Dynamic programming

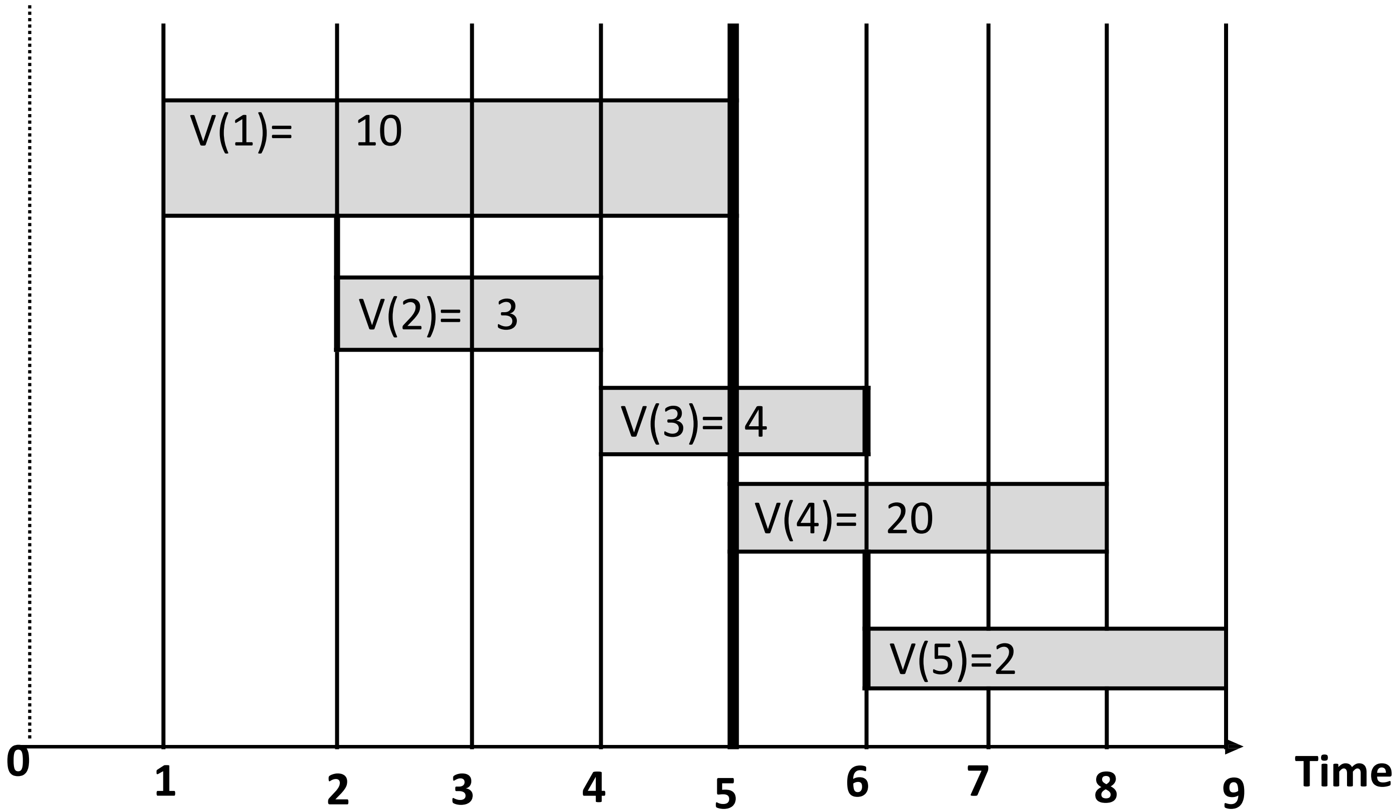
is a method used in mathematics and computer science to solve complex problems by breaking them down into simpler subproblems. By solving each subproblem only once and storing the results, it avoids redundant computations, leading to more efficient solutions for a wide range of problems.

Weighted Interval Scheduling

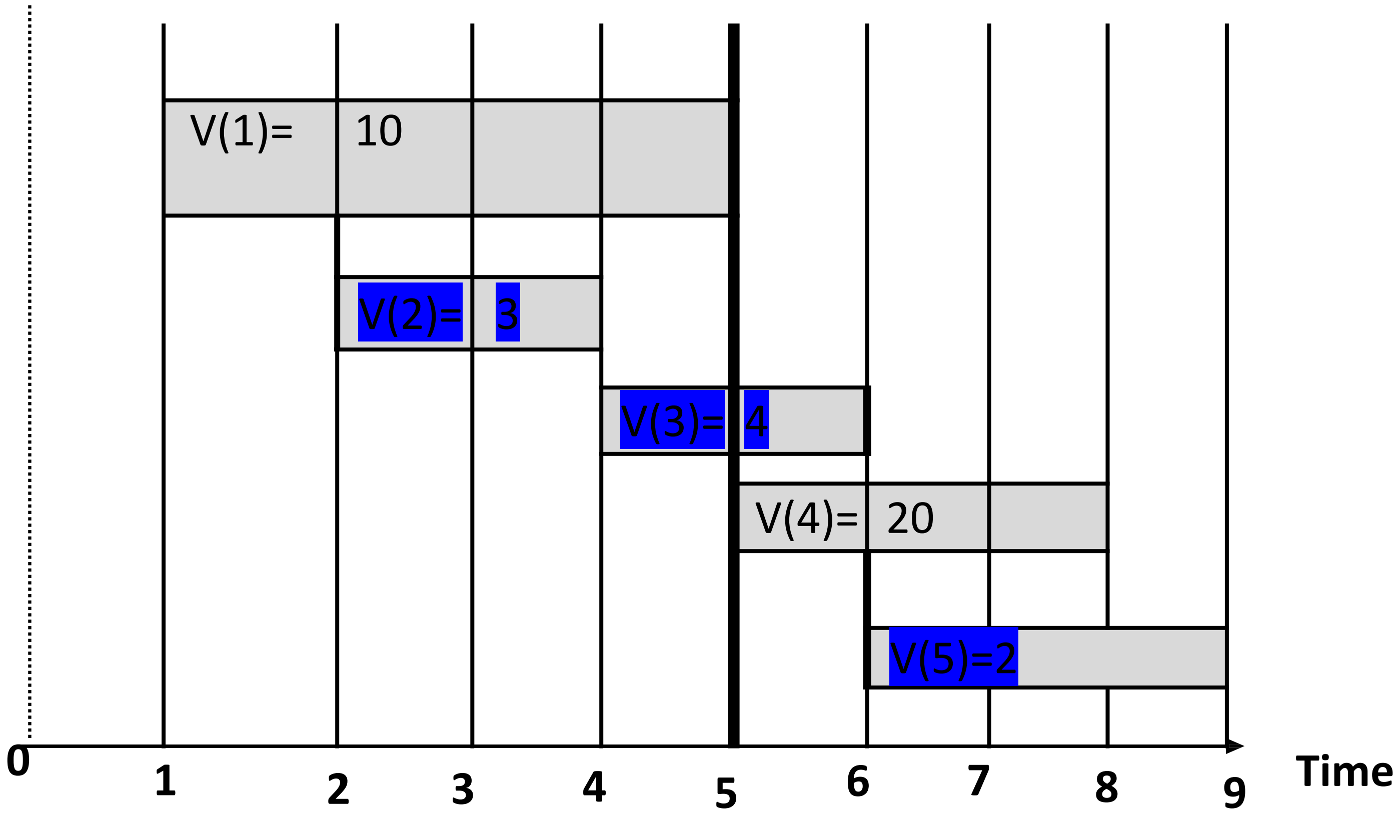
Problem and Goal

- Weighted interval scheduling problem.
  - Job j starts at s_j, finishes at f_j, and has weight or value v_j .
  - Two jobs compatible if they don't overlap.
  - Goal: find maximum weight subset of mutually compatible jobs.

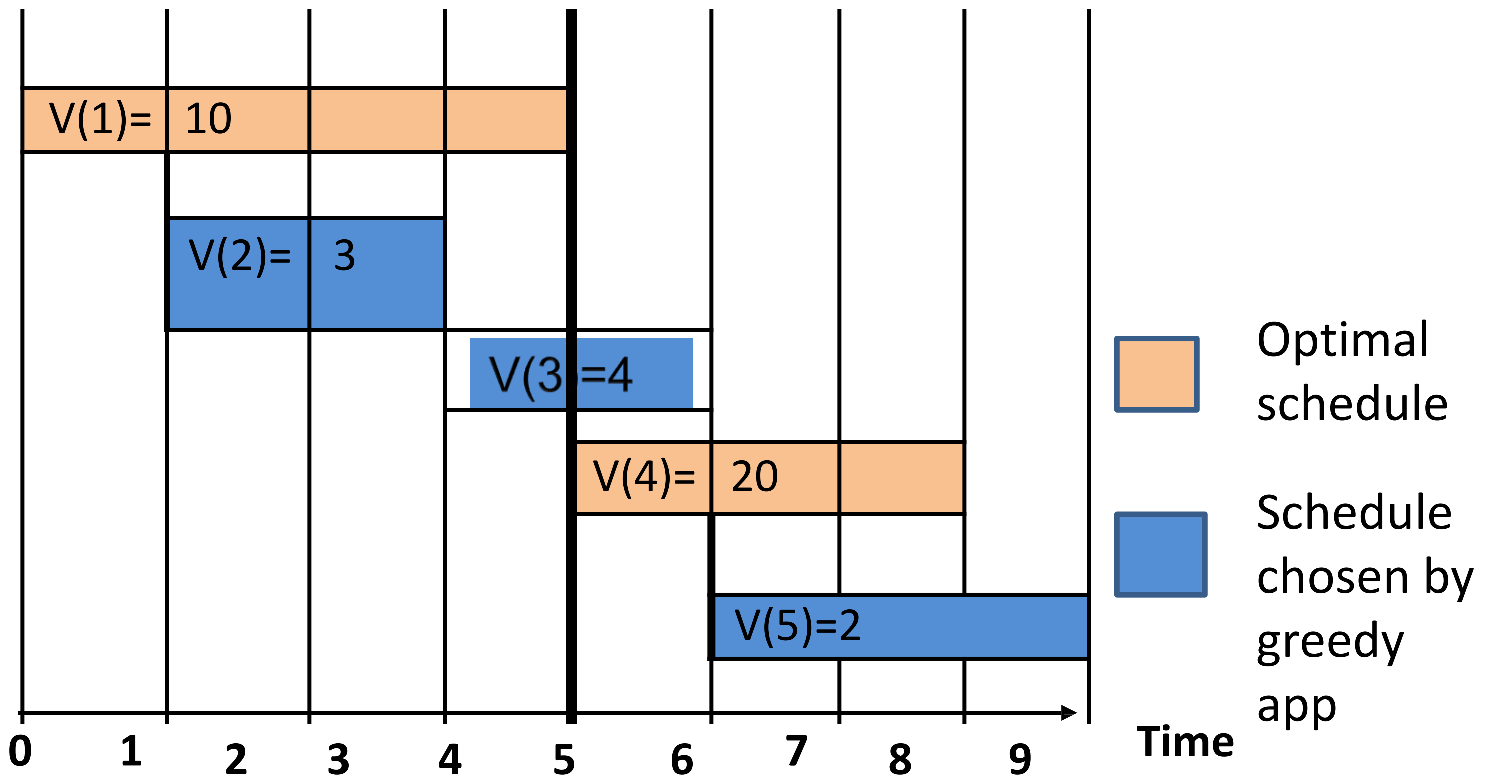
Greedy approach



Greedy approach



Greedy does not work



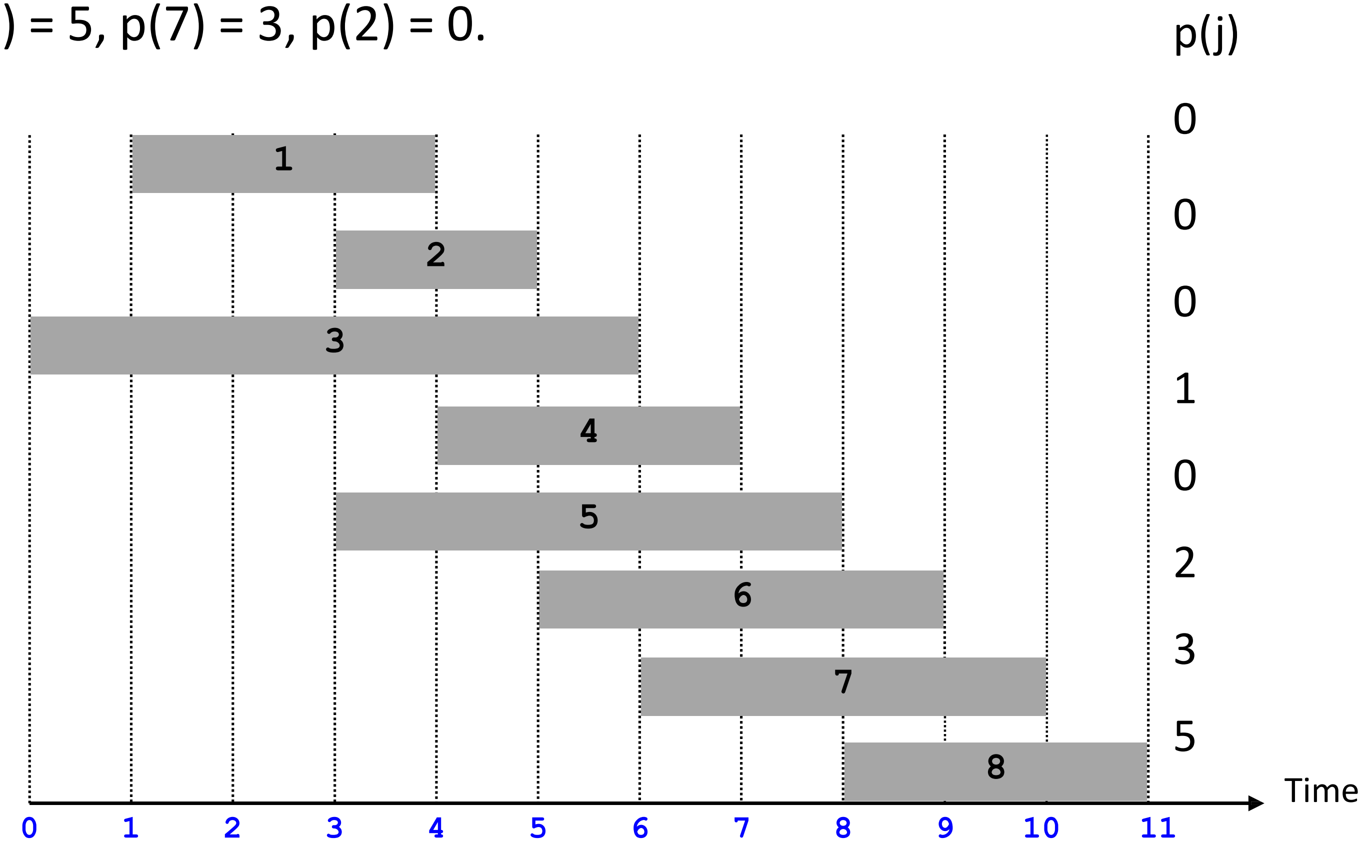
Greedy approach takes job 2, 3 and 5 as best schedule and makes profit of 9. While optimal schedule is job 1 and job4 making profit of 30 (10+20). Hence greedy will not work

Weighted Interval Scheduling

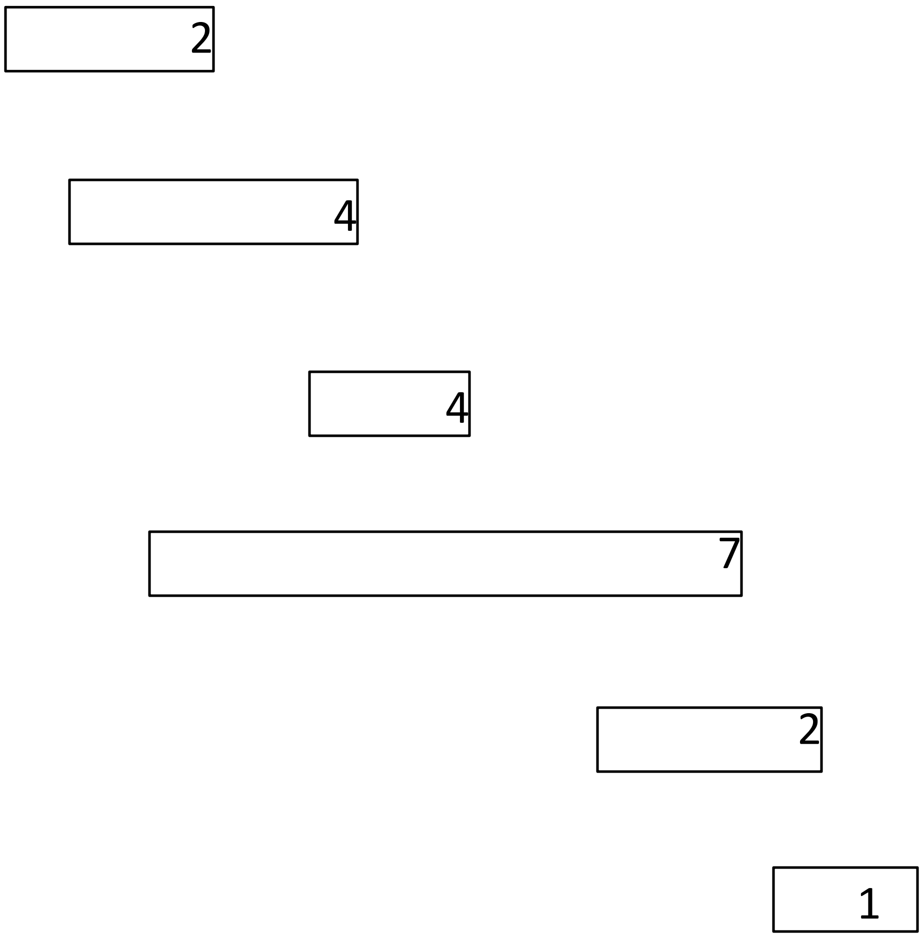
Notation. Order jobs by **finishing** time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Def. **p(j)** = ***largest*** index $i < j$ such that job i is compatible with j.

Ex: p(8) = 5, p(7) = 3, p(2) = 0.



p(j): the largest i < j such that f[i] <= s[j]



j	p(j)
1	0
2	0
3	1
4	0
5	3
6	3

Optimal solution

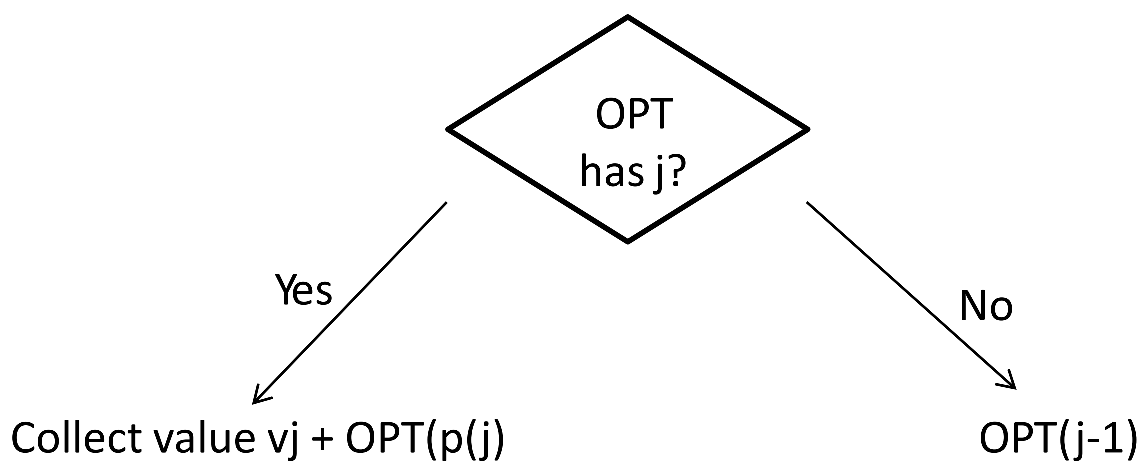
- <u>Notation.</u> ***OPT(j)*** = value of optimal solution to the problem consisting of job requests ***1, 2,…, j***.

Case 1: OPT selects job j

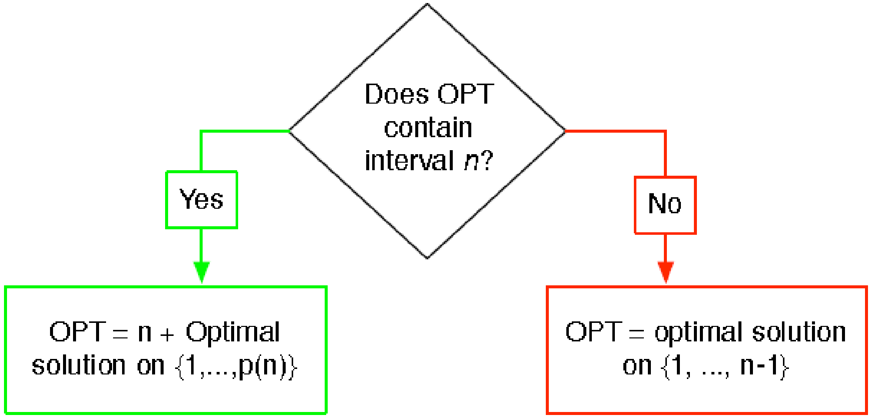
- collect profit v_j
- cant't use incompatible jobs {p(j)+1,p(j)+2,.....j-1}
- must include optimal solution to problem consisting of remaining compatible jobs 1,2,.....,p(j)

Case 2:OPT does not select job j.

- must include optimal solution to problem consisting of remaining compatible jobs 1,2,....,j-1



Recurrence relation using Optimal schedule



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

```
dp = [0] * n
```

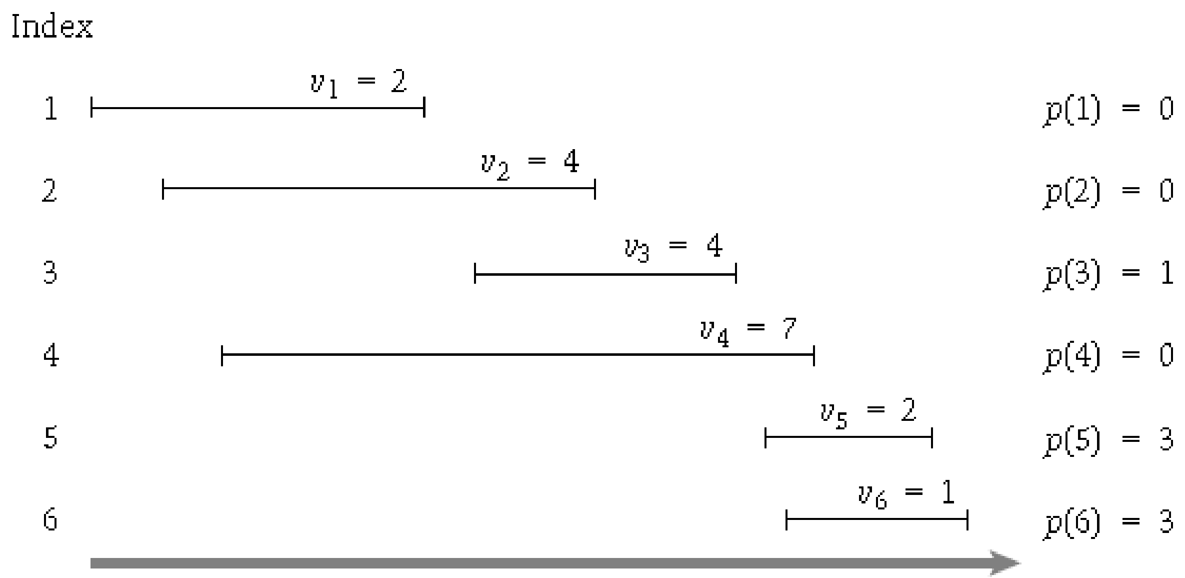
Algorithm for Recurrence relation

incl = V i + dp[index of Compatible prev job whose end time < start time of i] ?
excl = dp[i-1]
dp[i] = max(incl, excl)

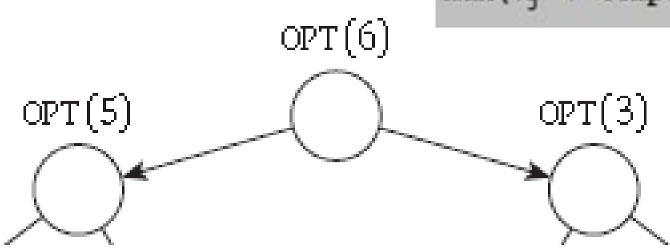
$$OPT(j) = \max \begin{cases} v_j + OPT(p(j)) & j \text{ in OPT solution} \\ OPT(j-1) & j \text{ not in solution} \\ 0 & j = 0 \end{cases}$$

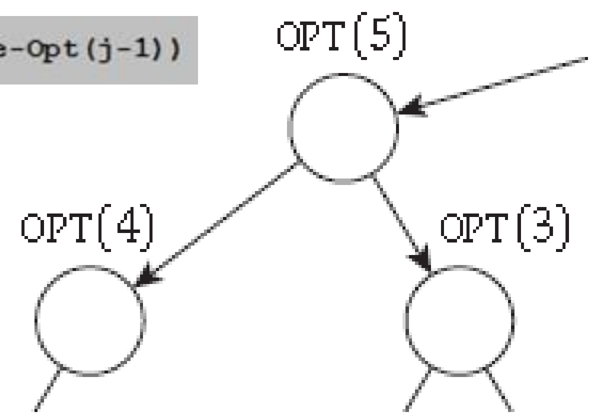
```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

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

Compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

Algorithm for Recurrence relation- Example

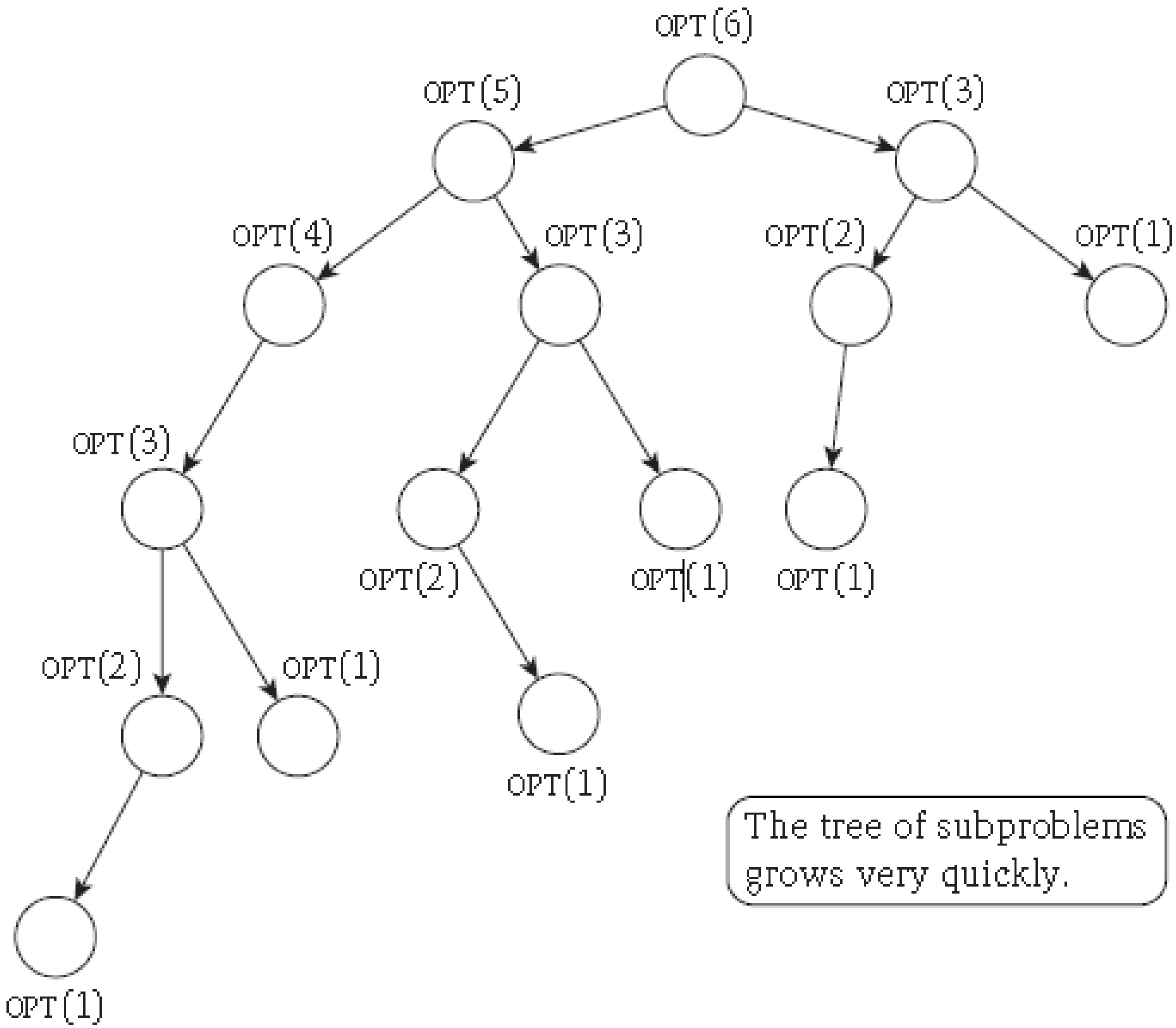


`max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))`



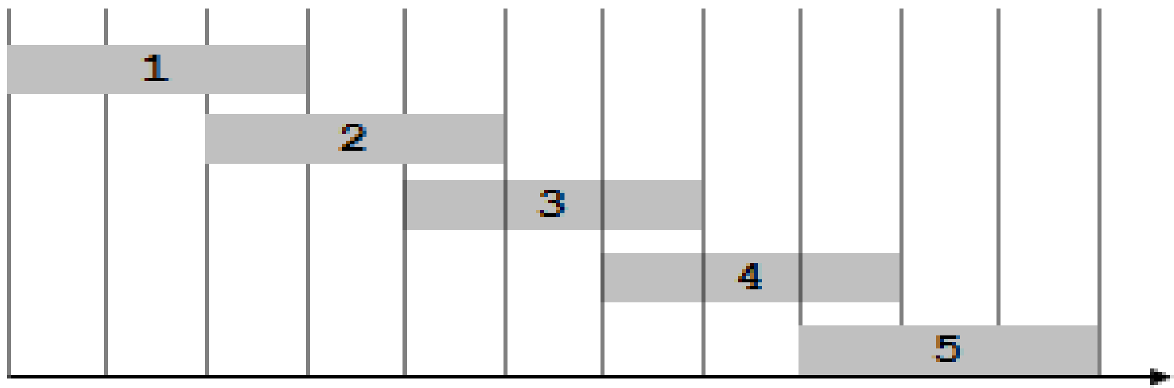


Algorithm for Recurrence relation- Example Brute force algorithm

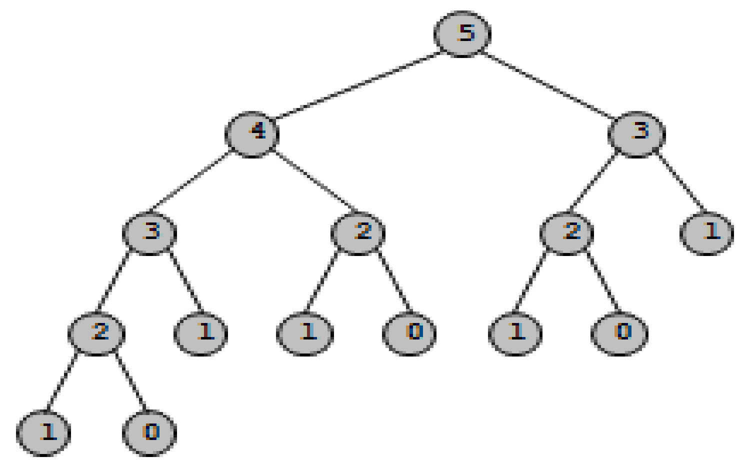


observation: Recursive algorithm fails spectacularly because of redundant sub-problem i.e Exponential algorithm

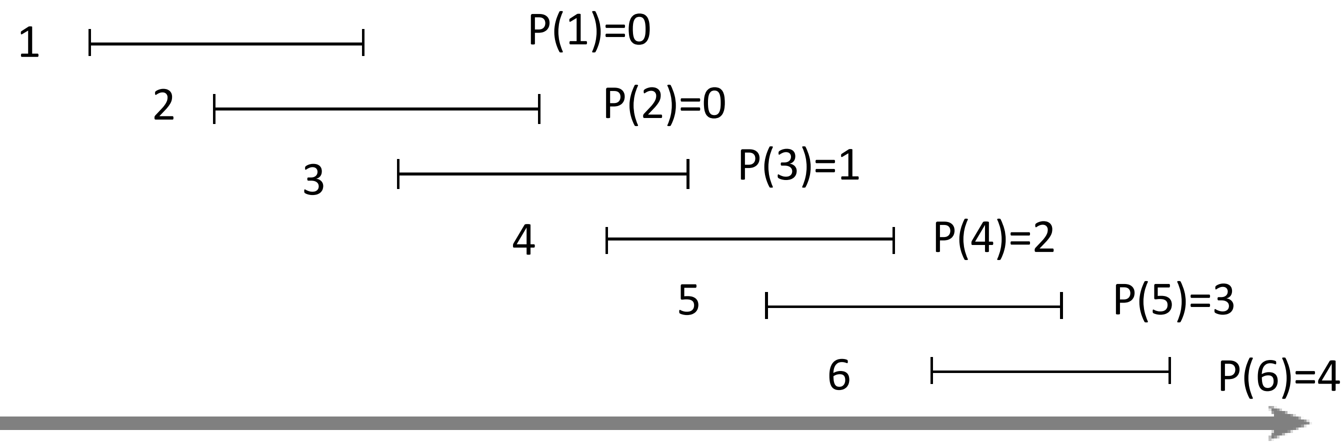
Ex. Number of recursive calls for family of layered instances grow like fibonacci sequence



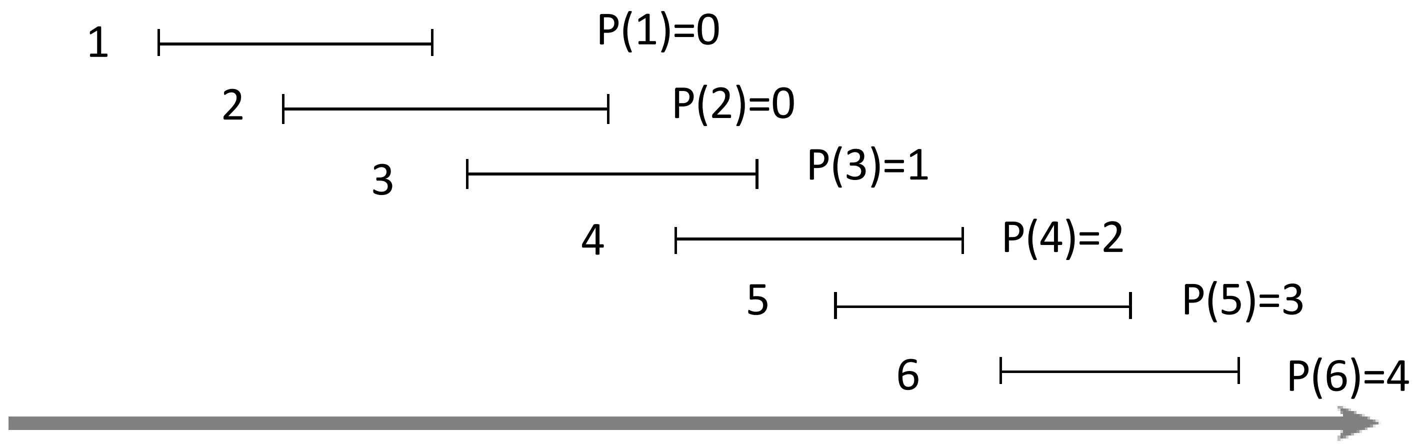
$p(1) = 0,\ p(j) = j-2$

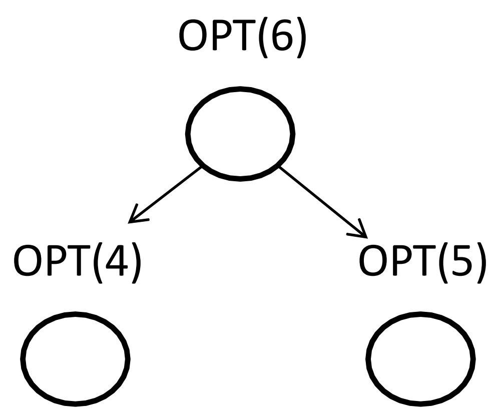


Algorithm for Recurrence relation- Example

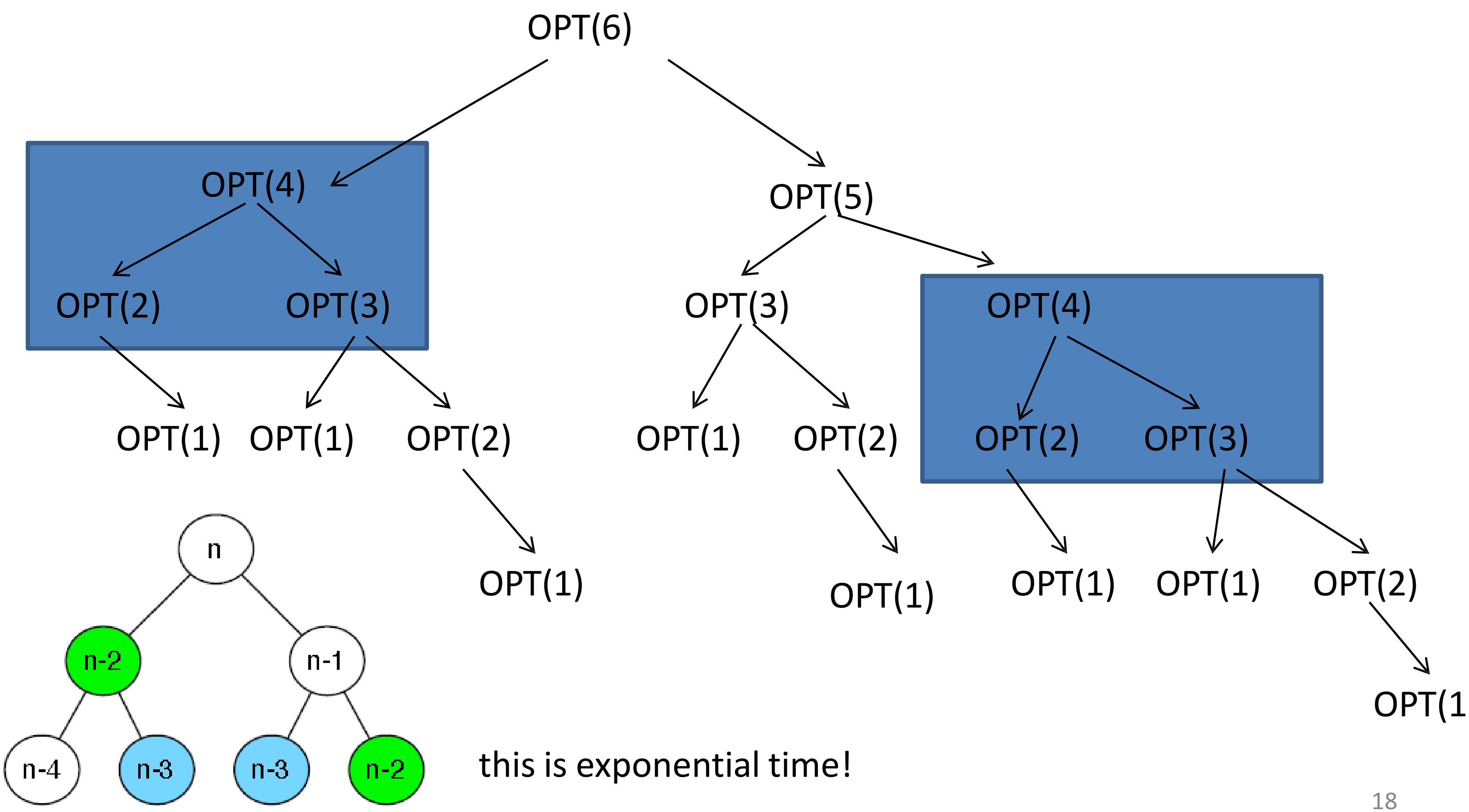


Algorithm for Recurrence relation- Example





Algorithm for Recurrence relation- Example



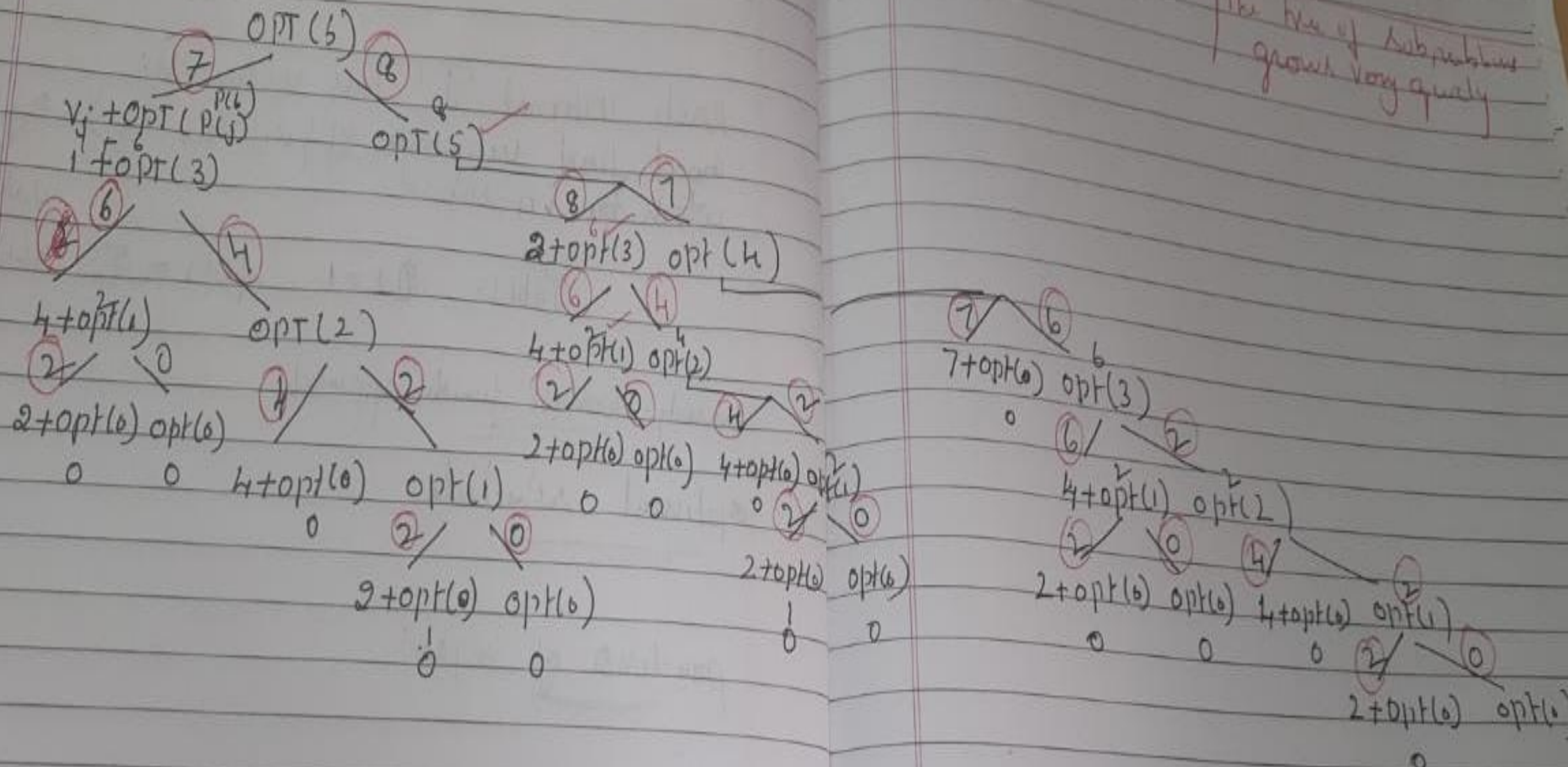
this is exponential time!

Memoization

② compute ... to finish time

② compute p (ordering)

③ compute OPT



The tree of subproblems grows very quickly

Memoizing the Recursion

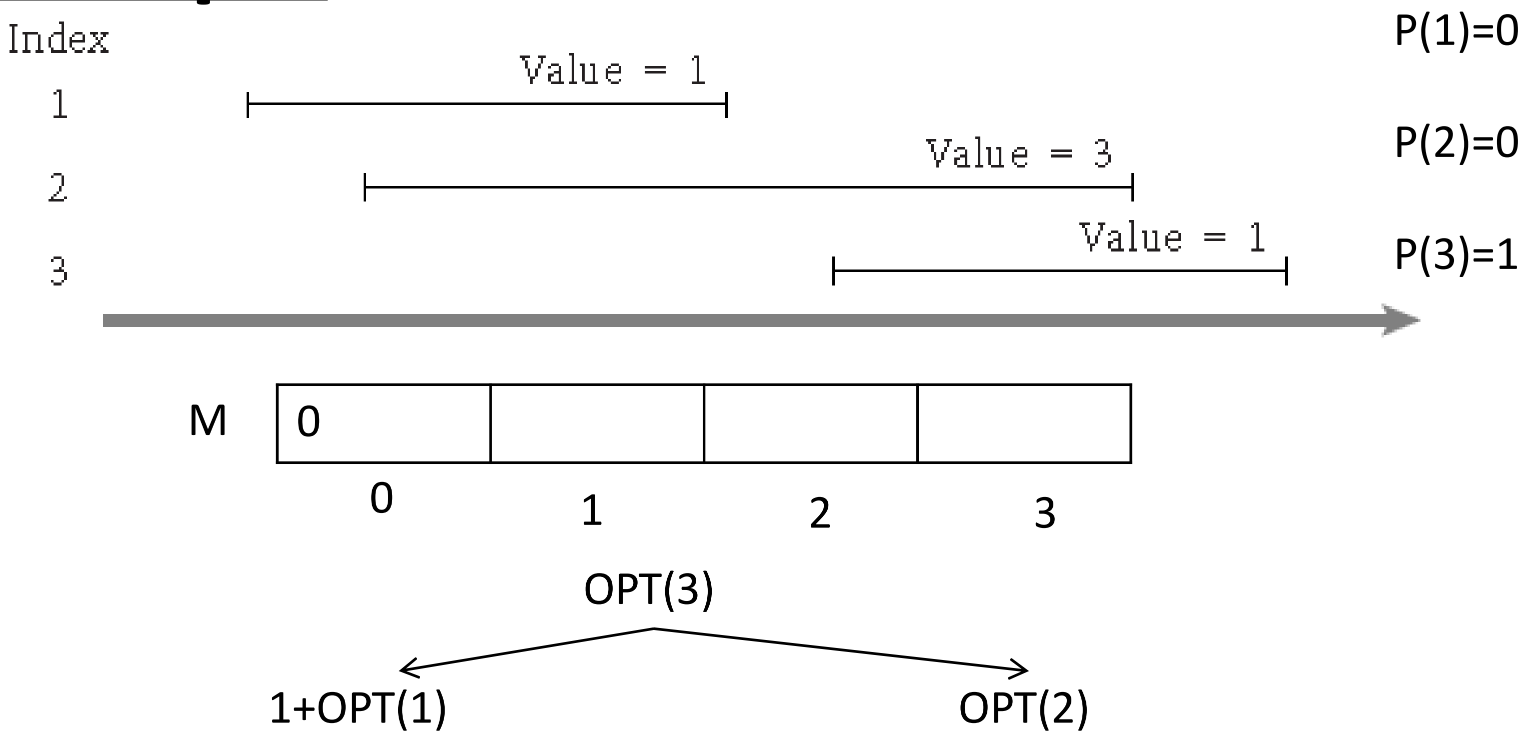
Problem: Repeatedly solving the same subproblem.

Solution: Save the answer for each subproblem as you compute it.

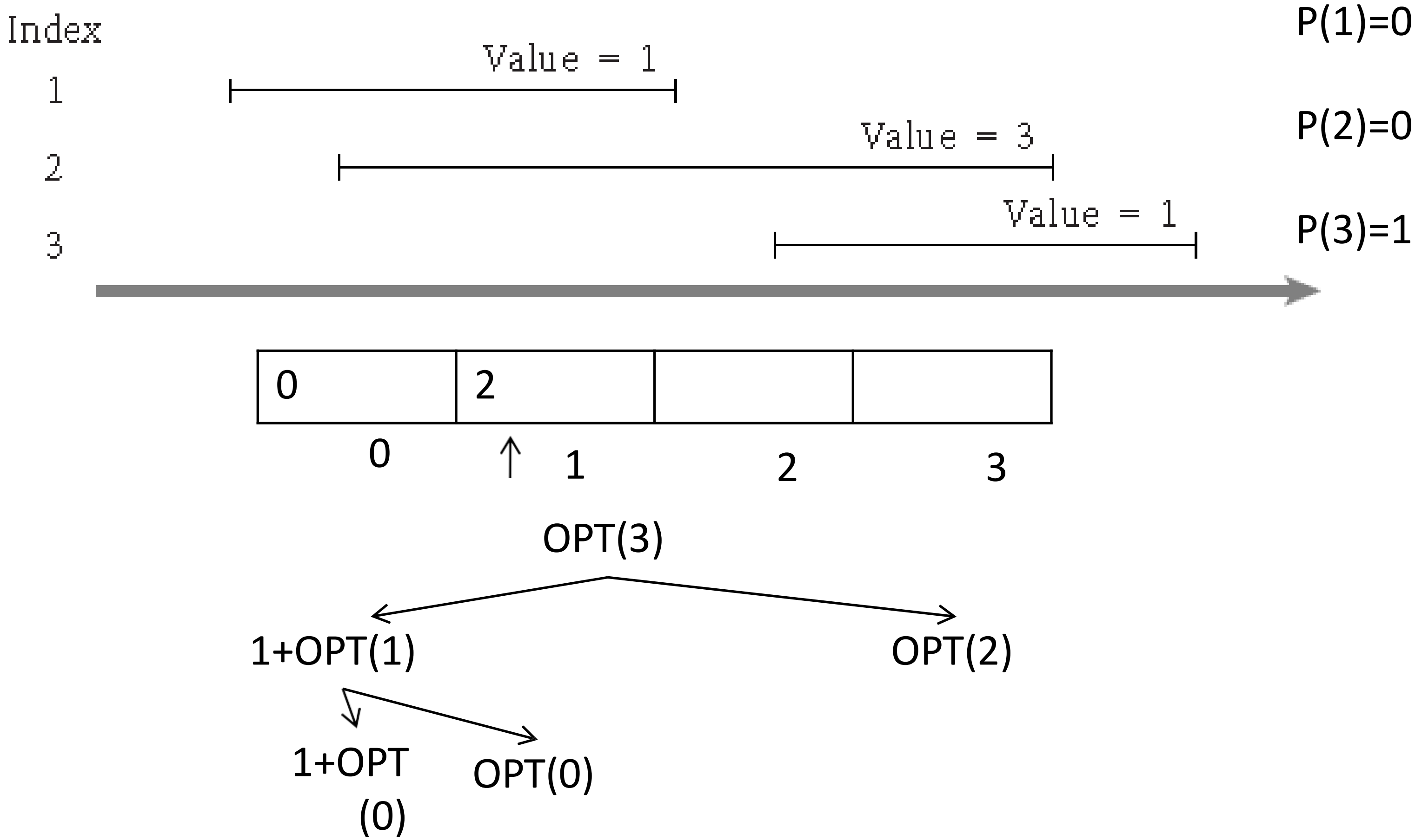
When you compute $OPT(j)$, save the value in a global array M.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty        global array
M[0] = 0

M-Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

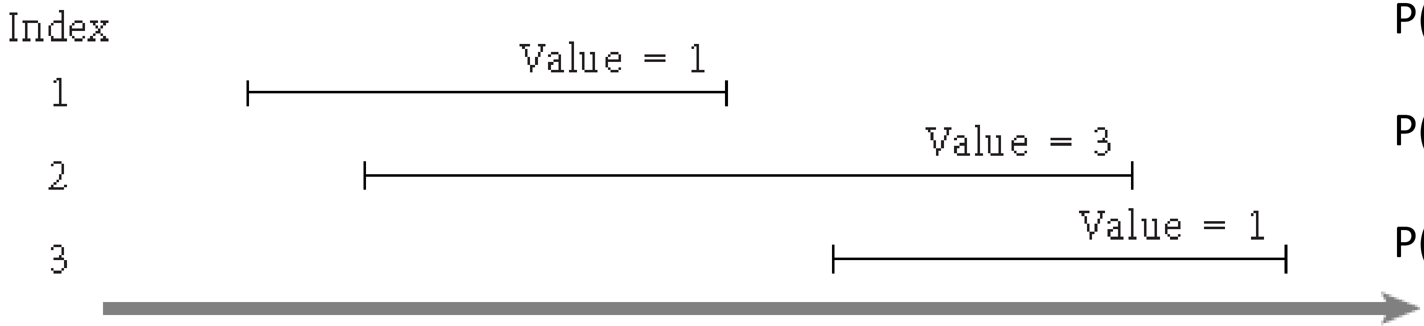
Memoizing the Recursion-Example



Memoizing the Recursion-Example



Memoizing the Recursion-Example



P(1)=0

P(2)=0

P(3)=1

0	2	3	3
0	1	2	3

OPT(3)

1+OPT(1)   OPT(2)

1+OPT (0)   OPT(0)   3+OPT(0)   OPT(1)

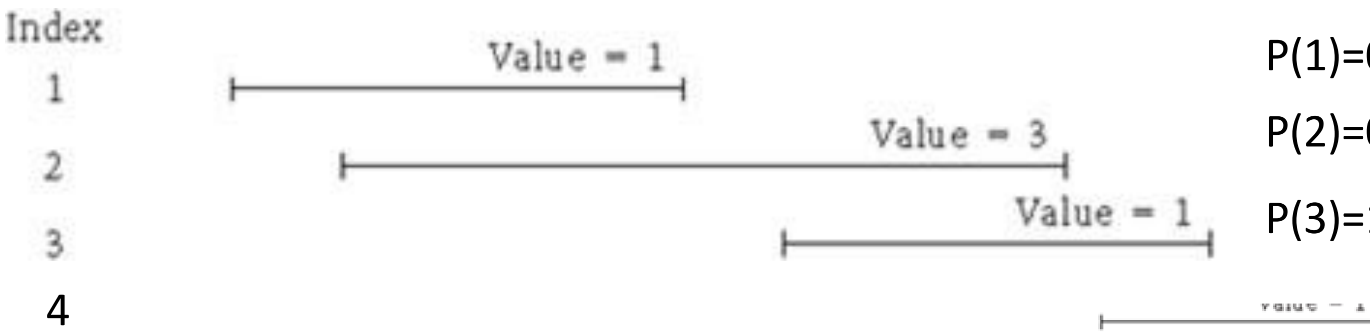
3+0=3   2

Memoizing the Recursion- Running time

- Fill in 1 array entry for every two calls to M-Compute-Opt. => O(n)

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty        global array
M[0] = 0

M-Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

Solve??



P(1)=0

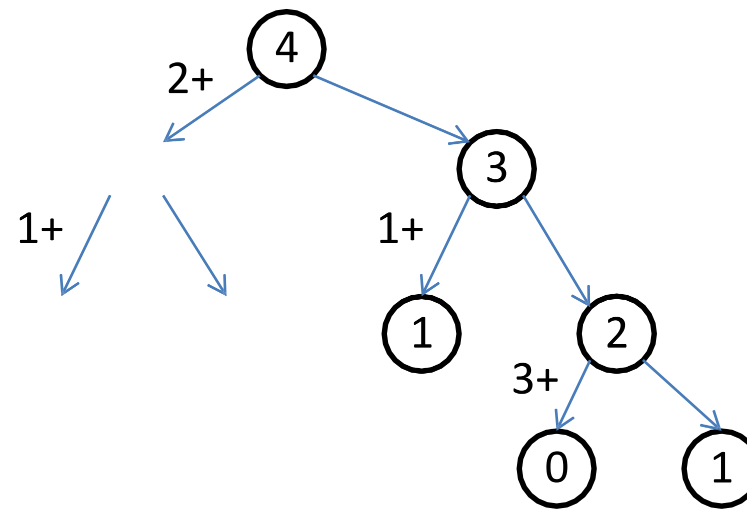
P(2)=0

P(3)=1

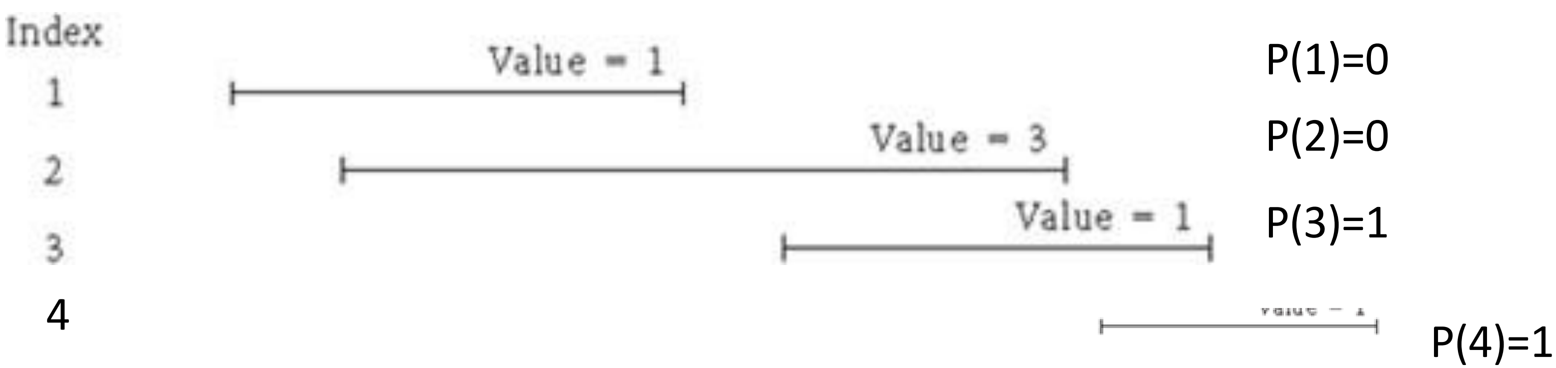
P(4)=2

Value =2

0 1 2 3 4

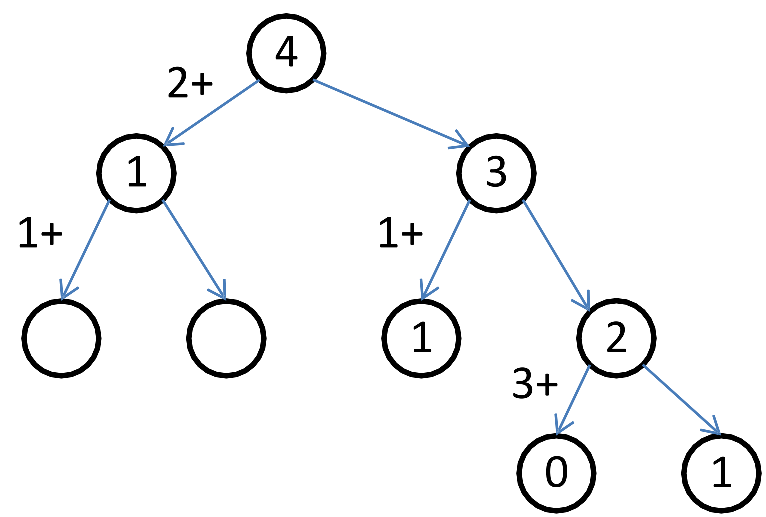


Solve??



Value =2

0	1	3	3	3
0	1	2	3	4



Iterative procedure

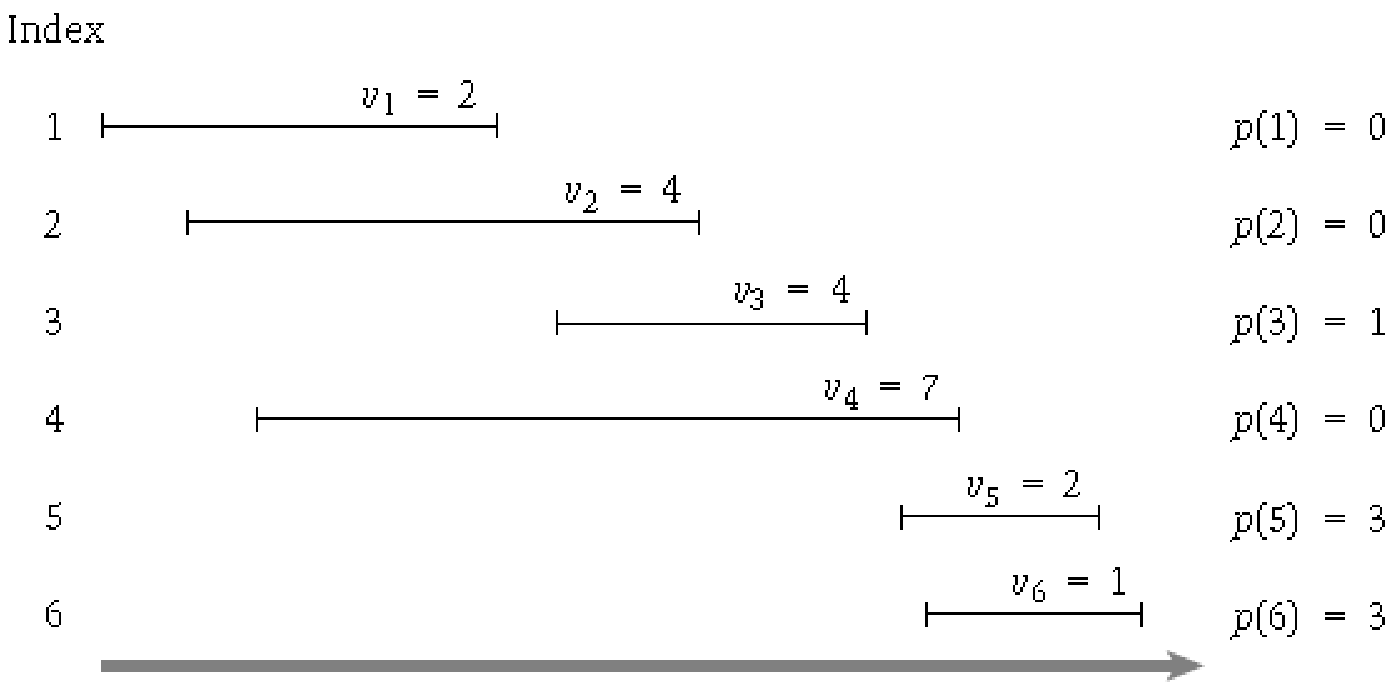
Input: n, $s_1, \ldots, s_n$, $f_1, \ldots, f_n$, $v_1, \ldots, v_n$

Sort jobs by finish times so that $f_1 \le f_2 \le \ldots \le f_n$.

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])
}
```

Example



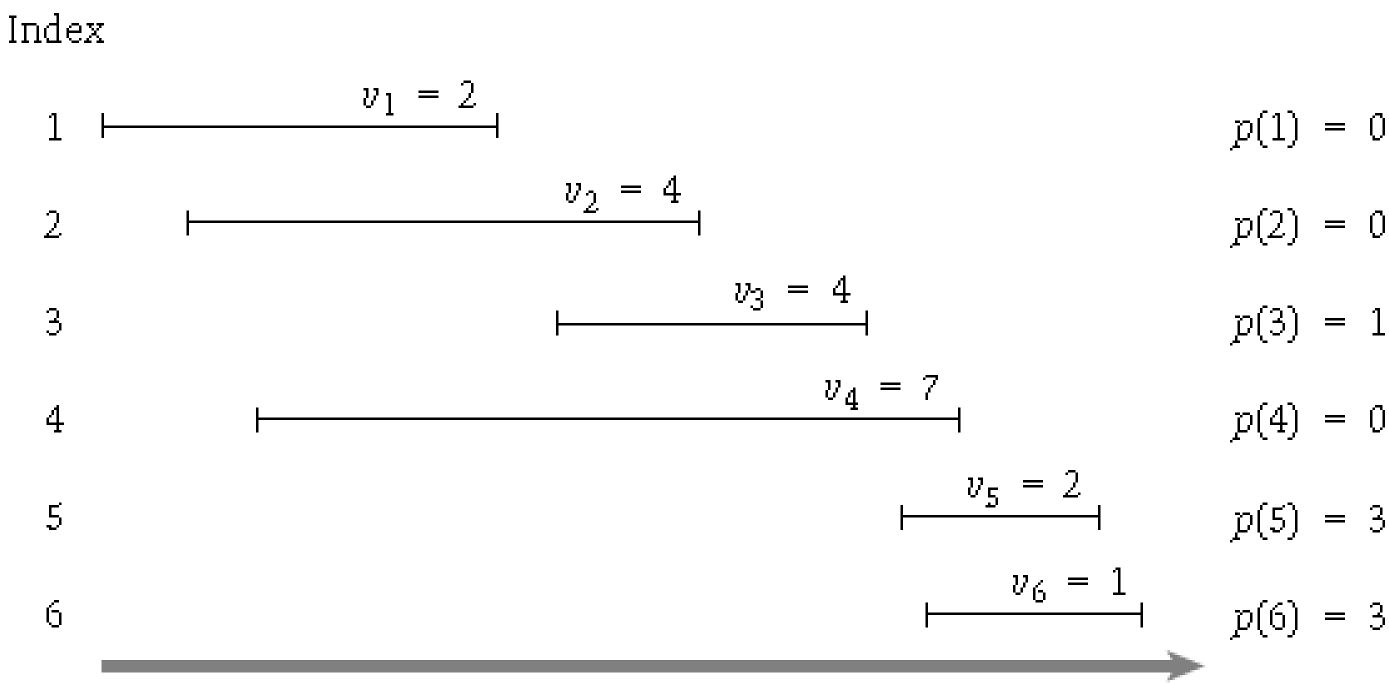
0	2				
0	1	2	3	4	5

j=1

v_j + M[p(j)] 2+M[p(1)]=2+M[0]=2+0 =2

M[j-1] 0

Example



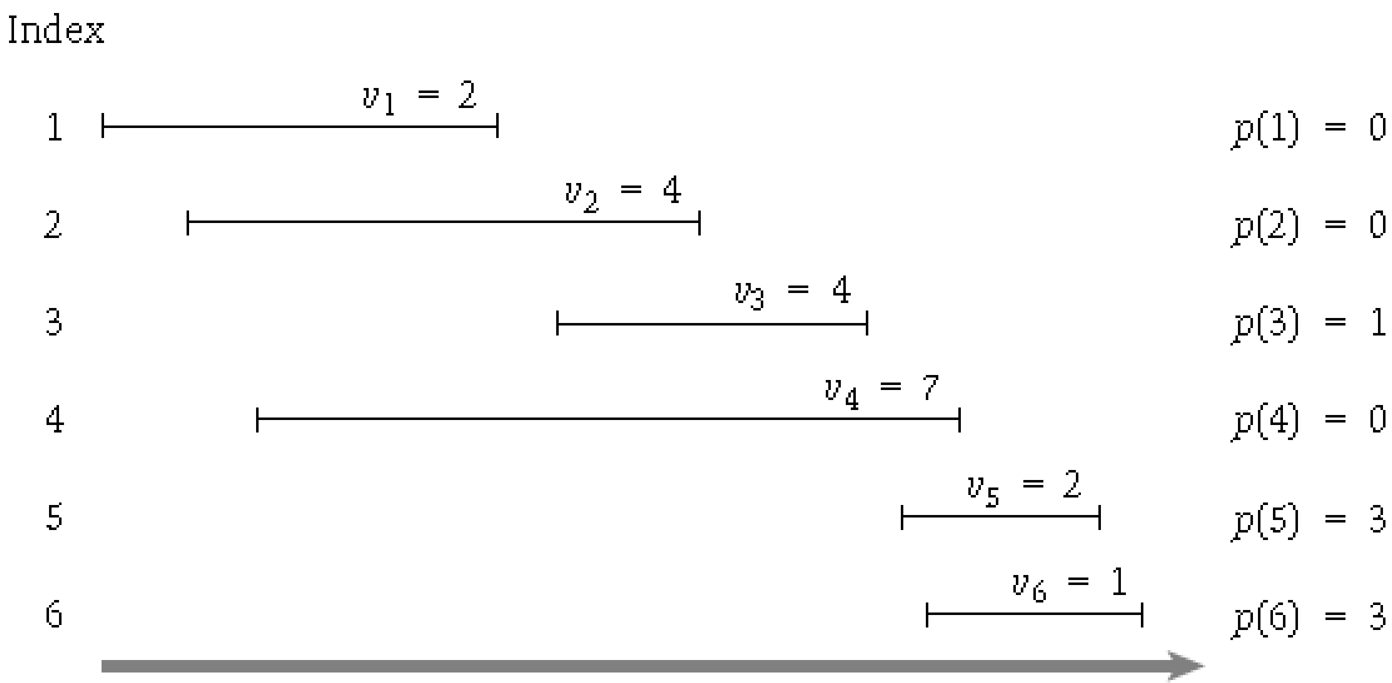
0	2	4			
0	1	2	3	4	5

j=2

$v_j + M[p(j)]$ 4+M[p(2)]=4+M[0]=4+0 =4

M[j-1] 2

Example



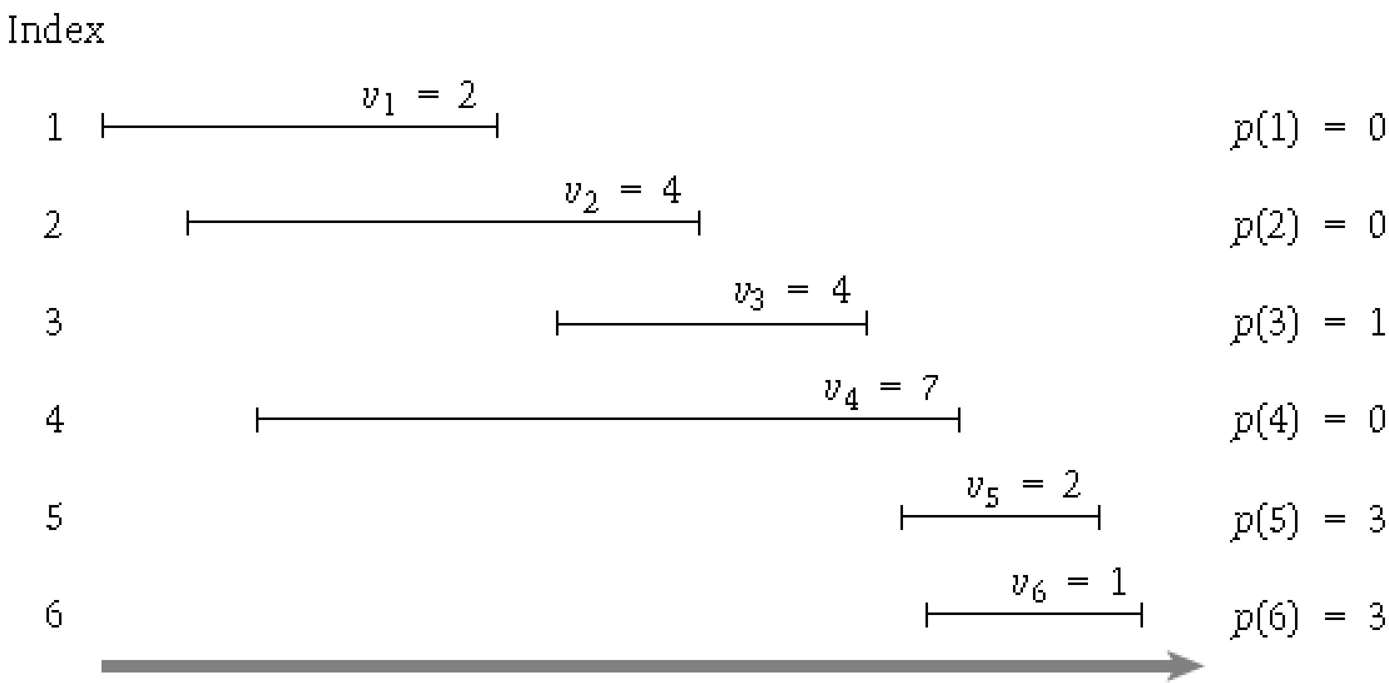
0	2	4	6		
0	1	2	3	4	5

j=3

v_j + M[p(j)] 4+M[p(3)]=4+M[1]=4+2=6

M[j-1] 4

Example



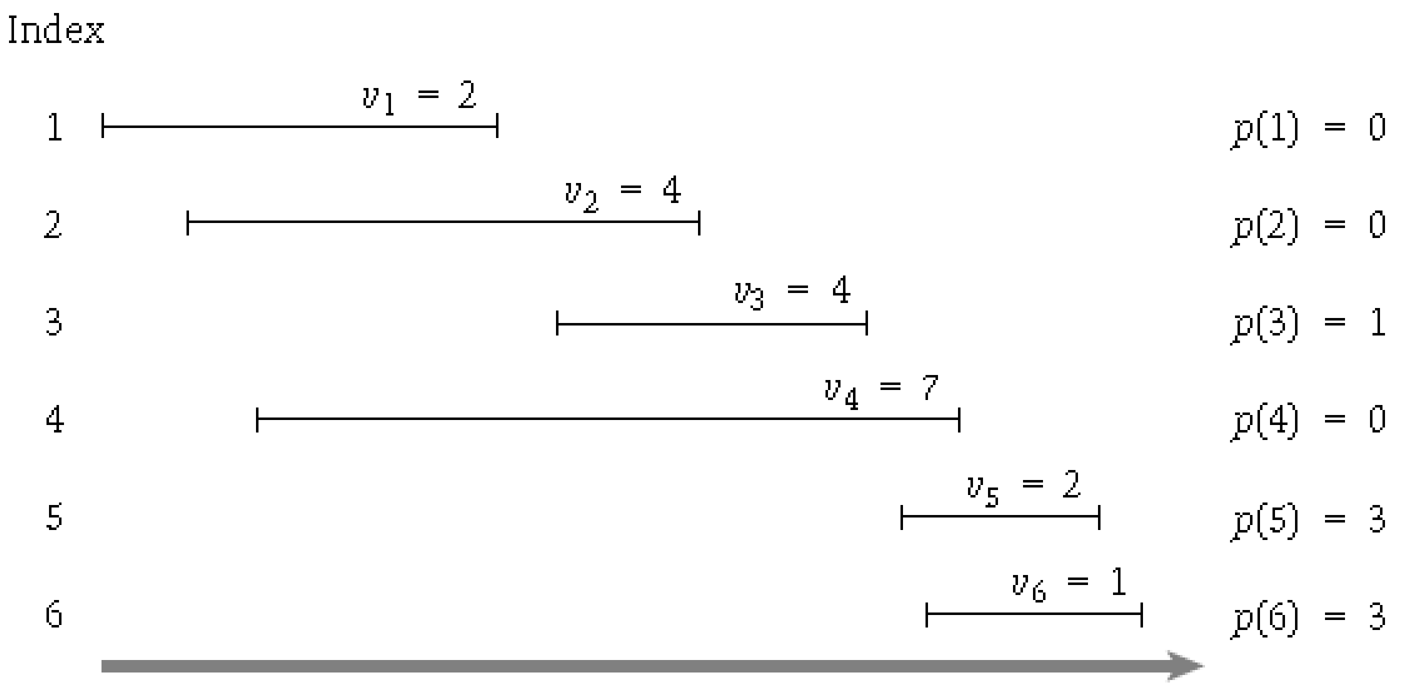
0	2	4	6	7	
0	1	2	3	4	5

j=4

$v_j + M[p(j)]$ 7+M[p(4)]=7+M[0]=7+0=7

M[j-1] 6

Example



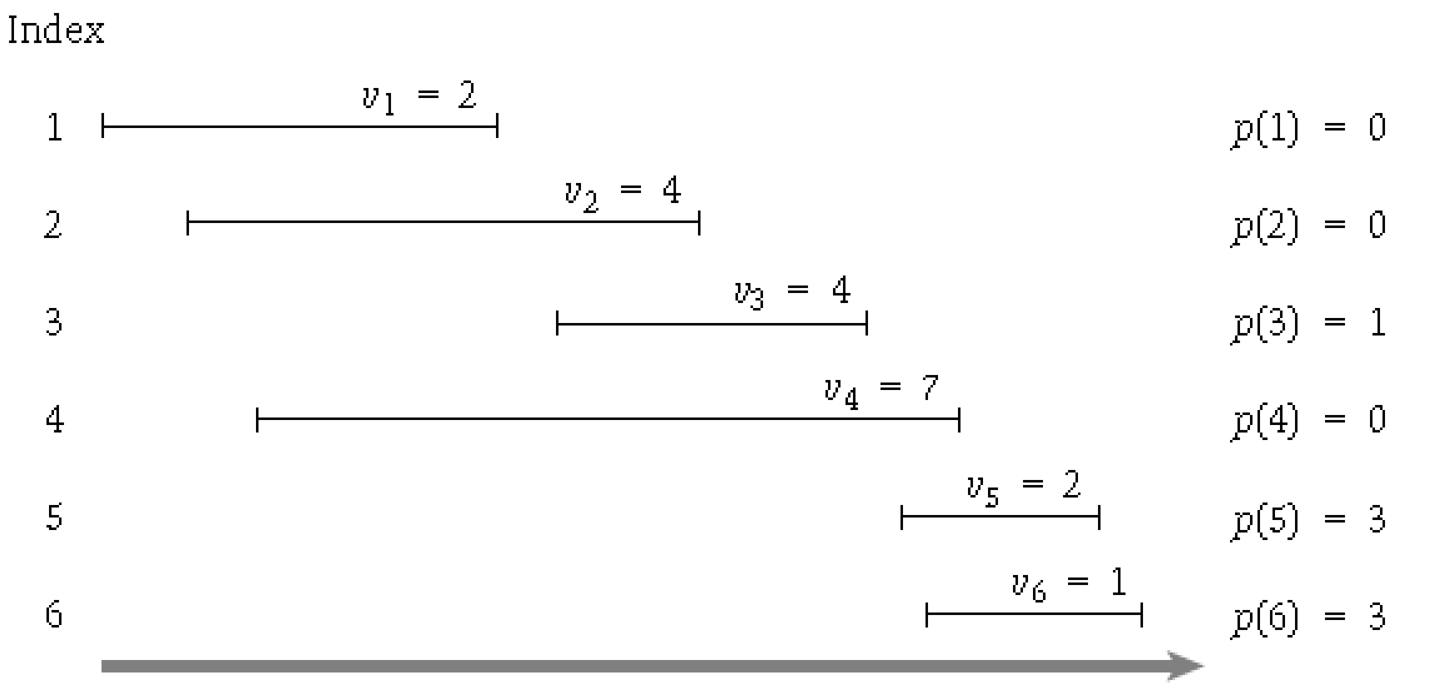
0	2	4	6	7	8
0	1	2	3	4	5

j=5

$v_j + M[p(j)]$ 2+M[p(5)]=2+M[3]=2+6=8

M[j-1] 7

Example

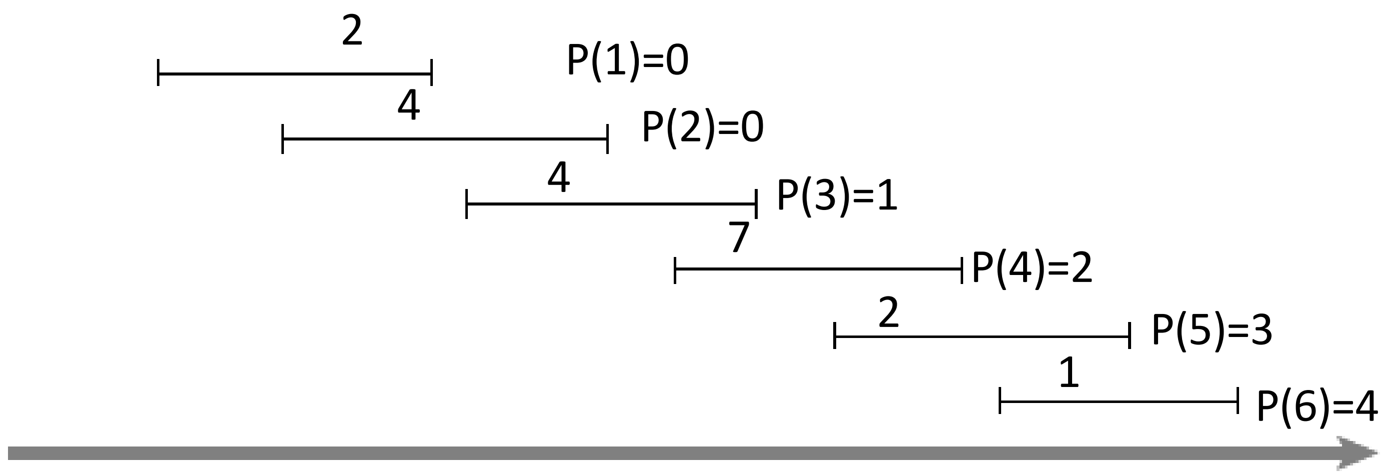


	0	2	4	6	7	8	8
j=6	0	1	2	3	4	5	6

$v_j + M[p(j)]$ $1+M[p(6)]=1+M[3]=1+6=7$

$M[j-1]$ 8

Solve??



0	1	2	3	4	5	6
0	2	4	6	11	11	12

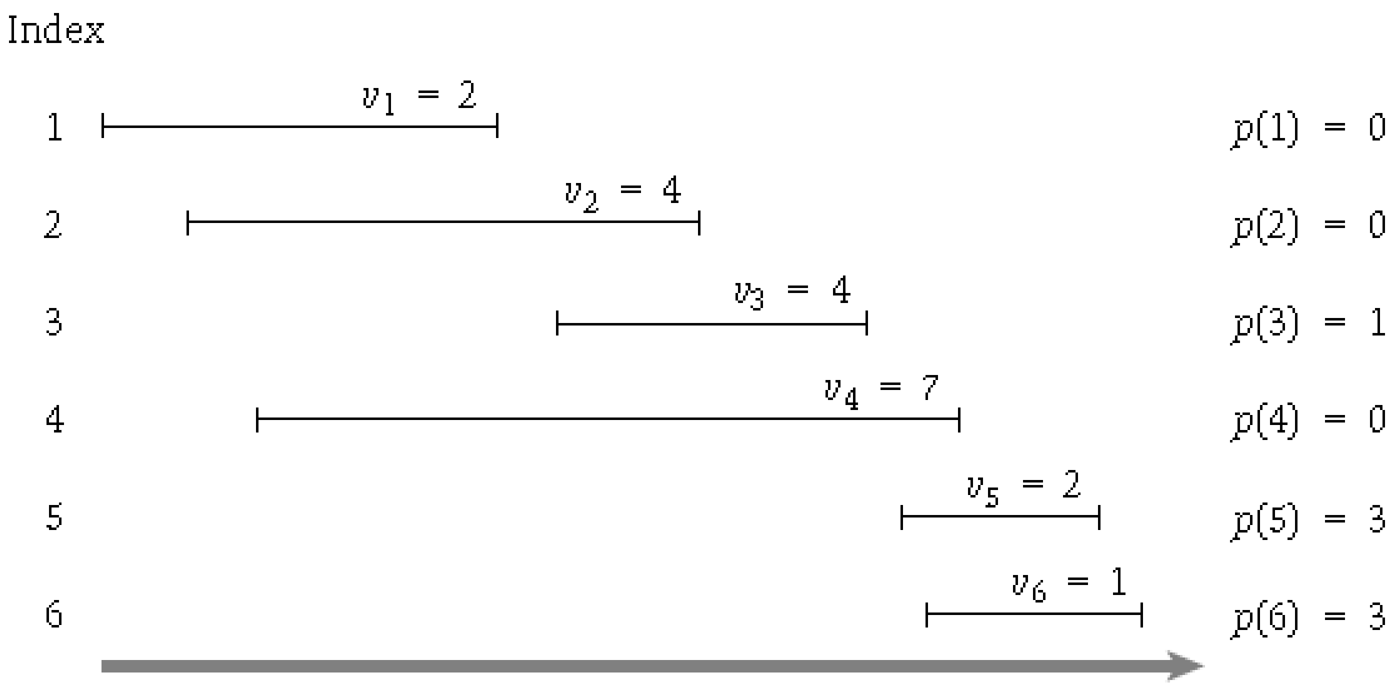
$v_j + M[p(j)]$

$M[j-1]$

Weighted Interval Scheduling: Finding a Solution

```
Run M-Compute-Opt(n) or Iterative-Compute-Opt(n)
Run Find-Solution(n)

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)
}
```

Example



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

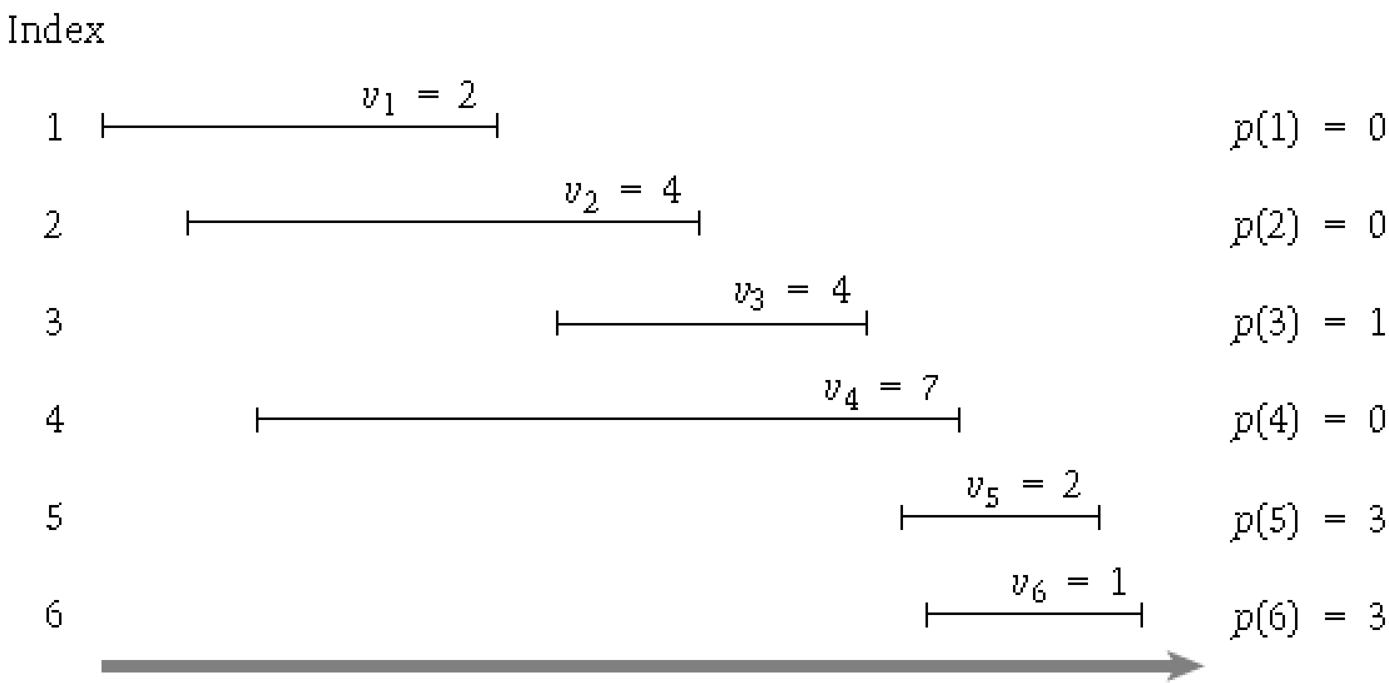
j=6

$v_j + M[p(j)]$ 1+M[p(6)]=1+M[3]=1+6=7

M[j-1] 8

Find-Solution(5)
Output={}

Example



0	2	4	6	7	8

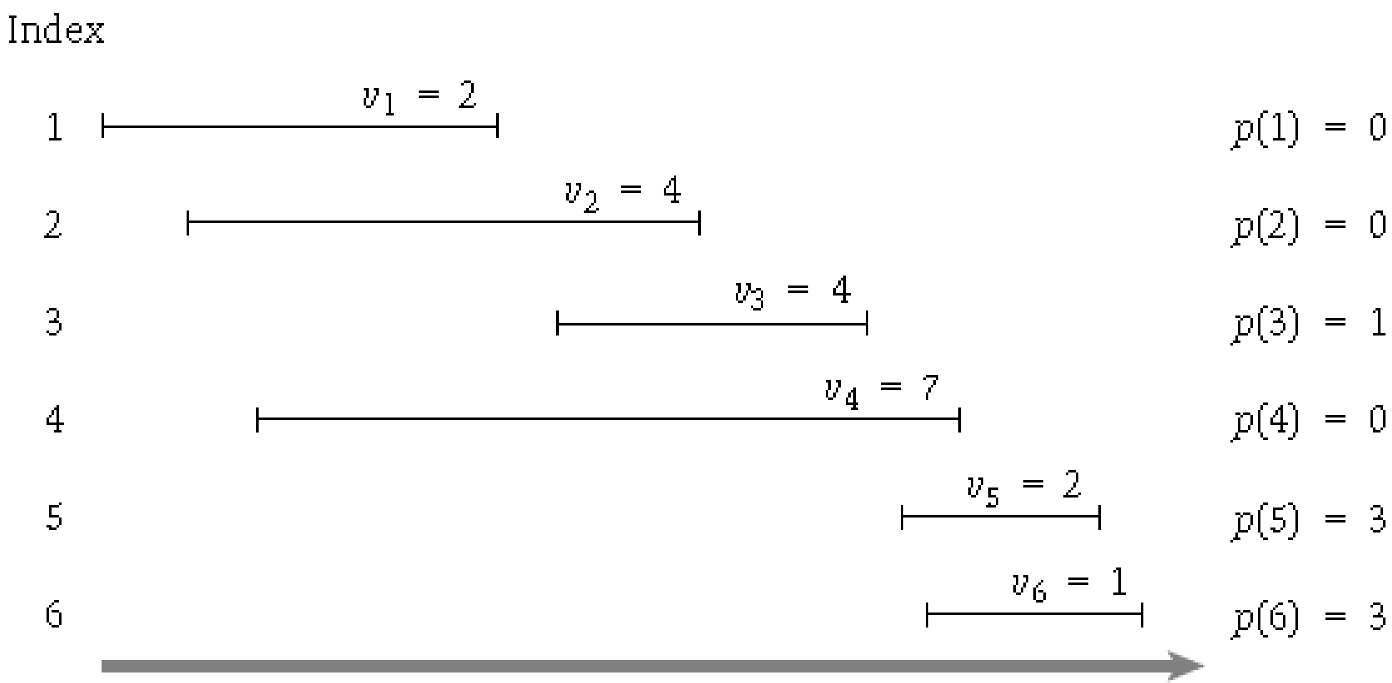
j=5

$v_j + M[p(j)]$ 2+M[p(5)]=2+M[3]=2+6=8

M[j-1] 7

Output={5}
Find-Solution(p(5))=Find-Solution(3)

Example



0	2	4	6		

j=3

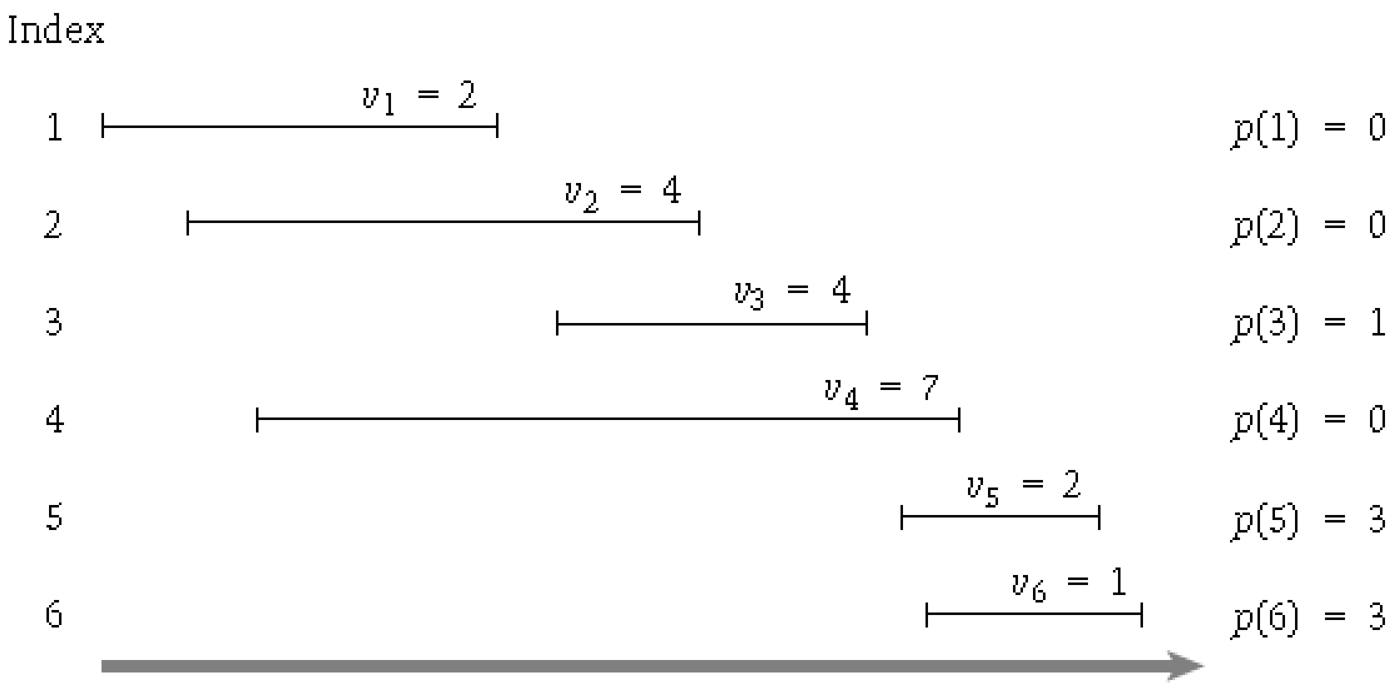
$v_j + M[p(j)]$ 4+M[p(3)]=4+M[1]=4+2=6

M[j-1] 4

Output={5,3}

Find-Solution(p(3))=Find-Solution(1)

Example



0	2				

j=1

v_j + M[p(j)] 2+M[p(1)]=2+M[0]=2+0=2

M[j-1] 0

Output={5,3,1}

Find-Solution(p(1))=Find-Solution(0)

Overall running time of weighted interval scheduling

- Sort the jobs according to fi.
- Compute p(j).
- Memoized compute (OPT(j)) or Iterative compute(OPT(j).
- Find solution(j).

To Compute P(j)

- Sort the requests in order of non-decreasing finish times. This step takes time O(n log n).
- For $1 \le j \le n$, find the largest $i < j$ s.t. $f_i \le s_j$, call it p(j).
- Since the requests are sorted in order of non-decreasing finish times, we can use binary search to find p(j) in time O(log j).

General DP Principles

- Optimal value of the original problem can be computed easily from some subproblems.
- There are only a polynomial # of subproblems.
- There is a “natural" ordering of the subproblems from smallest to largest such that you can obtain the solution for a subproblem by only looking at smaller subproblems.

Algorithm approaches

- Greedy. Build up a solution incrementally, myopically optimizing some local criterion.

- Divide-and-conquer. Break up a problem into sub- problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

- Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Subset sum and Knapsack

Subset sum problem and goal

Problem

- We are given *n items {1, . . . , n}, and each has a given* nonnegative weight w_i *(for i = 1, . . . , n).*
- *We are also given a bound W.*

***Goal**: We* would like to select a subset *S of the items so that* $\sum_{i \in S} w_i \leq W$ *and, subject* to this restriction, $\sum_{i \in S} w_i$ *is as large as possible.*

- Def. **OPT(n, W) = max** profit subset of items **1, …, n** with weight limit ***W***.

  - **Case 1:** ***OPT*** does not select item ***n i***.e. $n \notin OPT$
    - ***OPT*** selects **best** of **{ 1, 2, …, n-1 }** using weight limit ***W***

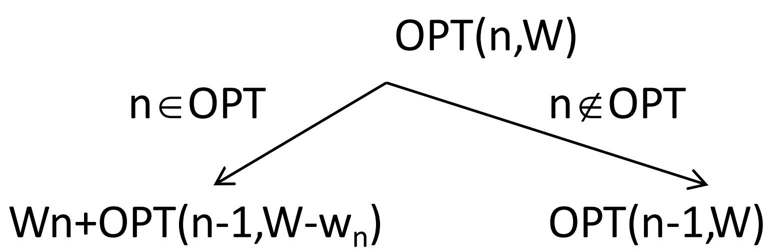
  - **Case 2:** **OPT** selects item ***n***. (*i*.e. $n \in OPT$)
    - new weight limit = $\mathbf{W - w_n}$
    - **OPT** selects **best** of **{ 1, 2, …, n–1 }** using **this new weight limit**

– **<u>Case 1:</u>** ***OPT*** does not select item ***n i***.e. $n \notin OPT$

  - ***OPT*** selects **best** of **{ 1, 2, …, n-1 }** using weight limit ***W***

– **<u>Case 2:</u>** **OPT** selects item ***n***. (***i***.e. $n \in OPT$)

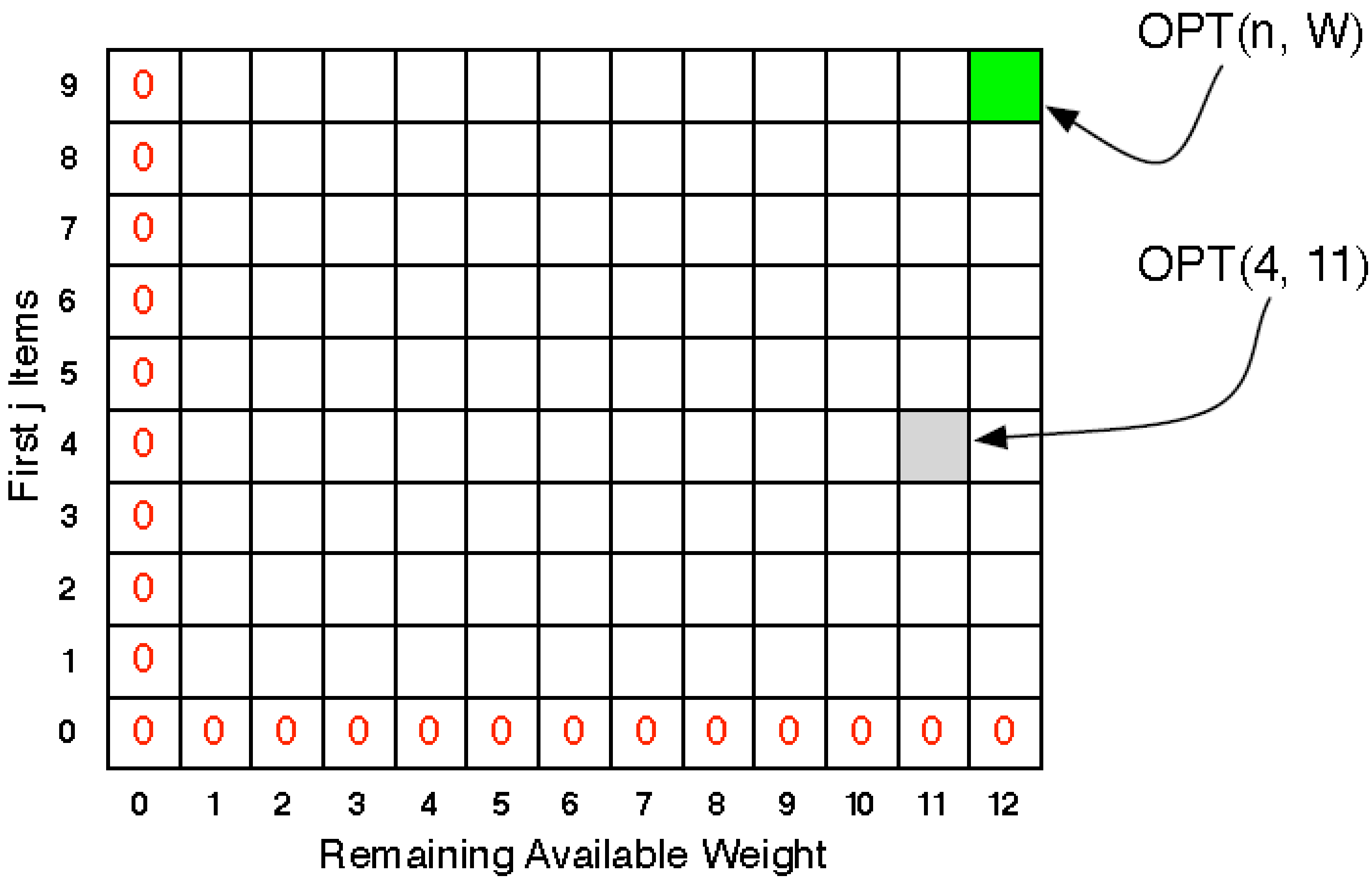
  - new weight limit = $\mathbf{W - w_n}$
  - **OPT** selects **best** of **{ 1, 2, …, n–1 }** using **this new weight limit**



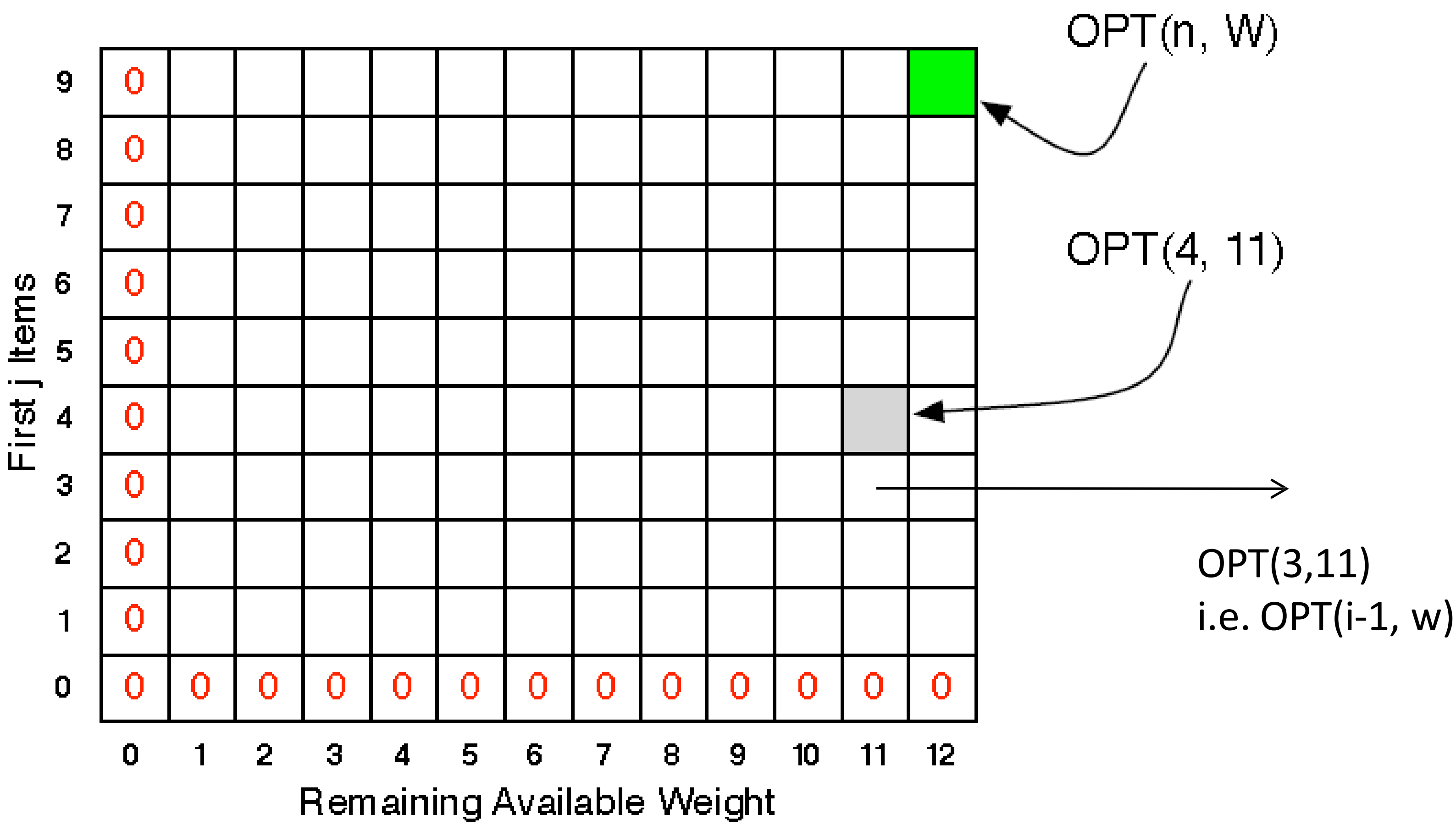
$$
\begin{tikzcd}
 & OPT(n,W) \arrow[dl, "n \in OPT"'] \arrow[dr, "n \notin OPT"] & \\
OPT(n-1, W-w_n) & & OPT(n-1, W)
\end{tikzcd}
$$

$$
OPT(n,W) = \begin{cases} 0 & \text{if } n = 0 \\ OPT(n-1, W) & \text{if } w_n > W \\ \max\{OPT(n-1, W), w_n + OPT(n-1, W-w_n)\} & \text{otherwise} \end{cases}
$$

(n,W) Table of solutions



(n,W) Table of solutions



Algorithm

```
Input: n, W, w1,w2.....wn

for w = 0 to W  M[0, w] = 0


for i = 1 to n                    // n items
   for w = 1 to W                 // weights from 1 to
      if (wi > w)                 max cap W
      M[i, w] = M[i-1, w]
      else
      M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}
      endfor
      endfor

      return M[n, W]
```

Problem

- Let's run our algorithm on the following data:
  - n = 4 (# of elements)
  - W = 5 (max weight)
  - Elements (weight): (2), (3), (4), (5)

Item	Weights
1	2
2	3
3	4
4	5

Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

// Initialize the base cases

for w = 0 to W

M[0,w] = 0

for i = 1 to n

M[i,0] = 0

Example

Items:
1: (2)
2: (3)
3: (4)
4: (5)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

$i = 1$
$w_i = 2$
$w = 1$

```
if (wi > w)
M[i, w] = M[i-1, w]
else
M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}
```

Example

Items:
1: (2)
2: (3)
3: (4)
4: (5)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	2			
2	0					
3	0					
4	0					

$i = 1$
$w_i = 2$
$w = 2$

```
if (wi > w)
M[i, w] = M[i-1, w]
else
M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}
```

Max{M[0,2],2+M[0,0]}

Example

Items:
1: (2)
2: (3)
3: (4)
4: (5)

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	2	2		
2	0					
3	0					
4	0					

$i = 1$
$w_i = 2$
$w = 3$
$w-w_i = 1$

```
if (wi > w)
M[i, w] = M[i-1, w]
else
M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}
```

```
Max{M[0,3],2+M[0,1]}
```

Example

Items:
1: (2)
2: (3)
3: (4)
4: (5)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	2	2	2	
2	0					
3	0					
4	0					

$i = 1$
$w_i = 2$
$w = 4$
$w\text{-}w_i = 2$

```
if (wi > w)
M[i, w] = M[i-1, w]
else
M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}
```

```
Max{M[0,4],2+M[0,2]}
```

Example

Items:

1: (2)

2: (3)

3: (4)

4: (5)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	2	2	2	2
2	0					
3	0					
4	0					

$i = 1$

$w_i = 2$

$w = 5$

$w\text{-}w_i = 3$

```
if (wi > w)
M[i, w] = M[i-1, w]
else
M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}
```

```
Max{M[0,5],2+M[0,3]}
```

Example

Items:
1: (2)
$\boxed{2: (3)}$
3: (4)
4: (5)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	2	2	2	2
2	0	0				
3	0					
4	0					

$i = 2$
$w_i = 3$
$w = 1$

```
if (wi > w)
M[i, w] = M[i-1, w]
else
M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}
```

M[2,1]=M[1,1]=0

Example

Items:

1: (2)

$\boxed{2: (3)}$

3: (4)

4: (5)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	2	2	2	2
2	0	0	2			
3	0					
4	0					

$i = 2$

$w_i = 3$

$\mathbf{w = 2}$

```
if (wi > w)
M[i, w] = M[i-1, w]
else
M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}
```

```
M[2,2]=M[1,2]=2
```

Example

Items:
1: (2)
2: (3)
3: (4)
4: (5)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	2	2	2	2
2	0	0	2	3		
3	0					
4	0					

$i = 2$
$w_i = 3$
$w = 3$
$w\text{-}w_i = 0$

```
if (wi > w)
M[i, w] = M[i-1, w]
else
M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}
```

```
Max{M[1,3],3+M[1,0]}
```

Example

Items:
1: (2)
2: (3)
3: (4)
4: (5)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	2	2	2	2
2	0	0	2	3	3	
3	0					
4	0					

$i = 2$
$w_i = 3$
$\mathbf{w = 4}$
$w\text{-}w_i = 1$

```
if (wi > w)
M[i, w] = M[i-1, w]
else
M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}
```

Max{M[1,4],3+M[1,1]}

Example

Items:

1: (2)

$\boxed{2: (3)}$

3: (4)

4: (5)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	2	2	2	2
2	0	0	2	3	3	5
3	0					
4	0					

$i = 2$

$w_i = 3$

$\mathbf{w = 5}$

$w\text{-}w_i = 2$

```
if (wi > w)
M[i, w] = M[i-1, w]
else
M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}
```

```
Max{M[1,5],3+M[1,2]}
```

Example

Items:

1: (2)

2: (3)

3: (4)

4: (5)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	2	2	2	2
2	0	0	2	3	3	5
3	0	0	2	3		
4	0					

$i = 3$

$w_i = 4$

$\mathbf{w = 1..3}$

$w\text{-}w_i = -3..-1$

```
if (wi > w)
M[i, w] = M[i-1, w]
else
M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}
```

Example

Items:
1: (2)
2: (3)
$\boxed{3: (4)}$
4: (5)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	2	2	2	2
2	0	0	2	3	3	5
3	0	0	2	3	4	
4	0					

$i = 3$
$w_i = 4$
$w = 4$
$w - w_i = 0$

```
if (wi > w)
M[i, w] = M[i-1, w]
else
M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}
```

```
Max{M[2,4],4+M[2,0]}
```

Example

Items:

1: (2)

2: (3)

3: (4)

4: (5)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	2	2	2	2
2	0	0	2	3	3	5
3	0	0	2	3	4	5
4	0					

$i = 3$

$w_i = 4$

$w = 5$

$w\text{-}w_i = 1$

```
if (wi > w)
M[i, w] = M[i-1, w]
else
M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}
```

Max{M[2,5],4+M[2,1]}

Example

Items:
1: (2)
2: (3)
3: (4)
4: (5)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	2	2	2	2
2	0	0	2	3	3	5
3	0	0	2	3	4	5
4	0	0	2	3	4	

$i = 4$
$w_i = 5$
$w = 1..4$
$w\text{-}w_i = -4..-1$

```
if (wi > w)
M[i, w] = M[i-1, w]
else
M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}
```

Example

Items:
1: (2)
2: (3)
3: (4)
4: (5)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	2	2	2	2
2	0	0	2	3	3	5
3	0	0	2	3	4	5
4	0	0	2	3	4	5

$i = 4$
$w_i = 5$
$w = 5$
$w - w_i = 0$

```
if (wi > w)
M[i, w] = M[i-1, w]
else
M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}
```

```
Max{M[3,5],5+M[3,0]}
```

Example

Items:

1: (2)

2: (3)

3: (4)

4: (5)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	2	2	2	2
2	0	0	2	3	3	5
3	0	0	2	3	4	5
4	0	0	2	3	4	5

We're DONE!!

The max possible value that can be obtained is 5.

Solve

Items	Weights
1	2
2	2
3	3

W=5

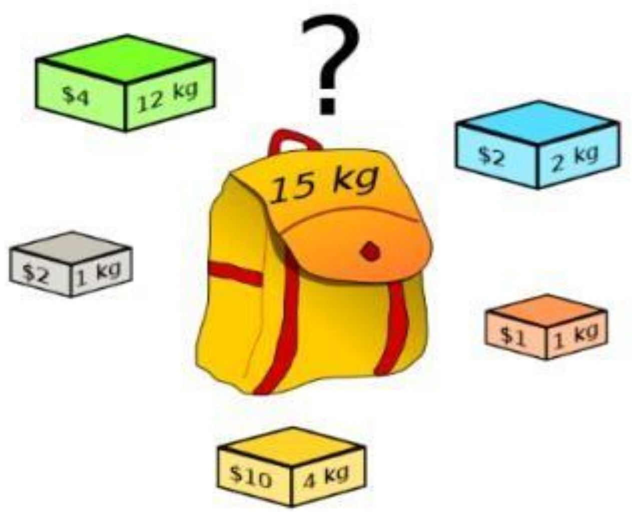
i/w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	2	2	2	2
2	0	0	2	2	4	4
3	0	0	2	3	4	5

$A=\{2,3,5,7,10\}$ $W=14$

Knapsack

- Given ***n*** objects and a "*knapsack*."
- Item ***i*** weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of ***W*** kilograms.
- **Goal:** fill knapsack so as to ***maximize*** total **SUM of values**.

Ex: { 3, 4 } has value 40.



W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$
\begin{tikzcd}
 & \mathrm{OPT}(n,W) \arrow[dl, "n \in \mathrm{OPT}"'] \arrow[dr, "n \notin \mathrm{OPT}"] & \\
\mathrm{OPT}(n-1, W-w_n) & & \mathrm{OPT}(n-1, W)
\end{tikzcd}
$$

$$
OPT(n,W) = \begin{cases} 0 & \text{if } n = 0 \\ OPT(n-1,W) & \text{if } w_n > W \\ \max\{OPT(n-1,W), w_n + OPT(n-1, W-w_n)\} & \text{otherwise} \end{cases}
$$

Algorithm

```
Input: n, W, w_1,…,w_N, v_1,…,v_N

for w = 0 to W  M[0, w] = 0

for i = 1 to n  for w = 1 to W
if (w_i > w)
M[i, w] = M[i-1, w]
else
M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}

return M[n, W]
```

Solve

Items	Weights
1	2
2	2
3	3

W=5

Solve

Items	Weights	Value
1	2	20
2	2	10
3	3	30

W=5

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0					
2	0					
3	0					

Solve

Items	Weights	Value
1	2	20
2	2	10
3	3	30

W=5

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	20	20	20	20
2	0	0	20	20	30	30
3	0	0	20	30	30	50

Find the items that belong to the Knapsack

- To know the ***items*** that make this maximum value, we need to trace back through the table.

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	20	20	20	20
2	0	0	20	20	30	30
3	0	0	20	30	30	50

Find the items that belong to the Knapsack

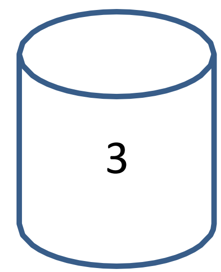
I	W
1	2
2	2
3	3

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	20	20	20	20
2	0	0	20	20	30	30
3	0	0	20	30	30	50

i=3
k=5

i=2
k=2

$i = n$, $k = W$ while i, k > 0
if $M[i, k] \neq M[i-1, k]$ then
mark the i^{th} *item as in the knapsack* $i = i-1, k = k-w_i$
else
$i = i-1$



Find the items that belong to the Knapsack

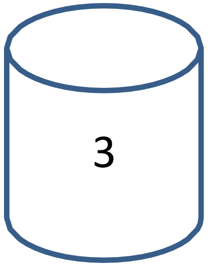
I	W
1	2
2	2
3	3

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	20	20	20	20
2	0	0	20	20	30	30
3	0	0	20	30	30	50

i=2
k=2

i=1
k=2

i = n , k = W while i, k > 0
if $M[i, k] \neq M[i-1, k]$ then
mark the i^{th} *item as in the knapsack* $i = i-1, k = k-w_i$
else
$i = i-1$



Find the items that belong to the Knapsack

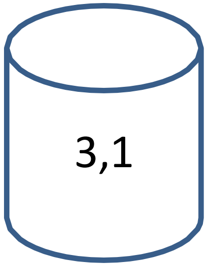
I	W
1	2
2	2
3	3

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	20	20	20	20
2	0	0	20	20	30	30
3	0	0	20	30	30	50

i=1
k=2

i=0
k=0

$i = n$, $k = W$ while $i, k > 0$
if $M[i, k] \neq M[i-1, k]$ then
mark the i^{th} *item as in the knapsack* $i = i-1$, $k = k-w_i$
else
$i = i-1$



Solve??

Item	Weight	Value
I_1	3	10
I_2	5	4
I_3	6	9
I_4	2	11

- **The maximum weight the knapsack can hold is 7.**