two vertices

Dynamic Programming

Overlapping Subproblems: is the property in which value of a subproblem is used several times..

```
                         fib(5)
                     /             \
               fib(4)                fib(3)
             /      \                /     \
         fib(3)      fib(2)         fib(2)    fib(1)
        /     \       /    \       /    \
    fib(2)   fib(1)  fib(1) fib(0) fib(1) fib(0)
   /    \
fib(1) fib(0)
```

Dynamic Programming

Overlapping Subproblems: is the property in which value of a subproblem is used several times..

```
                       fib(5)
                   /             \
             fib(4)                fib(3)
            /      \              /      \
       fib(3)      fib(2)       fib(2)      fib(1)
      /      \     /    \       /    \
  fib(2)   fib(1) fib(1) fib(0) fib(1) fib(0)
  /    \
fib(1) fib(0)
```

Dynamic programming: Techniques

Memoization:

involves storing
the results of repeated function calls
and reusing them when the function is called again.

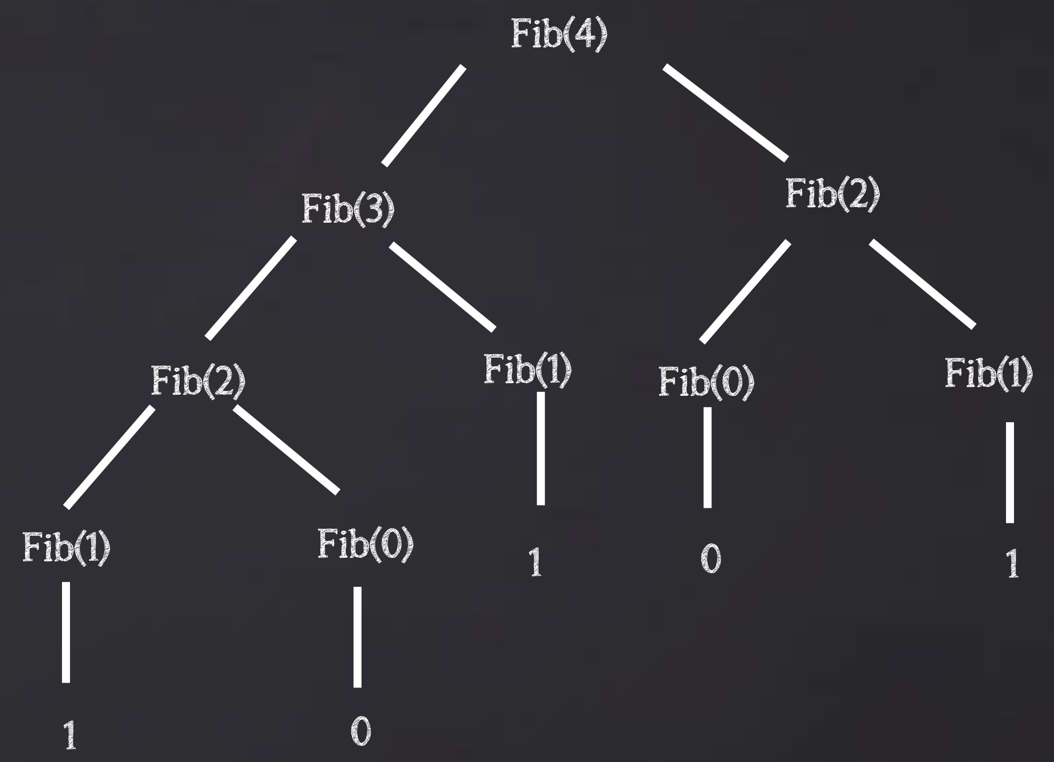
Dynamic Programming aims to find a recursion/iteration that can be efficiently memoized

Recursive Algorithm for Fibonacci

```
public static int Fib(int n) {
   if (n = 0)
       return 0;

   else if (n = 1)
        return 1;

   else
        return Fib(n-1) + Fib(n-2);
```



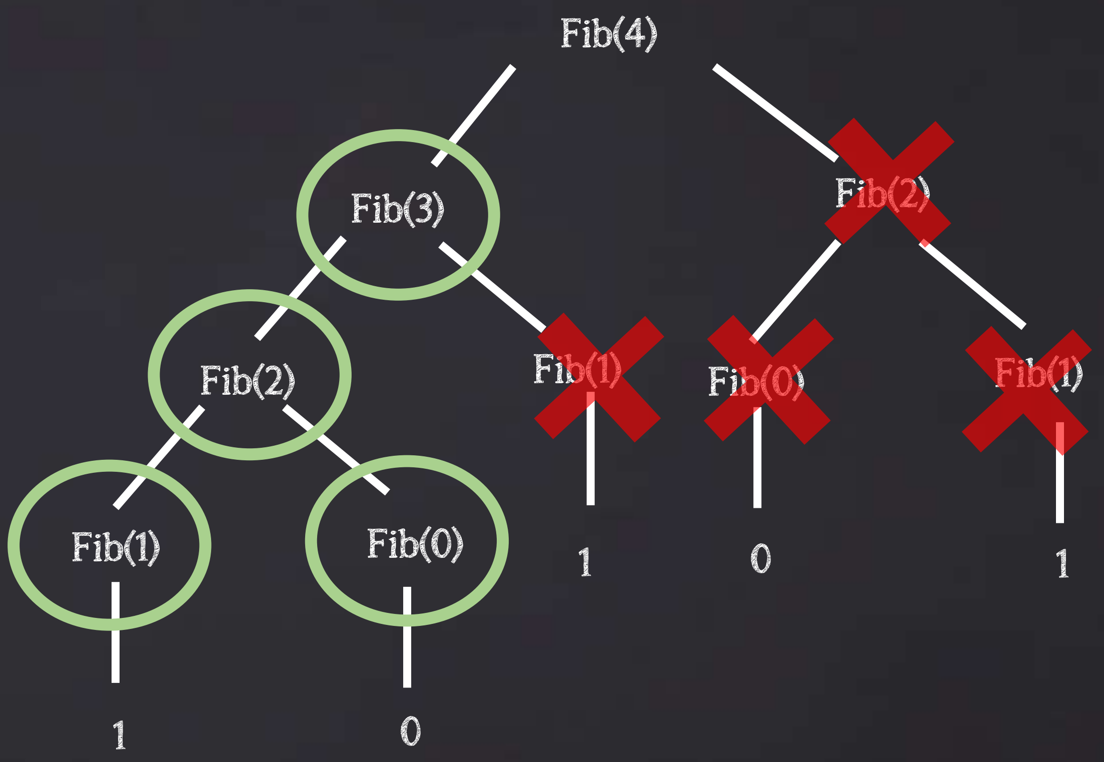
Memoized Recursive Algorithm for Fibonacci

Initialise array M
of size n such that M[i] = −1 for $0 \leq i < n$

```
public static int Fib(int n, int [] M)
{
   if (n = 0)
       M[0] = 0;

   else if (n = 1)
       M[1] = 1;
//checks if a value for n has been stored
   else if (M[n] != -1)
       return M[n];

   else {
        result = Fib(n-1) + Fib(n-2);
        M[n] = result;
       }
return result;
}
```



The memoized version
improves the algorithm from $O(2^n)$ to $O(n)$

Interval Scheduling Problem

We want to accept as many jobs as possible

Three attempts of solving it greedily do not yield optimal results

1. start with the earliest time
2. start with the shortest interval
3. start with the interval with fewest conflicts

An optimal solution is:

- Start with the earliest finish time

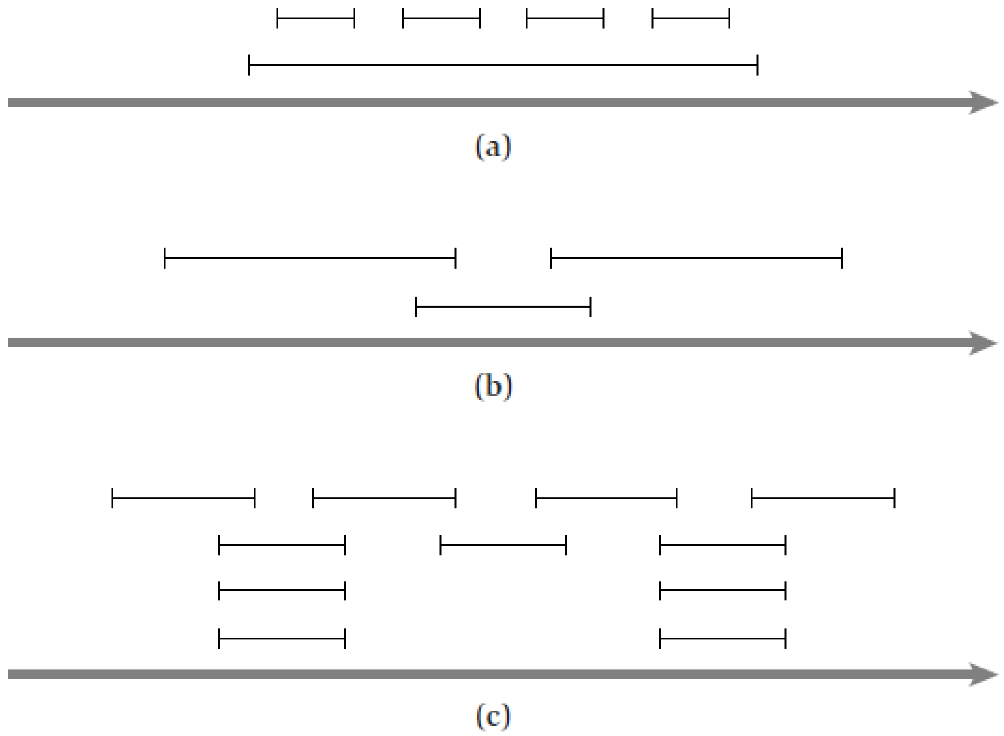


Figure 4.1 Some instances of the Interval Scheduling Problem on which natural greedy algorithms fail to find the optimal solution. In (a), it does not work to select the interval that starts earliest; in (b), it does not work to select the shortest interval; and in (c), it does not work to select the interval with the fewest conflicts.

Weighted Interval Scheduling Problem

When the intervals have weights or values,
no optimal greedy solution exists

The former optimal solution

- Start with the earliest finish time with no overlapping intervals

does not yield optimal results

Hence, we solve it recursively

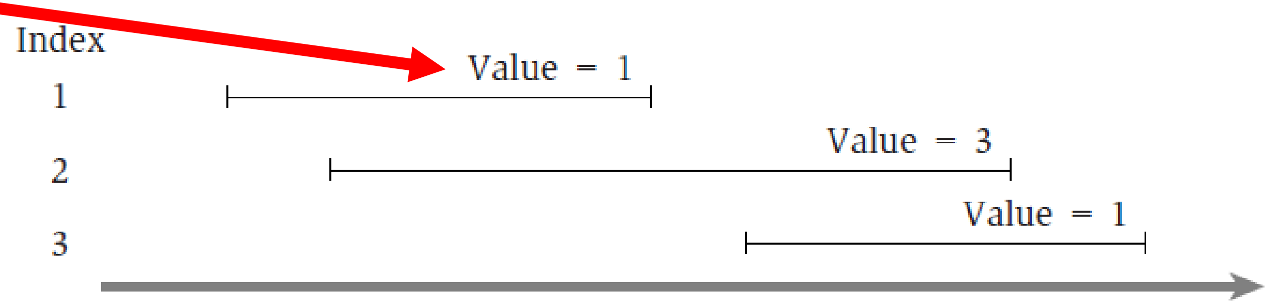


Figure 6.1 A simple instance of weighted interval scheduling.

Weighted Interval Scheduling Problem

We are given a set S={1, . . . , *n*} of *n activity requests*, where each activity is expressed as an interval $[s_i, f_i]$ from a given start time s_i to a given finish time f_i. , and the objective is to find a set of non-overlapping requests such that sum of values of the scheduled requests is maximum

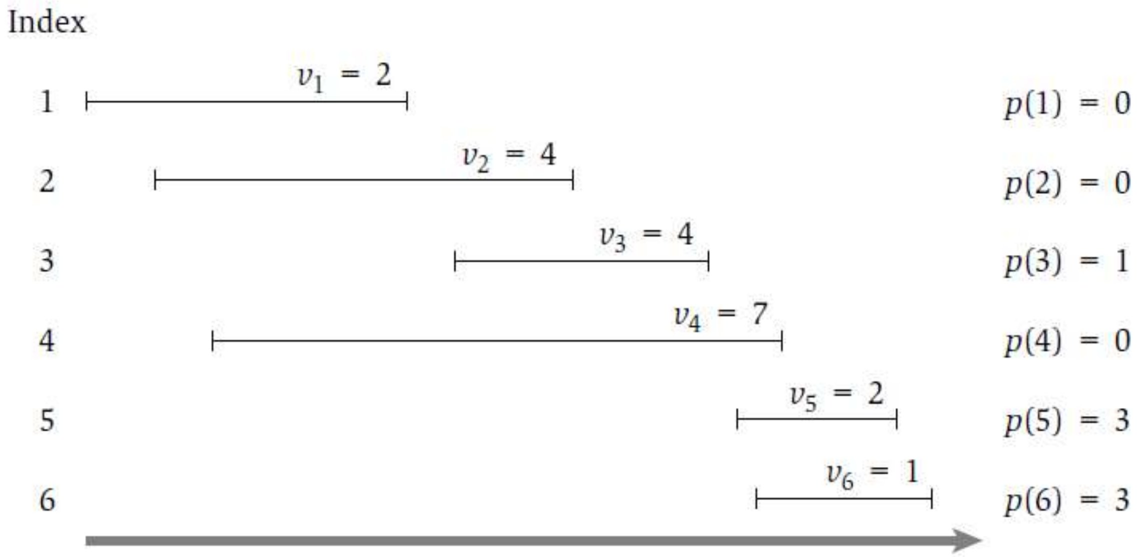


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

The requests are sorted in order of non-decreasing finish time:
$f_1 \leq f_2 \leq . . . \leq f_n$.

A request *i* comes before a request *j* if $i < j$

Define p(j), for an interval j, to be the largest index $i < j$ such that intervals *i* and *j* are disjoint.

Weighted Interval Scheduling Problem

Consider the last request $[s_n, f_n]$. There are two possibilities: either this request is in the optimal schedule or it is not.

If it is in the optimal schedule

- schedule this request (receiving the profit of v_n) and then eliminate all the requests whose intervals overlap this one. As requests have been sorted by finish time, this involves finding the largest index p(j) such that $f_j < s_n$.

If it is not in the optimal schedule,

- Ignore this request and compute the optimal solution of the first n-1 requests

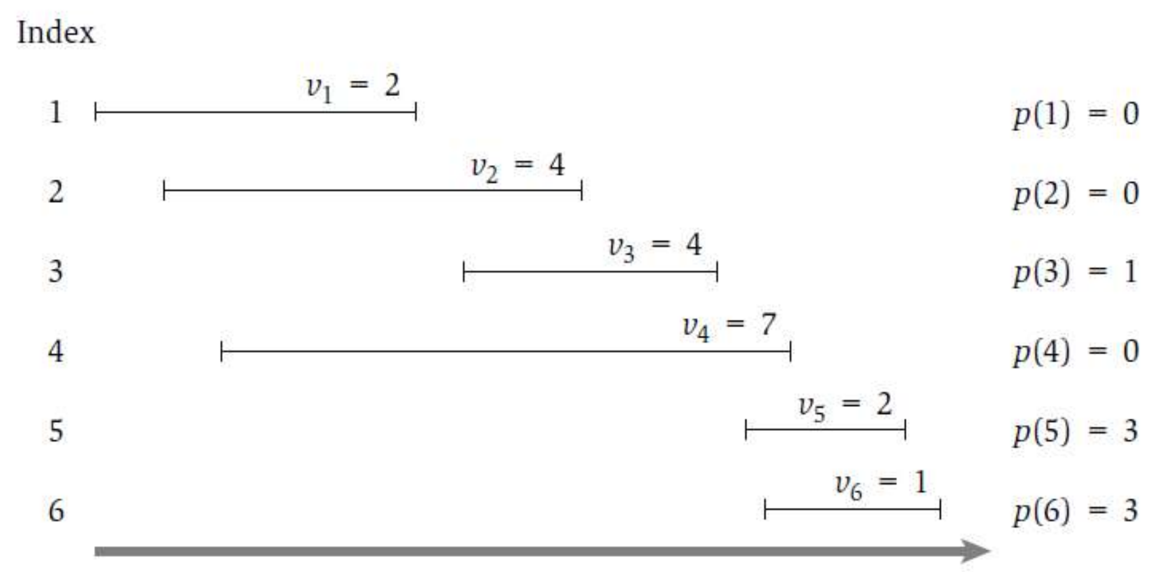


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

But we don't know the optimal solution, so how can we select among these two options?

The answer: compute the cost of both of them recursively, and take the better of the two.

Weighted Interval Scheduling Problem

To formalise the previous slide

For $0 \leq j \leq n$, let opt(j) denote the maximum possible value achievable if we consider just tasks $\{1, \ldots, j\}$ (assuming the tasks are given in order of finish time).

If job is in optimal schedule,
opt(j) = v_j + opt(p(j))

If job is NOT in optimal schedule,
opt(j) = opt(j-1)

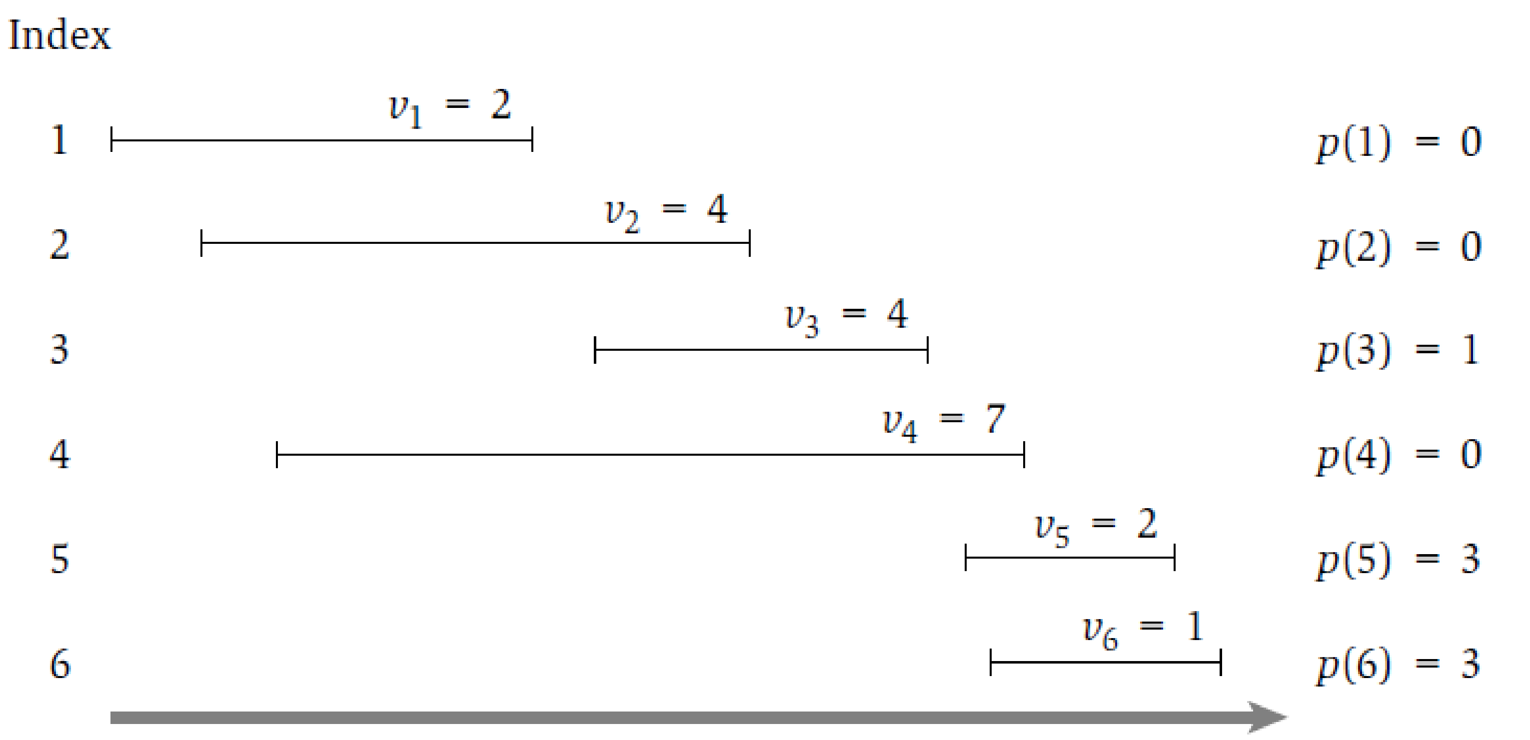
since we don't know the optimal schedule, we compute opt(j) for both cases and take the best

$$\text{opt}(j) = \max \begin{cases} v_j + \text{opt}(p(j)) \\ \text{opt}(j-1) \end{cases}$$

Recursive Algorithm

```
opt(int j)
{
   if (j = 0)
        return 0;

   else return max (opt(j-1), v[j] + opt(p[j]);
}
```



If job is in optimal schedule,
$opt(j) = v_j + opt(p(j))$

If job is NOT in optimal schedule,
$opt(j) = opt(j\text{-}1)$

$$opt(v_6) = \max \begin{cases} 1 + opt(v_3) \\ opt(v_5) \end{cases}$$

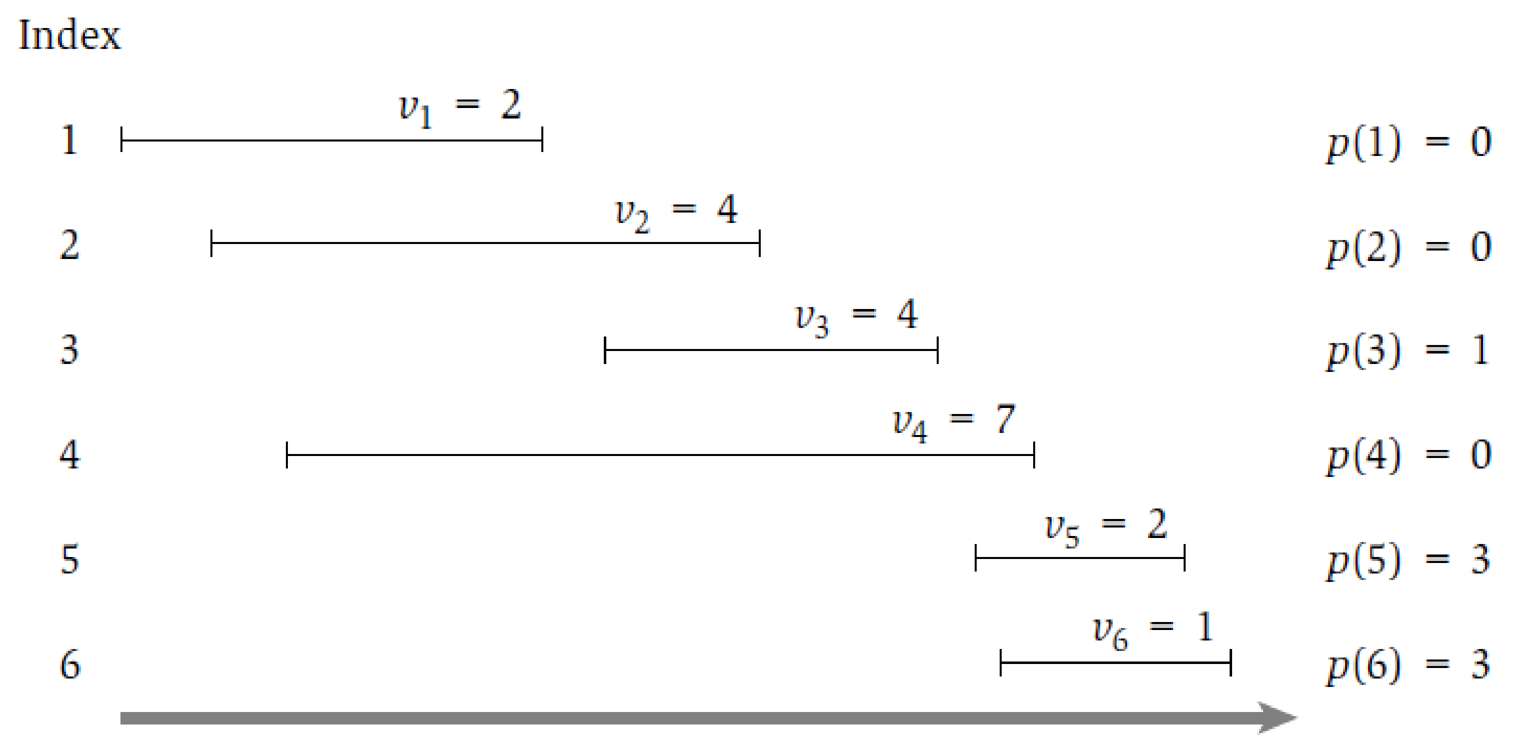
$$opt(v_4) = \max \begin{cases} 7 + 0 \\ opt(v_3) \end{cases}$$

$$opt(v_2) = \max \begin{cases} 4 + 0 \\ opt(v_1) \end{cases}$$

$$opt(v_5) = \max \begin{cases} 2 + opt(v_3) \\ opt(v_4) \end{cases}$$

$$opt(v_3) = \max \begin{cases} 4 + opt(v_1) \\ opt(v_2) \end{cases}$$

$$opt(v_1) = \max \begin{cases} 2 + 0 \\ 0 \end{cases}$$



If job is in optimal schedule,
opt(j) = v_j + opt(p(j))

If job is NOT in optimal schedule,
opt(j) = opt(j-1)

$$\mathrm{opt}(v_6) = \max \begin{cases} 1 + \mathrm{opt}(v_3) \\ \mathrm{opt}(v_5) \end{cases}$$

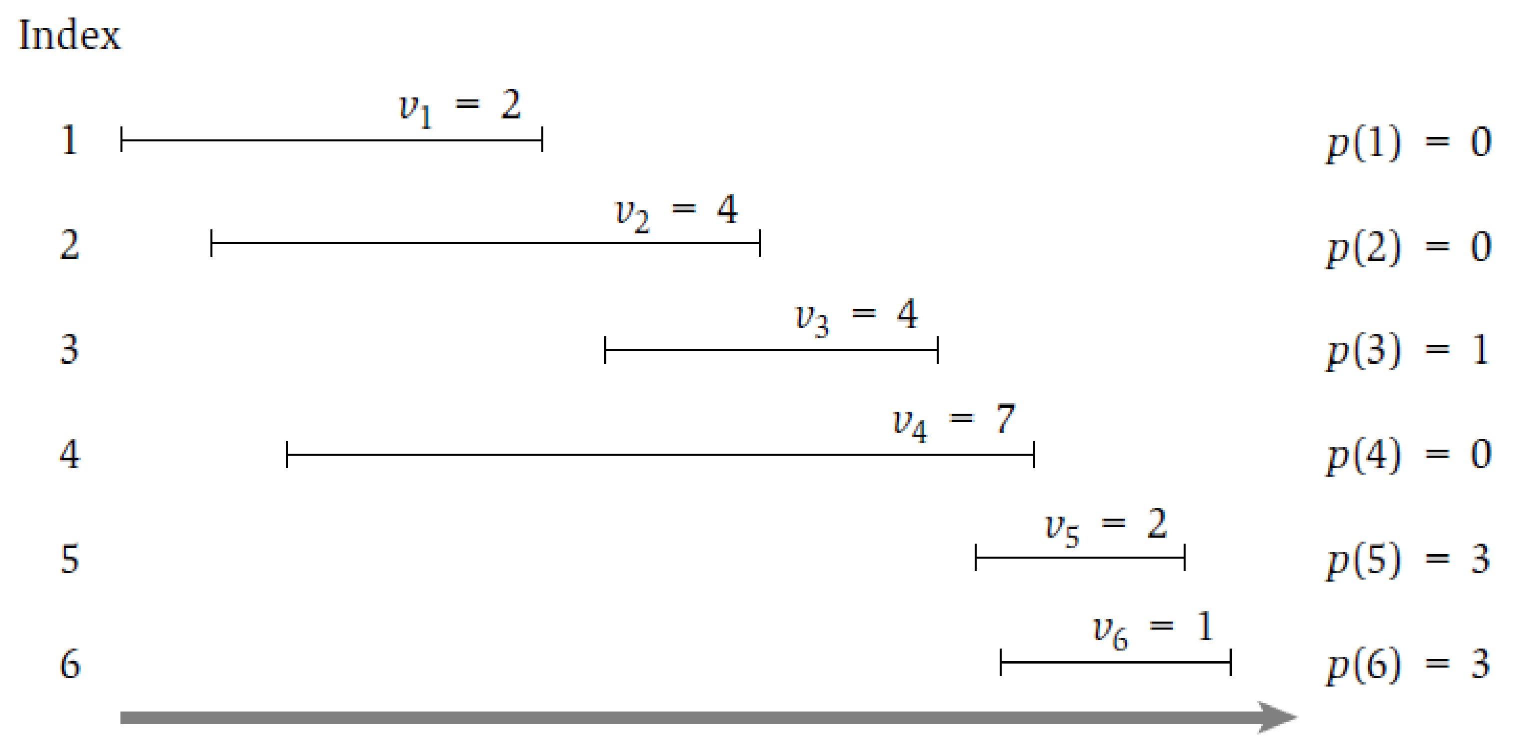
$$\mathrm{opt}(v_5) = \max \begin{cases} 2 + \mathrm{opt}(v_3) \\ \mathrm{opt}(v_4) \end{cases}$$

$$\mathrm{opt}(v_4) = \max \begin{cases} 7 + 0 \\ \mathrm{opt}(v_3) \end{cases}$$

$$\mathrm{opt}(v_3) = \max \begin{cases} 4 + \mathrm{opt}(v_1) \\ \mathrm{opt}(v_2) \end{cases}$$

$$\mathrm{opt}(v_2) = \max \begin{cases} 4 + 0 \\ \mathrm{opt}(v_1) \end{cases}$$

$$\mathrm{opt}(v_1) = 2$$

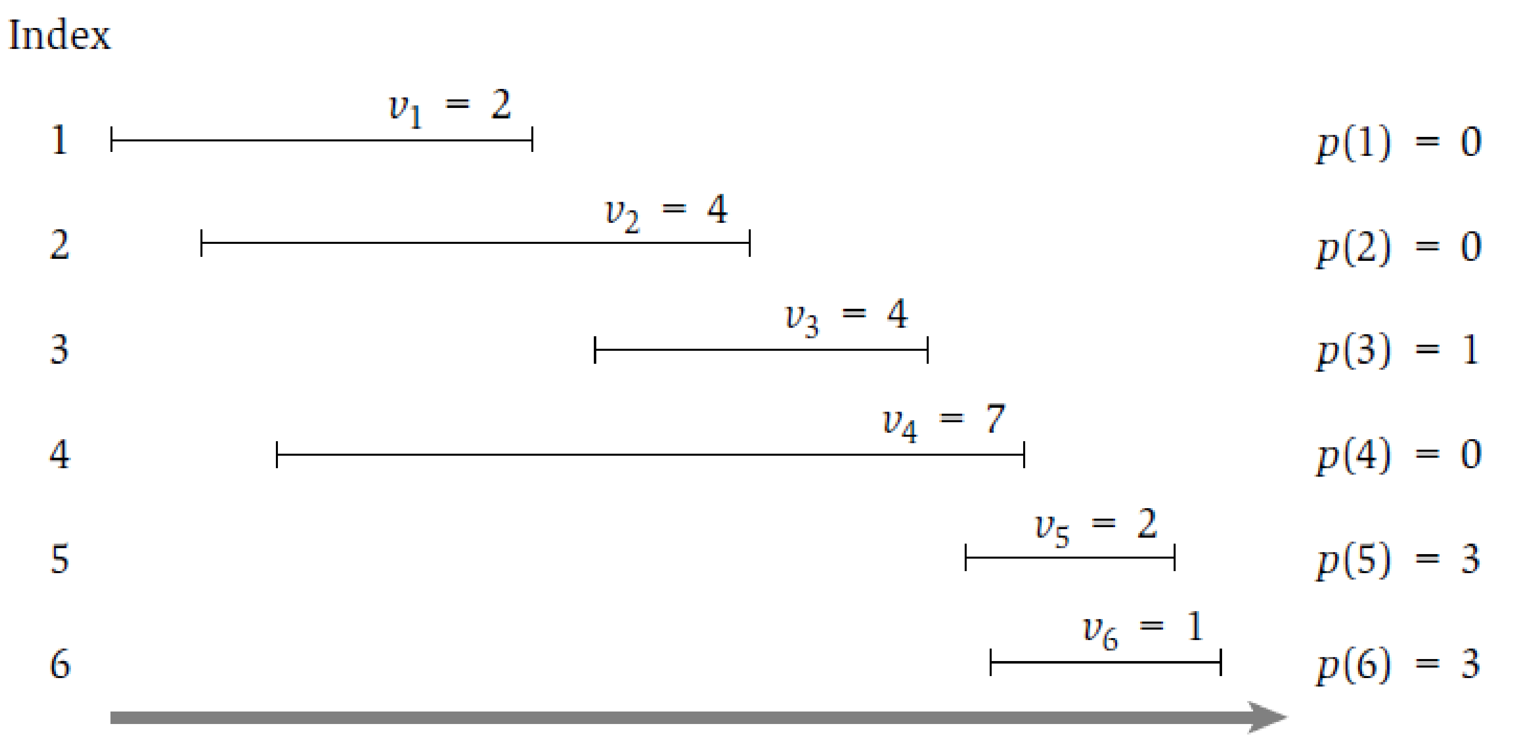


If job is in optimal schedule,
$opt(j) = v_j + opt(p(j))$

If job is NOT in optimal schedule,
$opt(j) = opt(j\text{-}1)$

$$opt(v_6) = \max \begin{cases} 1 + opt(v_3) \\ opt(v_5) \end{cases} \qquad opt(v_4) = \max \begin{cases} 7 + 0 \\ opt(v_3) \end{cases} \qquad opt(v_2) = 4$$

$$opt(v_5) = \max \begin{cases} 2 + opt(v_3) \\ opt(v_4) \end{cases} \qquad opt(v_3) = \max \begin{cases} 4 + opt(v_1) \\ opt(v_2) \end{cases} \qquad opt(v_1) = 2$$



If job is in optimal schedule,

$opt(j) = v_j + opt(p(j))$

If job is NOT in optimal schedule,

$opt(j) = opt(j-1)$

$$opt(v_6) = \max \begin{cases} 1 + opt(v_3) \\ opt(v_5) \end{cases}$$

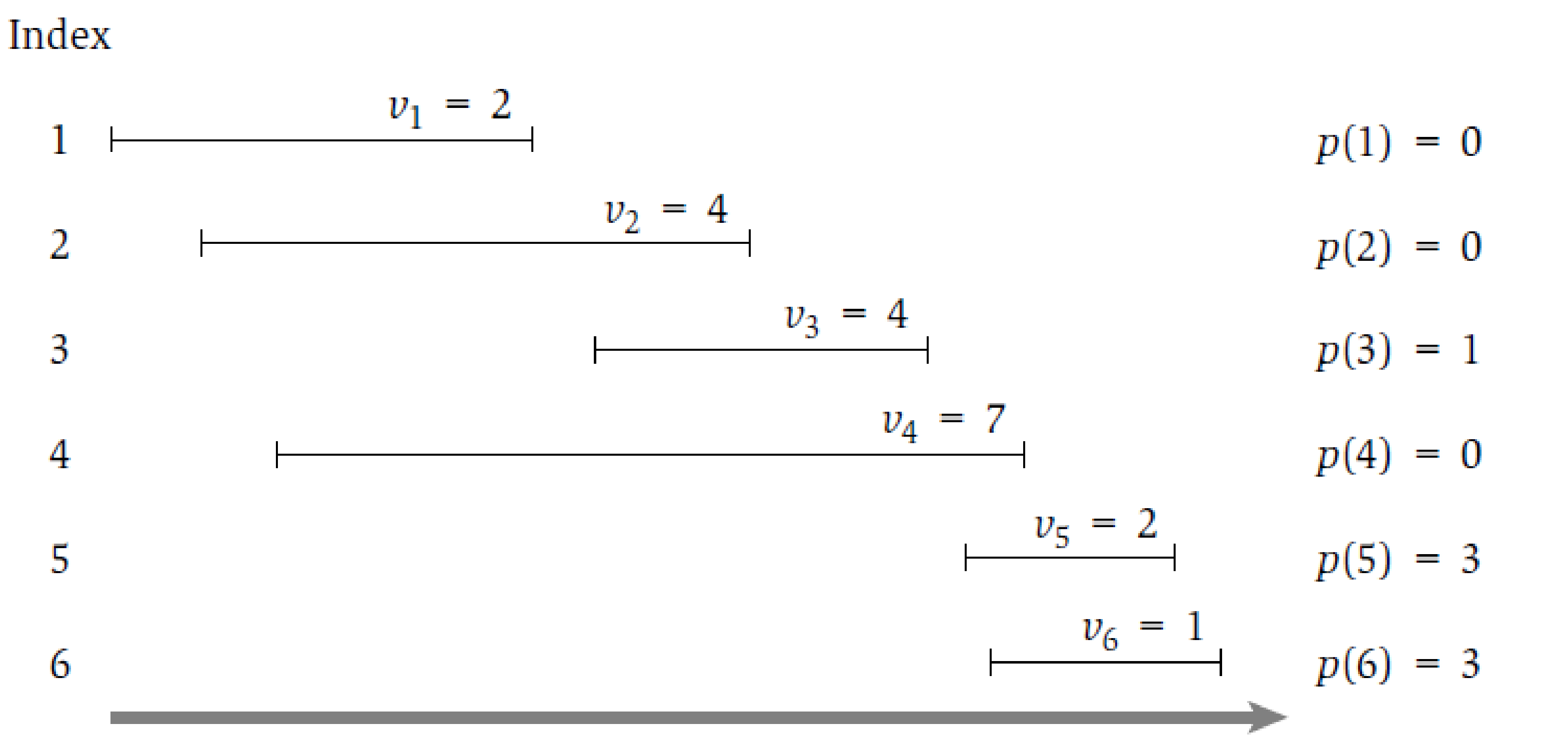
$$opt(v_4) = \max \begin{cases} 7 + 0 \\ opt(v_3) \end{cases}$$

$$opt(v_2) = 4$$

$$opt(v_5) = \max \begin{cases} 2 + opt(v_3) \\ opt(v_4) \end{cases}$$

$$opt(v_3) = 6$$

$$opt(v_1) = 2$$



If job is in optimal schedule,
$opt(j) = v_j + opt(p(j))$

If job is NOT in optimal schedule,
$opt(j) = opt(j-1)$

$$opt(v_6) = \max \begin{cases} 1 + opt(v_3) \\ opt(v_5) \end{cases}$$

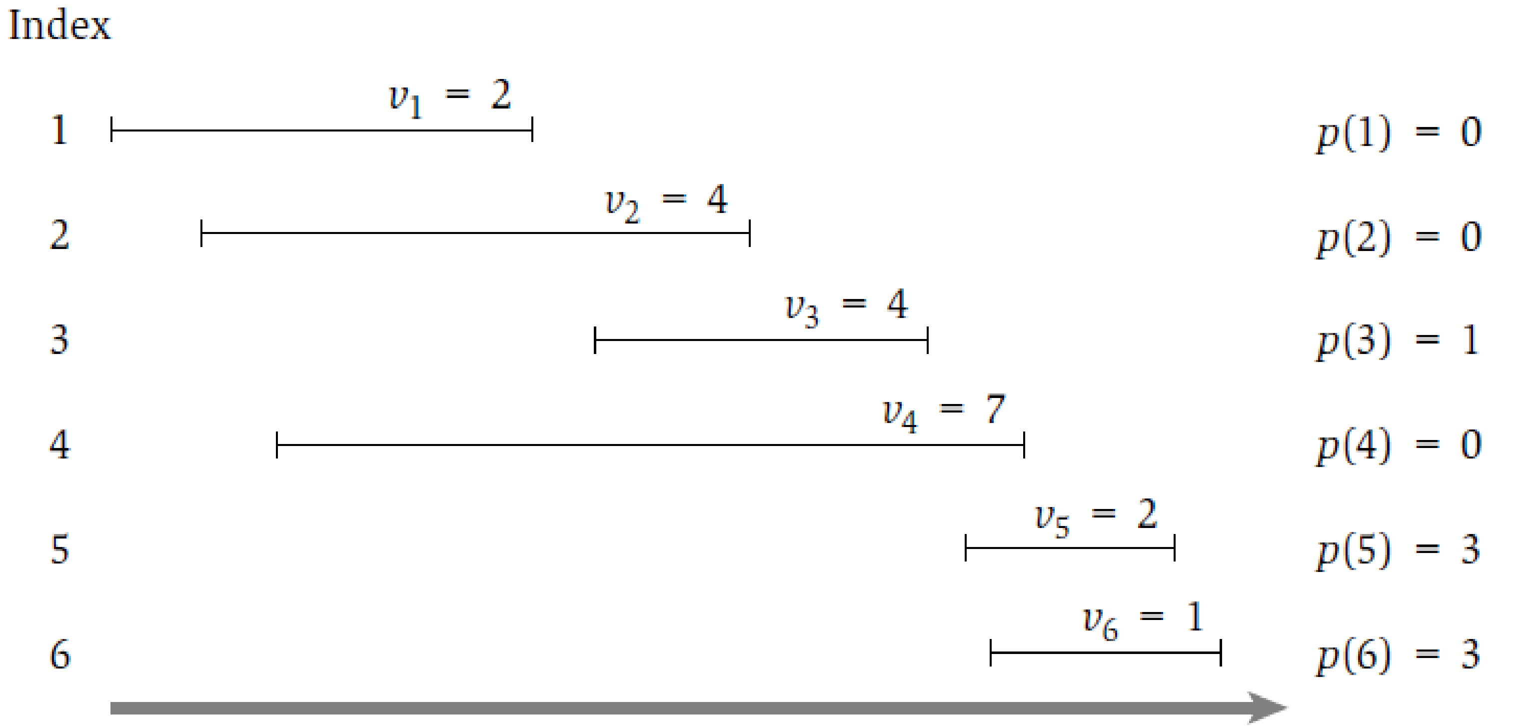
$opt(v_4) = 7$

$opt(v_2) = 4$

$$opt(v_5) = \max \begin{cases} 2 + opt(v_3) \\ opt(v_4) \end{cases}$$

$opt(v_3) = 6$

$opt(v_1) = 2$



If job is in optimal schedule,
$opt(j) = v_j + opt(p(j))$

If job is NOT in optimal schedule,
$opt(j) = opt(j-1)$

$$opt(v_6) = \max \begin{cases} 1 + opt(v_3) \\ opt(v_5) \end{cases}$$

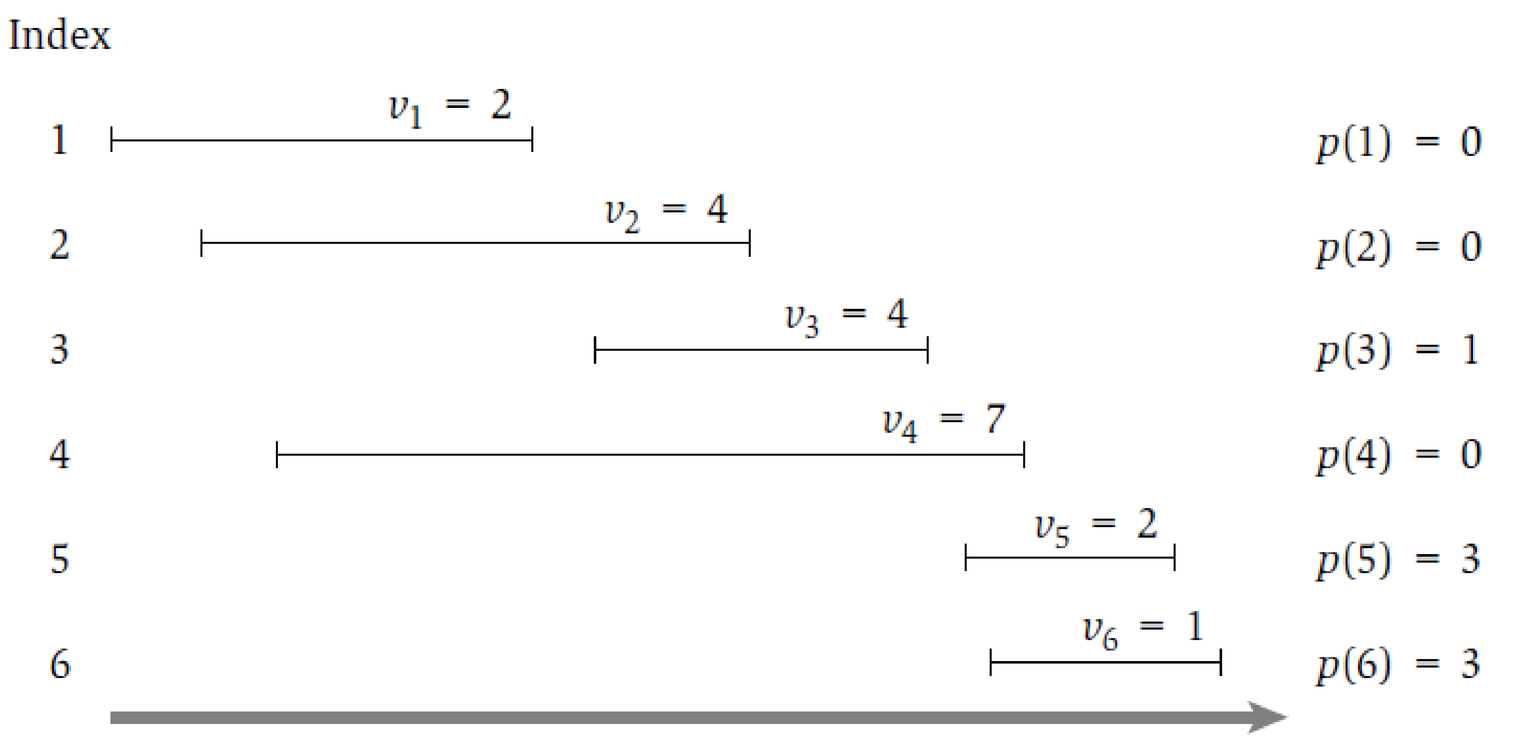
$opt(v_4) = 7$

$opt(v_2) = 4$

$opt(v_5) = 8$

$opt(v_3) = 6$

$opt(v_1) = 2$



If job is in optimal schedule,
$opt(j) = v_j + opt(p(j))$

If job is NOT in optimal schedule,
$opt(j) = opt(j-1)$

$opt(v_6) = 7$

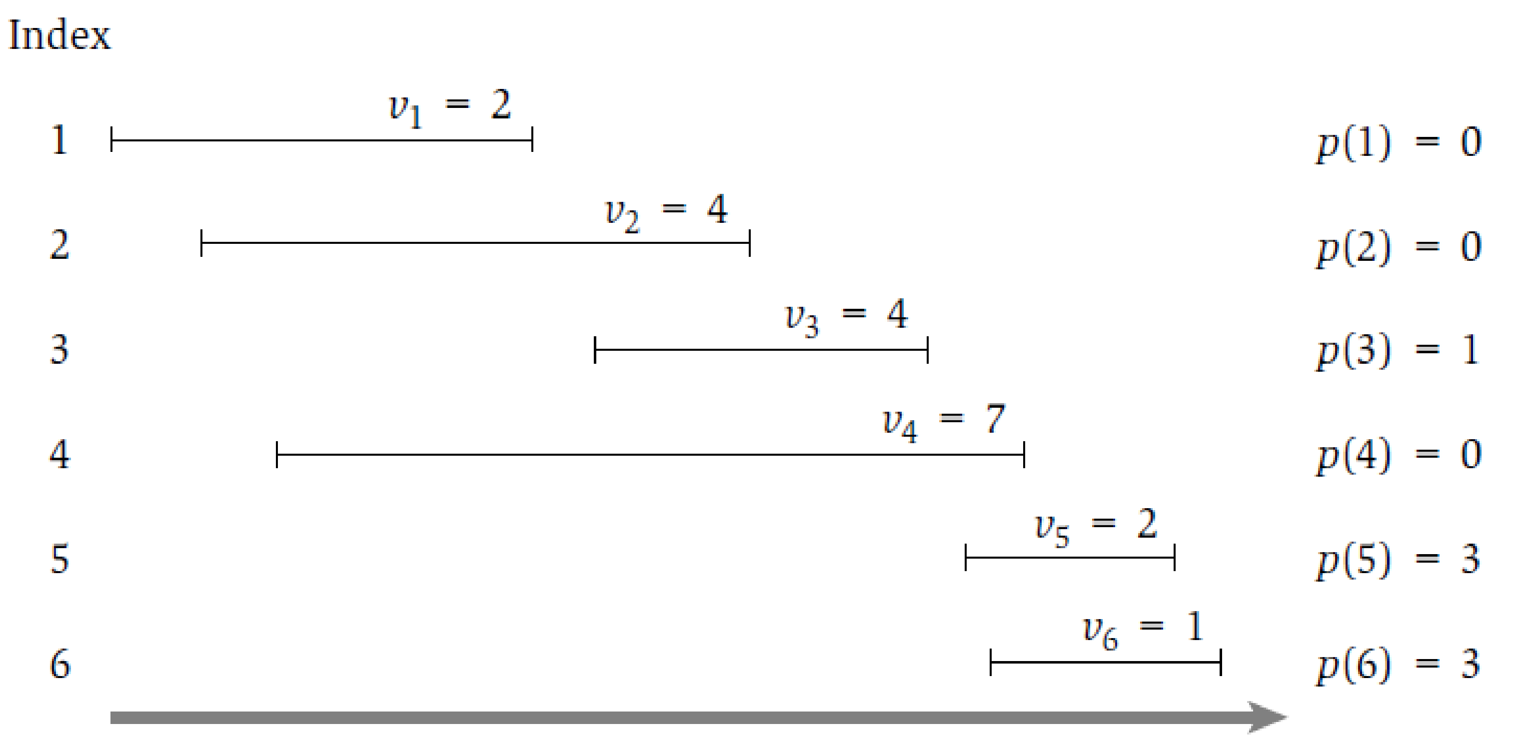
$opt(v_5) = 8$

$opt(v_4) = 7$

$opt(v_3) = 6$

$opt(v_2) = 4$

$opt(v_1) = 2$



If job is in optimal schedule,

$\text{opt}(j) = v_j + \text{opt}(p(j))$

If job is NOT in optimal schedule,

$\text{opt}(j) = \text{opt}(j\text{-}1)$

$\text{opt}(v_6) = 7$ $\text{opt}(v_4) = 7$ $\text{opt}(v_2) = 4$

$\text{opt}(v_5) = 8$ $\text{opt}(v_3) = 6$ $\text{opt}(v_1) = 2$

The maximum value to be obtained is 8 which is gotten if one takes request 5 because one will be able to take requests 3 and 1 as well.

Weighted Interval Scheduling Problem

Memoized Recursive Algorithm

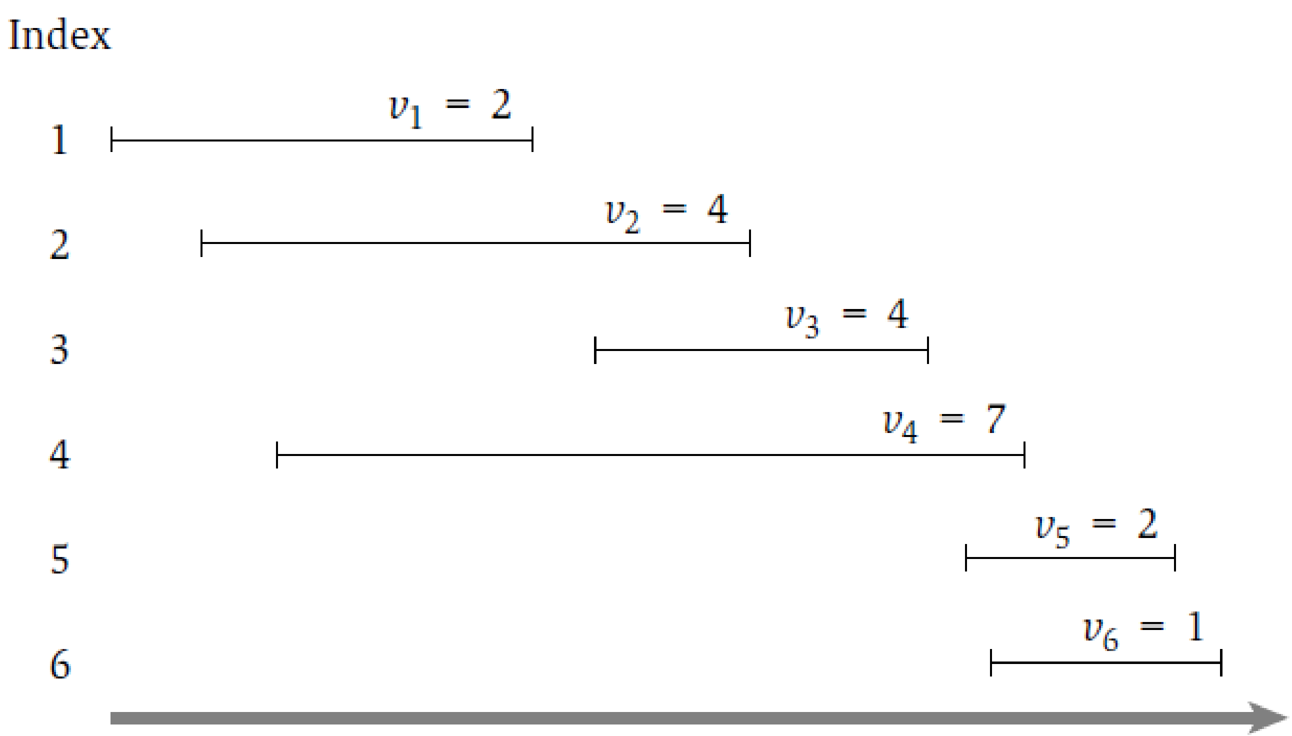
Initialise array M of size n such that $0 \leq j < n$

```
memoized-WIS(j) {
    if (j == 0) return 0                            // basis case - no requests
    else if (M[j] has been computed) return M[j]
    else {
        leaveWeight = memoized-WIS(j-1)             // total weight if we leave j
        takeWeight = v[j] + memoized-WIS(p[j])      // total weight if we take j
        if ( leaveWeight > takeWeight ) {
            M[j] = leaveWeight                      // better to leave j

        } else {
            M[j] = takeWeight                       // better to take j

        }
        return M[j]                                 // return final weight
    }
}
```

The memoized version runs in O(n) time.



$p(1) = 0$
$p(2) = 0$
$p(3) = 1$
$p(4) = 0$
$p(5) = 3$
$p(6) = 3$

If job is in optimal schedule,
$opt(j) = v_j + opt(p(j))$

If job is NOT in optimal schedule,
$opt(j) = opt(j\text{-}1)$

The memorized version gets rid of repeated function calls

$$opt(v_6) = \max \begin{cases} 1 + \cancel{opt(v_3)} \\ \cancel{opt(v_5)} \end{cases}$$

$$opt(v_5) = \max \begin{cases} 2 + \cancel{opt(v_3)} \\ \cancel{opt(v_4)} \end{cases}$$

$$opt(v_4) = \max \begin{cases} 7 + 0 \\ \cancel{opt(v_3)} \end{cases}$$

$$opt(v_3) = \max \begin{cases} 4 + \cancel{opt(v_1)} \\ \cancel{opt(v_2)} \end{cases}$$

$$opt(v_2) = \max \begin{cases} 4 + 0 \\ \cancel{opt(v_1)} \end{cases}$$

$$opt(v_1) = \max \begin{cases} 2 + 0 \\ 0 \end{cases}$$