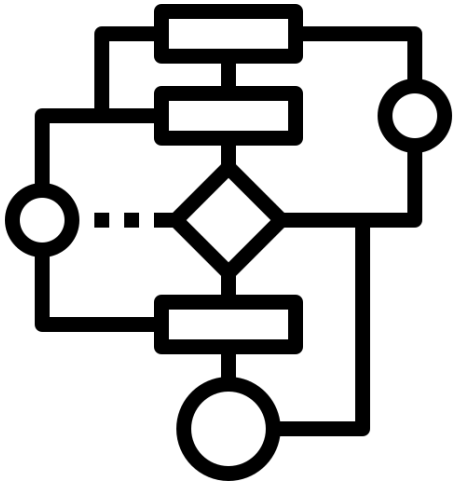
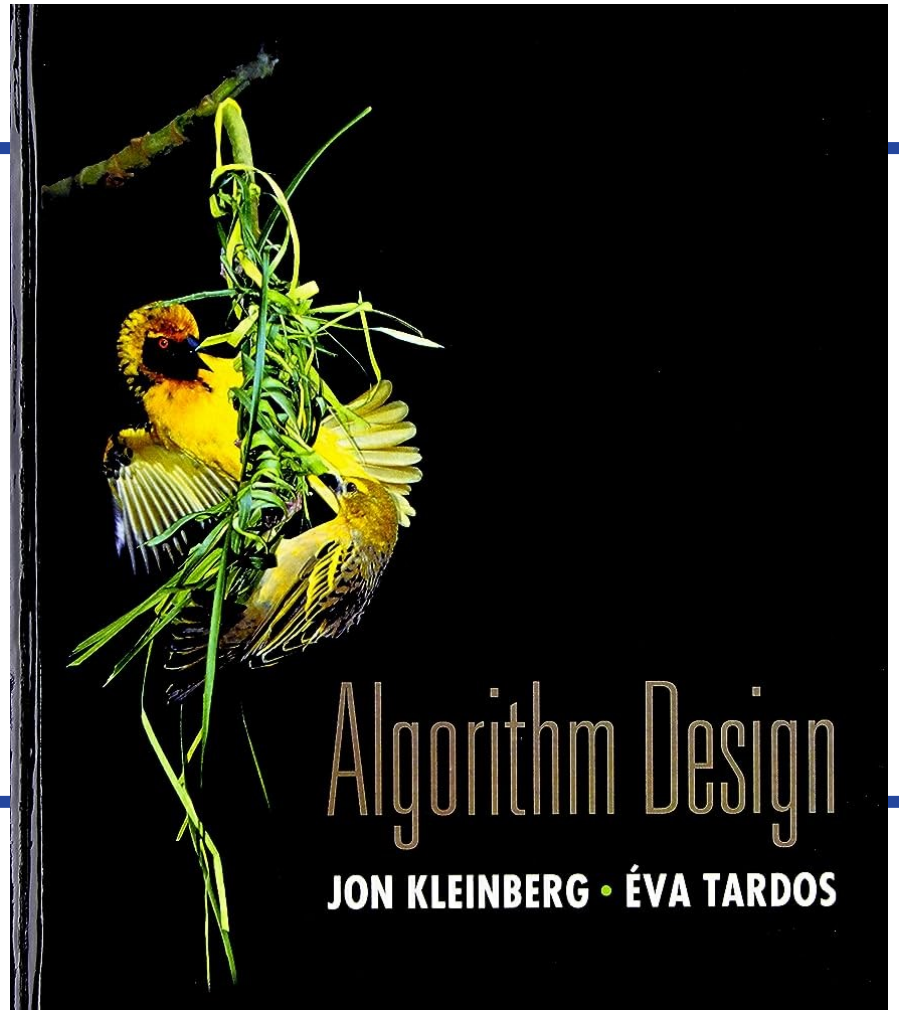


CS 310: Algorithms

Lecture 19

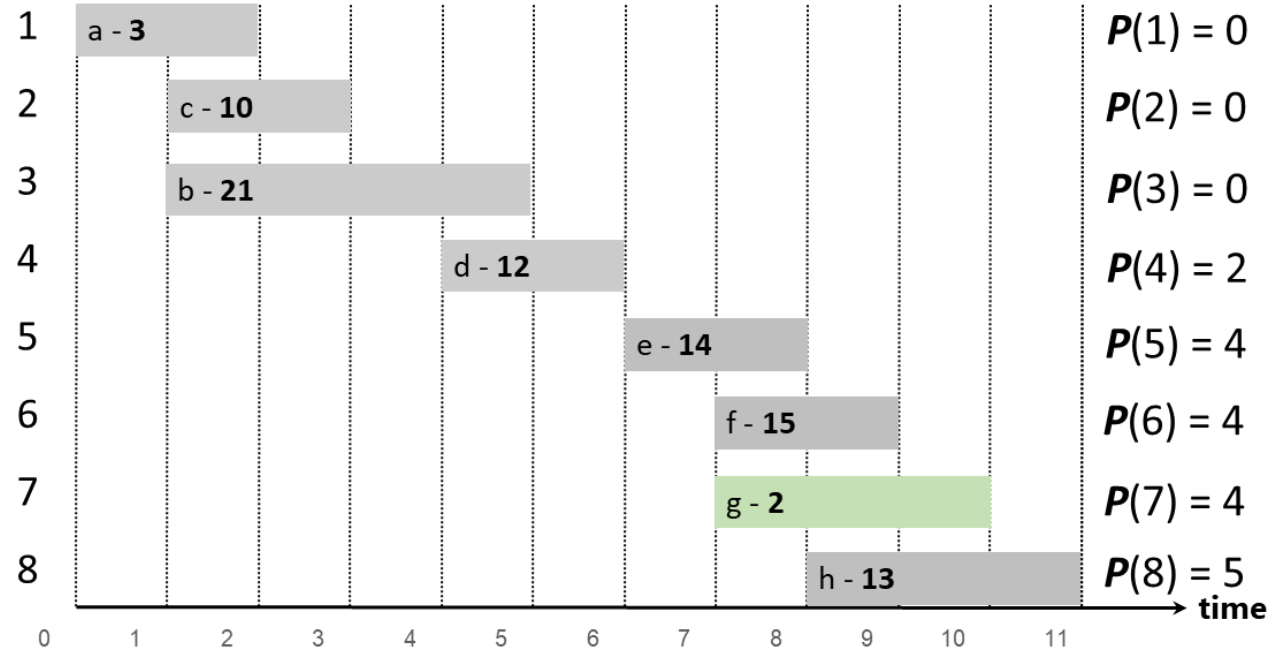
Instructor: Naveed Anwar Bhatti



Chapter 6:
Dynamic Programming

Section :
Weighed Interval Scheduling Problem

Weighted Interval Scheduling Problem



$$\textbf{Max\_Value}(n) = \max \begin{cases} 0 & \text{if } n=0 \\ W_n + \textbf{Max\_Value}(P(n)) & \text{if } W_n \in \text{Optimal Jobs} \\ \textbf{Max\_Value}(n-1) & \text{if } W_n \notin \text{Optimal Jobs} \end{cases}$$

Weighted Interval Scheduling Problem

Brute-force Algorithm

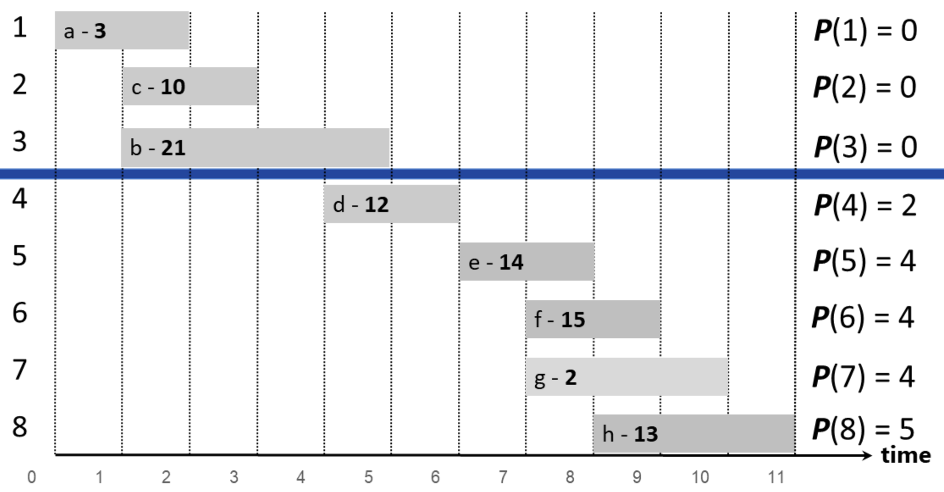
```
Input: n, s1,…,sn , f1,…,fn , w1,…,wn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Max_Value(n) {
   if (n = 0)
      return 0
   else
      return max(Wn + Max_Value(p(n)), Max_Value(n-1))
}
```

O(nlogn)

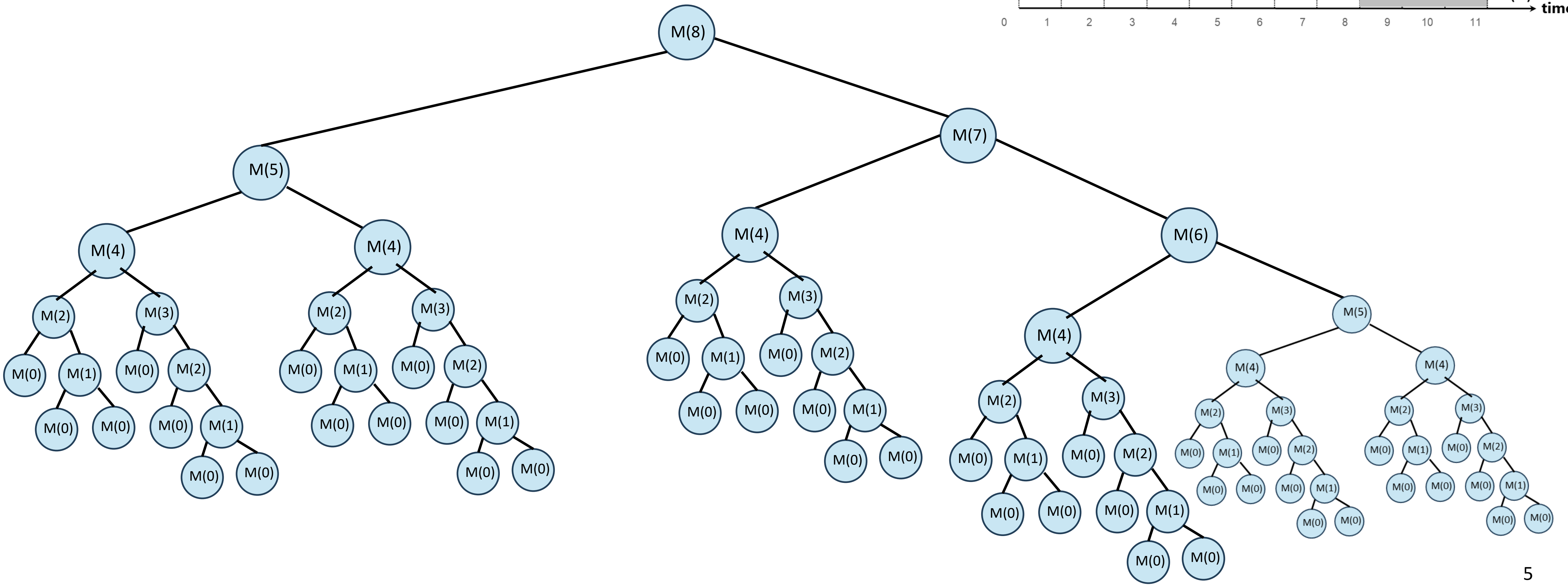
O(nlogn)

O(2^n)

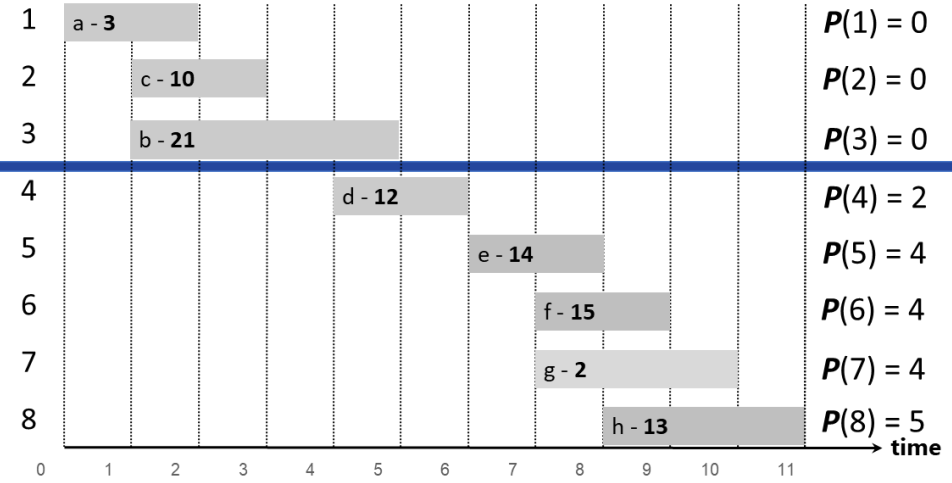
Weighted Interval Scheduling Problem



$$\texttt{Max\_Value}(n) = \mathit{max} \begin{cases} 0 & \text{if } n=0 \\ W_n + \texttt{Max\_Value}(P(n)) & \text{if } W_n \in \text{Optimal Jobs} \\ \texttt{Max\_Value}(n-1) & \text{if } W_n \notin \text{Optimal Jobs} \end{cases}$$



Weighted Interval Scheduling Problem

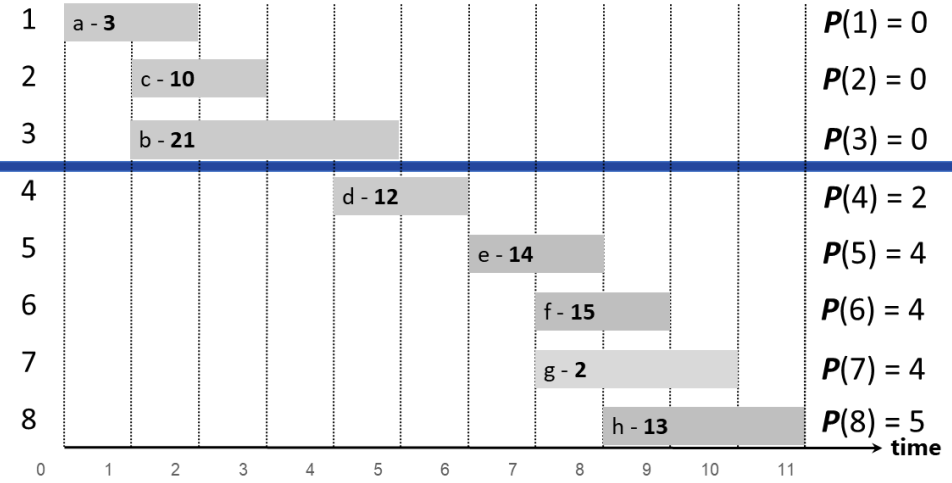


Memoization: Store results of each sub-problem in a cache; lookup as needed.

```
Input: n, s1,…,sn , f1,…,fn , w1,…,wn

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

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

Max_Value(n) {
   if (M[n] is -1)
      M[n] = max(wn + Max_Value(p(n)), Max_Value(n-1))
   return M[n]
}
```

M[ ]	0
0	0
1	-1
2	-1
3	-1
4	-1
5	-1
6	-1
7	-1
8	-1

Weighted Interval Scheduling Problem



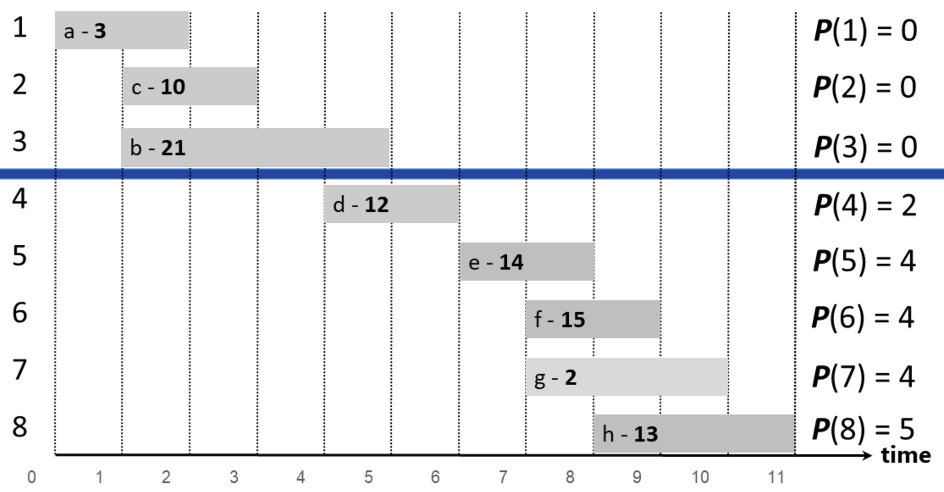
Memoization: Store results of each sub-problem in a cache; lookup as needed.

```
Input: n, s1,…,sn , f1,…,fn , w1,…,wn

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

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

Max_Value(n) {
   if (M[n] is -1)
      M[n] = max(wn + Max_Value(p(n)), Max_Value(n-1))
   return M[n]
}
```

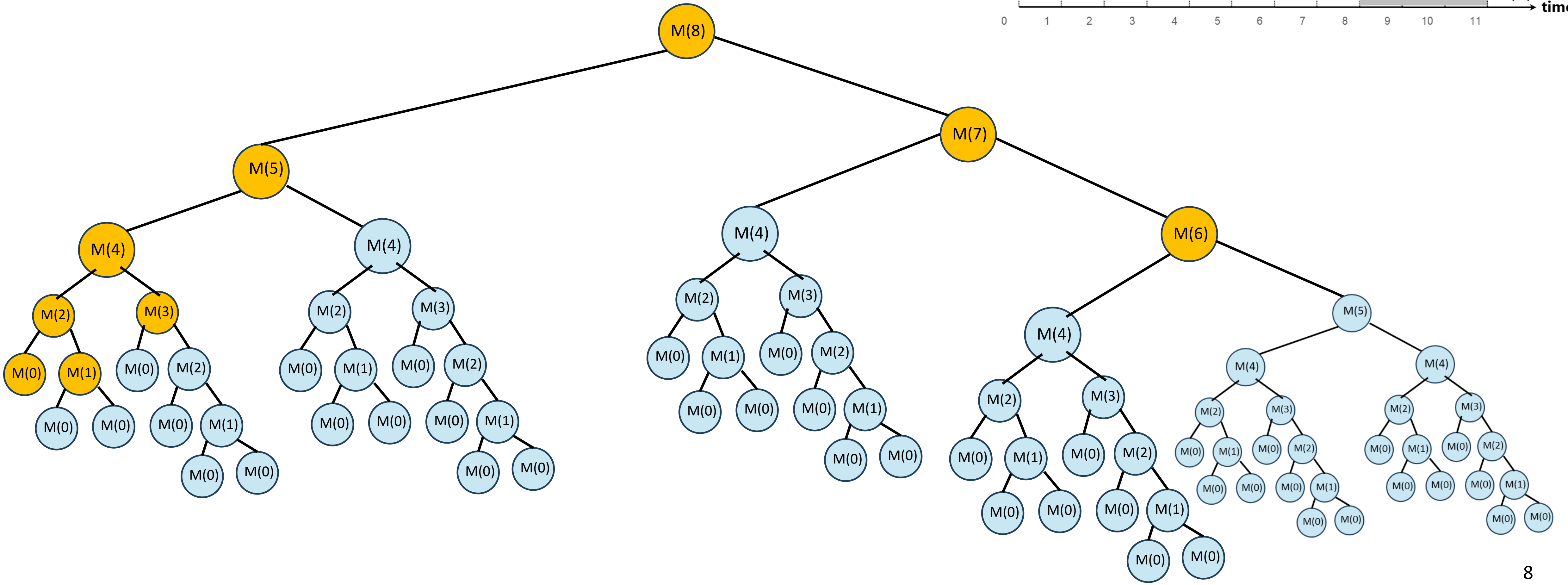
M[ ]	0
	1
	2
	3
	4
	5
	6
	7
	8

0
3
10
21
22
36
37
37
49

Weighted Interval Scheduling Problem



$$\textbf{Max\_Value}(n) = \max \begin{cases} 0 & \text{if } n=0 \\ W_n + \textbf{Max\_Value}(P(n)) & \text{if } W_n \in \text{Optimal Jobs} \\ \textbf{Max\_Value}(n-1) & \text{if } W_n \notin \text{Optimal Jobs} \end{cases}$$

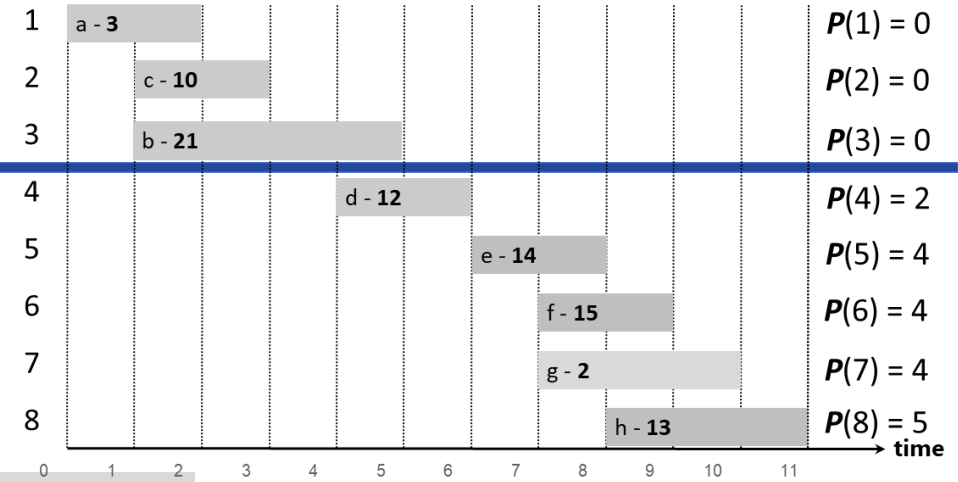


Weighted Interval Scheduling Problem

Overall Time Complexity

```
Input: n, s1,…,sn , f1,…,fn , w1,…,wn

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

for j = 1 to n                                                 O(n)
   M[j] = empty ← global array
M[0] = 0

Max_Value(n) {                                                 O(n)
   if (M[n] is empty)
      M[n] = max(wn + Max_Value(p(n)), Max_Value(n-1))
   return M[n]
}
```

WISP: Finding a Solution



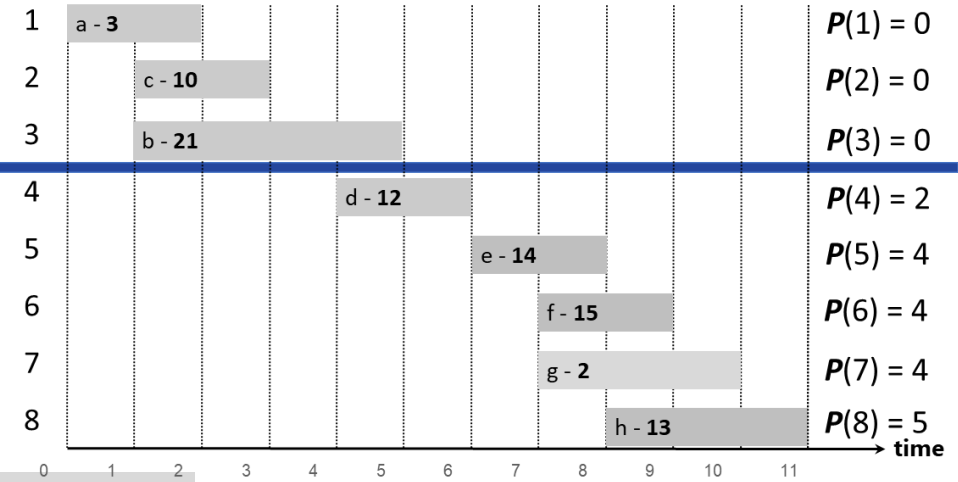
Q: Dynamic programming algorithms computes optimal value. How can we get the solution itself?

A: Do some post-processing.

```
Run Max_Value(n)
Run Find-Solution(n)

Find-Solution(j) {
    if (j = 0)
        output nothing
    else if (w_j + M[p(j)] > M[j-1])
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}
```

0	0
1	3
2	10
3	21
4	22
5	36
6	37
7	37
8	49

WISP: Finding a Solution



Q: Dynamic programming algorithms computes optimal value. How can we get the solution itself?

A: Do some post-processing.

```
Run Max_Value(n)
Run Find-Solution(n)

Find-Solution(j) {
    if (j = 0)
        output nothing
    else if (w_j + M[p(j)] > M[j-1])
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}
```

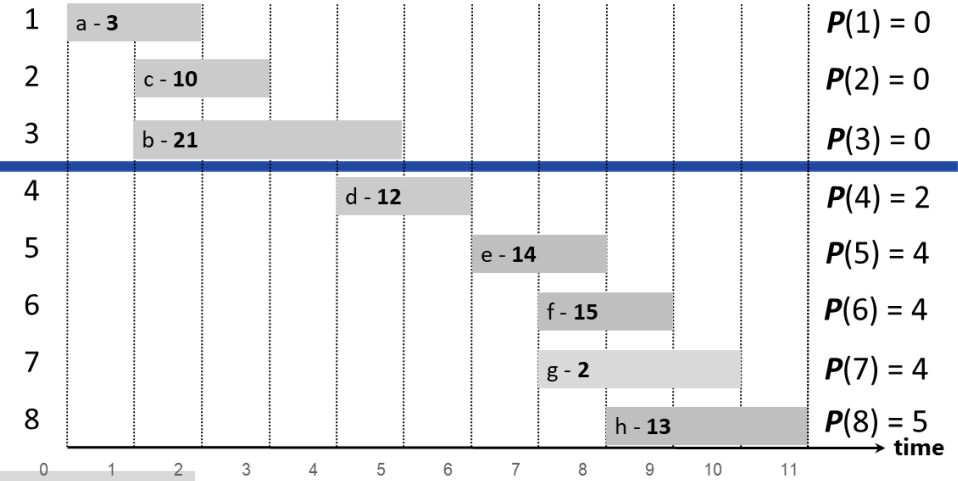
j = 8

13 + 36 > 37

8

0	0
1	3
2	10
3	21
4	22
5	36
6	37
7	37
8	49

WISP: Finding a Solution



Q: Dynamic programming algorithms computes optimal value. How can we get the solution itself?

A: Do some post-processing.

```
Run Max_Value(n)
Run Find-Solution(n)

Find-Solution(j) {
    if (j = 0)
        output nothing
    else if (w_j + M[p(j)] > M[j-1])
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}
```

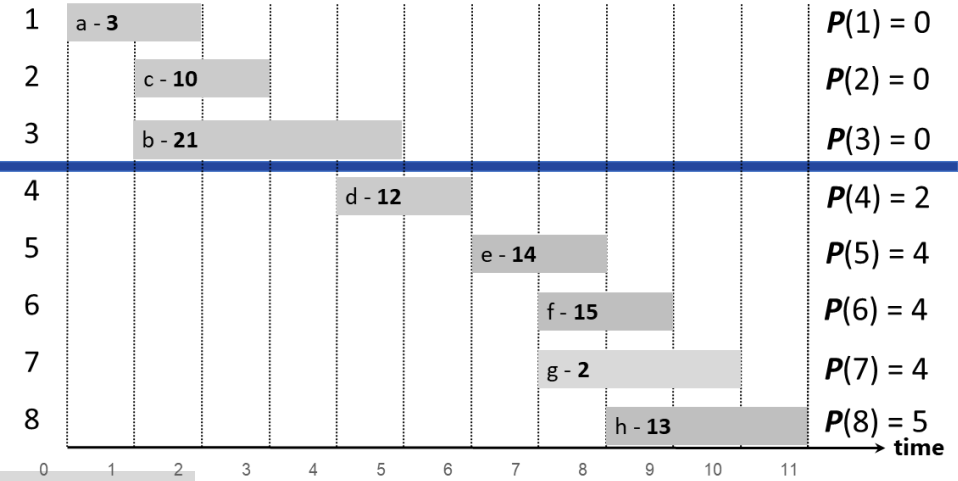
j = 5

14 + 22 > 22

0	0
1	3
2	10
3	21
4	22
5	36
6	37
7	37
8	49

8 5

WISP : Finding a Solution



Q: Dynamic programming algorithms computes optimal value. How can we get the solution itself?

A: Do some post-processing.

```
Run Max_Value(n)
Run Find-Solution(n)

Find-Solution(j) {
    if (j = 0)
        output nothing
    else if (w_j + M[p(j)] > M[j-1])
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}
```

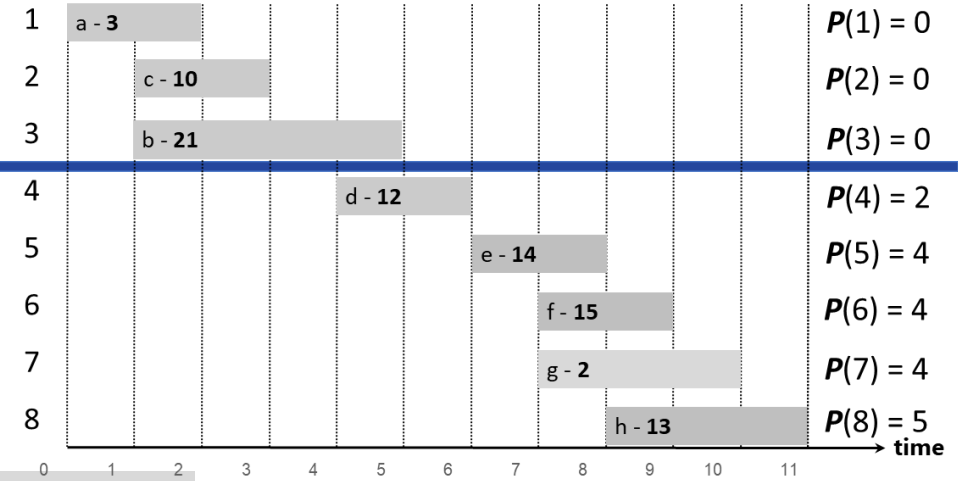
j = 4

12 + 10 > 21

0	0
1	3
2	10
3	21
4	22
5	36
6	37
7	37
8	49

8 5 4

WISP : Finding a Solution



Q: Dynamic programming algorithms computes optimal value. How can we get the solution itself?

A: Do some post-processing.

```
Run Max_Value(n)
Run Find-Solution(n)

Find-Solution(j) {
    if (j = 0)
        output nothing
    else if (w_j + M[p(j)] > M[j-1])
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}
```

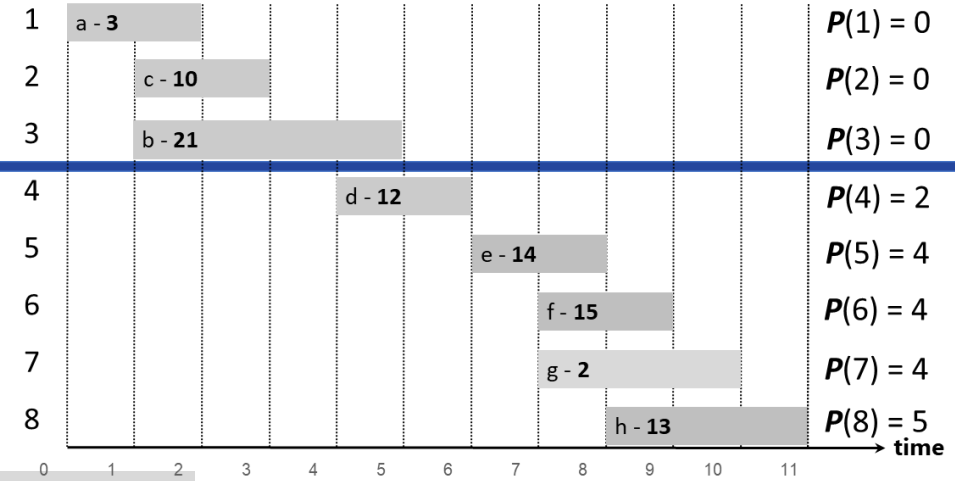
j = 2

10 + 0 > 3

8	5	4	2

0	0
1	3
2	10
3	21
4	22
5	36
6	37
7	37
8	49

WISP : Finding a Solution



Q: Dynamic programming algorithms computes optimal value. How can we get the solution itself?

A: Do some post-processing.

```
Run Max_Value(n)
Run Find-Solution(n)

Find-Solution(j) {
    if (j = 0)
        output nothing
    else if (w_j + M[p(j)] > M[j-1])
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}
```

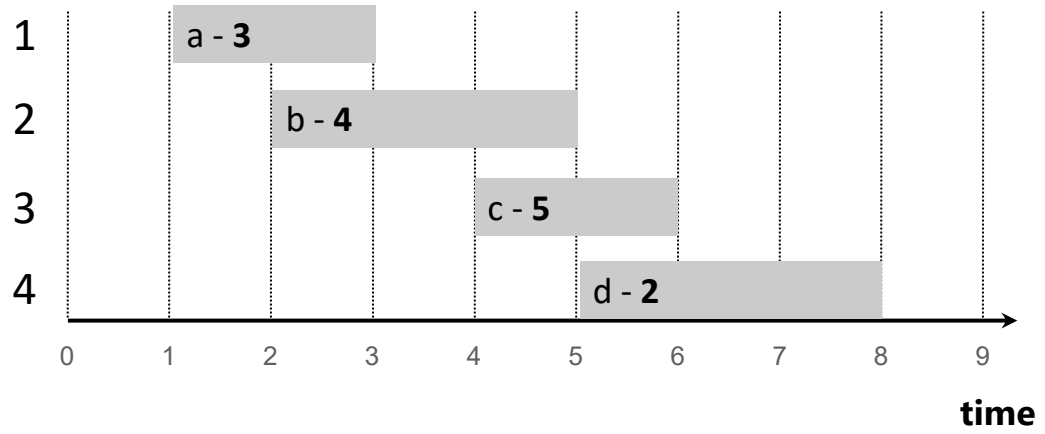
j = 0

8	5	4	2

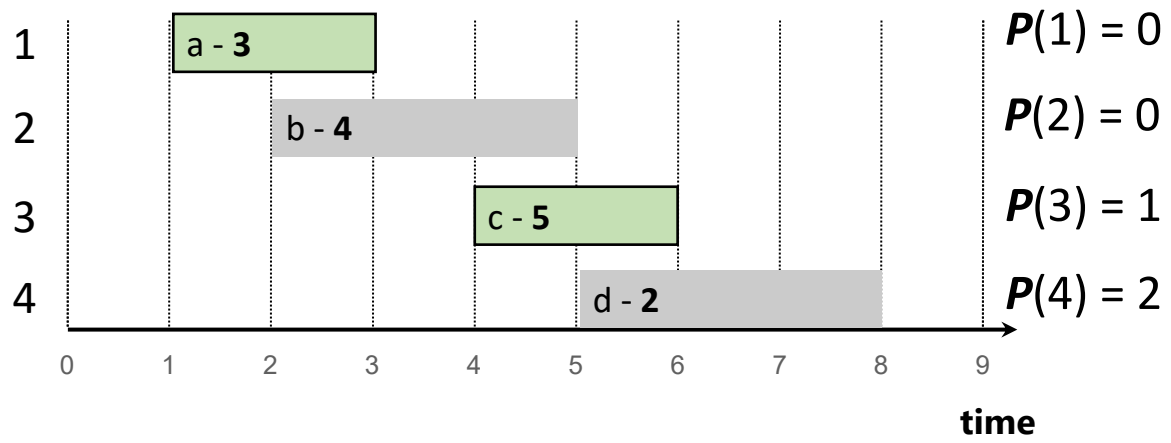
0	0
1	3
2	10
3	21
4	22
5	36
6	37
7	37
8	49

- # of recursive calls $\leq n \Rightarrow O(n)$.

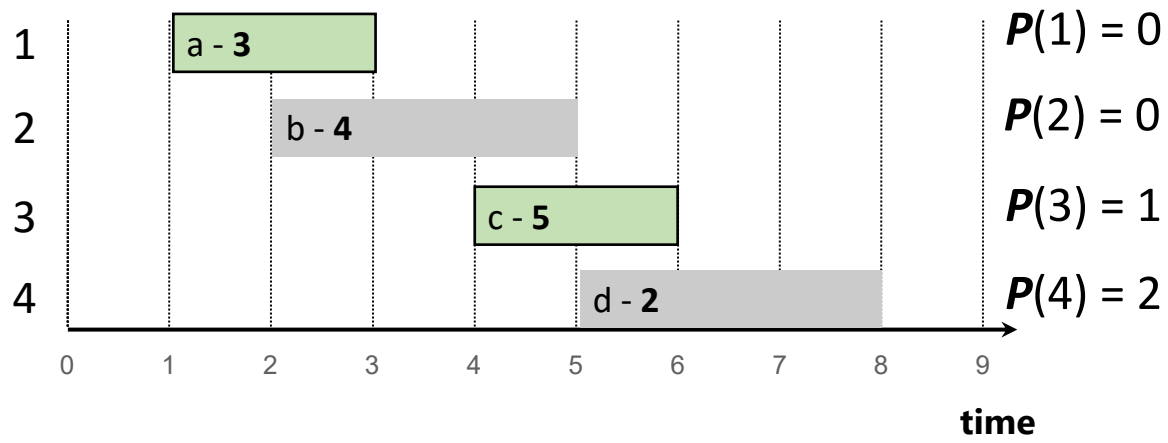
WISP : Finding a Solution



WISP : Finding a Solution



WISP : Finding a Solution

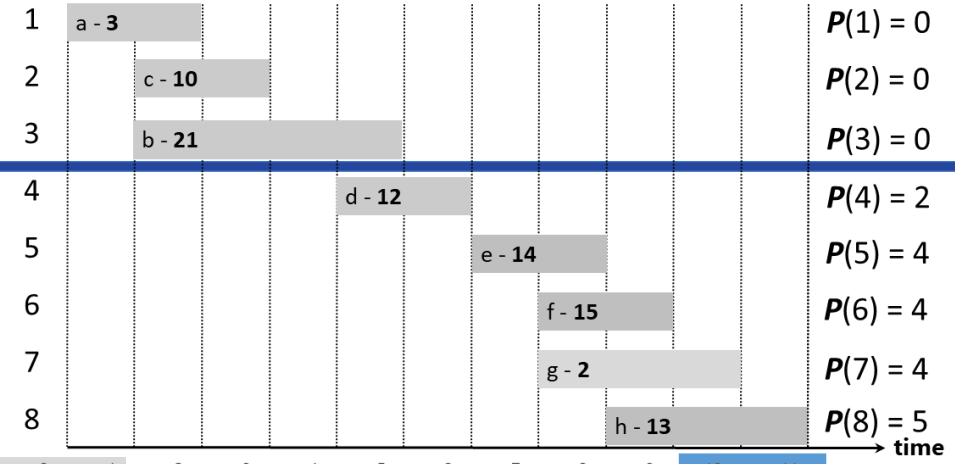


0	0
1	3
2	4
3	8
4	8

```
Run Max_Value(n)
Run Find-Solution(n)

Find-Solution(j) {
    if (j = 0)
        output nothing
    else if (w_j + M[p(j)] > M[j-1])
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}
```

WISP : Bottom-Up

- Bottom-up dynamic programming. Unwind recursion.




```
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)

Iterative_Max_Value {
   M[0] = 0
   for j = 1 to n
      M[j] = max(wj + M[p(j)], M[j-1])
}
```

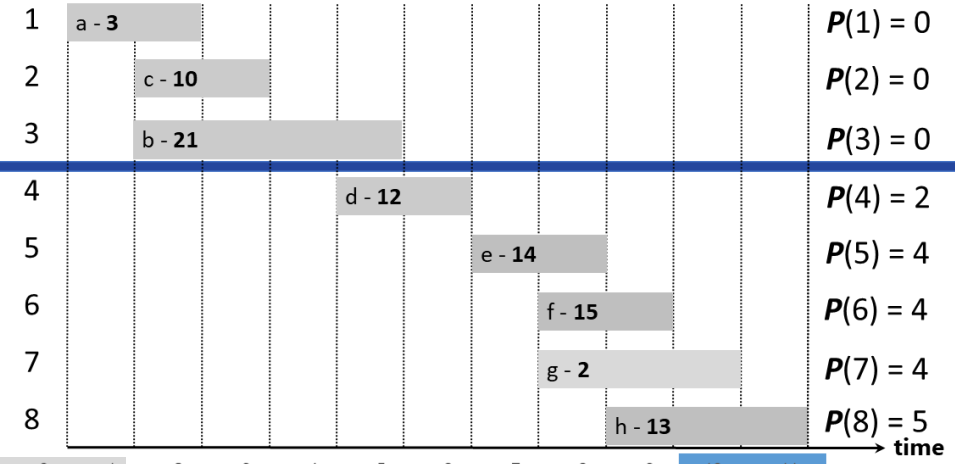
j = 1

(3+0,0)

0	0
1	
2	
3	
4	
5	
6	
7	
8	

WISP : Bottom-Up

- Bottom-up dynamic programming. Unwind recursion.




```
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)

Iterative_Max_Value {
   M[0] = 0
   for j = 1 to n
      M[j] = max(wj + M[p(j)], M[j-1])
}
```

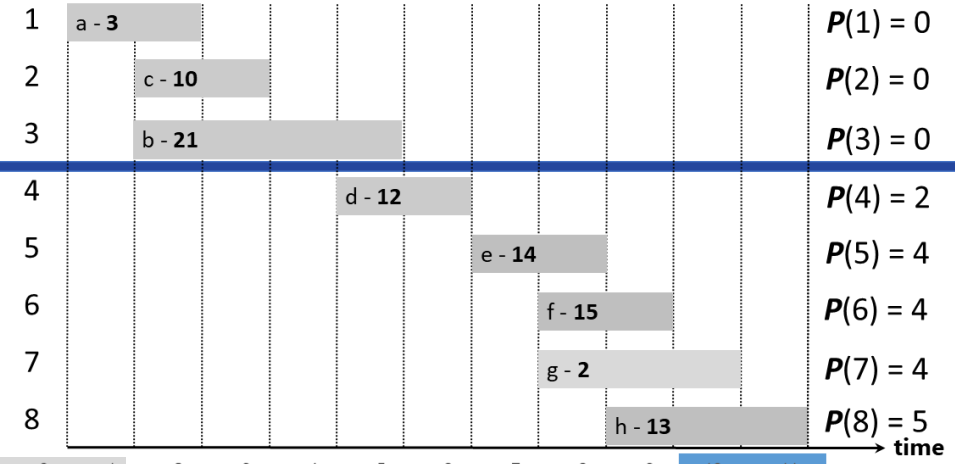
j = 1

(3+0,0)

0	0
1	3
2	
3	
4	
5	
6	
7	
8	

WISP : Bottom-Up

- Bottom-up dynamic programming. Unwind recursion.




```
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)

Iterative_Max_Value {
   M[0] = 0
   for j = 1 to n
      M[j] = max(wj + M[p(j)], M[j-1])
}
```

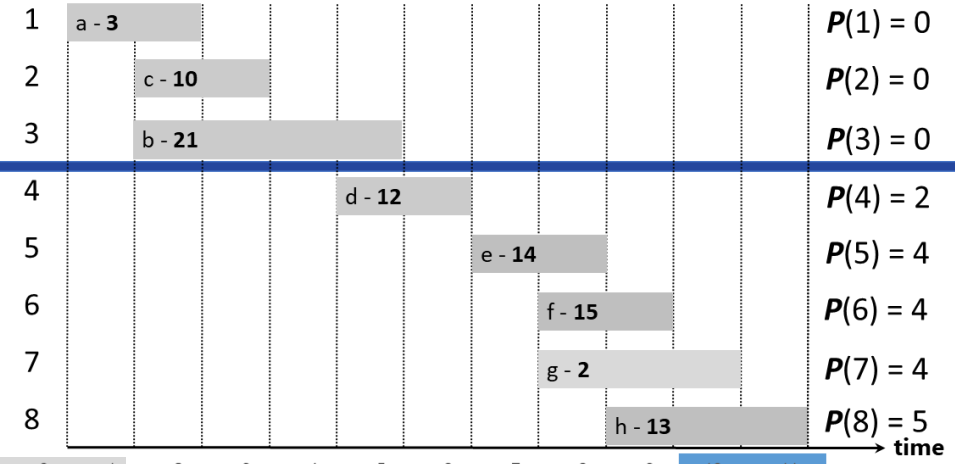
j = 2

(10 + 0 , 3)

0	0
1	3
2	
3	
4	
5	
6	
7	
8	

WISP : Bottom-Up

- Bottom-up dynamic programming. Unwind recursion.




```
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)

Iterative_Max_Value {
   M[0] = 0
   for j = 1 to n
      M[j] = max(wj + M[p(j)], M[j-1])
}
```

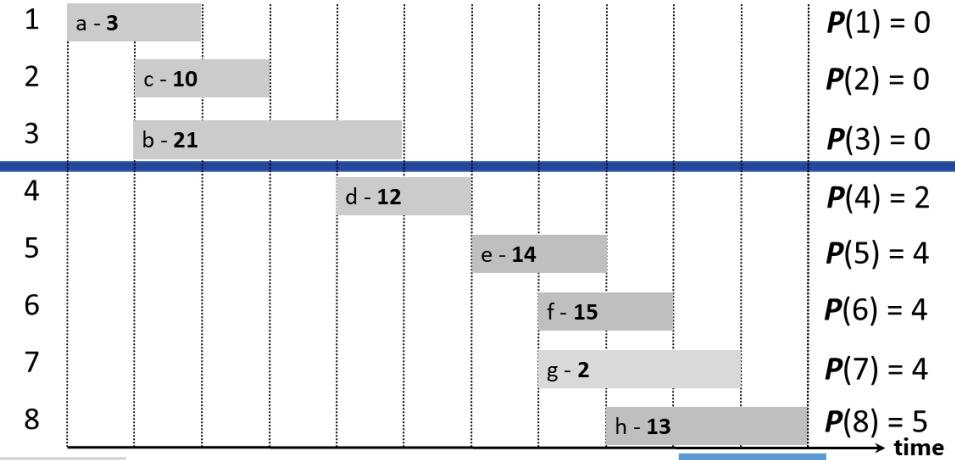
j = 2

(10 + 0 , 3)

0	0
1	3
2	10
3	
4	
5	
6	
7	
8	

WISP : Bottom-Up

- Bottom-up dynamic programming. Unwind recursion.




```
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)

Iterative_Max_Value {
   M[0] = 0
   for j = 1 to n
      M[j] = max(wj + M[p(j)], M[j-1])
}
```

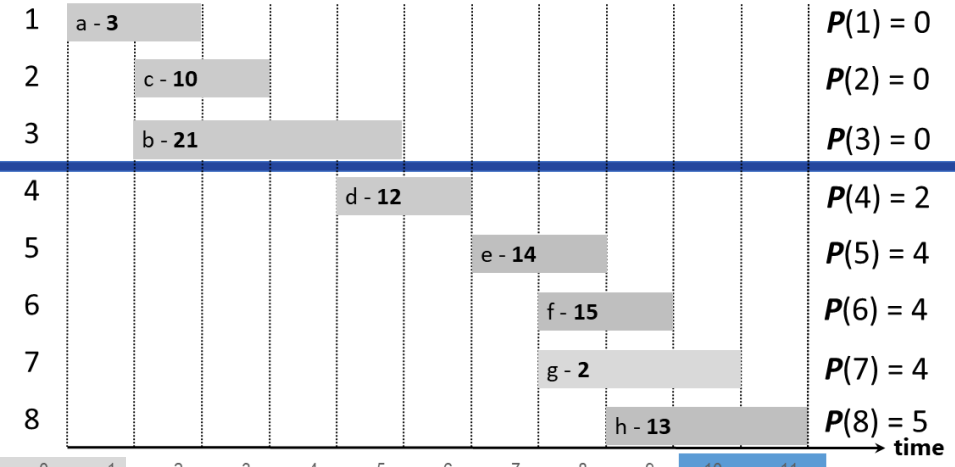
j = 3

(21 + 0 , 10)

0	0
1	3
2	10
3	
4	
5	
6	
7	
8	

WISP : Bottom-Up

- Bottom-up dynamic programming. Unwind recursion.




```
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)

Iterative_Max_Value {
   M[0] = 0
   for j = 1 to n
      M[j] = max(wj + M[p(j)], M[j-1])
}
```

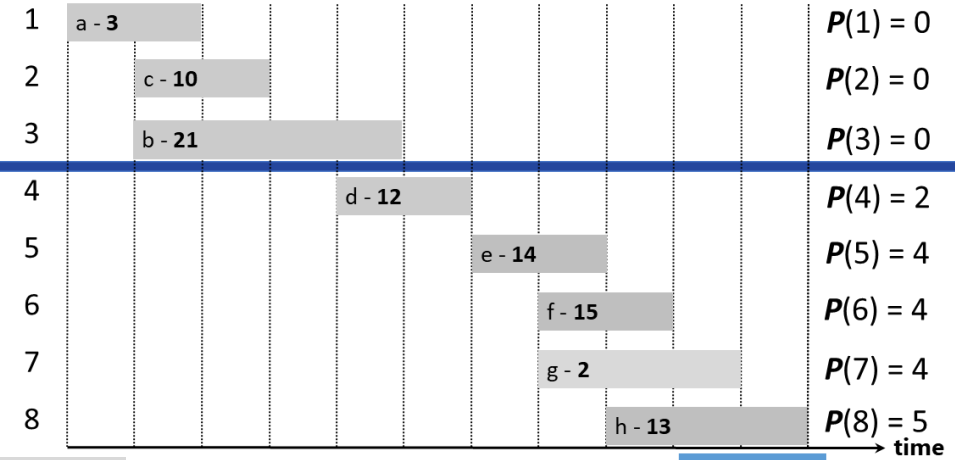
j = 3

(21 + 0 , 10)

0	0
1	3
2	10
3	21
4	
5	
6	
7	
8	

WISP : Bottom-Up

- Bottom-up dynamic programming. Unwind recursion.



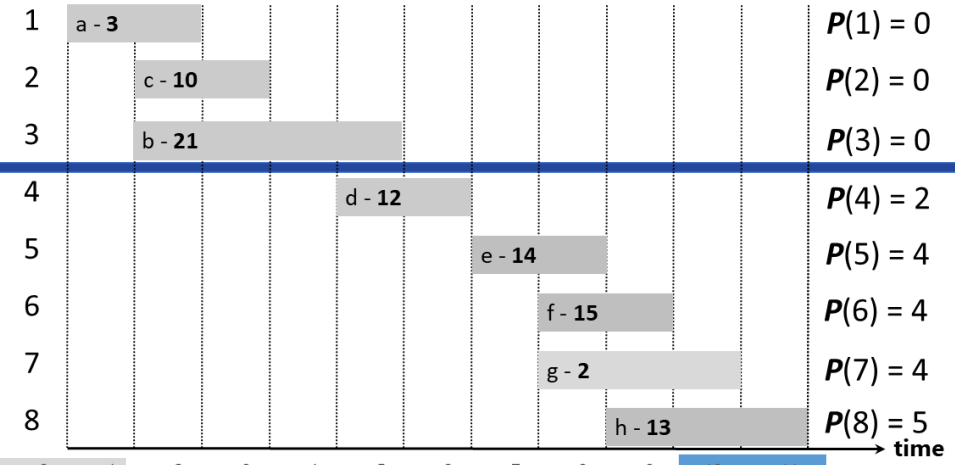

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Iterative_Max_Value {
   M[0] = 0
   for j = 1 to n
      M[j] = max(wj + M[p(j)], M[j-1])
}
```

j = 4

(12 + 10 , 21)

0	0
1	3
2	10
3	21
4	
5	
6	
7	
8	

WISP : Bottom-Up



- Bottom-up dynamic programming. Unwind recursion.

```
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)

Iterative_Max_Value {
   M[0] = 0
   for j = 1 to n
      M[j] = max(wj + M[p(j)], M[j-1])
}
```

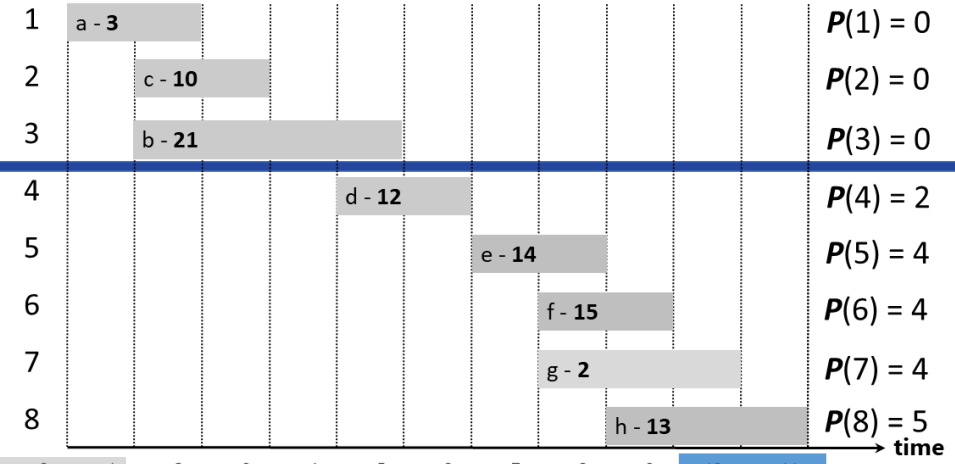
j = 4

(12 + 10 , 21)

0	0
1	3
2	10
3	21
4	22
5	
6	
7	
8	

WISP : Bottom-Up

- Bottom-up dynamic programming. Unwind recursion.




```
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)

Iterative_Max_Value {
   M[0] = 0
   for j = 1 to n
      M[j] = max(wj + M[p(j)], M[j-1])
}
```

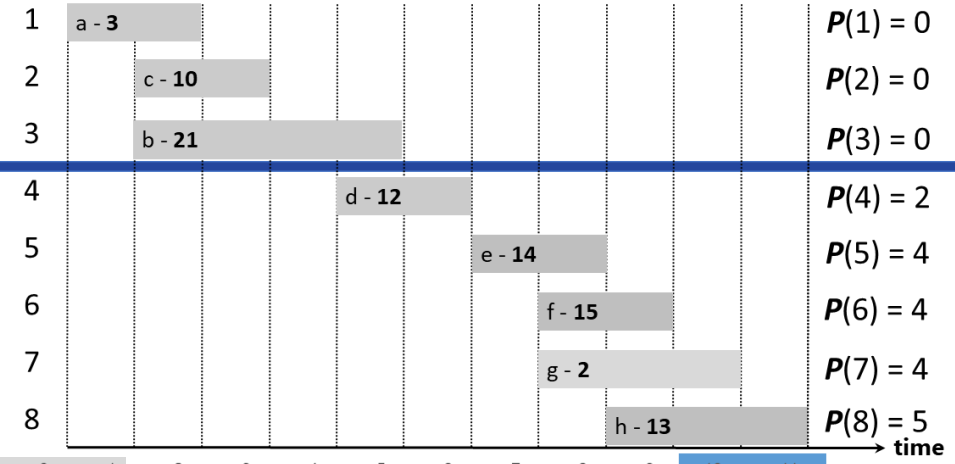
j = 5

(14 + 22 , 22)

0	0
1	3
2	10
3	21
4	22
5	
6	
7	
8	

WISP : Bottom-Up

- Bottom-up dynamic programming. Unwind recursion.




```
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)

Iterative_Max_Value {
   M[0] = 0
   for j = 1 to n
      M[j] = max(wj + M[p(j)], M[j-1])
}
```

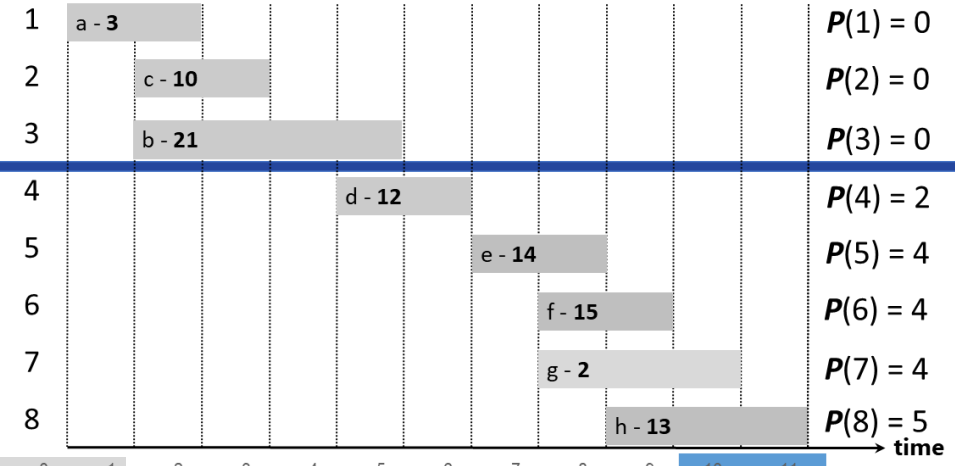
j = 5

(14 + 22 , 22)

0	0
1	3
2	10
3	21
4	22
5	36
6	
7	
8	

WISP : Bottom-Up

- Bottom-up dynamic programming. Unwind recursion.



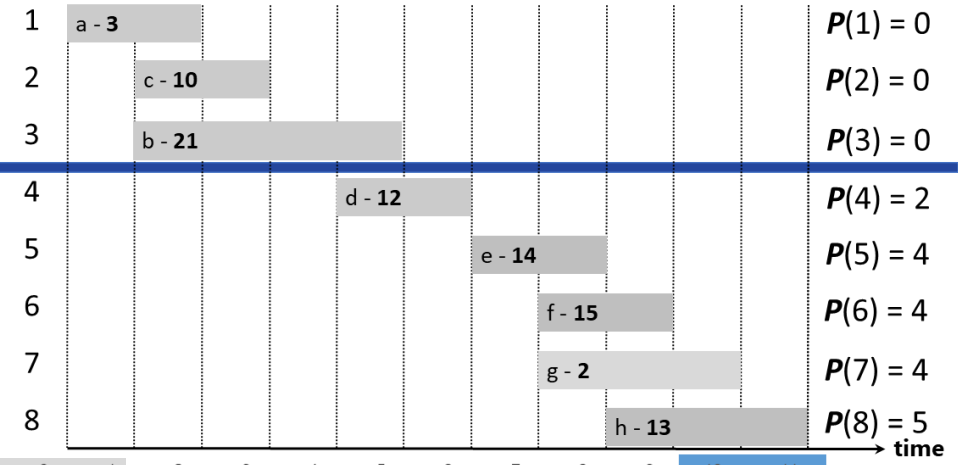

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Iterative_Max_Value {
   M[0] = 0
   for j = 1 to n
      M[j] = max(wj + M[p(j)], M[j-1])
}
```

j = 6

(15 + 22 , 22)

0	0
1	3
2	10
3	21
4	22
5	36
6	
7	
8	

WISP : Bottom-Up



- Bottom-up dynamic programming. Unwind recursion.

```
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)

Iterative_Max_Value {
   M[0] = 0
   for j = 1 to n
      M[j] = max(wj + M[p(j)], M[j-1])
}
```

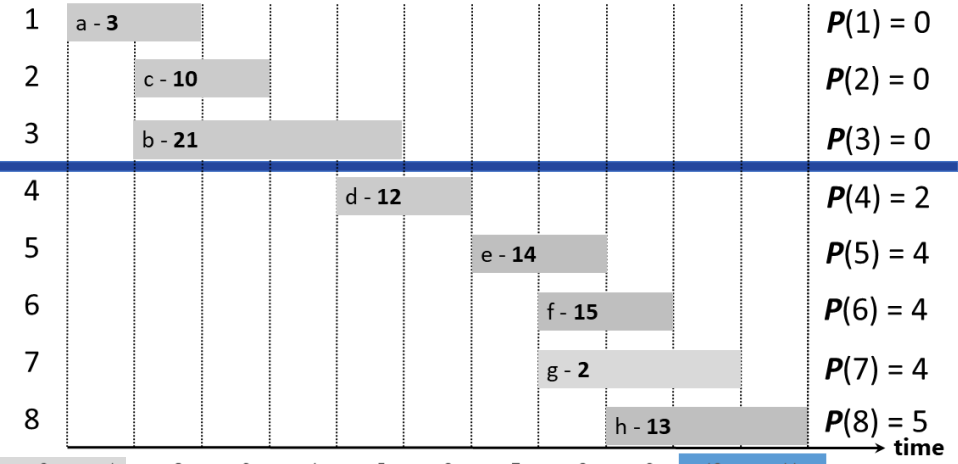
j = 6

(15 + 22 , 22)

0	0
1	3
2	10
3	21
4	22
5	36
6	37
7	
8	

WISP : Bottom-Up

- Bottom-up dynamic programming. Unwind recursion.




```
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)

Iterative_Max_Value {
   M[0] = 0
   for j = 1 to n
      M[j] = max(wj + M[p(j)], M[j-1])
}
```

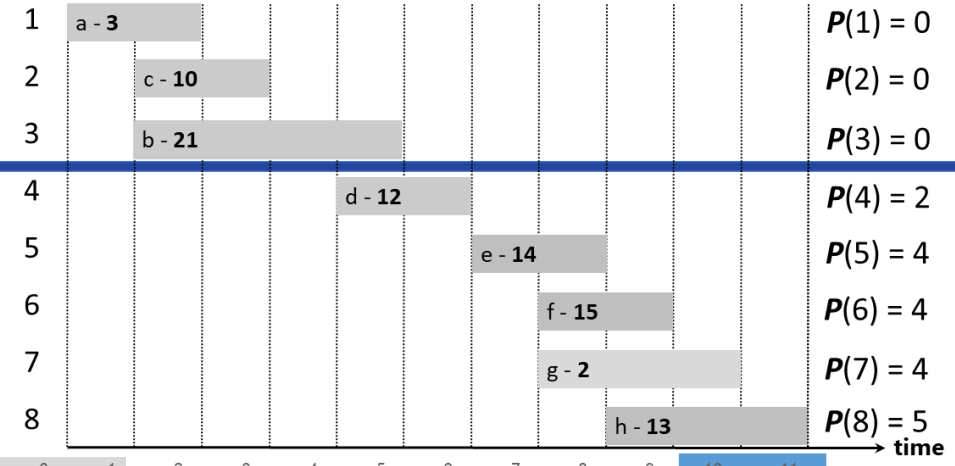
j = 7

(2 + 22 , 37)

0	0
1	3
2	10
3	21
4	22
5	36
6	37
7	
8	

WISP : Bottom-Up

- Bottom-up dynamic programming. Unwind recursion.




```
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)

Iterative_Max_Value {
   M[0] = 0
   for j = 1 to n
      M[j] = max(wj + M[p(j)], M[j-1])
}
```

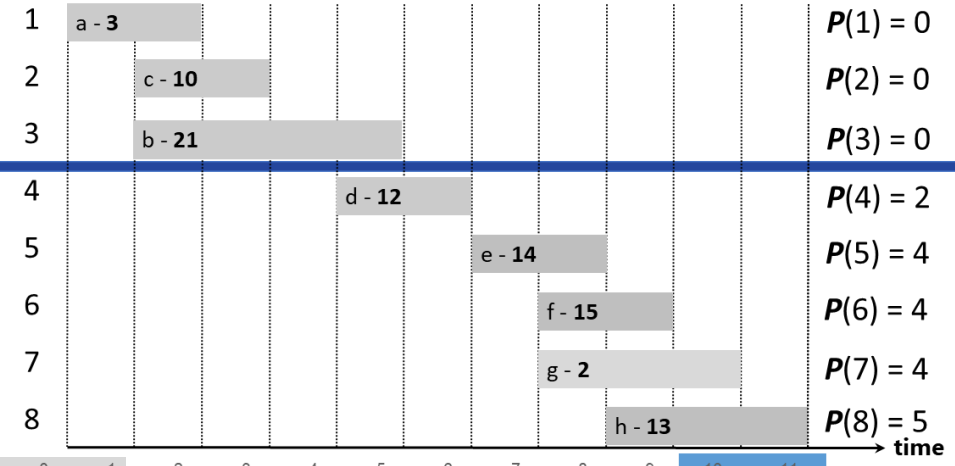
j = 7

(2 + 22 , 37)

0	0
1	3
2	10
3	21
4	22
5	36
6	37
7	37
8	

WISP : Bottom-Up

- Bottom-up dynamic programming. Unwind recursion.




```
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)

Iterative_Max_Value {
   M[0] = 0
   for j = 1 to n
      M[j] = max(wj + M[p(j)], M[j-1])
}
```

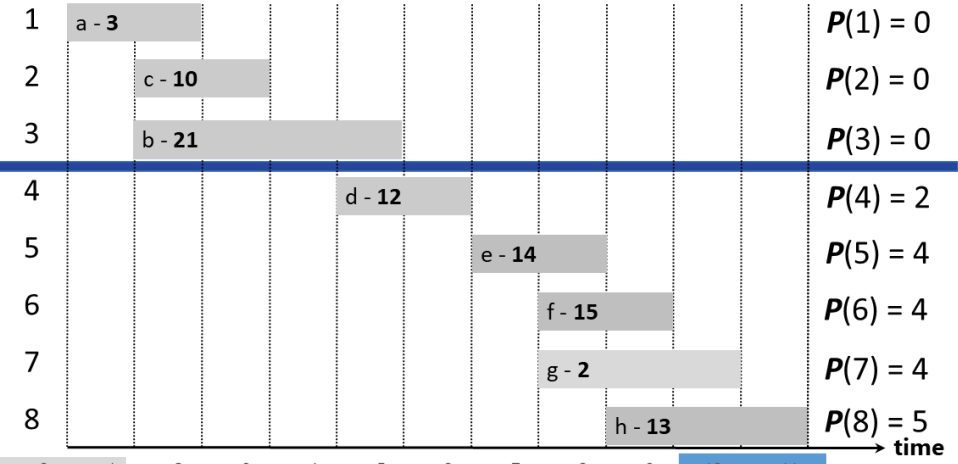
j = 8

(13 + 36 , 37)

0	0
1	3
2	10
3	21
4	22
5	36
6	37
7	37
8	

WISP : Bottom-Up

- Bottom-up dynamic programming. Unwind recursion.

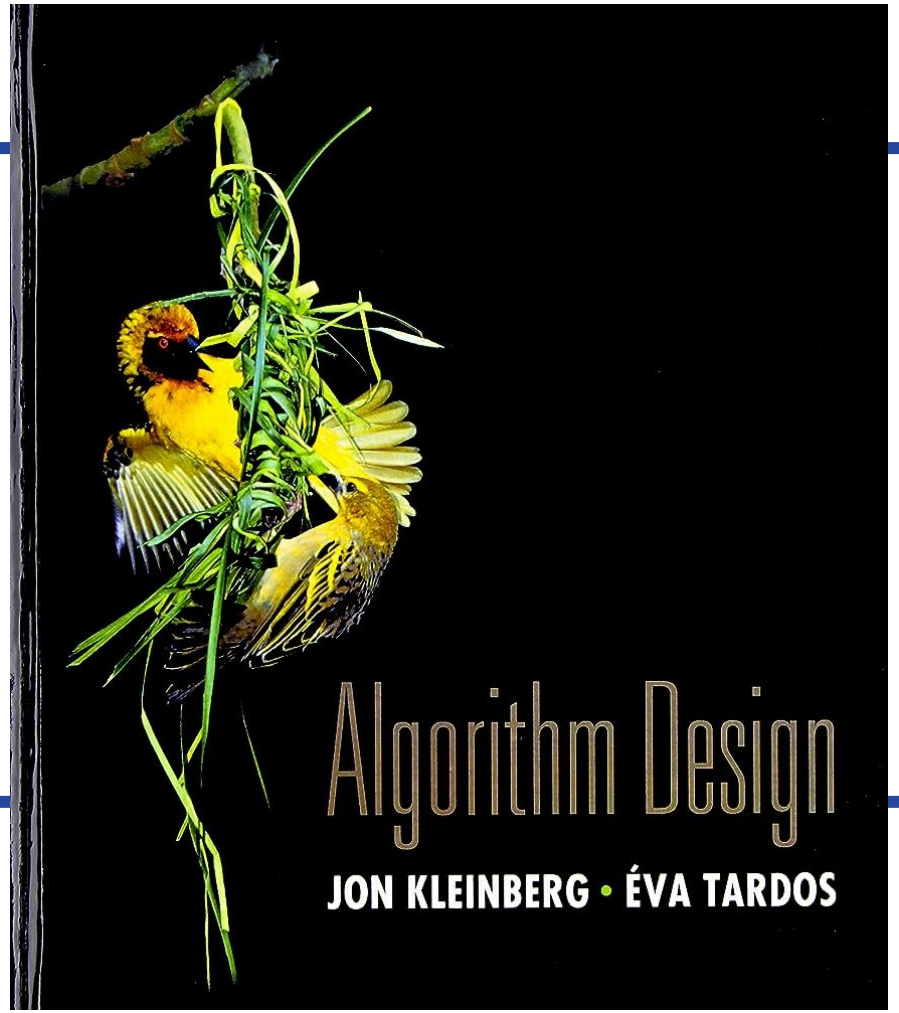



```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Iterative_Max_Value {
   M[0] = 0
   for j = 1 to n
      M[j] = max(wj + M[p(j)], M[j-1])
}
```

j = 8

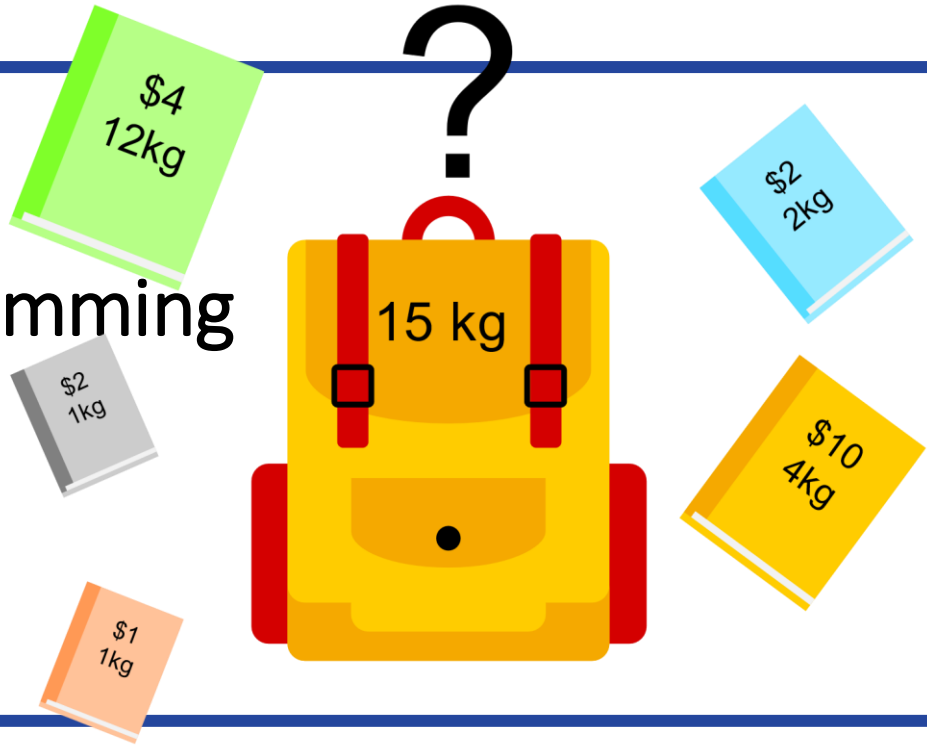
(13 + 36 , 37)

0	0
1	3
2	10
3	21
4	22
5	36
6	37
7	37
8	49



Chapter 6:
Dynamic Programming

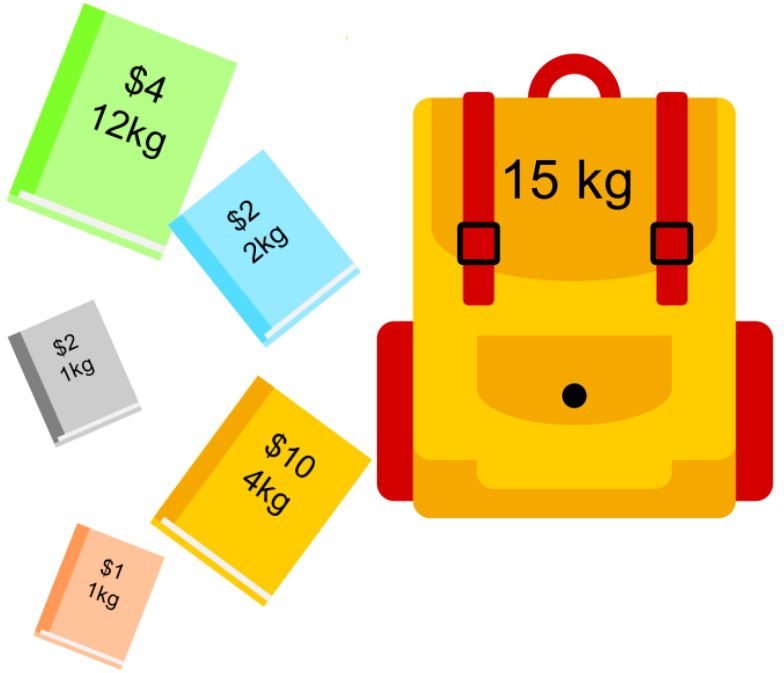
Section :
Knapsack Problem



Knapsack Problem

Knapsack problem:

- Given **n** objects ∈ ***U*** set with **weighs** $w_i > 0$ kilograms and has **value** $v_i > 0$
- And a "**knapsack**" with capacity of **C** kilograms
- **Goal:** Fill knapsack to maximize total value.



Output:

A subset $S \subset U$

Capacity constraint: $\sum_{a_i \in S} w_i \leq C$

Objective: Maximize $\sum_{a_i \in S} v_i$

Applications:

- Logistic problem involving transportation of freights
- A container/truck has a fixed maximum capacity
- Bunch of items each has a certain volume and a profit (return)
- Transporter would like to select items to maximize its profit

Knapsack Problem

Greedy Approach

- Select the **most profitable** item
- Add if it fits remaining capacity
- Repeat

C = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Problem

Greedy Approach

- Select the **most profitable** item
- Add if it fits remaining capacity
- Repeat

C = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

28 + 6 + 1 = **35**

Knapsack Problem

Greedy Approach

- Select the **least weighted** item
- Add if it fits remaining capacity
- Repeat

C = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Problem

Greedy Approach

- Select the **least weighted** item
- Add if it fits remaining capacity
- Repeat

C = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

1 + 6 + 18 = **25**

Knapsack Problem

Greedy Approach

- Select the item with highest $\frac{V_i}{W_i}$ ratio
- Add if it fits remaining capacity
- Repeat

C = 11

Item	Value	Weight	Ratio
1	1	1	1
2	6	2	3
3	18	5	3.6
4	22	6	3.6
5	28	7	4

28+ 6 + 1 = **35**

Knapsack Problem

Greedy Approach

- Select the item with highest $\frac{V_i}{W_i}$ ratio
- Add if it fits remaining capacity
- Repeat

C = 11

Item	Value	Weight	Ratio
1	1	1	1
2	6	2	3
3	18	5	3.6
4	22	6	3.6
5	28	7	4

28+ 6 + 1 = **35**

Knapsack Problem: Dynamic Programming

Our goal is to find **`Max_Value(n)`**

Either $Item_n$ is not part of the solution

- v_n is not counted in **`Max_Value(n)`**
- Some subset of $Item_1, \ldots, Item_{n-1}$ is solution set
- Analyze **`Max_Value(n-1)`**

C = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Or $Item_n$ is part of the solution

- v_n is counted in opt-val(n)
- Some subset of $Item_1, \ldots, Item_{n-1}$ is in solution set in addition to $Item_n$
- Analyze **`Max_Value(n-1)`**

We don't even know if we have enough room for more

Knapsack Problem: Dynamic Programming

Our goal is to find **`Max_Value(n)`**

Either $Item_n$ is not part of the solution

- v_n is not counted in **`Max_Value(n,C)`**
- Some subset of $Item_1, \ldots, Item_{n-1}$ is solution set
- Remaining capacity is C
- Analyze **`Max_Value(n-1,C)`**

C = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Or $Item_n$ is part of the solution

- v_n is counted in **`Max_Value(n,C)`**
- Some subset of $Item_1, \ldots, Item_{n-1}$ is in solution set in addition to $Item_n$
- Remaining capacity is C - w_n
- Analyze **`Max_Value(n-1,C-`** w_n**`)`**

Knapsack Problem: Dynamic Programming

Our goal is to find **`Max_Value(n)`**

$$\texttt{Max\_Value}(n) = max \begin{cases} 0 & \text{if } C <= 0 \\ 0 & \text{if } n = 0 \\ v_n + \texttt{Max\_Value(n-1, C-}\boldsymbol{w_n}\texttt{)} & \text{if } Item_n \in \text{Optimal Set} \\ \texttt{Max\_Value(n-1, C)} & \text{if } Item_n \notin \text{Optimal Set} \end{cases}$$

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

C = 11

Knapsack Problem: Dynamic Programming

Brute-force Algorithm

```
Input: n, v_1,…,v_n , w_1,…,w_n , C

Max_Weight(n) {
   if (n == 0 || C == 0)                                          O(nlogn)
      return 0

   if (w_n > C)                                                   O(nlogn)
      return Max_Value(n-1,C)

   else
      return max(Max_Value(n-1,C), v_i + Max_Value(n-1,C- w_n))   O(2^n)
}
```

Knapsack Problem: Dynamic Programming

Memoization – Top-down approach

```
Input: n, v1,…,vn , w1,…,wn , C

Max_Weight(n) {
   if (n == 0 || C == 0)
      return 0

   if (M[n][C] != -1)
      return M[n][C]

   if (wn > C)
      M[n][C] = Max_Value(n-1,C)

   else
      M[n][C] =max(Max_Value(n-1,C), vn + Max_Value(n-1,C- wn))

   return M[n][C]
}
```

Knapsack Problem: Dynamic Programming

Tabular – Bottom-up approach

```
Input: n, w_1,…,w_N, v_1,…,v_N

for c = 0 to C
   M[0, c] = 0

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

return M[n, C]
```

Knapsack Problem: Dynamic Programming

W + 1 →

n + 1 ↓	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

n + 1 ↓	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
{ 1 }	-1	1	1	1	1	1	1	-1	-1	1	-1	1
{ 1, 2 }	-1	-1	-1	-1	7	7	7	-1	-1	-1	-1	7
{ 1, 2, 3 }	-1	-1	-1	-1	7	18	-1	-1	-1	-1	-1	25
{ 1, 2, 3, 4 }	-1	-1	-1	-1	7	-1	-1	-1	-1	-1	-1	40
{ 1, 2, 3, 4, 5 }	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	40

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

C = 11

Knapsack Problem: Dynamic Programming

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

C = 11

```
Find_solution(n,C) {
   if (n == 0 || C == 0)
      return 0

   else
      if (Max_Value(n-1,C-w_n) + v_i > Max_Value(n-1,C))
         print n
         return Find_solution(n-1,C-w_n)

      else
         return Find_solution(n-1,C)
}
```

n = 5

Knapsack Problem: Dynamic Programming

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

C = 11

```
Find_solution(n,C) {
   if (n == 0 || C == 0)
      return 0

   else
      if (Max_Value(n-1,C-w_n) + v_i > Max_Value(n-1,C))
         print n
         return Find_solution(n-1,C-w_n)

      else
         return Find_solution(n-1,C)
}
```

n = 5

(28+7, 40)

Knapsack Problem: Dynamic Programming

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

C = 11

```
Find_solution(n,C) {
   if (n == 0 || C == 0)
      return 0

   else
      if (Max_Value(n-1,C-w_n) + v_i > Max_Value(n-1,C))
         print n
         return Find_solution(n-1,C-w_n)

      else
         return Find_solution(n-1,C)
}
```

n = 4

(22+18, 25)

Knapsack Problem: Dynamic Programming

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

C = 11

```
Find_solution(n,C) {
   if (n == 0 || C == 0)
      return 0

   else
      if (Max_Value(n-1,C-w_n) + v_i > Max_Value(n-1,C))
         print n
         return Find_solution(n-1,C-w_n)

      else
         return Find_solution(n-1,C)
}
```

n = 3

(18+0, 7)

OPT: { 4, 3 }
value = 22 + 18 = 40

Thanks a lot

If you are taking a Nap, **wake up**.....***Lecture OVER***