

# CSE 421

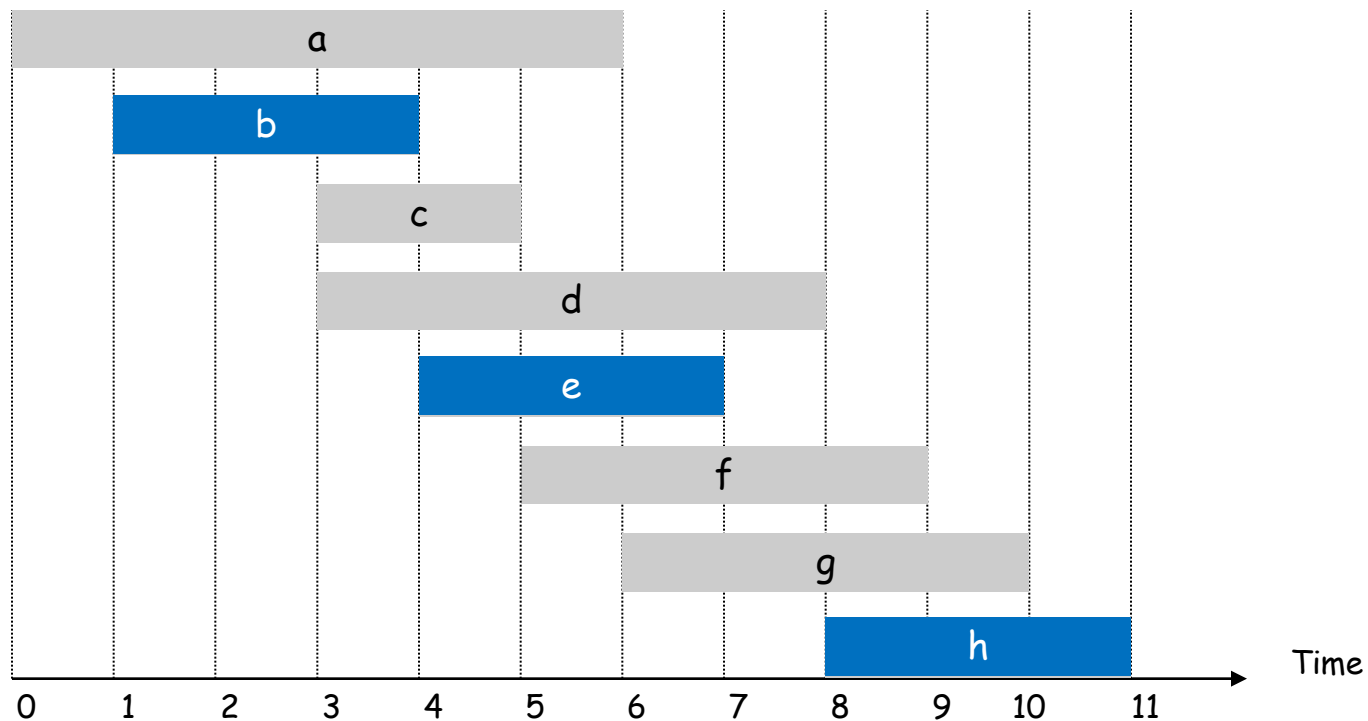
## Weighted Interval Scheduling, Knapsack, RNA Secondary Structure

Shayan Oveis Gharan

# Weighted Interval Scheduling

# Interval Scheduling

- Job  $j$  starts at  $s(j)$  and finishes at  $f(j)$  and has **weight**  $w_j$
- Two jobs **compatible** if they don't overlap.
- Goal: find maximum **weight** subset of mutually compatible jobs.



# Weighted Job Scheduling by Induction

**Sorting Idea:** Label jobs by finishing time  $f(1) \leq \dots \leq f(n)$

Let  $OPT(j)$  denote the  $OPT$  solution of  $1, \dots, j$

To solve  $OPT(j)$ : **Characterize** the Optimum solution.

**Case 1:**  $OPT(j)$  has job  $j$ .

- So, all jobs  $i$  that are not compatible with  $j$  are not  $OPT(j)$
- Let  $p(j)$  = largest index  $i < j$  such that job  $i$  is compatible with  $j$ .
- So  $OPT(j) = OPT(p(j)) \cup \{j\}$ .

**Case 2:**  $OPT(j)$  does not select job  $j$ .

- Then,  $OPT(j) = OPT(j - 1)$

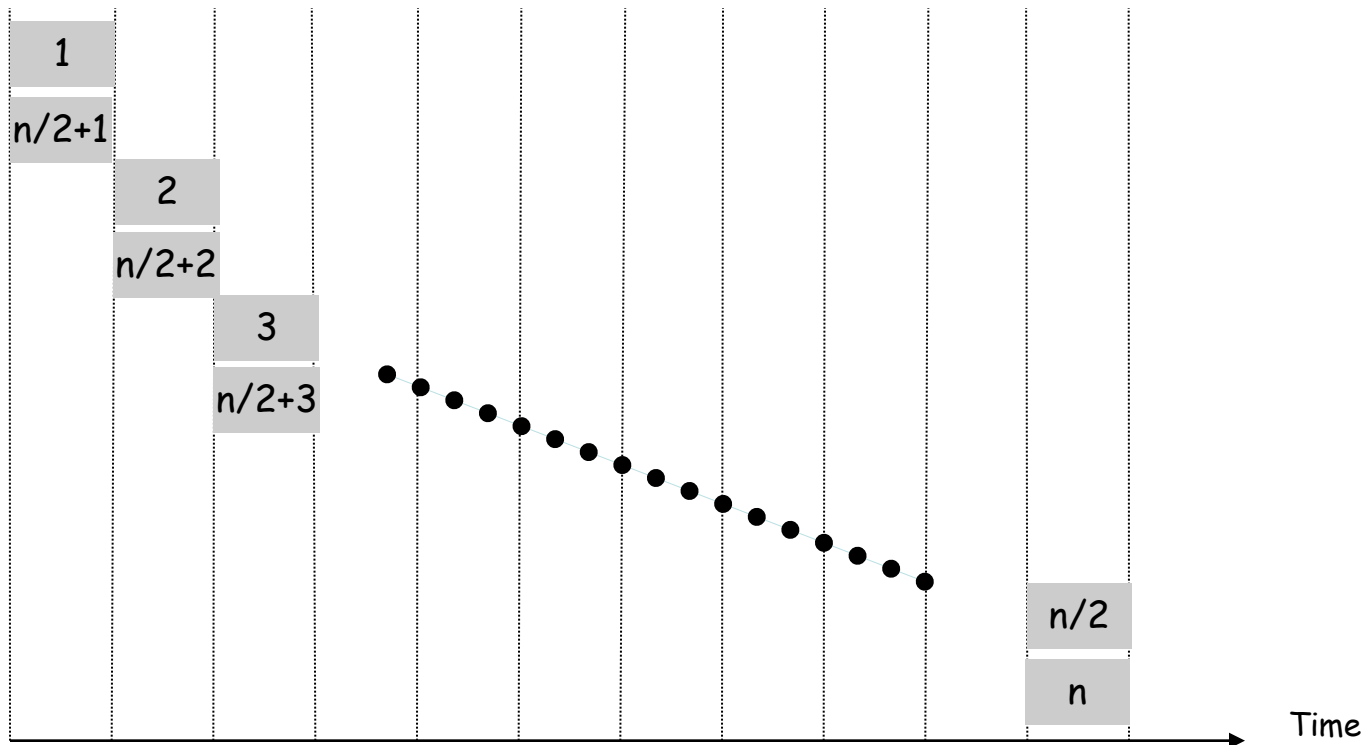
$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max(w_j + OPT(p(j)), OPT(j - 1)) & \text{o. w.} \end{cases}$$

# Impact of Sorting

Suppose we do not sort.

Consider jobs  $n/2+1, \dots, n$ . These decisions have no impact on one another.

How many subproblems do we get?  $\geq 2^{\lceil n/2 \rceil}$



# Algorithm

**Input:**  $n, s(1), \dots, s(n)$  and  $f(1), \dots, f(n)$  and  $w_1, \dots, w_n$ .

**Sort** jobs by finish times so that  $f(1) \leq f(2) \leq \dots \leq f(n)$ .

**Compute**  $p(1), p(2), \dots, p(n)$

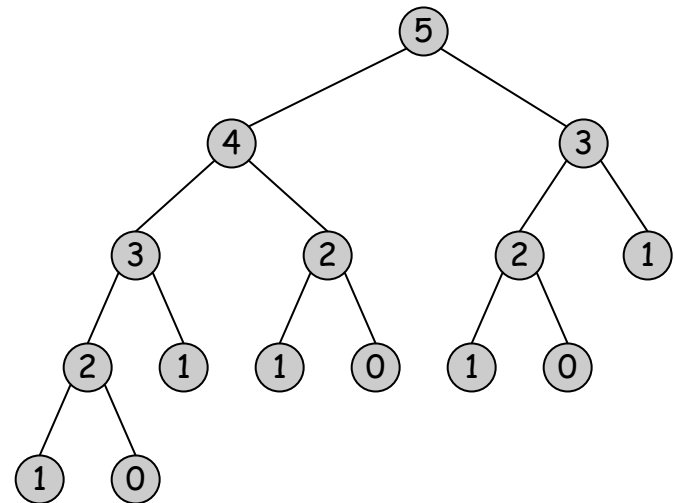
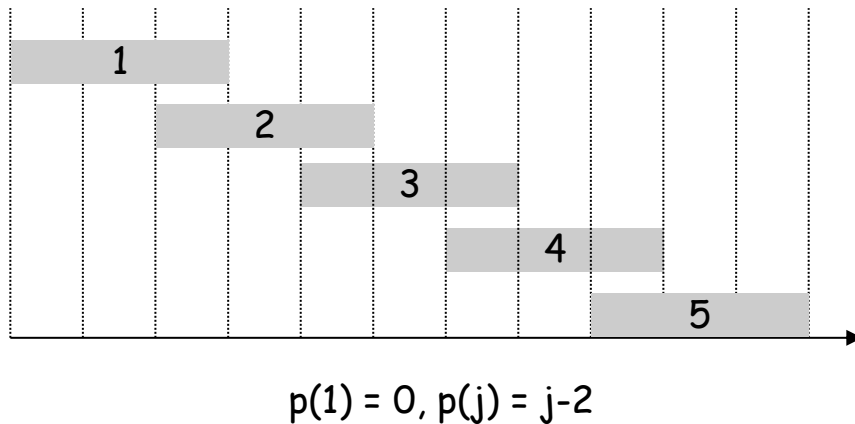
```
Compute-Opt(j) {  
    if (j = 0)  
        return 0  
    else  
        return max( $w_j + \text{Compute-Opt}(p(j))$ ,  $\text{Compute-Opt}(j-1)$ )  
}
```

# Recursive Algorithm Fails

Even though we have only  $n$  subproblems, we do not **store** the solution to the subproblems

➤ So, we may re-solve the same problem many many times.

**Ex.** Number of recursive calls for family of "layered" instances grows like Fibonacci sequence



# Algorithm with Memoization

**Memoization.** Compute and Store the solution of each sub-problem in a cache the first time that you face it. lookup as needed.

**Input:**  $n, s(1), \dots, s(n)$  and  $f(1), \dots, f(n)$  and  $w_1, \dots, w_n$ .

**Sort** jobs by finish times so that  $f(1) \leq f(2) \leq \dots f(n)$ .

**Compute**  $p(1), p(2), \dots, p(n)$

**for**  $j = 1$  to  $n$

$M[j] = \text{empty}$

$M[0] = 0$

**M-Compute-Opt**( $j$ ) {

**if** ( $M[j]$  is empty)

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

**return**  $M[j]$

}



# Bottom up Dynamic Programming

You can also avoid recursion

- recursion may be easier conceptually when you use induction

**Input:**  $n, s(1), \dots, s(n)$  and  $f(1), \dots, f(n)$  and  $w_1, \dots, w_n$ .

**Sort** jobs by finish times so that  $f(1) \leq f(2) \leq \dots \leq f(n)$ .

**Compute**  $p(1), p(2), \dots, p(n)$

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

**Output** M[n]

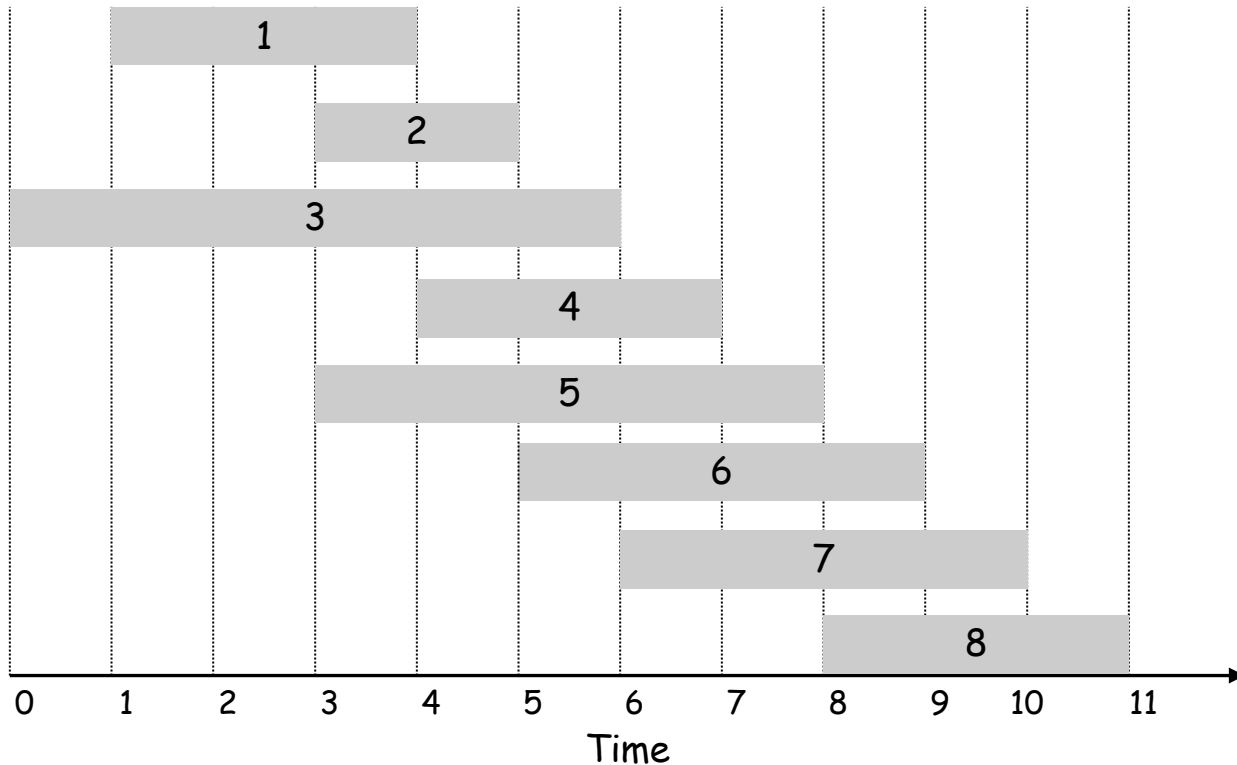
**Claim:** M[j] is value of OPT(j)

**Timing:** Easy. Main loop is O(n); sorting is O(n log n)

# Example

Label jobs by finishing time:  $f(1) \leq \dots \leq f(n)$ .

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

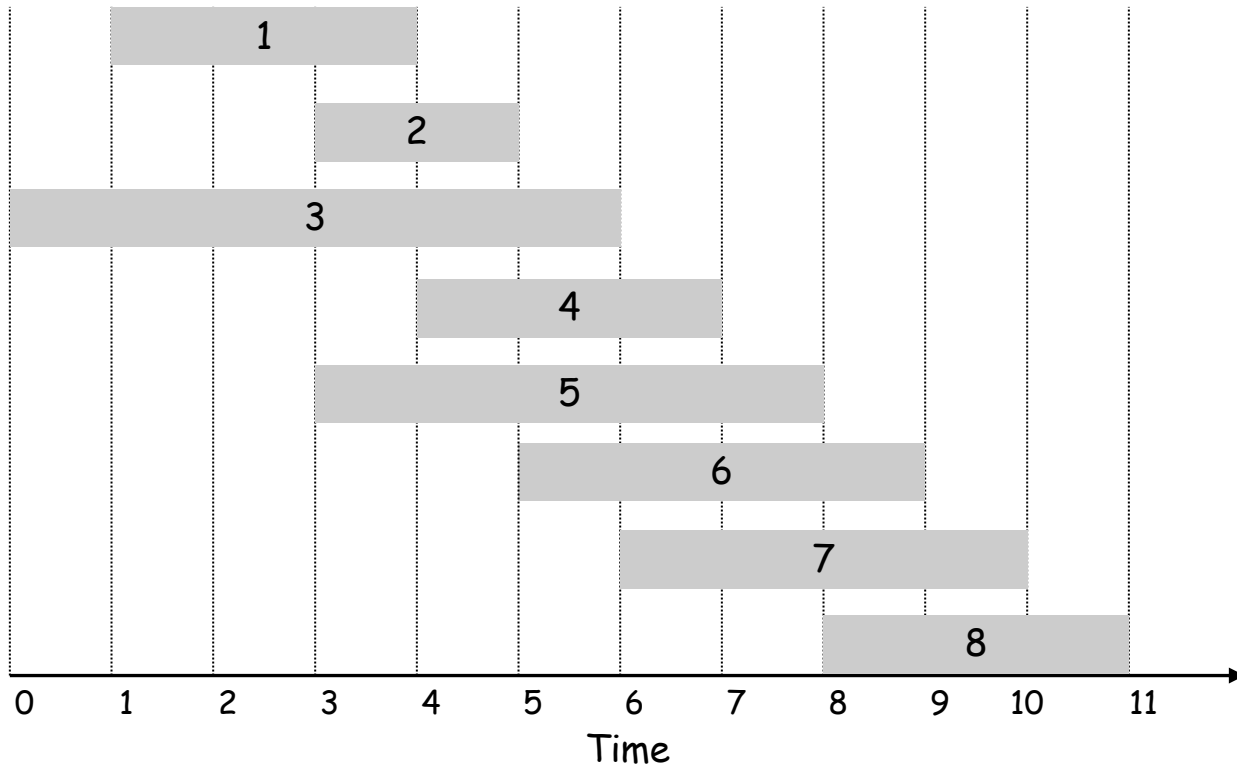


$j$	$w_j$	$p(j)$	$OPT(j)$
0			0
1	3	0	
2	4	0	
3	1	0	
4	3	1	
5	4	0	
6	3	2	
7	2	3	
8	4	5	

# Example

Label jobs by finishing time:  $f(1) \leq \dots \leq f(n)$ .

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

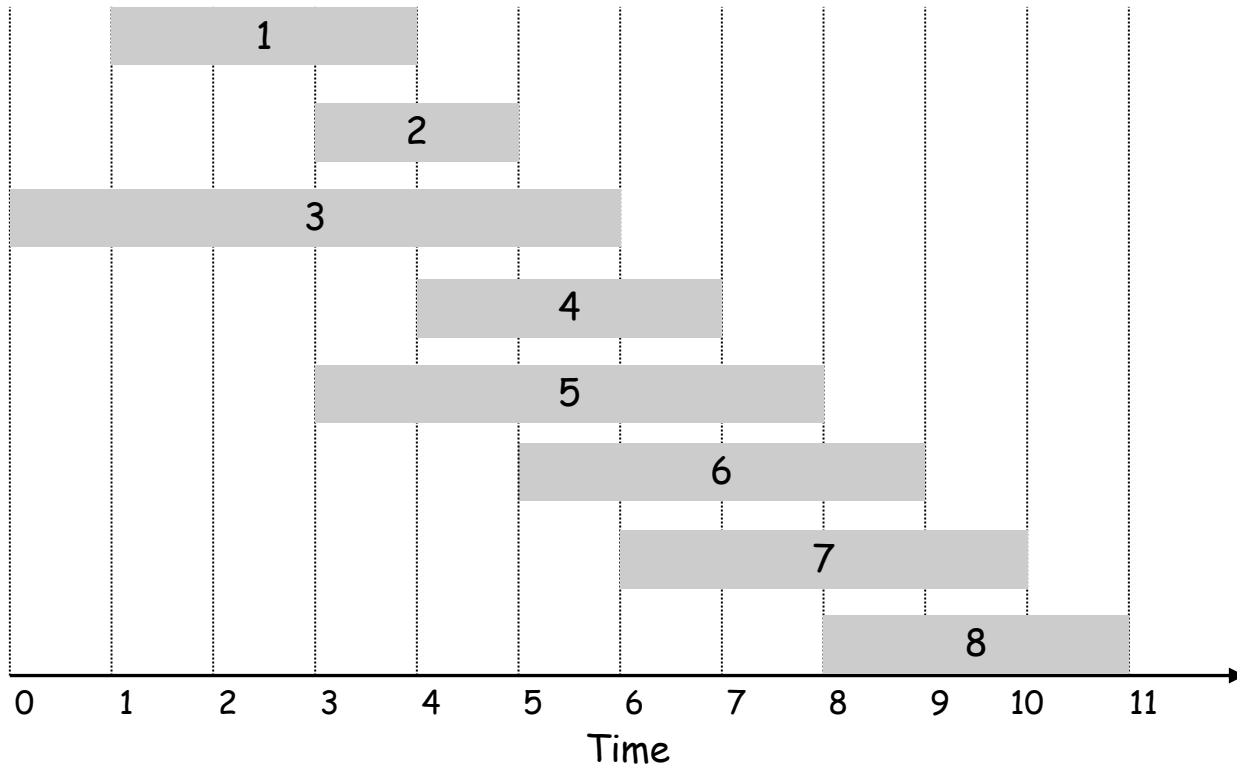


$j$	$w_j$	$p(j)$	OPT( $j$ )
0			0
1	3	0	3
2	4	0	
3	1	0	
4	3	1	
5	4	0	
6	3	2	
7	2	3	
8	4	5	

# Example

Label jobs by finishing time:  $f(1) \leq \dots \leq f(n)$ .

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

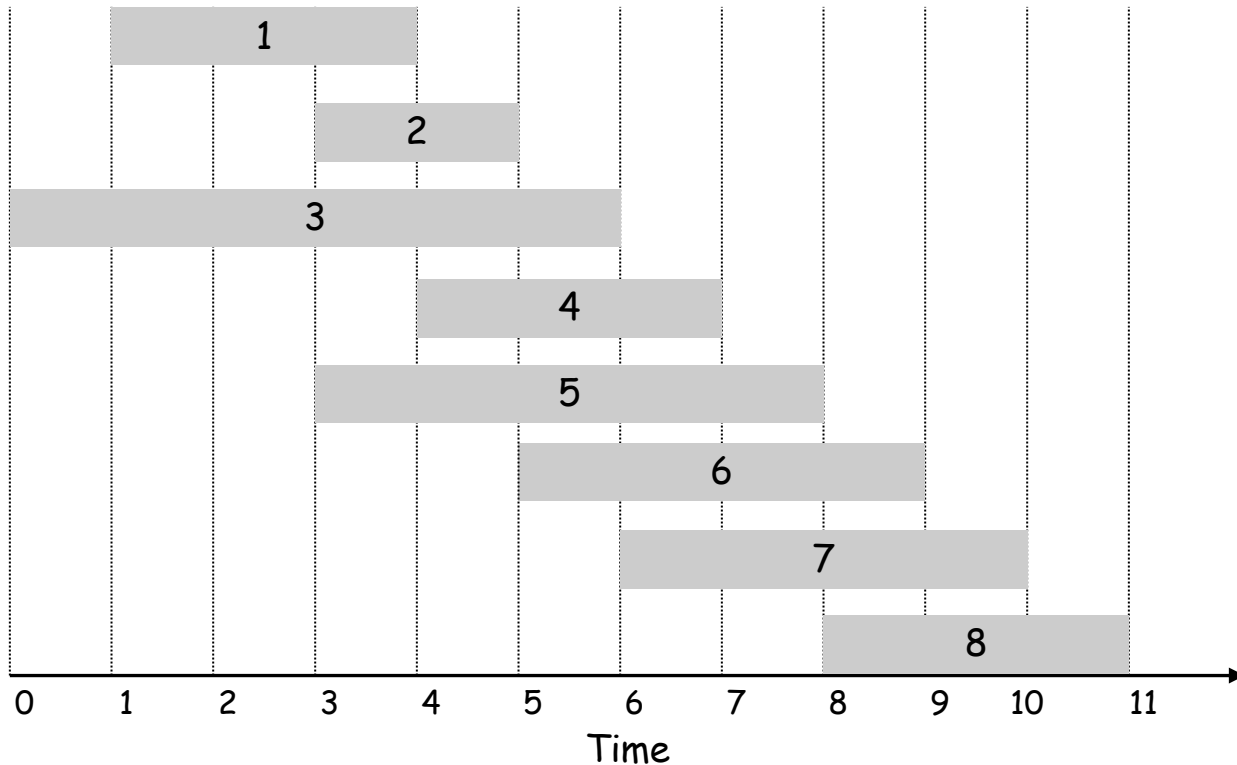


$j$	$w_j$	$p(j)$	OPT( $j$ )
0			0
1	3	0	3
2	4	0	4
3	1	0	
4	3	1	
5	4	0	
6	3	2	
7	2	3	
8	4	5	

# Example

Label jobs by finishing time:  $f(1) \leq \dots \leq f(n)$ .

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

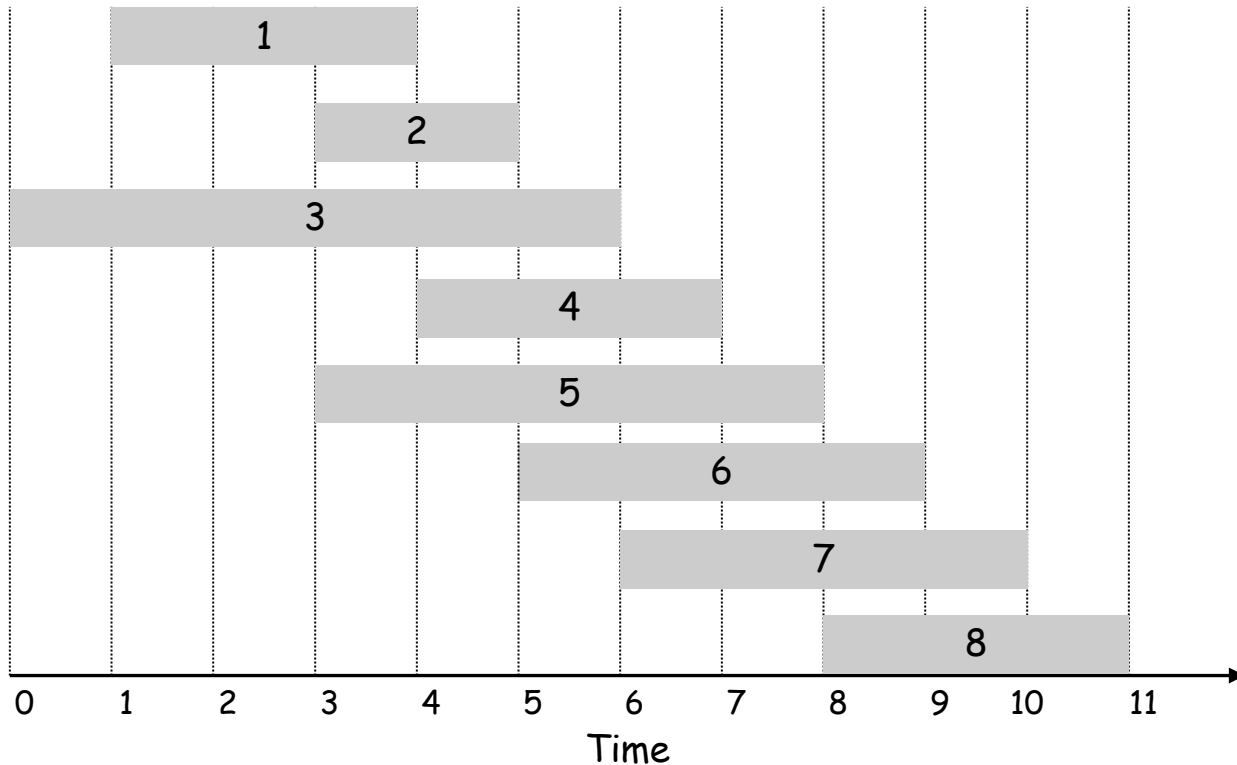


$j$	$w_j$	$p(j)$	OPT( $j$ )
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	
5	4	0	
6	3	2	
7	2	3	
8	4	5	

# Example

Label jobs by finishing time:  $f(1) \leq \dots \leq f(n)$ .

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

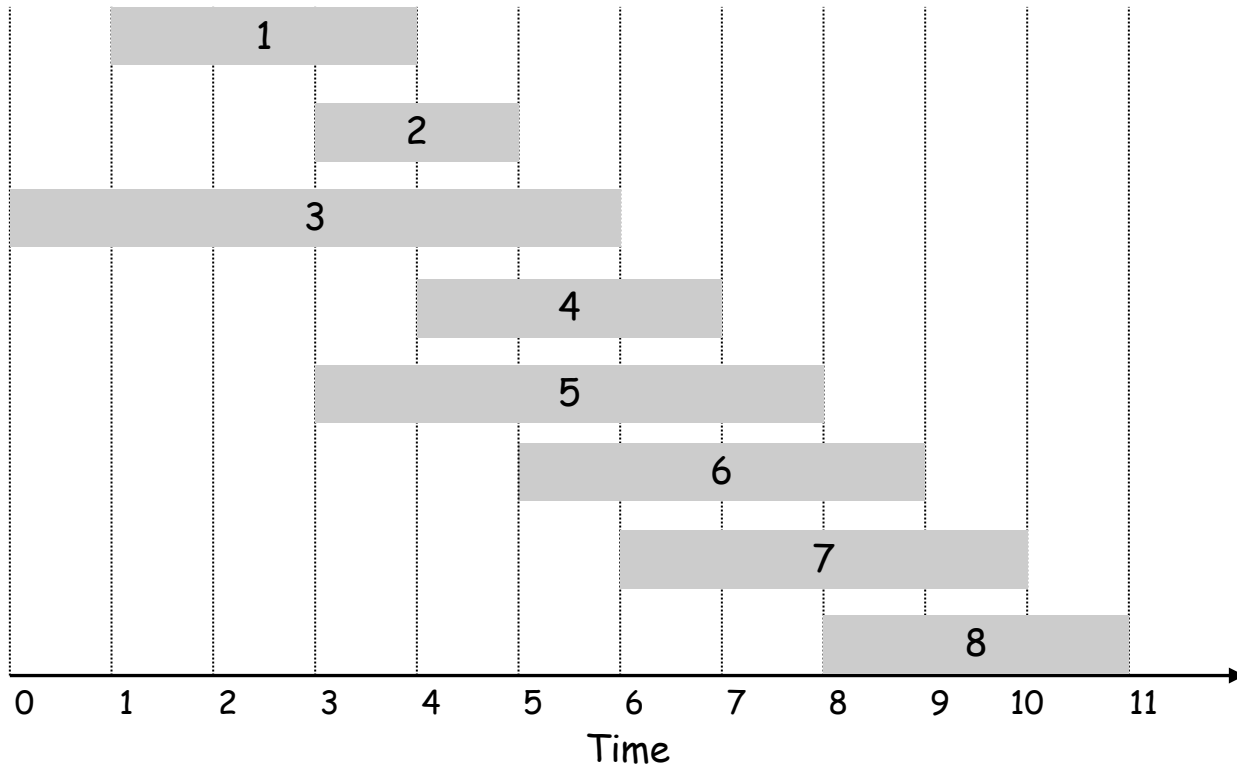


j	$w_j$	$p(j)$	OPT(j)
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	
6	3	2	
7	2	3	
8	4	5	

# Example

Label jobs by finishing time:  $f(1) \leq \dots \leq f(n)$ .

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

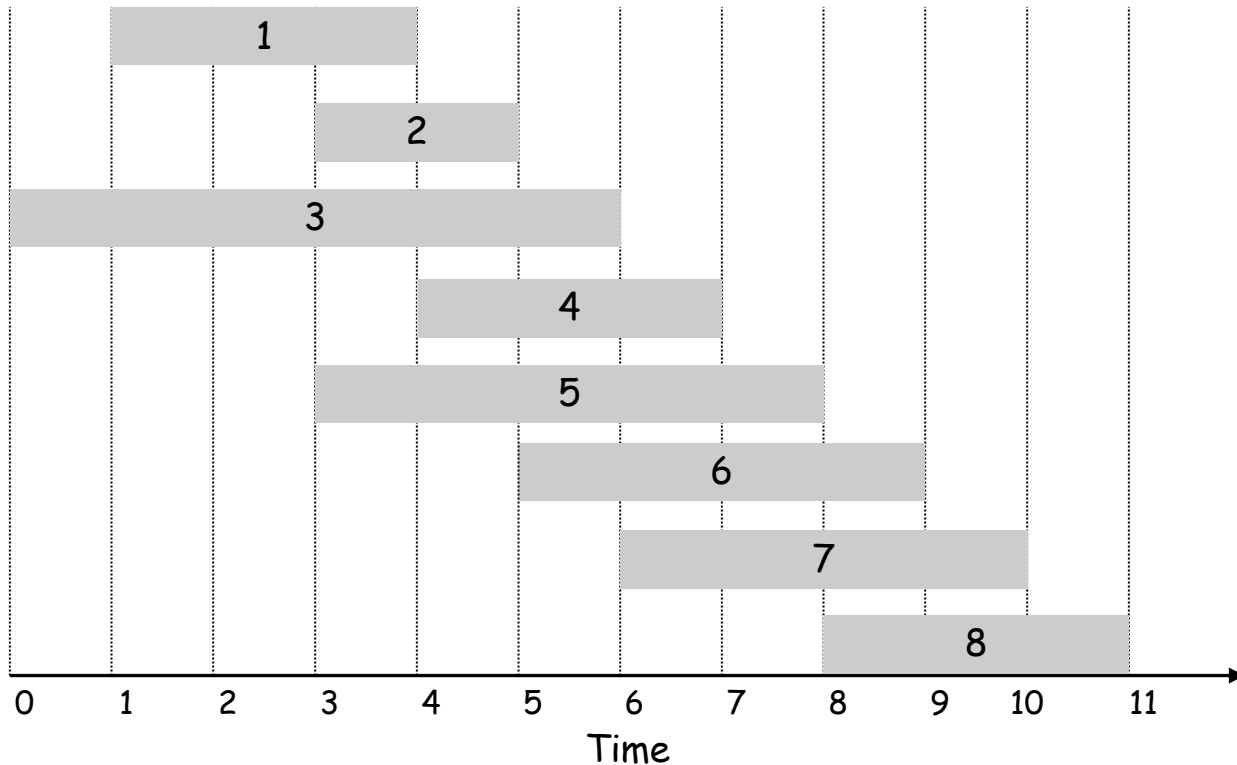


j	$w_j$	$p(j)$	OPT(j)
0			$\emptyset$
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	6
6	3	2	
7	2	3	
8	4	5	

# Example

Label jobs by finishing time:  $f(1) \leq \dots \leq f(n)$ .

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



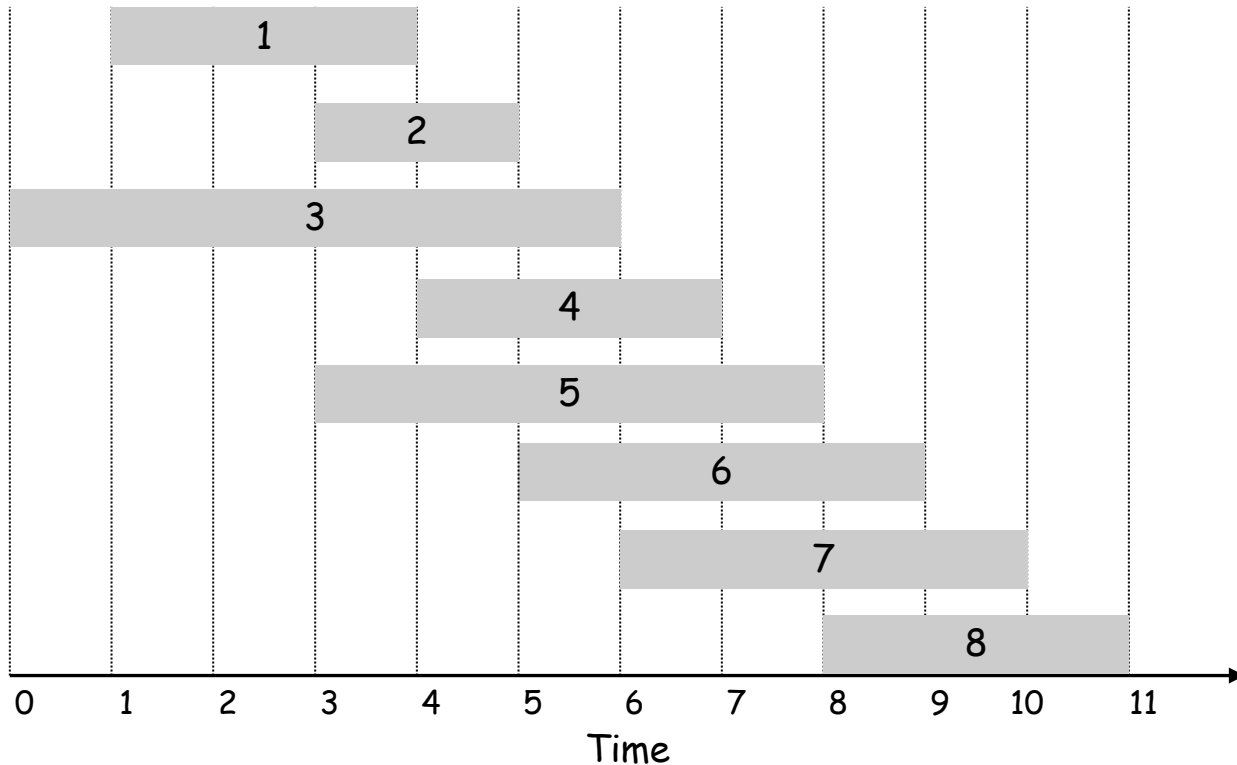
$j$	$w_j$	$p(j)$	$OPT(j)$
0			$\emptyset$
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	6
6	3	2	7
7	2	3	
8	4	5	



# Example

Label jobs by finishing time:  $f(1) \leq \dots \leq f(n)$ .

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

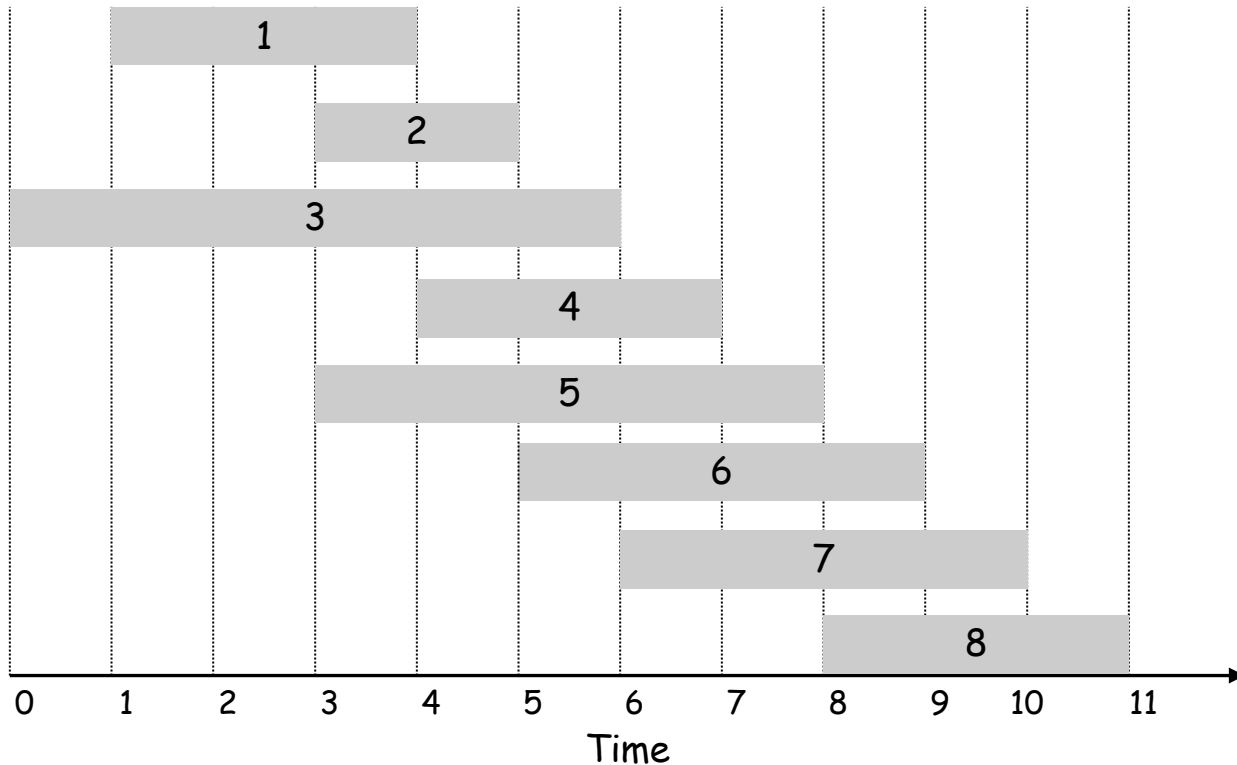


$j$	$w_j$	$p(j)$	OPT( $j$ )
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	6
6	3	2	7
7	2	3	7
8	4	5	

# Example

Label jobs by finishing time:  $f(1) \leq \dots \leq f(n)$ .

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

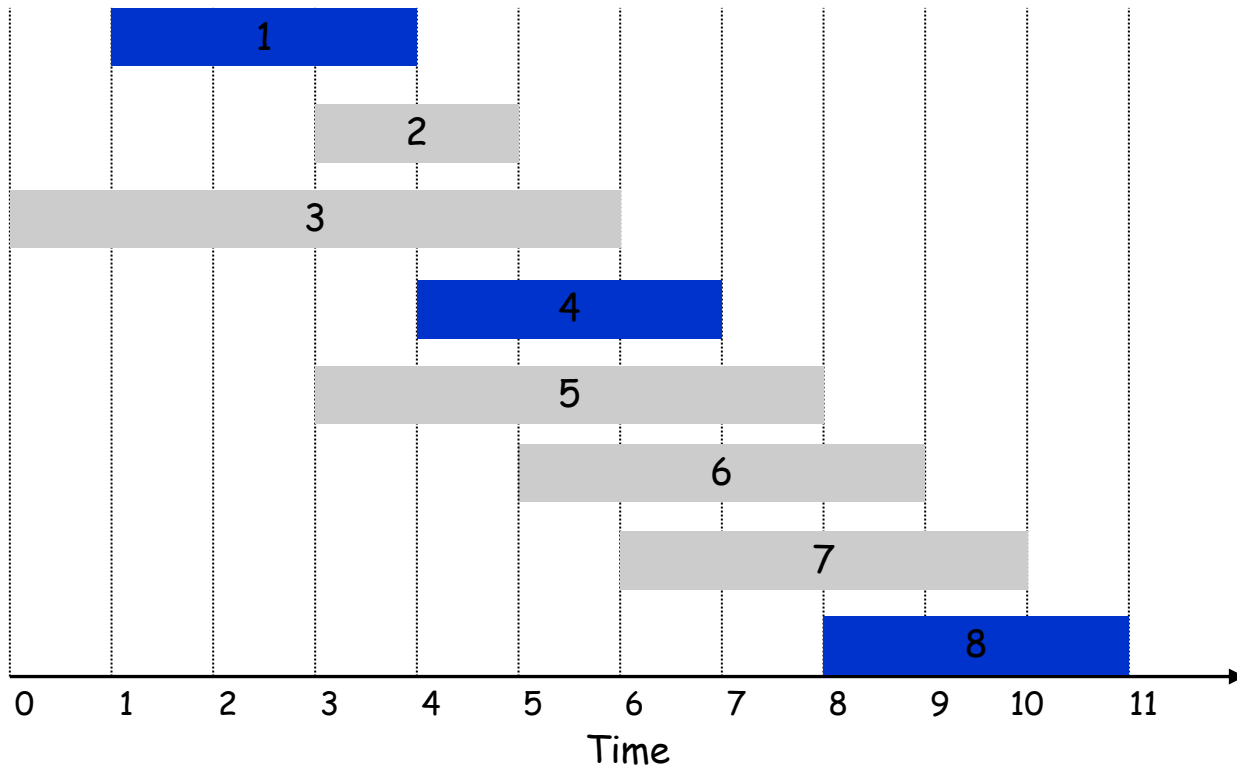


$j$	$w_j$	$p(j)$	$OPT(j)$
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	6
6	3	2	7
7	2	3	7
8	4	5	10

# Example

Label jobs by finishing time:  $f(1) \leq \dots \leq f(n)$ .

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



$j$	$w_j$	$p(j)$	$OPT(j)$
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	6
6	3	2	7
7	2	3	7
8	4	5	<b>10</b>

# Knapsack Problem

# Knapsack Problem

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 value.

**Ex:** OPT is  $\{ 3, 4 \}$  with value 40.

$$W = 11$$

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

**Greedy:** repeatedly add item with maximum ratio  $v_i / w_i$ .

**Ex:**  $\{ 5, 2, 1 \}$  achieves only value = 35  $\Rightarrow$  greedy not optimal.

# Dynamic Programming: First Attempt

Let  $OPT(i)$  = Max value of subsets of items  $1, \dots, i$  of weight  $\leq W$ .

**Case 1:**  $OPT(i)$  does not select item  $i$

- In this case  $OPT(i) = OPT(i - 1)$

**Case 2:**  $OPT(i)$  selects item  $i$

- In this case, item  $i$  does not immediately imply we have to reject other items
- The problem does not reduce to  $OPT(i - 1)$  because we now want to pack as much value into box of weight  $\leq W - w_i$

**Conclusion:** We need more subproblems, we need to strengthen IH.



# Stronger DP (Strengthening Hypothesis)

Let  $OPT(i, w)$  = Max value subset of items  $1, \dots, i$  of weight at most  $w$   
where  $0 \leq w \leq W$

**Case 1:**  $OPT(i, w)$  selects item  $i$

- In this case,  $OPT(i, w) = v_i + OPT(i - 1, w - w_i)$

**Case 2:**  $OPT(i, w)$  does not select item  $i$

- In this case,  $OPT(i, w) = OPT(i - 1, w)$ .

Take best of the two



Therefore,

$$OPT(i, w) = \begin{cases} 0 & \text{If } i = 0 \\ OPT(i - 1, w) & \text{If } w_i > w \\ \max(OPT(i - 1, w), v_i + OPT(i - 1, w - w_i)) & \text{o.w.,} \end{cases}$$

# DP for Knapsack

$v_i, w_i = 10$   
 $w = 0$

```
Compute-OPT(i,w)
  if M[i,w] == empty
    if (i==0)
      M[i,w]=0
    else if (wi > w)
      M[i,w]=Comp-OPT(i-1,w)
    else
      M[i,w]= max {Comp-OPT(i-1,w), vi + Comp-OPT(i-1,w-wi) }
  return M[i, w]
```

Adv: possibly faster  
computes fewer entries  
recursive

```
for w = 0 to W
  M[0, w] = 0
for i = 1 to n
  for w = 1 to W
    if (wi > w)
      M[i, w] = M[i-1, w]
    else
      M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}
return M[n, W]
```

Non-recursive

Advantage: Optimize memory.



# DP for Knapsack

←  $W + 1$  →

		0	1	2	3	4	5	6	7	8	9	10	11
$n + 1$	ϕ	0	0	0	0	0	0	0	0	0	0	0	0
	{ 1 }	0											
	{ 1, 2 }	0											
	{ 1, 2, 3 }	0											
	{ 1, 2, 3, 4 }	0											
	{ 1, 2, 3, 4, 5 }	0											

$W = 11$

```

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

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

# DP for Knapsack

←————— W + 1 —————→

		0	1	2	3	4	5	6	7	8	9	10	11
<div style="display: flex; align-items: center;"> <div style="border-left: 1px solid black; border-right: 1px solid black; height: 100px; margin-right: 5px;"></div> <div style="display: flex; flex-direction: column; align-items: center; justify-content: space-between;"> <span>n + 1</span> <span>↓</span> </div> </div>	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, 2, 3 }	0											
	{ 1, 2, 3, 4 }	0											
	{ 1, 2, 3, 4, 5 }	0											

W = 11

```

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

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

# DP for Knapsack

←  $W + 1$  →

		0	1	2	3	4	5	6	7	8	9	10	11
$n + 1$	$\phi$	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								
	{1, 2, 3}	0	1										
	{1, 2, 3, 4}	0	1										
	{1, 2, 3, 4, 5}	0	1										

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

$W = 11$

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

```

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

# DP for Knapsack

←  $W + 1$  →

		0	1	2	3	4	5	6	7	8	9	10	11
$n + 1$	$\phi$	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					
	{1,2,3,4}	0	1										
	{1,2,3,4,5}	0	1										

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

$W = 11$

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

```

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

# DP for Knapsack

←  $W + 1$  →

		0	1	2	3	4	5	6	7	8	9	10	11
$n + 1$	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		
	{1,2,3,4,5}	0	1										

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

$W = 11$

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

```

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

# DP for Knapsack

←————— W + 1 —————→

		0	1	2	3	4	5	6	7	8	9	10	11
<div style="display: flex; align-items: center;"> <div style="border-left: 1px solid black; border-right: 1px solid black; height: 100px; margin-right: 5px;"></div> <div style="display: flex; flex-direction: column; align-items: center; justify-content: space-between; width: 20px;"> <span>n + 1</span> <span>↓</span> </div> </div>	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

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

W = 11

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

```

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

# Knapsack Problem: Running Time

Running time:  $\Theta(n \cdot W)$

- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete.

Knapsack approximation algorithm:

There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum in time  $\text{Poly}(n, \log W)$ .

# DP Ideas so far

- You may have to define an ordering to decrease #subproblems
- You may have to strengthen DP, equivalently the induction, i.e., you may have to carry more information to find the Optimum.
- This means that sometimes we may have to use two dimensional or three dimensional induction
- To think about DP try to characterize OPT.
- To write down your solution explicitly and carefully write  $OPT(i, \dots)$



# RNA Secondary Structure



# RNA Secondary Structure (Formal)

**Secondary structure.** A set of pairs  $S = \{ (b_i, b_j) \}$  that satisfy:

[Watson-Crick.]

- $S$  is a *matching* and
- each pair in  $S$  is a Watson-Crick pair: A-U, U-A, C-G, or G-C.

[No sharp turns.]: The ends of each pair are separated by at least 4 intervening bases. If  $(b_i, b_j) \in S$ , then  $i < j - 4$ .

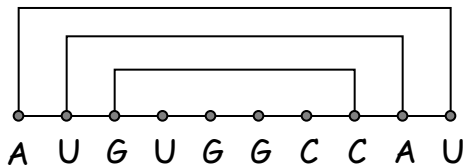
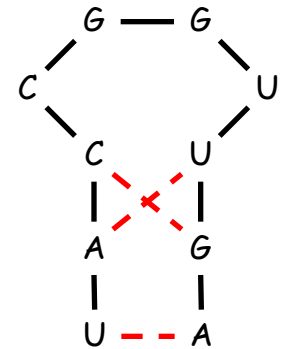
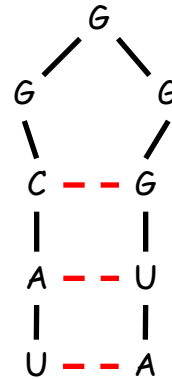
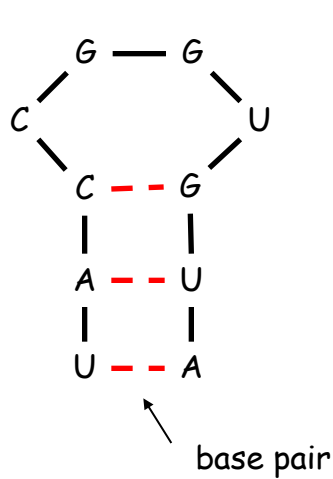
[Non-crossing.] If  $(b_i, b_j)$  and  $(b_k, b_l)$  are two pairs in  $S$ , then we cannot have  $i < k < j < l$ .

**Free energy:** Usual hypothesis is that an RNA molecule will maximize total free energy.

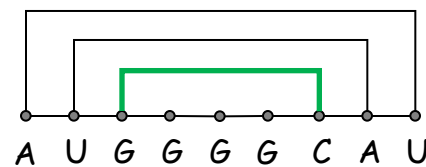
↑  
approximate by number of base pairs

**Goal:** Given an RNA molecule  $B = b_1b_2\dots b_n$ , find a secondary structure  $S$  that maximizes the number of base pairs.

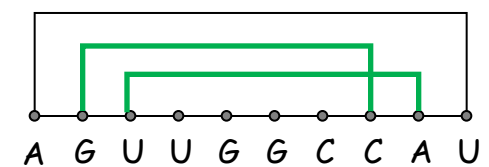
# Secondary Structure (Examples)



ok



~~sharp turn~~



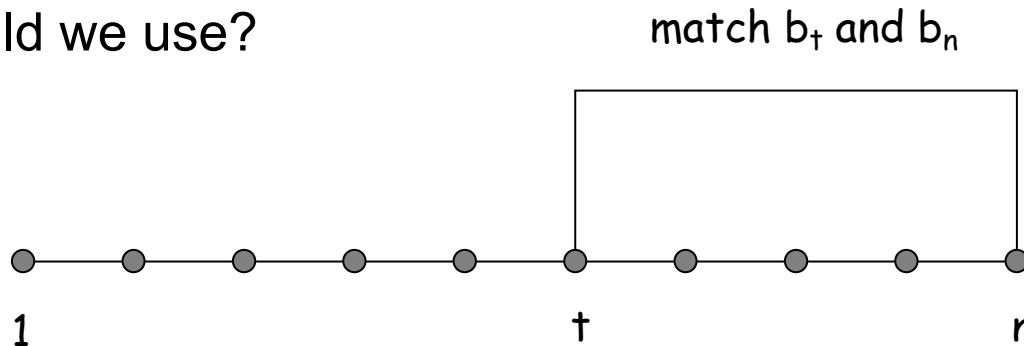
~~crossing~~

# DP: First Attempt

**First attempt.** Let  $OPT(n)$  = maximum number of base pairs in a secondary structure of the substring  $b_1b_2\dots b_n$ .

Suppose  $b_n$  is matched with  $b_t$  in  $OPT(n)$ .

What IH should we use?



**Difficulty:** This naturally reduces to two subproblems

- Finding secondary structure in  $b_1, \dots, b_{t-1}$ , i.e.,  $OPT(t-1)$
- Finding secondary structure in  $b_{t+1}, \dots, b_{n-1}$ , ???

# DP: Second Attempt

**Definition:**  $OPT(i, j)$  = maximum number of base pairs in a secondary structure of the substring  $b_i, b_{i+1}, \dots, b_j$

The most important part of a correct DP; It fixes IH

**Case 1:** If  $j - i \leq 4$ .

- $OPT(i, j) = 0$  by no-sharp turns condition.

**Case 2:** Base  $b_j$  is not involved in a pair.

- $OPT(i, j) = OPT(i, j-1)$

**Case 3:** Base  $b_j$  pairs with  $b_t$  for some  $i \leq t < j - 4$

- non-crossing constraint **decouples** resulting sub-problems
- $OPT(i, j) = \max_{t: b_i \text{ pairs with } b_t} \{ 1 + OPT(i, t - 1) + OPT(t + 1, j - 1) \}$

# Recursive Code

Let  $M[i,j]$ =empty for all  $i,j$ .

```
Compute-OPT(i,j){
  if (j-i <= 4)
    return 0;
  if (M[i,j] is empty)
    M[i,j]=Compute-OPT(i,j-1)
  for t=i to j-5 do
    if ( $b_t, b_j$  is in {A-U, U-A, C-G, G-C})
      M[i,j]=max(M[i,j], 1+Compute-OPT(i,t-1) +
        Compute-OPT(t+1,j-1))
  return M[j]
}
```

Does this code terminate?  
What are we inducting on?

# Formal Induction

Let  $OPT(i, j)$  = maximum number of base pairs in a secondary structure of the substring  $b_i, b_{i+1}, \dots, b_j$

**Base Case:**  $OPT(i, j) = 0$  for all  $i, j$  where  $|j - i| \leq 4$ .

**IH:** For some  $\ell \geq 4$ , Suppose we have computed  $OPT(i, j)$  for all  $i, j$  where  $|i - j| \leq \ell$ .

**IS:** Goal: We find  $OPT(i, j)$  for all  $i, j$  where  $|i - j| = \ell + 1$ . Fix  $i, j$  such that  $|i - j| = \ell + 1$ .

**Case 1:** Base  $b_j$  is not involved in a pair.

- $OPT(i, j) = OPT(i, j-1)$  [this we know by IH since  $|i - (j - 1)| = \ell$ ]

**Case 2:** Base  $b_j$  pairs with  $b_t$  for some  $i \leq t < j - 4$

- $OPT(i, j) = \max_{t: b_i \text{ pairs with } b_t} \{ 1 + OPT(i, t - 1) + OPT(t + 1, j - 1) \}$

We know by IH since difference  $\leq \ell$



# Bottom-up DP

```
for k = 1, 2, ..., n-1
  for i = 1, 2, ..., n-1
    j = i + k
    if (j-i <= 4)
      M[i,j]=0;
    else
      M[i,j]=M[i,j-1]
      for t=i to j-5 do
        if ( $b_t, b_j$  is in {A-U, U-A, C-G, G-C})
          M[i,j]=max(M[i,j], 1+ M[i,t-1] + M[t+1,j-1])

return M[1, n]
}
```

4	0	0	0	↗
3	0	0	↗	↗
2	0	↗	↗	↗
1	↗	↗	↗	↗
	6	7	8	9

j

Running Time:  $O(n^3)$

# Lesson

We may not always induct on  $i$  or  $w$  to get to smaller subproblems.

We may have to induct on  $|i - j|$  or  $i + j$  when we are dealing with more complex problems, e.g., intervals