

# CSE 421 Winter 2025

## Lecture 12: Dynamic Programming



Nathan Brunelle

<http://www.cs.uw.edu/421>

# Algorithmic Paradigms

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



**Divide-and-conquer:** Break up a problem into sub-problems (each typically a constant factor smaller), 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.

# Algorithm Design Techniques

## Dynamic Programming:

- Technique for making building solution to a problem based on solutions to smaller subproblems (recursive ideas).
- The subproblems just have to be smaller, but don't need to be a constant-factor smaller like divide and conquer.
- *Useful when the same subproblems show up over and over again*
- The final solution is simple iterative code when the following holds:
  - *The parameters to all the subproblems are predictable in advance*

# Dynamic Programming History

Bellman. [1950s] Pioneered the systematic study of dynamic programming.

## Etymology

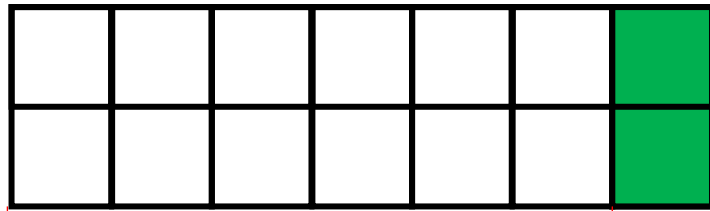
- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.

"it's impossible to use dynamic in a pejorative sense"  
"something not even a Congressman could object to"

Reference: Bellman, R. E. *Eye of the Hurricane, An Autobiography*.

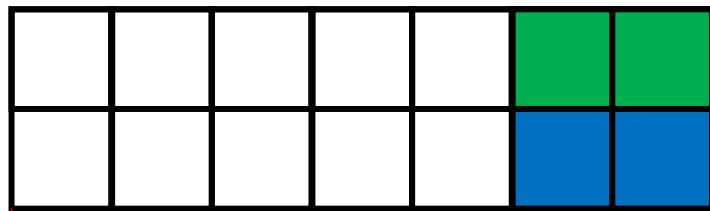
How many ways are there to tile a  $2 \times n$  board with dominoes?

Two ways to fill the final column:



$$Tile(n) = Tile(n - 1) + Tile(n - 2)$$

$n - 1$



$$Tile(0) = Tile(1) = 1$$

$n - 2$

How to compute  $Tile(n)$ ?

Tile(n):

if  $n < 2$ :

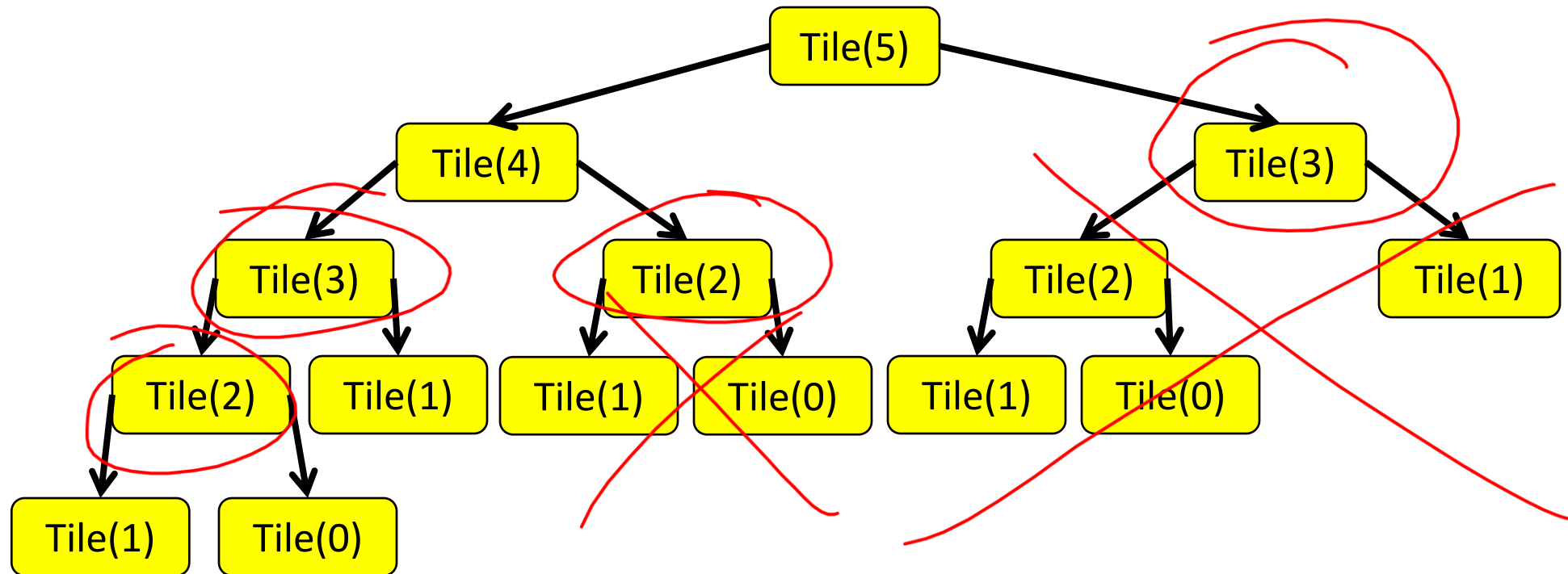
return 1

return  $Tile(n-1)+Tile(n-2)$

Running Time?



# Recursion Tree

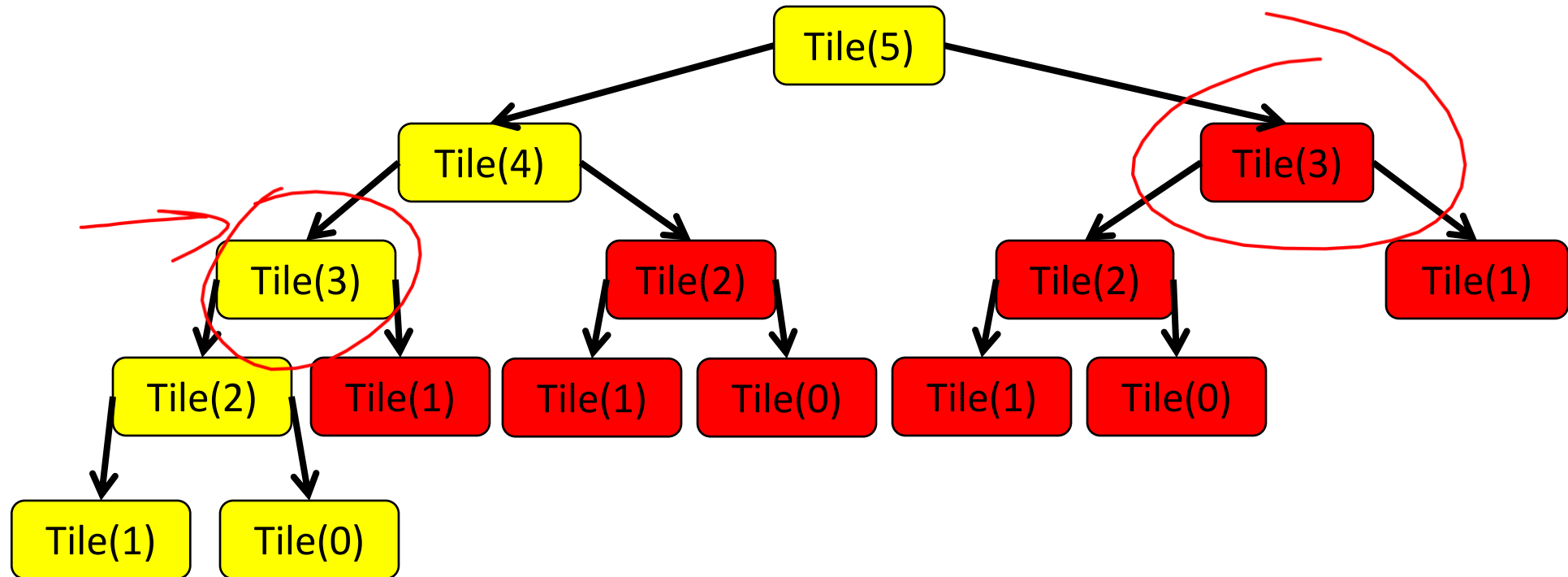


Many redundant calls!

Run time:  $\Omega(2^n)$

Better way: Use Memory!

# Recursion Tree



Many redundant calls!

Run time:  $\Omega(2^n)$

Better way: Use Memory!



# Computing $Tile(n)$ with Memory

Initialize Memory M

$Tile(n)$ :

if  $n < 2$ :

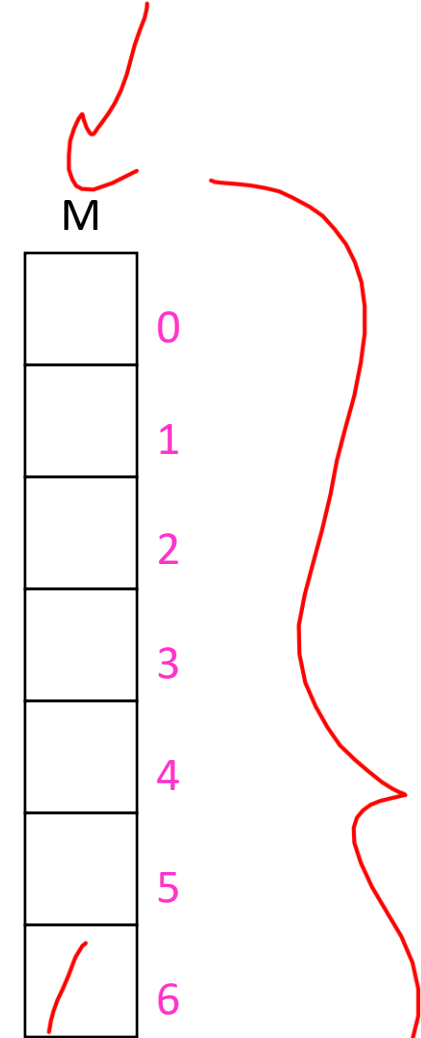
return 1

if  $M[n]$  is filled:

return  $M[n]$

$M[n] = Tile(n-1) + Tile(n-2)$

return  $M[n]$



Technique: “memoization” (note no “r”)

# Computing $Tile(n)$ with Memory - “Top Down”

Initialize Memory M

Tile(n):

if  $n < 2$ :

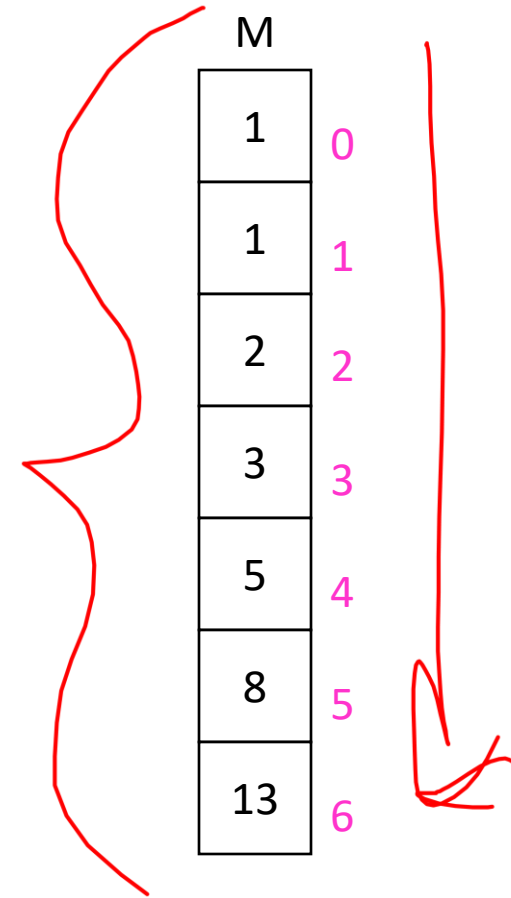
return 1

if M[n] is filled:

return M[n]

M[n] = Tile(n-1)+Tile(n-2)

return M[n]



# Computing $Tile(n)$ with Memory - “Top Down”

Initialize Memory M

Tile(n):

if  $n < 2$ :

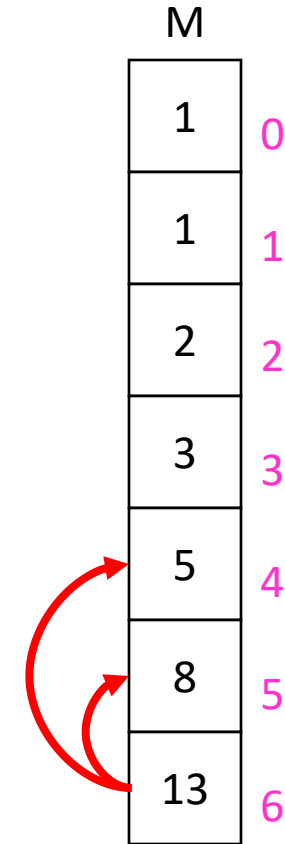
return 1

if M[n] is filled:

return M[n]

M[n] = Tile(n-1)+Tile(n-2)

return M[n]



Recursive calls happen in a predictable order

# *Tile*(*n*) with Memory - “Bottom Up”

Tile(*n*):

Initialize Memory M

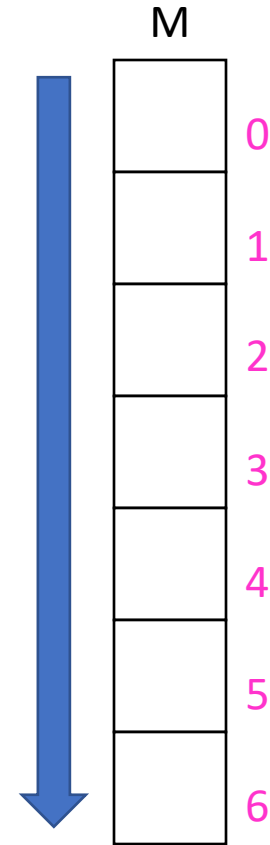
$$M[0] = 1$$

$$M[1] = 1$$

→ for *i* = 2 to *n*: \*

$$\rightarrow M[i] = M[i-1] + M[i-2]$$

return M[*n*]



# Better $Tile(n)$ with Memory - “Bottom Up”

Tile(n):

$M[0] = 1$

$M[1] = 1$

answer = -1

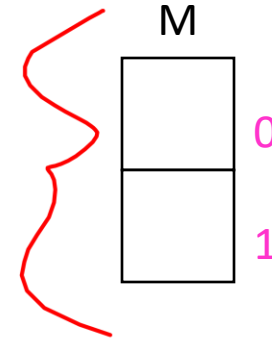
for  $i = 2$  to  $n$ :

answer =  $M[0] + M[1]$

$M[0] = M[1]$

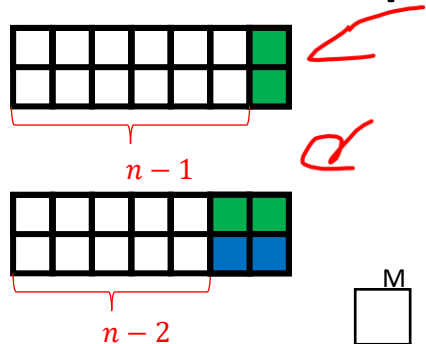
$M[1] = \text{answer}$

return  $M[1]$

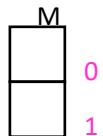
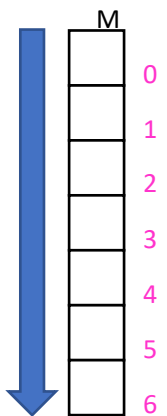
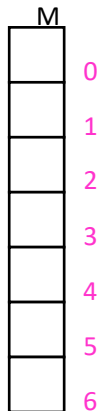


Observation: We only need to remember the last two subproblems!

# Four Steps to Dynamic Programming



Conclusion: a 1-dimensional memory of size  $n$



1. Formulate the answer with a recursive structure
  - What are the options for the last choice?
  - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
  - Figure out the possible values of all parameters in the recursive calls.
  - How many subproblems (options for last choice) are there?
  - What are the parameters needed to identify each?
  - How many different values could there be per parameter?
3. Specify an order of evaluation.
  - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
  - With this step: a “Bottom-up” (iterative) algorithm
  - Without this step: a “Top-down” (recursive) algorithm
4. See if there's a way to save space
  - Is it possible to reuse some memory locations?

# Top-Down DP Idea

```
def myDPalgo(problem):  
    if mem[problem] not blank: // Check if we've solved this already  
        return mem[problem]  
    if baseCase(problem): // Check if this is a base case  
        solution = solve(problem)  
        mem[problem] = solution // Always save your solution before returning  
        return solution  
    for subproblem of problem:  
        subsolutions.append(myDPalgo(subproblem)) // solve each subproblem  
    solution = selectAndExtend(subsolutions) // Pick the subproblem to use  
    mem[problem] = solution // Always save your solution before returning  
    return solution
```

# Bottom-Up DP Idea

```
def myDPalgo(problem):
```

```
    for each baseCase: // Identify which subproblems are base cases
```

```
        solution = solve(baseCase)
```

```
        mem[baseCase] = solution // Save the solution for reuse
```

```
    for each subproblem in bottom-up order:
```

```
        // The order should be chosen so that every subsolution is
```

```
        // guaranteed to already be in memory when it's needed
```

```
        solution = selectAndExtend(subsolutions)
```

```
        mem[subproblem] = solution // Save the solution for reuse
```

```
    return mem[problem]
```



# Weighted Interval Scheduling

**Input:** Like interval scheduling each request  $i$  has start and finish times  $s_i$  and  $f_i$ . Each request  $i$  also has an associated **value** or **weight**  $v_i$

$v_i$  might be

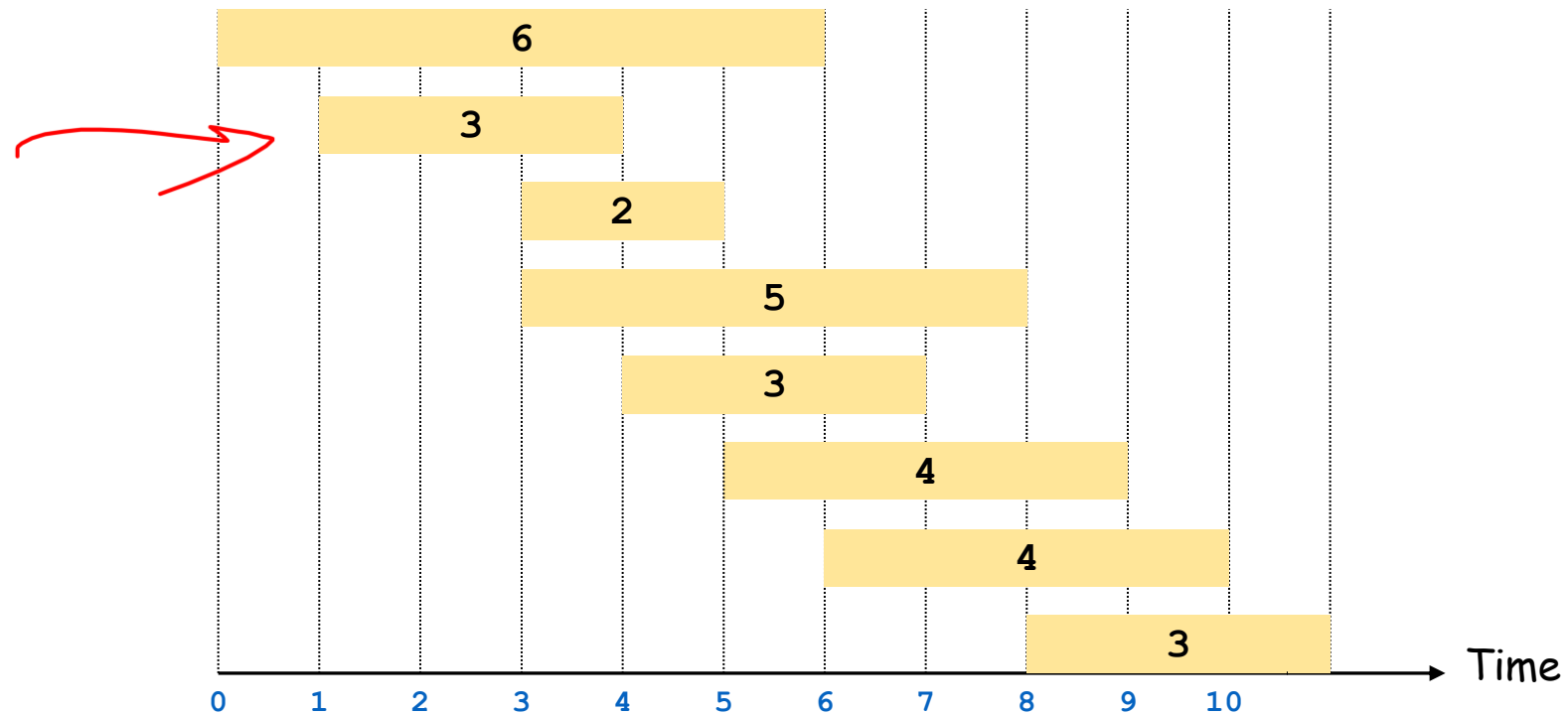
- the amount of money we get from renting out the resource
- the amount of time the resource is being used ( $v_i = f_i - s_i$ )

**Find:** A maximum-weight compatible subset of requests.

# Weighted Interval Scheduling

**Input:** Set of jobs with start times, finish times, and **weights**

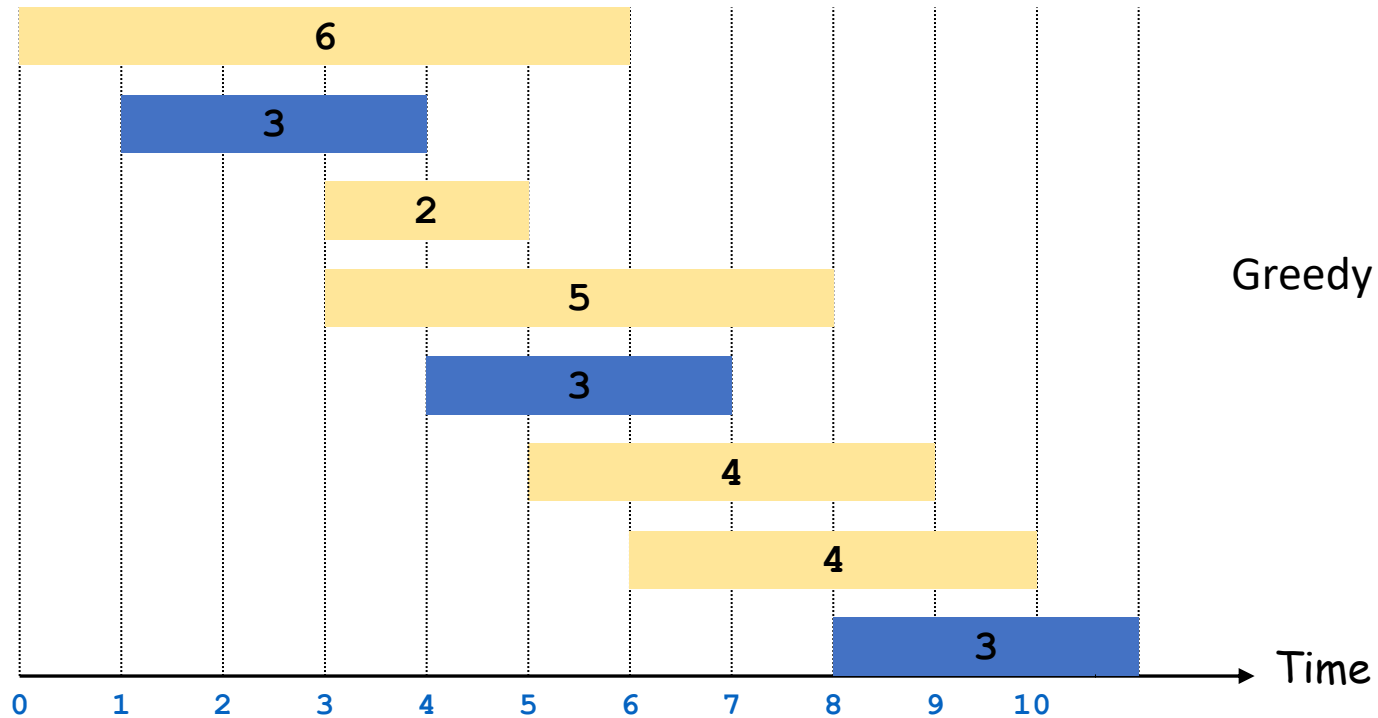
**Goal:** Find **maximum weight** subset of mutually compatible jobs.



# Weighted Interval Scheduling

**Input:** Set of jobs with start times, finish times, and **weights**

**Goal:** Find **maximum weight** subset of mutually compatible jobs.



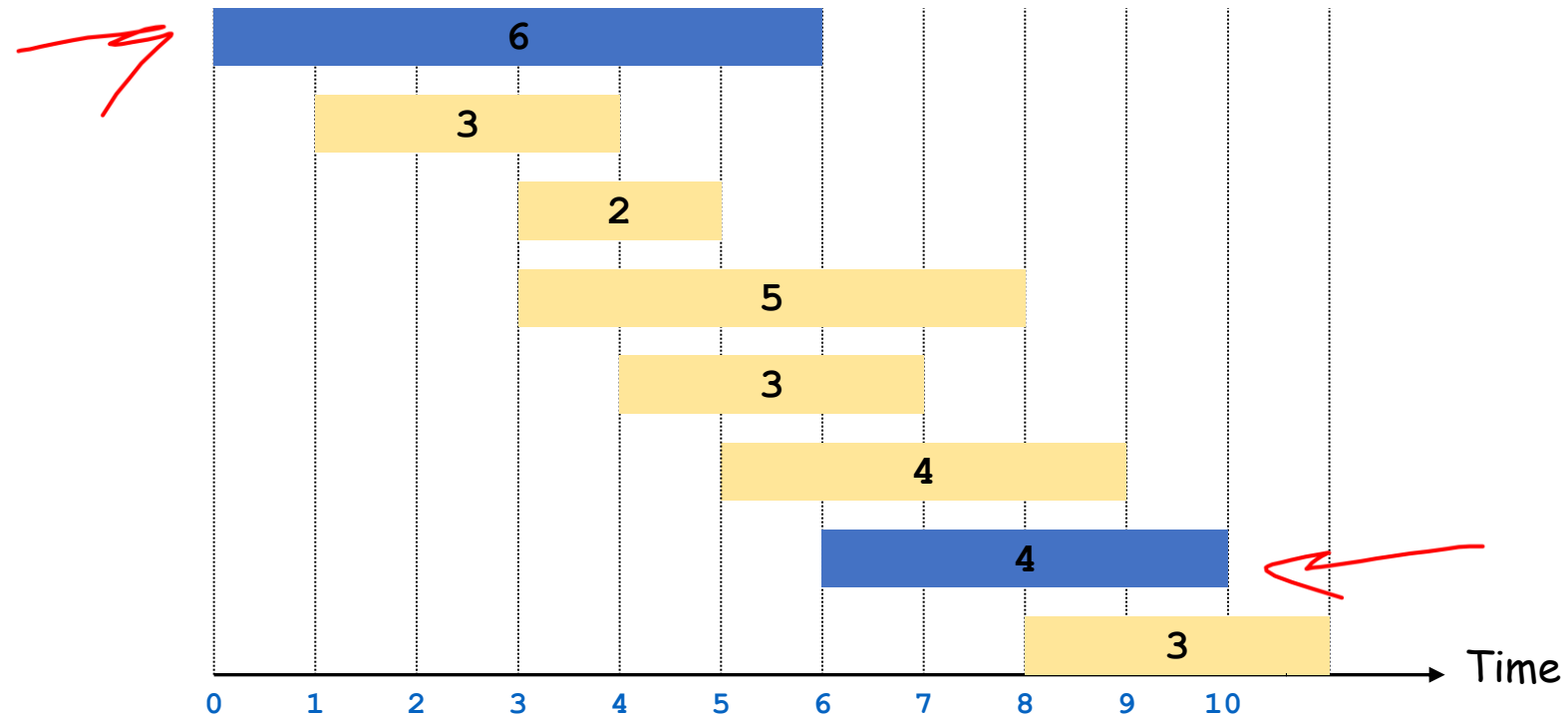
Greedy by finish times would give 9

U

# Weighted Interval Scheduling

**Input:** Set of jobs with start times, finish times, and **weights**

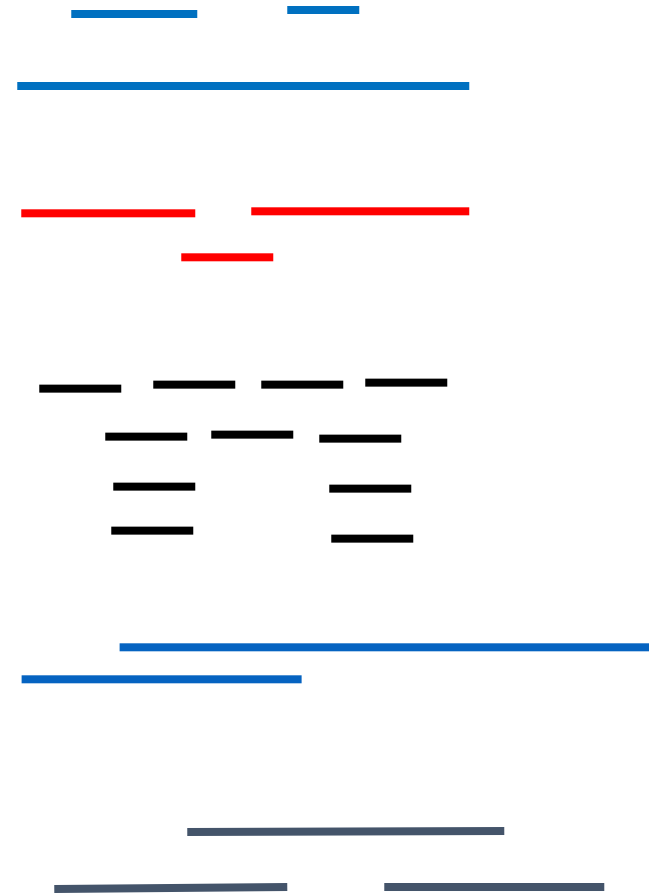
**Goal:** Find **maximum weight** subset of mutually compatible jobs.



Optimal yields 10

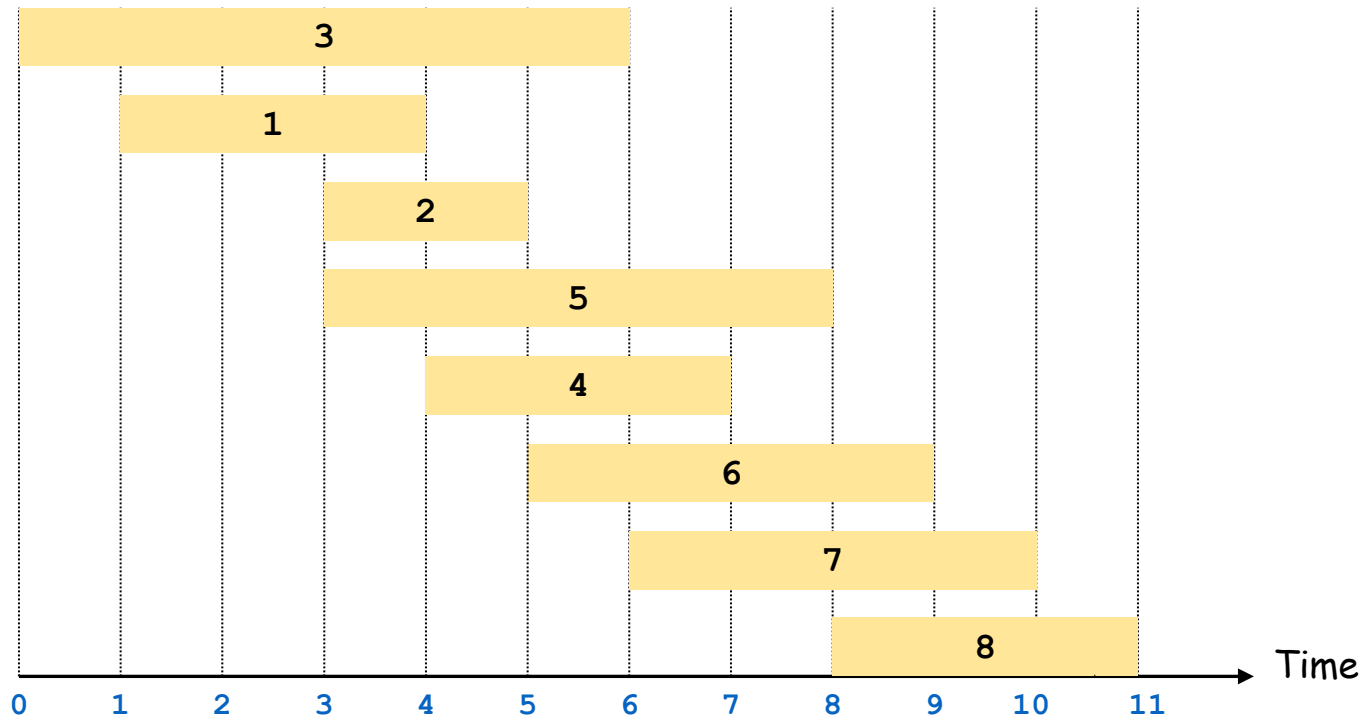
# Greedy Algorithms for Weighted Interval Scheduling?

- What criterion should we try?
  - Earliest start time  $s_i$ 
    - Doesn't work
  - Shortest request time  $f_i - s_i$ 
    - Doesn't work
  - Fewest conflicts
    - Doesn't work
  - Earliest finish time  $f_i$ 
    - Doesn't work
  - Largest value/weight  $v_i$ 
    - Doesn't work



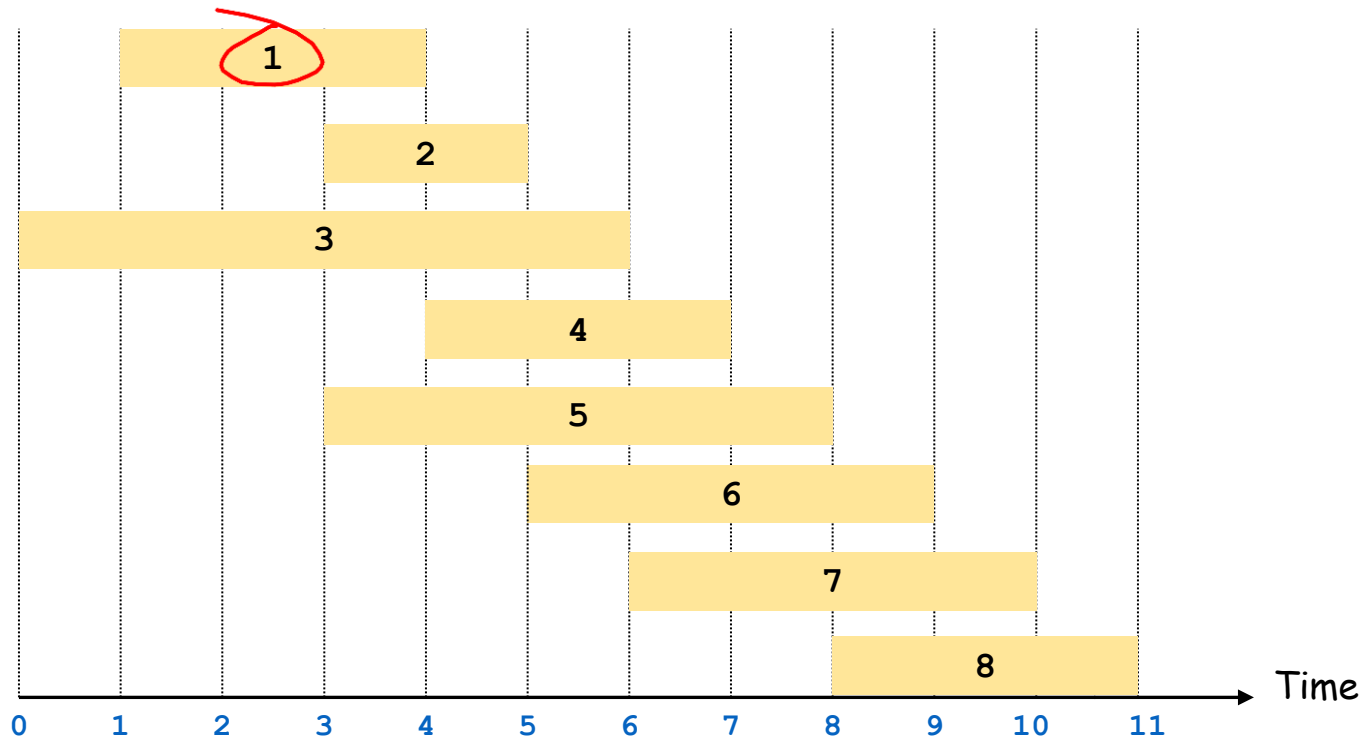
# Weighted Interval Scheduling

Notation: Label jobs by finishing time:  $f_1 \leq f_2 \leq \dots \leq f_n$ .



# Weighted Interval Scheduling

Notation: Label jobs by finishing time:  $f_1 \leq f_2 \leq \dots \leq f_n$ .



# Weighted Interval Scheduling – Four Steps

1. Formulate the answer with a recursive structure
  - What are the options for the last choice?
  - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
  - Figure out the possible values of all parameters in the recursive calls.
  - How many subproblems (options for last choice) are there?
  - What are the parameters needed to identify each?
  - How many different values could there be per parameter?
3. Specify an order of evaluation.
  - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
  - With this step: a “Bottom-up” (iterative) algorithm
  - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
  - Is it possible to reuse some memory locations?



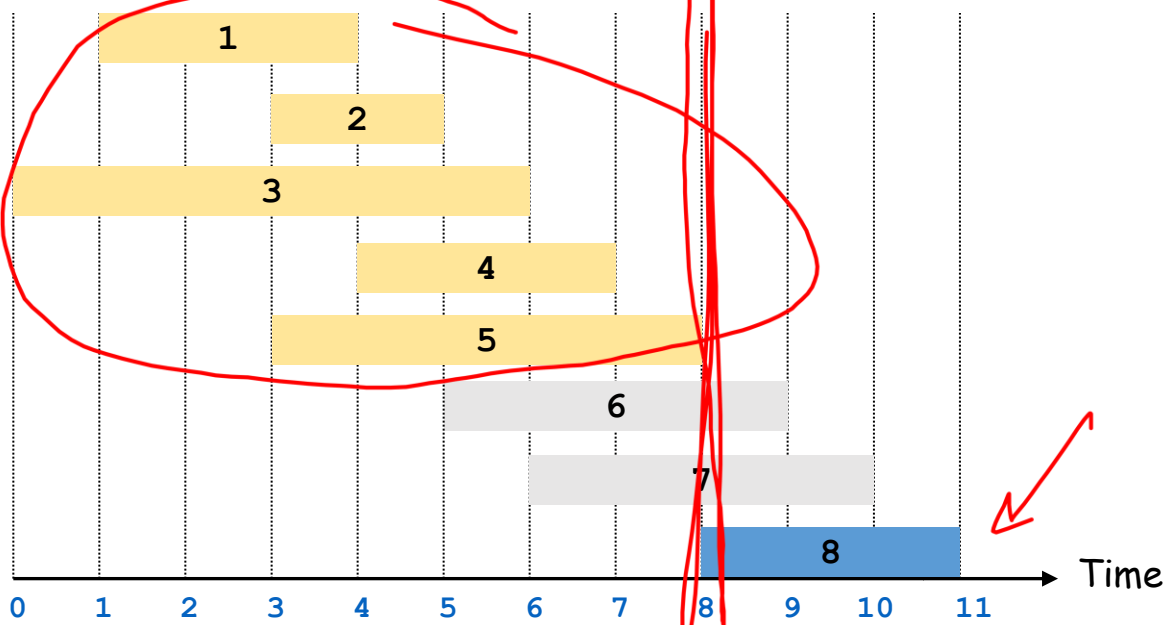
# Towards Dynamic Programming: Step 1 – Recursive Algorithm

Suppose that we first sort the requests by finish time  $f_i$  so  $f_1 \leq f_2 \leq \dots \leq f_n$ .

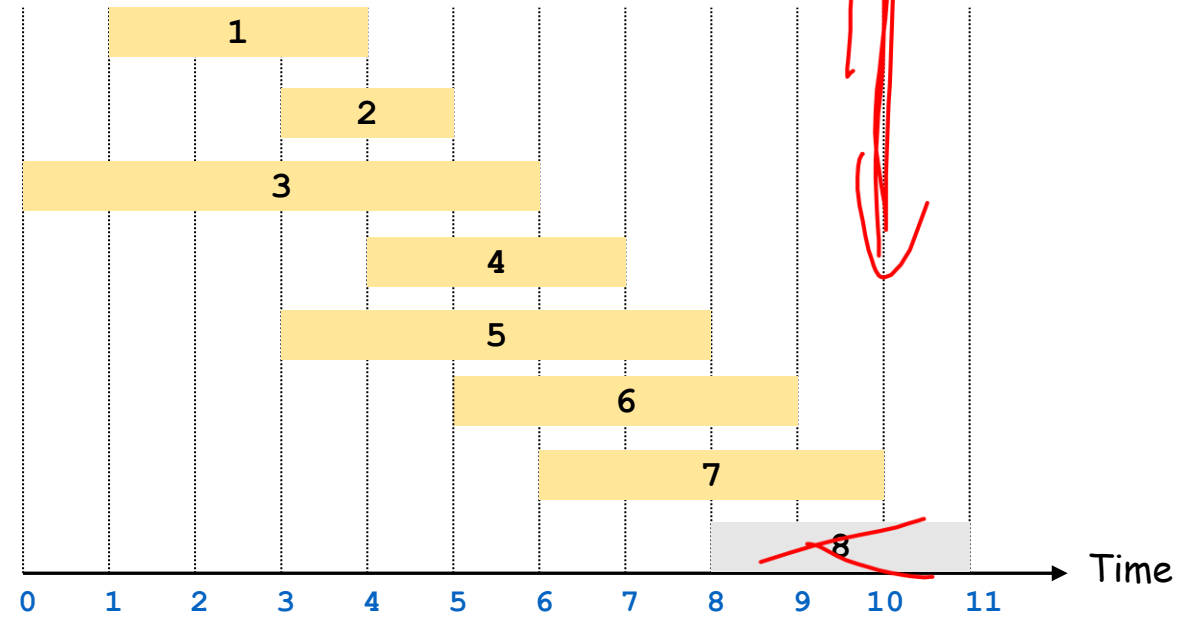
We now want

- a recursive solution that makes calls to smaller problems and
- the indices for those smaller problems to be convenient, so we first focus on the options for the *last* request, request  $n$ .

Option 1: Include the last request

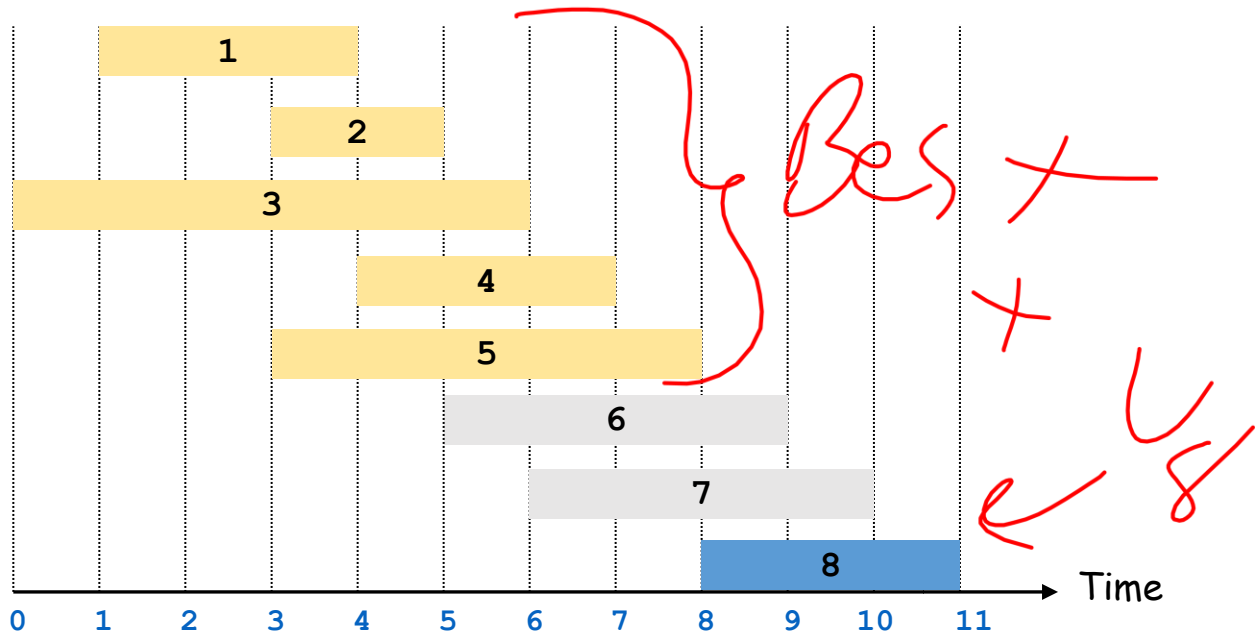


Option 2: Exclude the last request

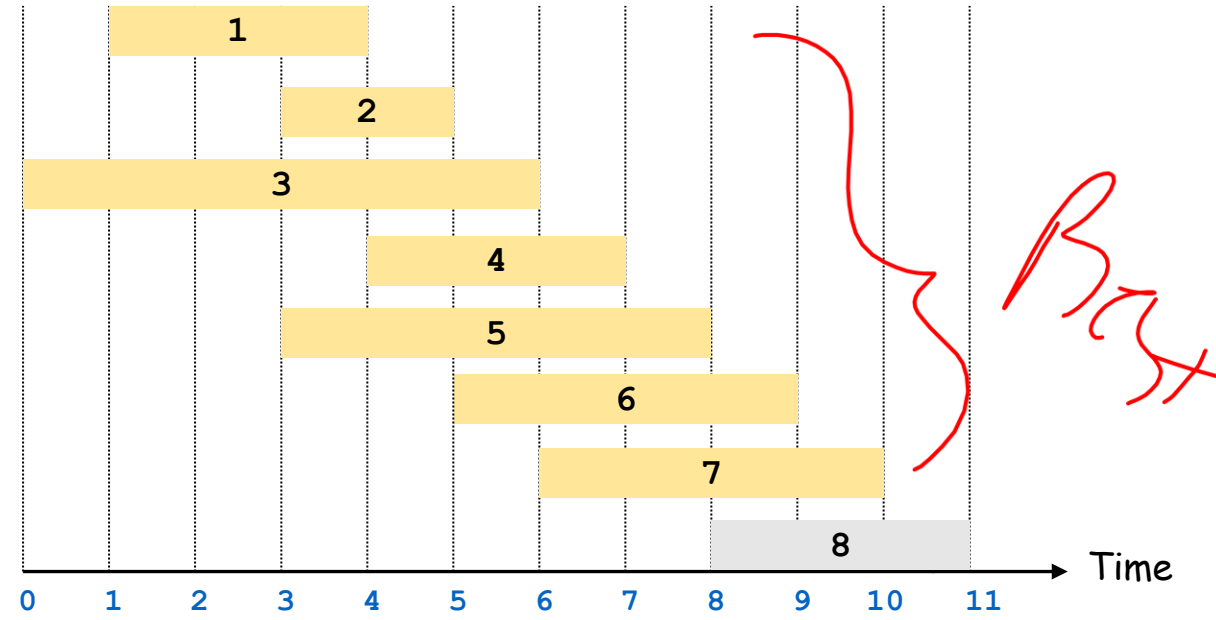


# Towards Dynamic Programming: Step 1 – Recursive Algorithm

Option 1: Include the last request



Option 2: Exclude the last request



After making this choice, the best solution possible is given by:

- The value of the solution to subproblem consisting of everything compatible
- Plus the value of the last request

After making this choice, the best solution possible is given by:

- The value of the solution to subproblem consisting of everything except the last request

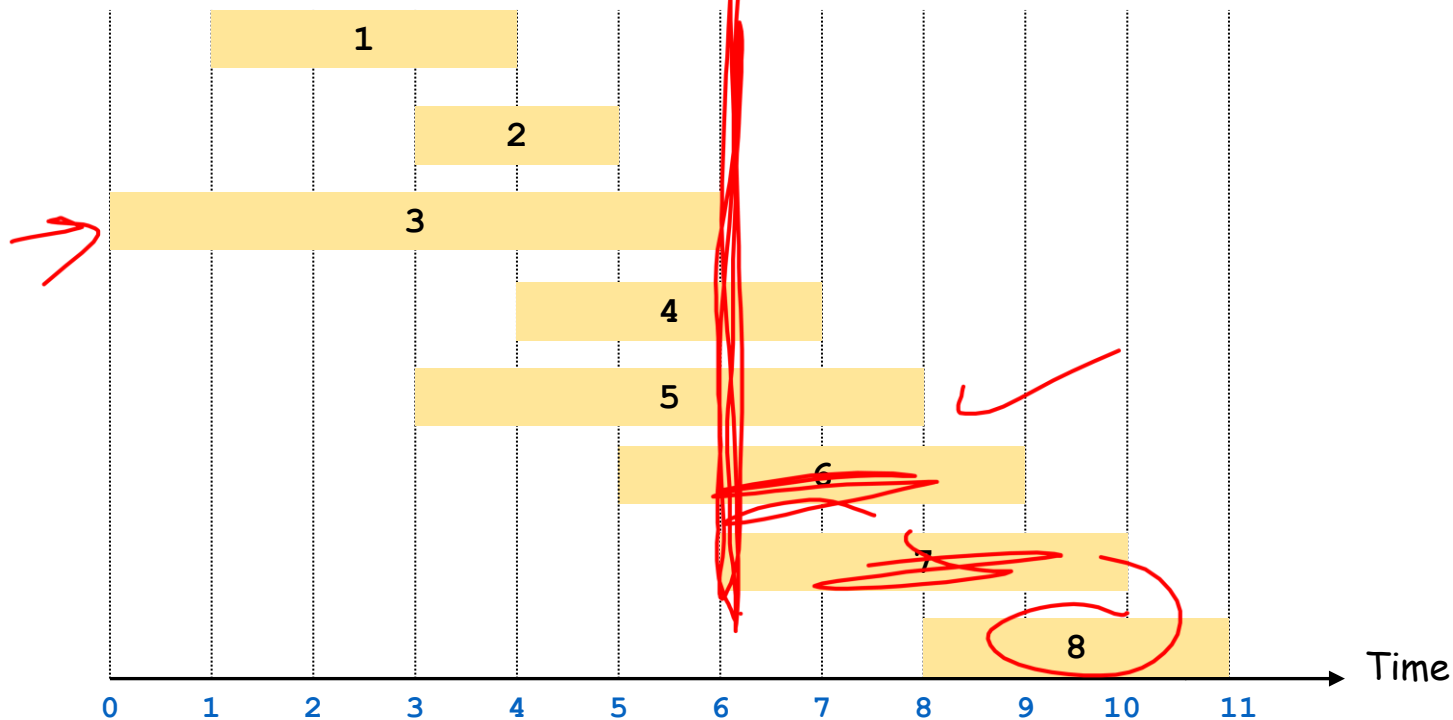
It will be convenient to be able to prune incompatible requests quickly...

# Weighted Interval Scheduling

**Notation:** Label jobs by finishing time:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Defn:**  $p(j)$  = largest index  $i < j$  s.t. job  $i$  is compatible with  $j$ .

**Example:**  $p(8) = 5$ ,  $p(7) = 3$ ,  $p(2) = 0$

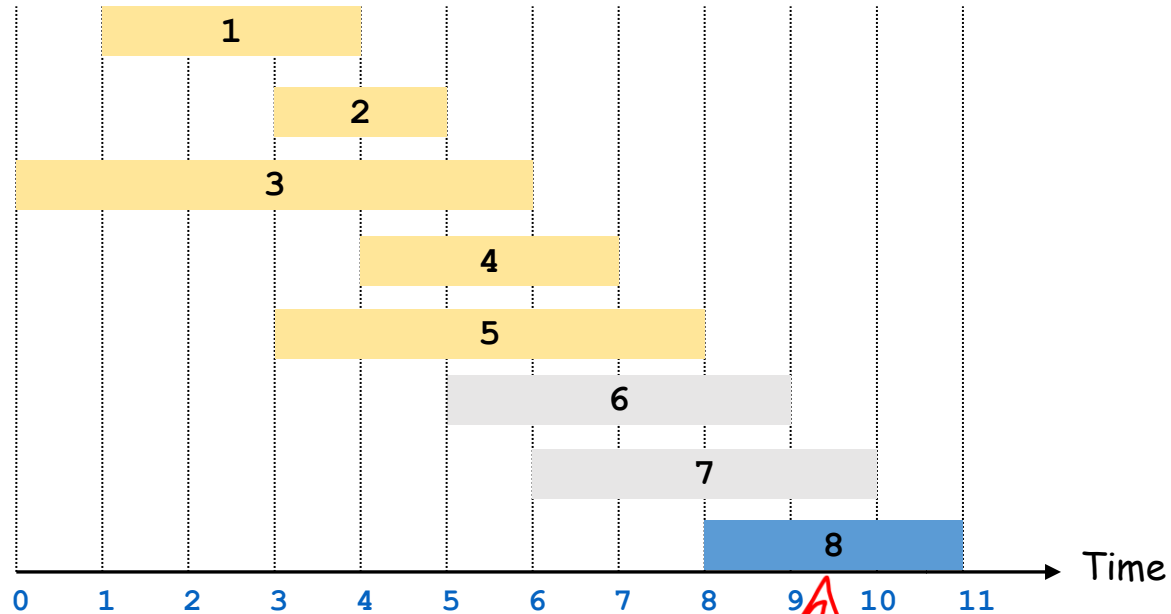


↓

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

# Towards Dynamic Programming: Step 1 – Recursive Algorithm

Option 1: Include the last request

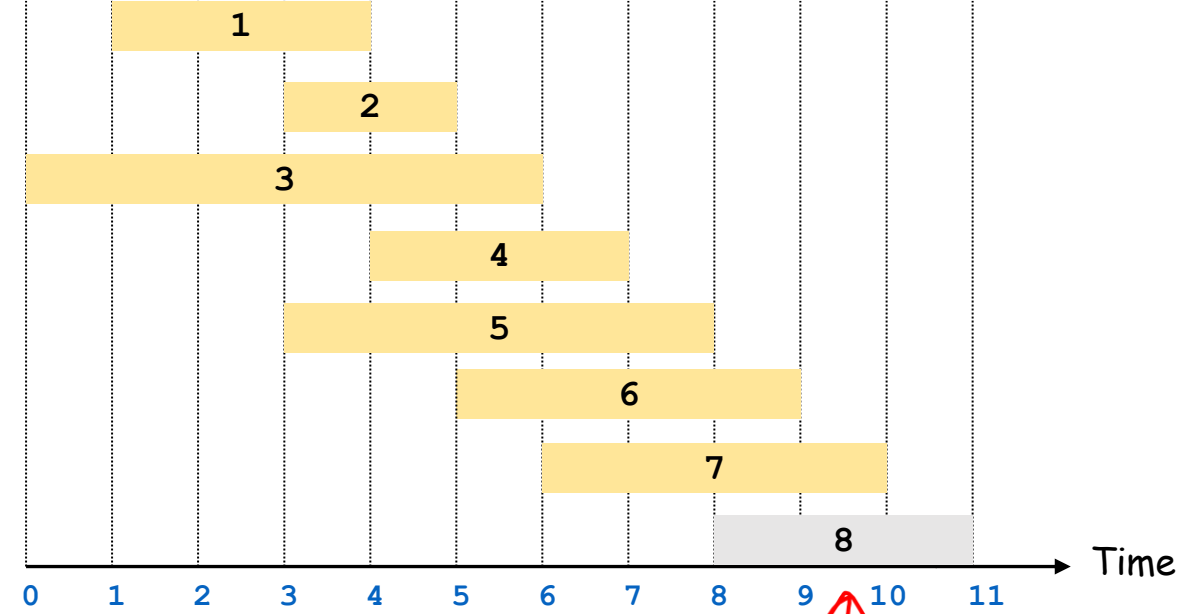


After making this choice, the best solution possible is given by:

- The value of the solution to subproblem consisting of everything compatible
- Plus the value of the last request

$$OPT(p(j)) + v_j$$

Option 2: Exclude the last request



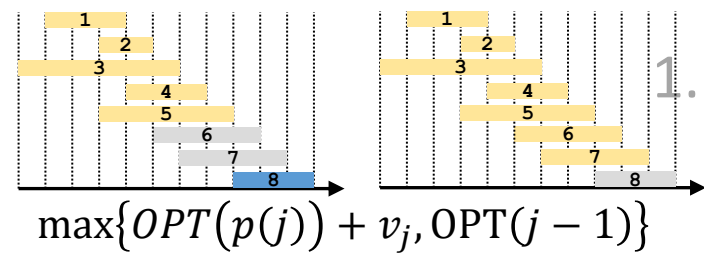
After making this choice, the best solution possible is given by:

- The value of the solution to subproblem consisting of everything except the last request

$$OPT(j - 1)$$

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j - 1)\}$$

# Weighted Interval Scheduling – Four Steps



1. Formulate the answer with a recursive structure
  - What are the options for the last choice?
  - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
  - Figure out the possible values of all parameters in the recursive calls.
  - How many subproblems (options for last choice) are there?
  - What are the parameters needed to identify each?
  - How many different values could there be per parameter?
3. Specify an order of evaluation.
  - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
  - With this step: a “Bottom-up” (iterative) algorithm
  - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
  - Is it possible to reuse some memory locations?

# Towards Dynamic Programming: Step 2 – Memory Structure

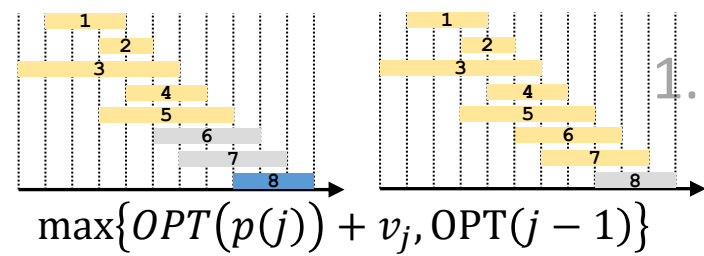
$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j - 1)\}$$

Subproblems are identified by a single parameter  
1-dimensional array

That parameter is the last-ending compatible request  
length is the number of requests

<i>j</i>	OPT[ <i>j</i> ]
0	0
1	
2	
3	
4	
5	
6	
7	
8	

# Weighted Interval Scheduling – Four Steps



$j$	$OPT[j]$
0	0
1	
2	
3	
4	
5	
6	
7	
8	

1. Formulate the answer with a recursive structure
  - What are the options for the last choice?
  - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
  - Figure out the possible values of all parameters in the recursive calls.
  - How many subproblems (options for last choice) are there?
  - What are the parameters needed to identify each?
  - How many different values could there be per parameter?
3. Specify an order of evaluation.
  - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
  - With this step: a “Bottom-up” (iterative) algorithm
  - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
  - Is it possible to reuse some memory locations?

# Top-Down DP Idea

```
def myDPalgo(problem):  
    if mem[problem] not blank: // Check if we've solved this already  
        return mem[problem]  
    if baseCase(problem): // Check if this is a base case  
        solution = solve(problem)  
        mem[problem] = solution // Always save your solution before returning  
        return solution  
    for subproblem of problem:  
        subsolutions.append(myDPalgo(subproblem)) // solve each subproblem  
    solution = selectAndExtend(subsolutions) // Pick the subproblem to use  
    mem[problem] = solution // Always save your solution before returning  
    return solution
```



# Weighted Interval Scheduling Top-Down DP

**WIS(j):**

if OPT[j] not blank: // Check if we've solved this already

return OPT[j]

if j==0: // Check if this is a base case

mem[j] = 0 // Always save your solution before returning

return mem[j]

includej = WIS(p(j)) // Solve each subproblem

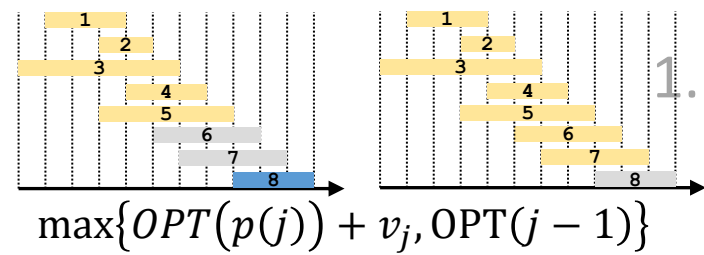
excludej = WIS(j-1) // Solve each subproblem

solution = max(includej+value[j], excludej) // Pick the subproblem to use

mem[j] = solution // Always save your solution before returning

return solution

# Weighted Interval Scheduling – Four Steps



$j$	$OPT[j]$
0	0
1	
2	
3	
4	
5	
6	
7	
8	

1. Formulate the answer with a recursive structure
  - What are the options for the last choice?
  - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
  - Figure out the possible values of all parameters in the recursive calls.
  - How many subproblems (options for last choice) are there?
  - What are the parameters needed to identify each?
  - How many different values could there be per parameter?
3. Specify an order of evaluation.
  - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
  - With this step: a “Bottom-up” (iterative) algorithm
  - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
  - Is it possible to reuse some memory locations?

# Towards Dynamic Programming: Step 3 – Order of Evaluation

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j - 1)\}$$

For any given cell  $j$ , which other cells might I need?

- $j - 1$
- $p(j)$

It's hard to know in advance what  $p(j)$  might be, but certainly  $p(j) < j$

Order: increasing order of  $j$  will work

$j$	$OPT[j]$
0	0
1	
2	
3	
4	
5	
6	
7	
8	



# Bottom-Up DP Idea

```
def myDPalgo(problem):  
    for each baseCase: // Identify which subproblems are base cases  
        solution = solve(baseCase)  
        mem[baseCase] = solution // Save the solution for reuse  
    for each subproblem in bottom-up order:  
        // The order should be chosen so that every subsolution is  
        // guaranteed to already be in memory when it's needed  
        solution = selectAndExtend(subsolutions)  
        mem[subproblem] = solution // Save the solution for reuse  
    return mem[problem]
```

# Weighted Interval Scheduling Bottom-Up DP

**WIS(*j*):**

OPT[0] = 0 // Save the solution for the base case

for each *i* = 1 up to *j*:

// The order should be chosen so that every subsolution is

// guaranteed to already be in memory when it's needed

solution = max(OPT[*p*(*i*)]+value[*i*], OPT[*i* - 1])

mem[*i*] = solution // Save the solution for reuse

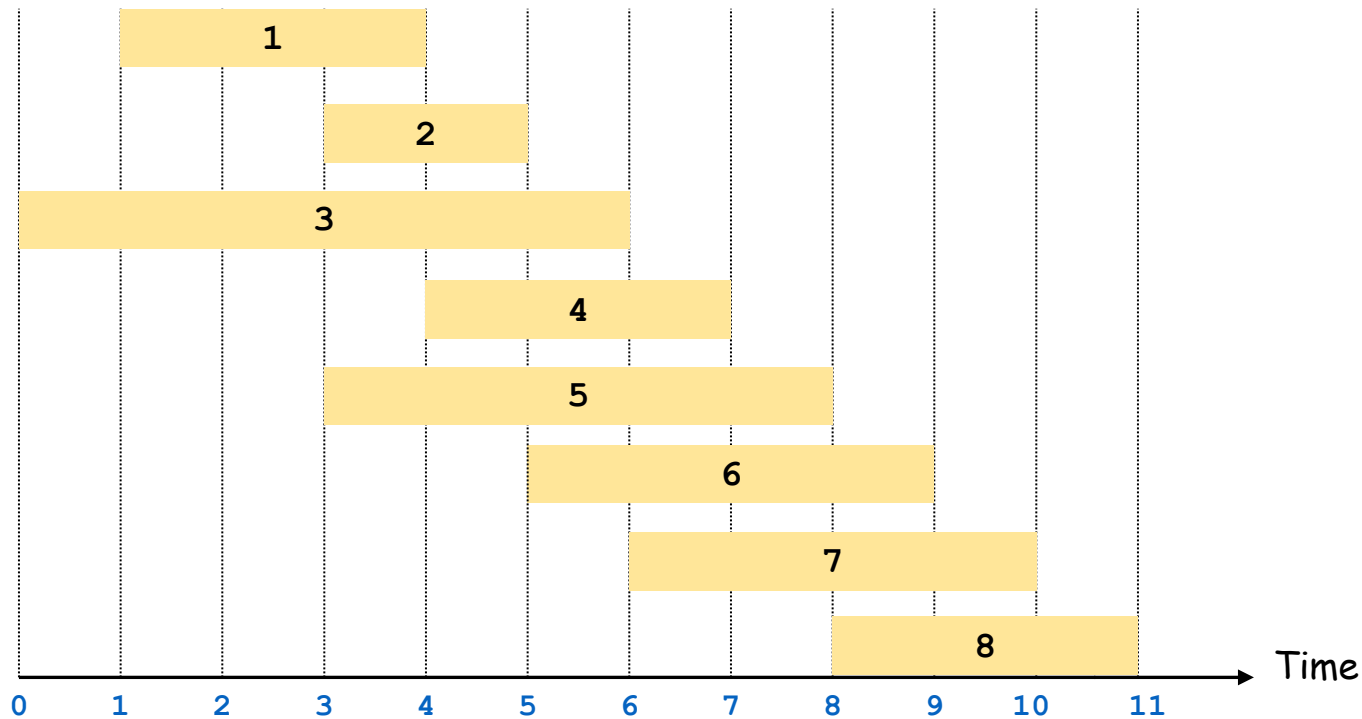
return OPT[*j*]

# Example Execution (iterative)

**Notation:** Label jobs by finishing time:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Defn:**  $p(j)$  = largest index  $i < j$  s.t. job  $i$  is compatible with  $j$ .

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j - 1)\}$$



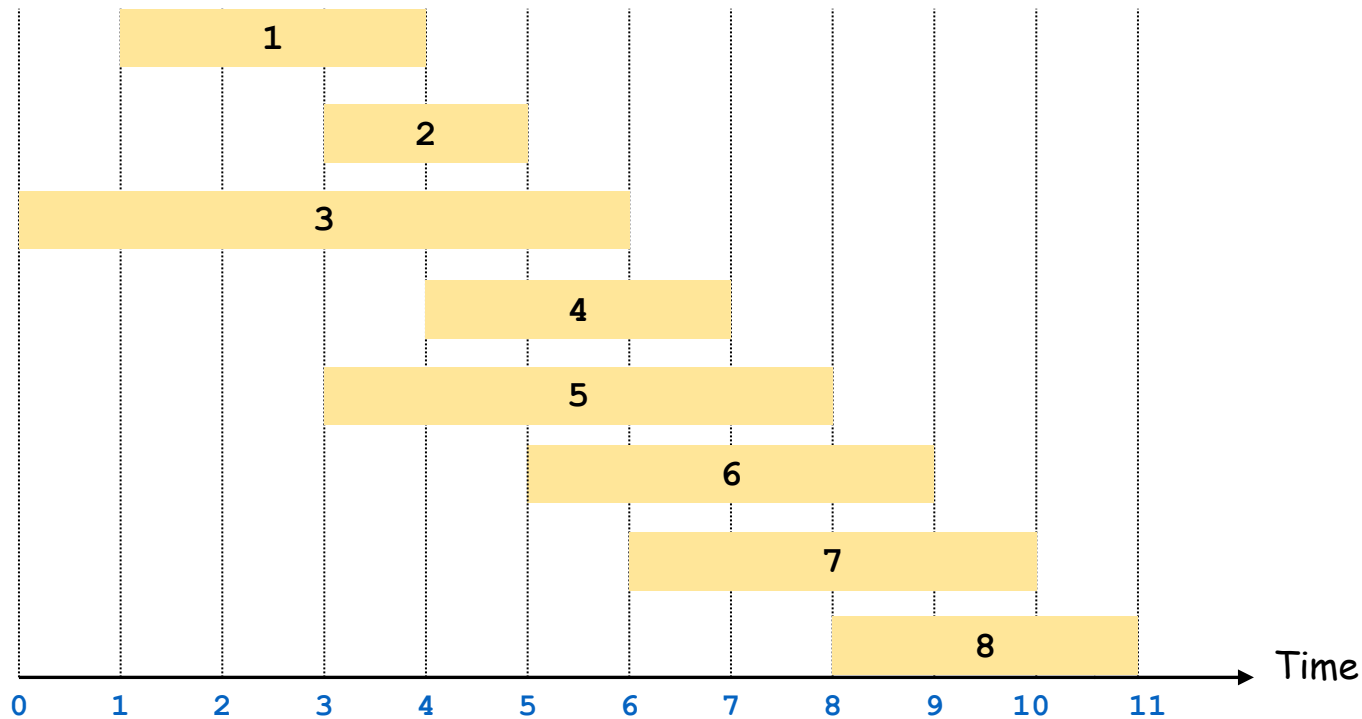
$j$	$v_j$	$p(j)$	$OPT[j]$
0	-	-	0
1	3	0	
2	2	0	
3	6	0	
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

# Example Execution (iterative)

**Notation:** Label jobs by finishing time:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Defn:**  $p(j)$  = largest index  $i < j$  s.t. job  $i$  is compatible with  $j$ .

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j - 1)\}$$



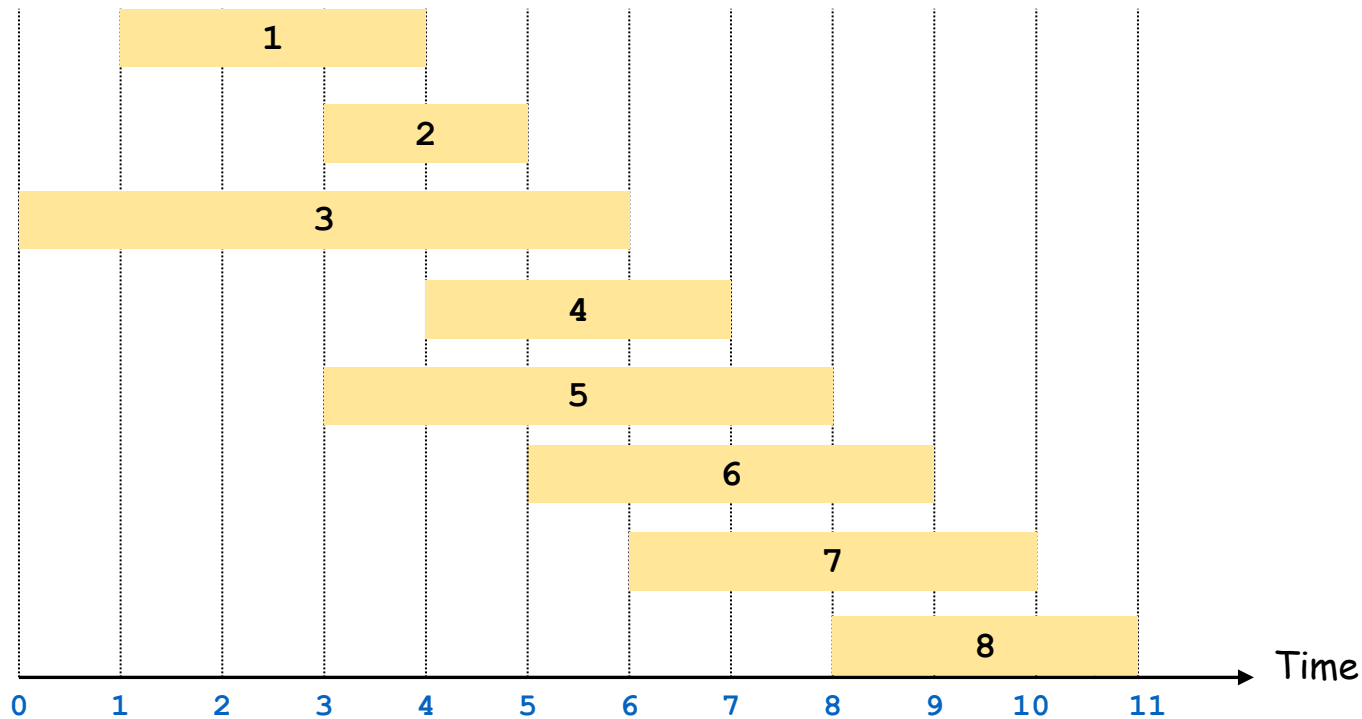
$j$	$v_j$	$p(j)$	$OPT[j]$
0	-	-	0
1	3	0	3
2	2	0	
3	6	0	
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

# Example Execution (iterative)

**Notation:** Label jobs by finishing time:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Defn:**  $p(j)$  = largest index  $i < j$  s.t. job  $i$  is compatible with  $j$ .

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j - 1)\}$$



$j$	$v_j$	$p(j)$	$OPT[j]$
0	-	-	0
1	3	0	3
2	2	0	
3	6	0	
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

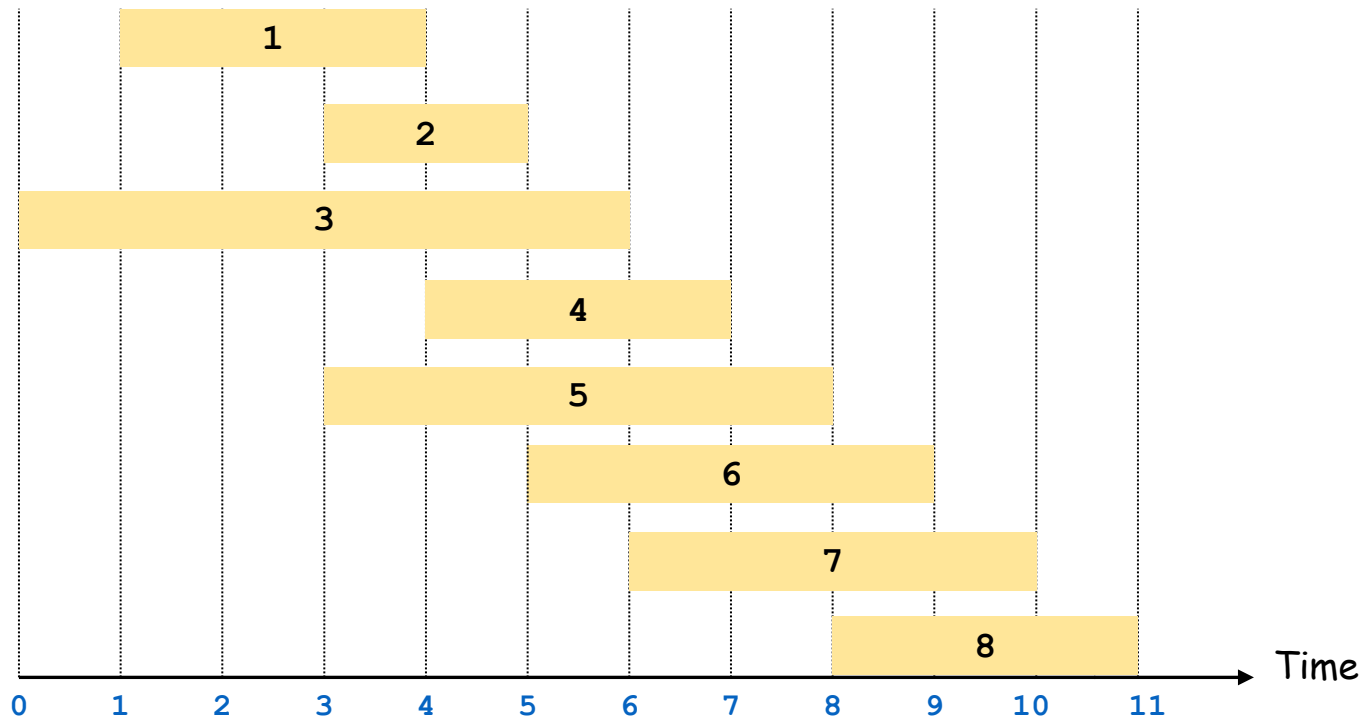


# Example Execution (iterative)

**Notation:** Label jobs by finishing time:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Defn:**  $p(j)$  = largest index  $i < j$  s.t. job  $i$  is compatible with  $j$ .

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j - 1)\}$$



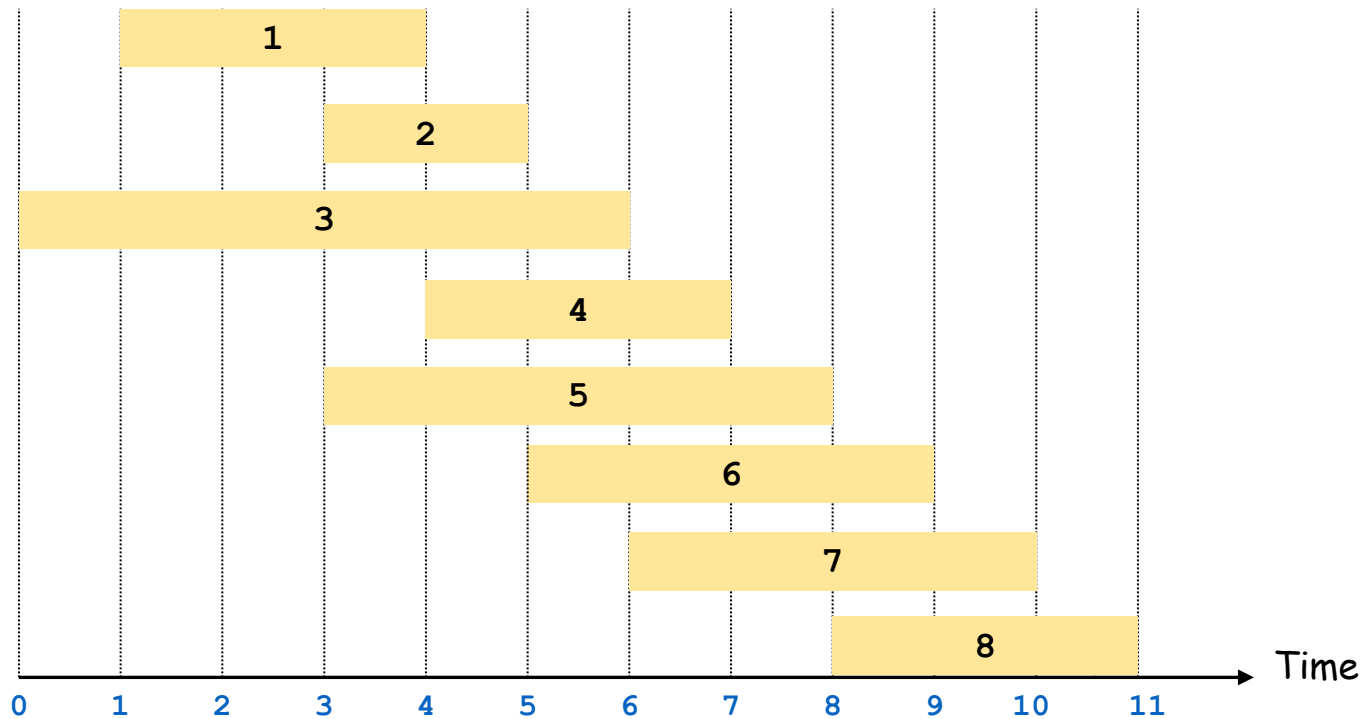
$j$	$v_j$	$p(j)$	$OPT[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

# Example Execution (iterative)

**Notation:** Label jobs by finishing time:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Defn:**  $p(j)$  = largest index  $i < j$  s.t. job  $i$  is compatible with  $j$ .

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j - 1)\}$$



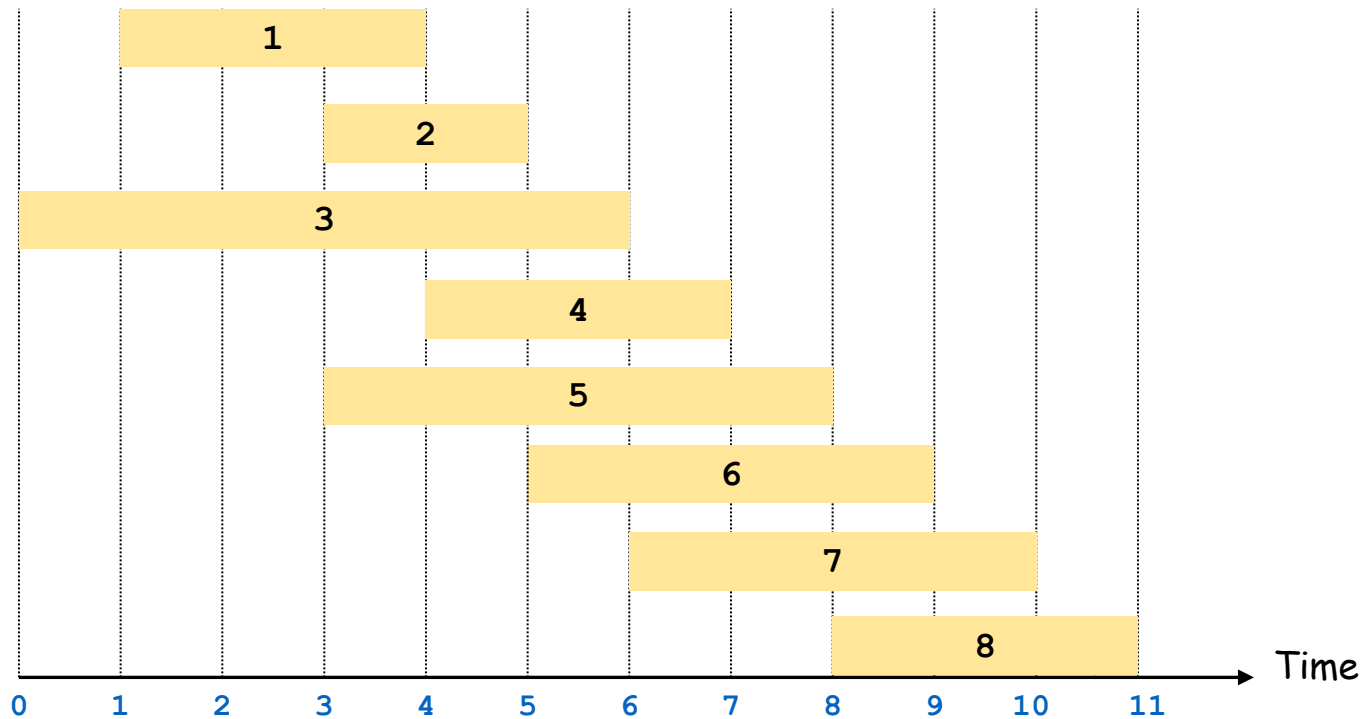
$j$	$v_j$	$p(j)$	$OPT[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

# Example Execution (iterative)

**Notation:** Label jobs by finishing time:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Defn:**  $p(j)$  = largest index  $i < j$  s.t. job  $i$  is compatible with  $j$ .

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j - 1)\}$$



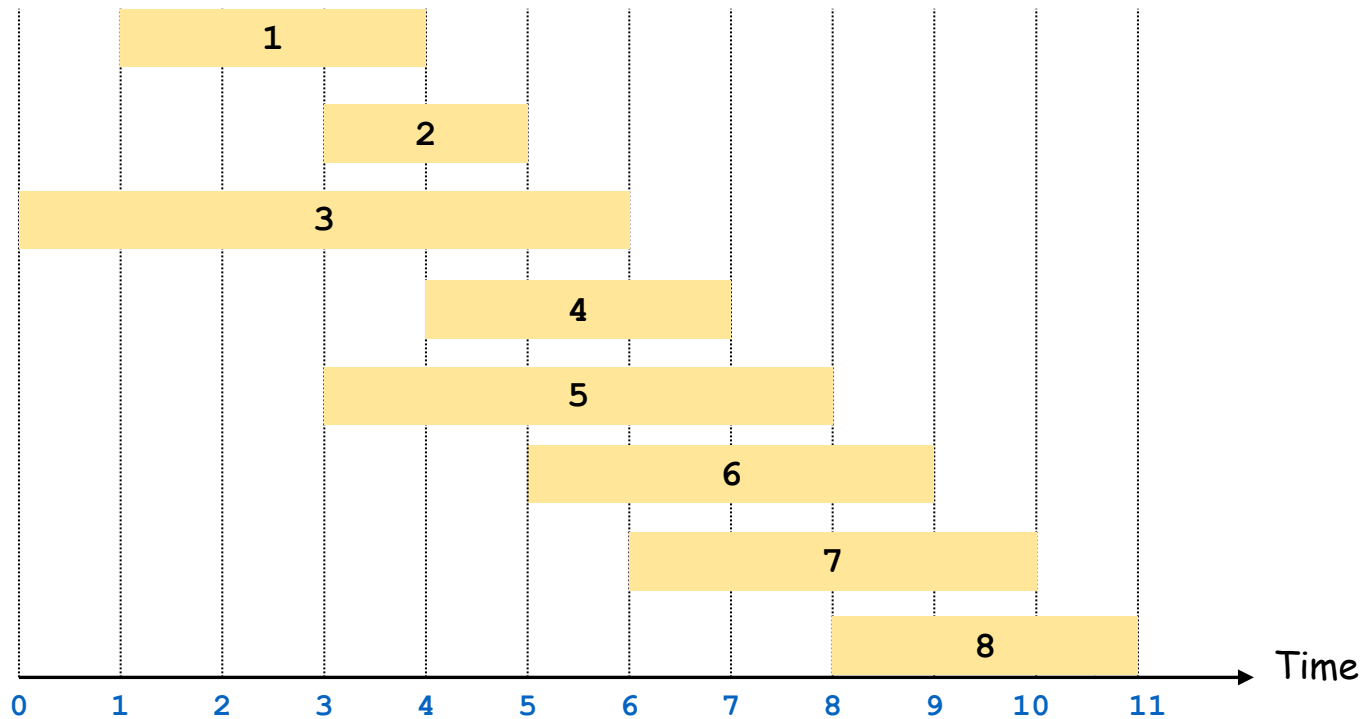
$j$	$v_j$	$p(j)$	$OPT[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

# Example Execution (iterative)

**Notation:** Label jobs by finishing time:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Defn:**  $p(j)$  = largest index  $i < j$  s.t. job  $i$  is compatible with  $j$ .

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j - 1)\}$$



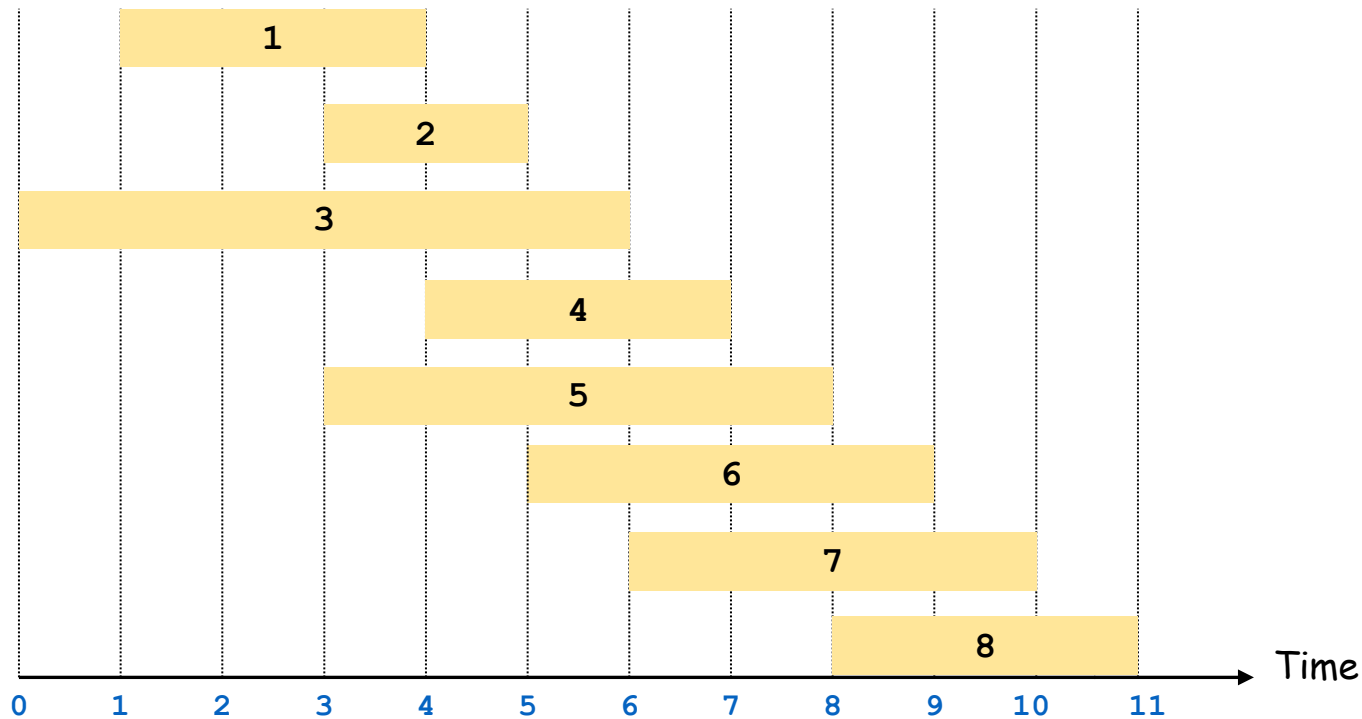
$j$	$v_j$	$p(j)$	$OPT[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

# Example Execution (iterative)

**Notation:** Label jobs by finishing time:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Defn:**  $p(j)$  = largest index  $i < j$  s.t. job  $i$  is compatible with  $j$ .

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j - 1)\}$$



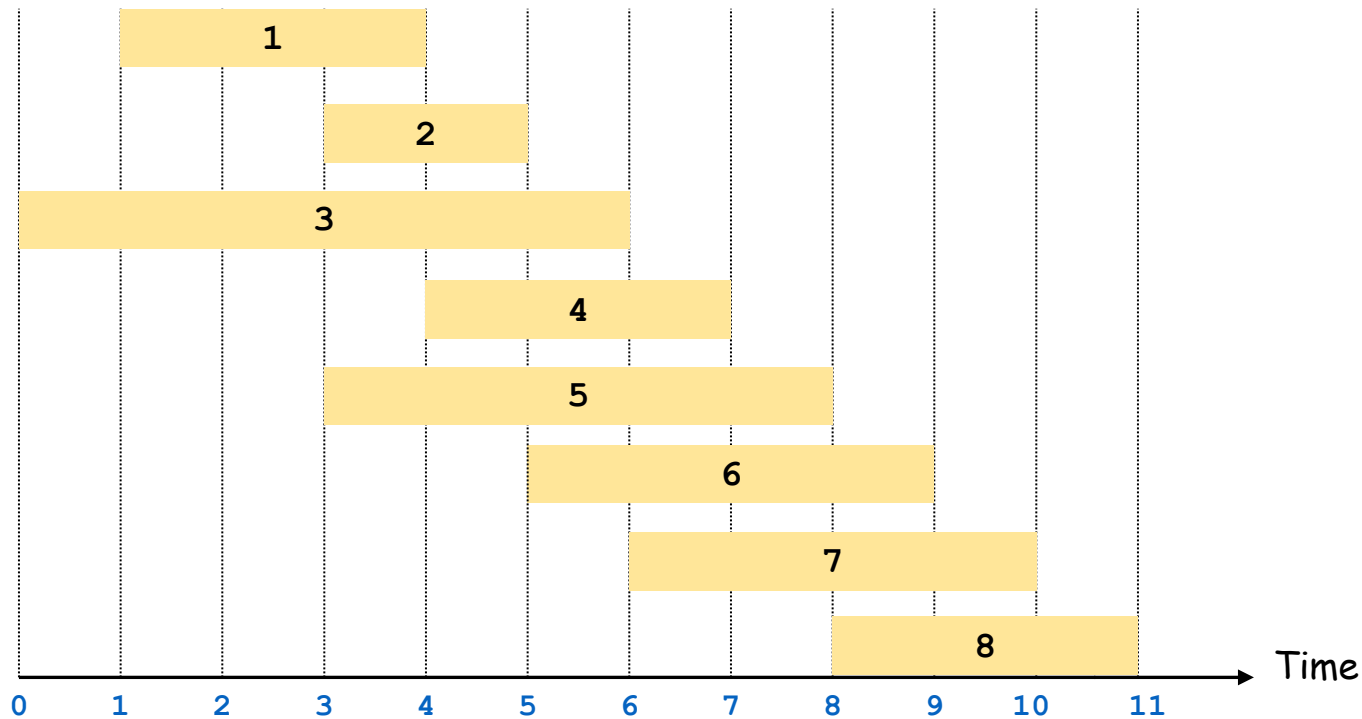
$j$	$v_j$	$p(j)$	$OPT[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	
6	4	2	
7	4	3	
8	3	5	

# Example Execution (iterative)

**Notation:** Label jobs by finishing time:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Defn:**  $p(j)$  = largest index  $i < j$  s.t. job  $i$  is compatible with  $j$ .

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j - 1)\}$$



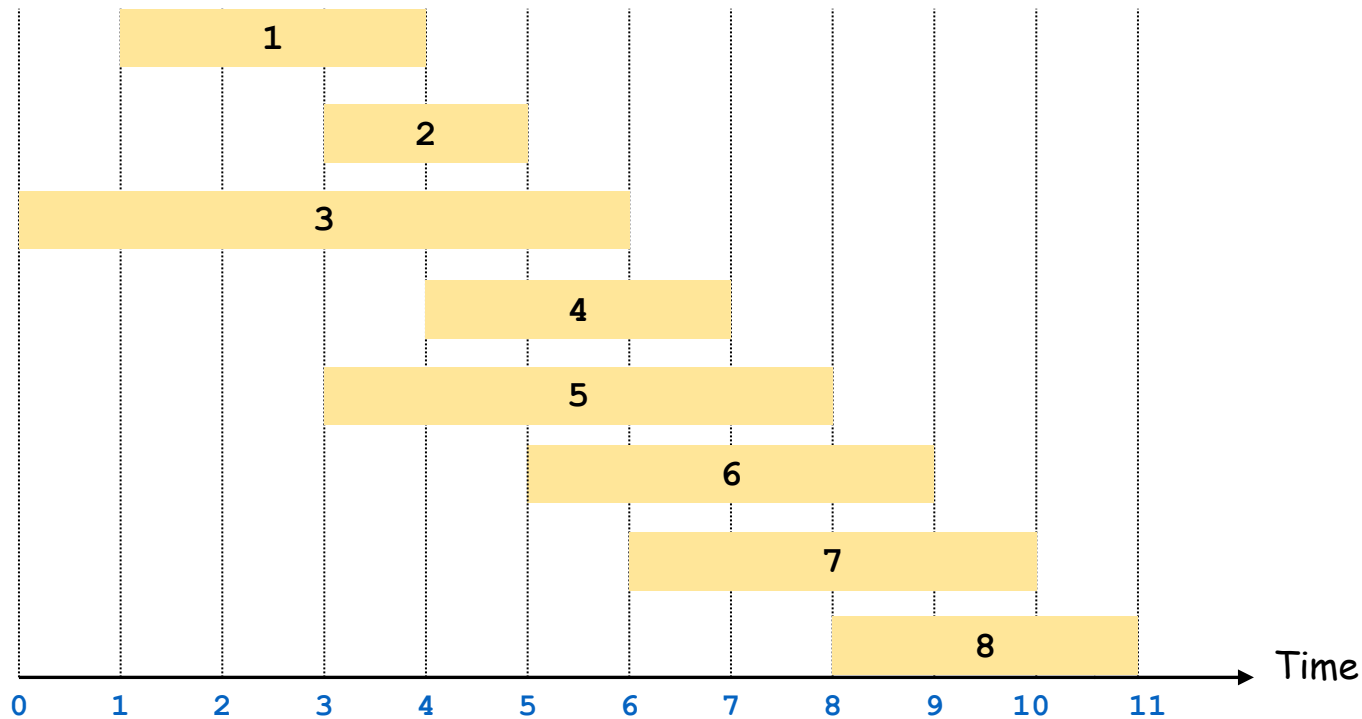
$j$	$v_j$	$p(j)$	$OPT[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	5
6	4	2	
7	4	3	
8	3	5	

# Example Execution (iterative)

**Notation:** Label jobs by finishing time:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Defn:**  $p(j)$  = largest index  $i < j$  s.t. job  $i$  is compatible with  $j$ .

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j - 1)\}$$



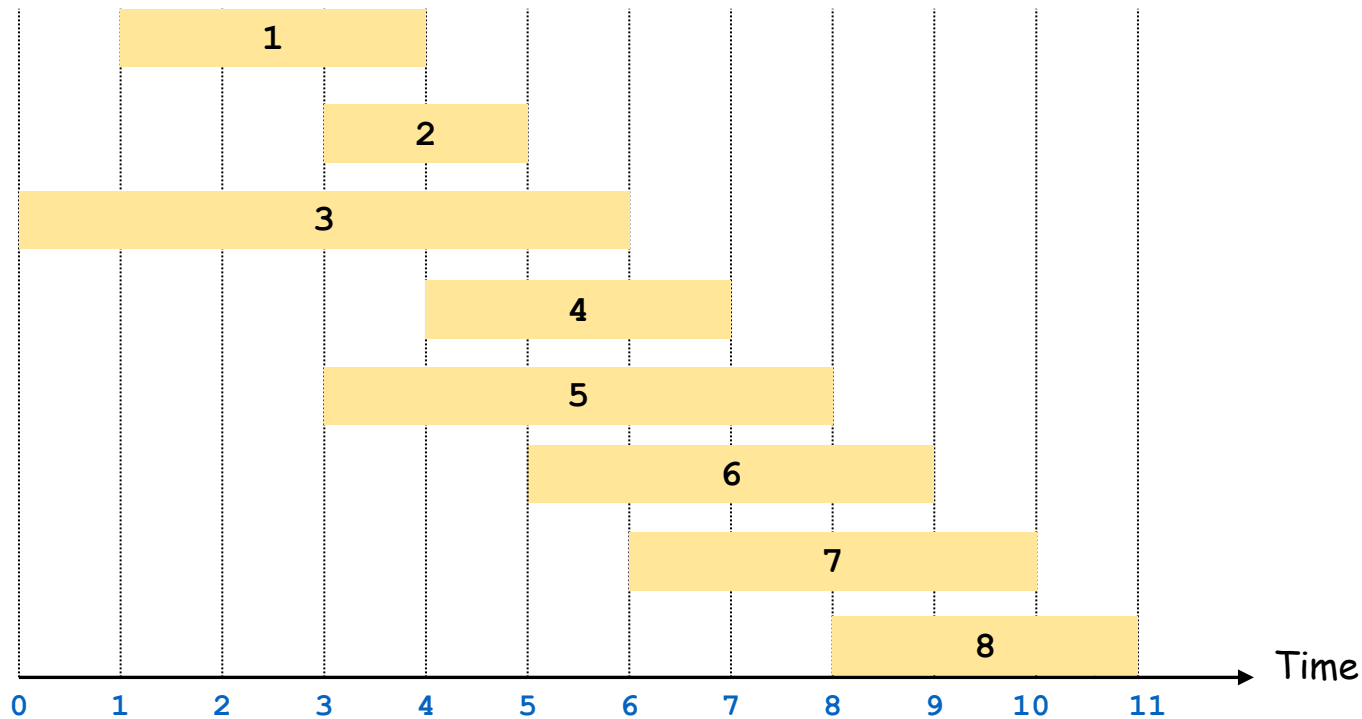
$j$	$v_j$	$p(j)$	$OPT[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	6
6	4	2	7
7	4	3	
8	3	5	

# Example Execution (iterative)

**Notation:** Label jobs by finishing time:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Defn:**  $p(j)$  = largest index  $i < j$  s.t. job  $i$  is compatible with  $j$ .

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j - 1)\}$$



$j$	$v_j$	$p(j)$	$OPT[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	6
6	4	2	7
7	4	3	
8	3	5	

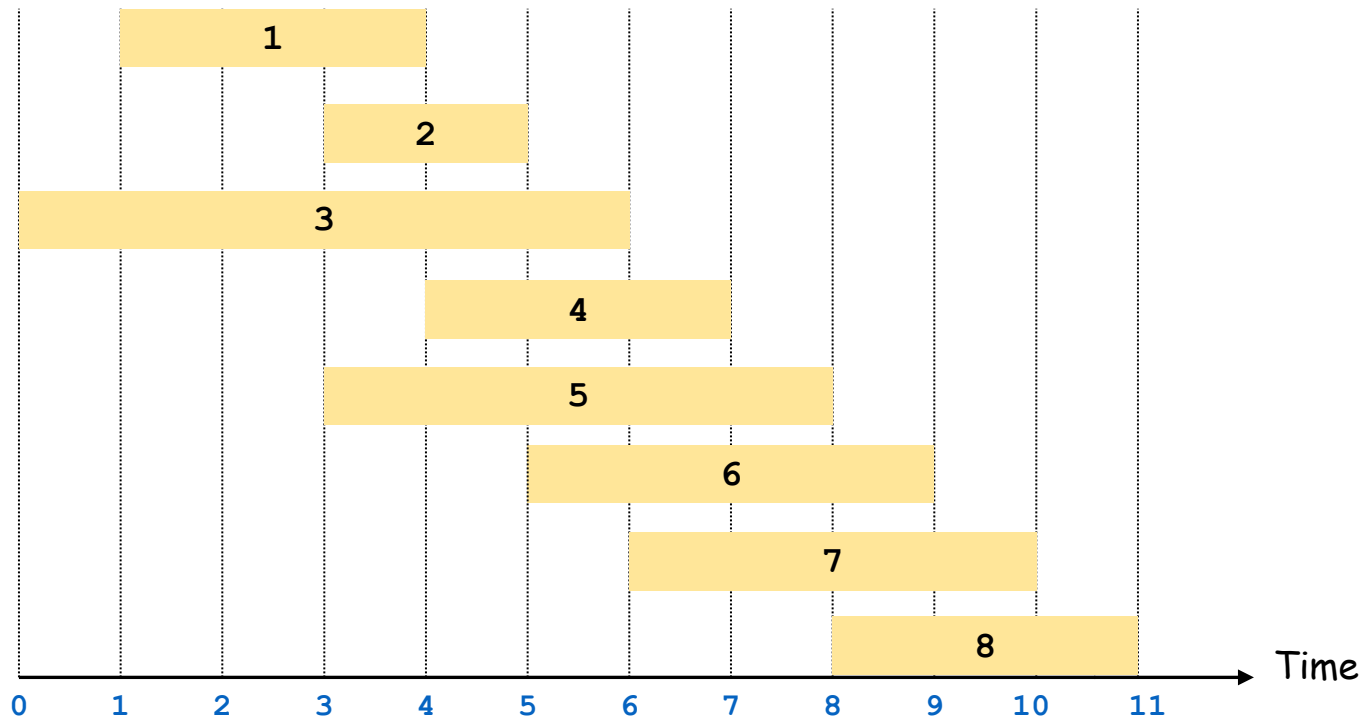


# Example Execution (iterative)

**Notation:** Label jobs by finishing time:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Defn:**  $p(j)$  = largest index  $i < j$  s.t. job  $i$  is compatible with  $j$ .

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j - 1)\}$$



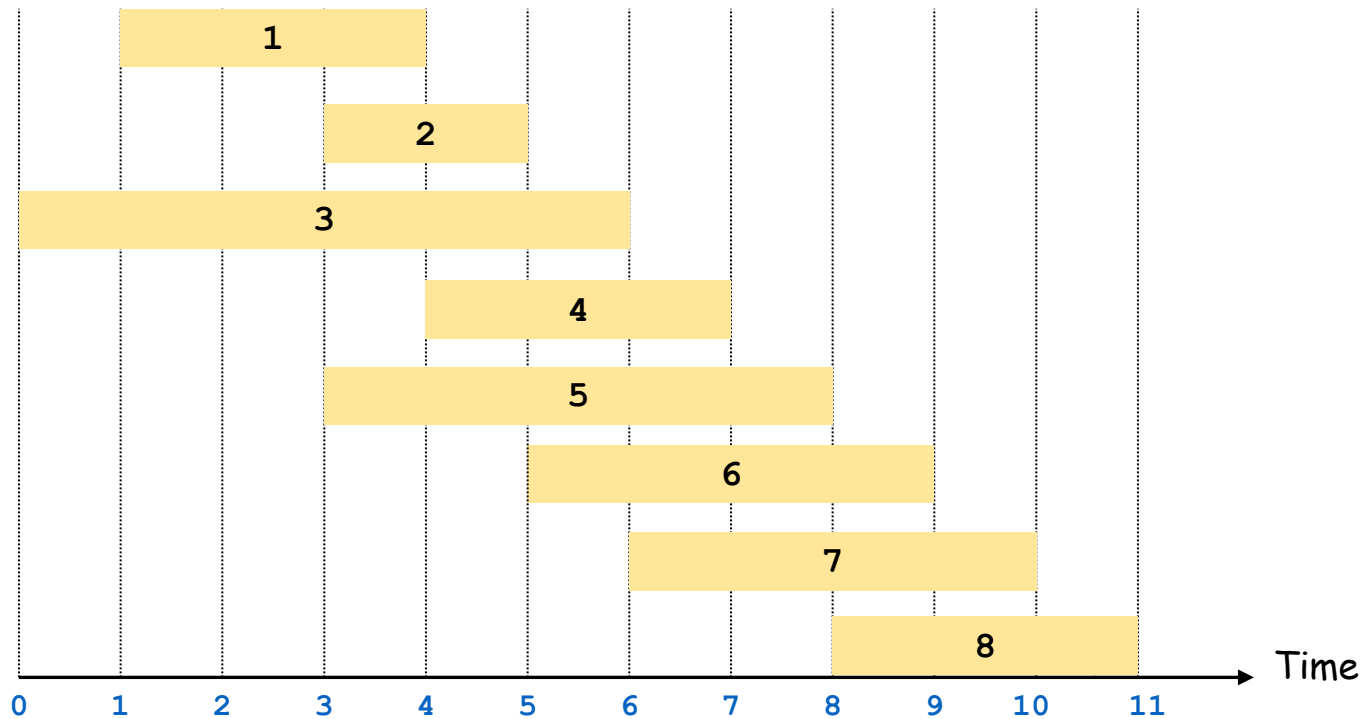
$j$	$v_j$	$p(j)$	$OPT[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	6
6	4	2	7
7	4	3	10
8	3	5	

# Example Execution (iterative)

**Notation:** Label jobs by finishing time:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Defn:**  $p(j)$  = largest index  $i < j$  s.t. job  $i$  is compatible with  $j$ .

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j - 1)\}$$



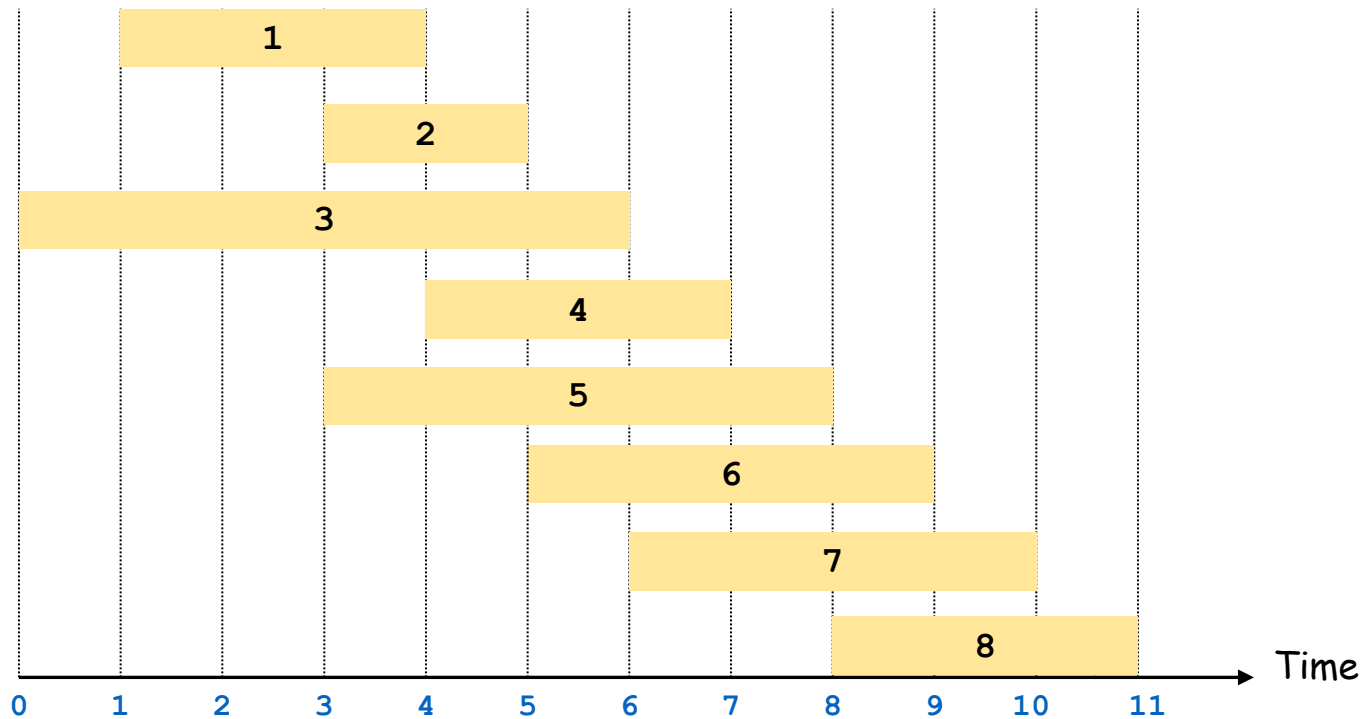
$j$	$v_j$	$p(j)$	$OPT[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	6
6	4	2	7
7	4	3	10
8	3	5	

# Example Execution (iterative)

**Notation:** Label jobs by finishing time:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Defn:**  $p(j)$  = largest index  $i < j$  s.t. job  $i$  is compatible with  $j$ .

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j - 1)\}$$



$j$	$v_j$	$p(j)$	$OPT[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	6
6	4	2	7
7	4	3	10
8	3	5	10

# Weighted Interval Scheduling: Finding the Solution

So far we have computed the value  $\text{OPT}(n)$  but we probably want to know what that solution  $\text{OPT}$  actually is!

We can do this, too, by keeping track of which option was better at each step.

Define  $\text{Used}[j] = \begin{cases} 1 & \text{solution with value } \text{OPT}(j) \text{ includes request } j \\ 0 & \text{otherwise} \end{cases}$

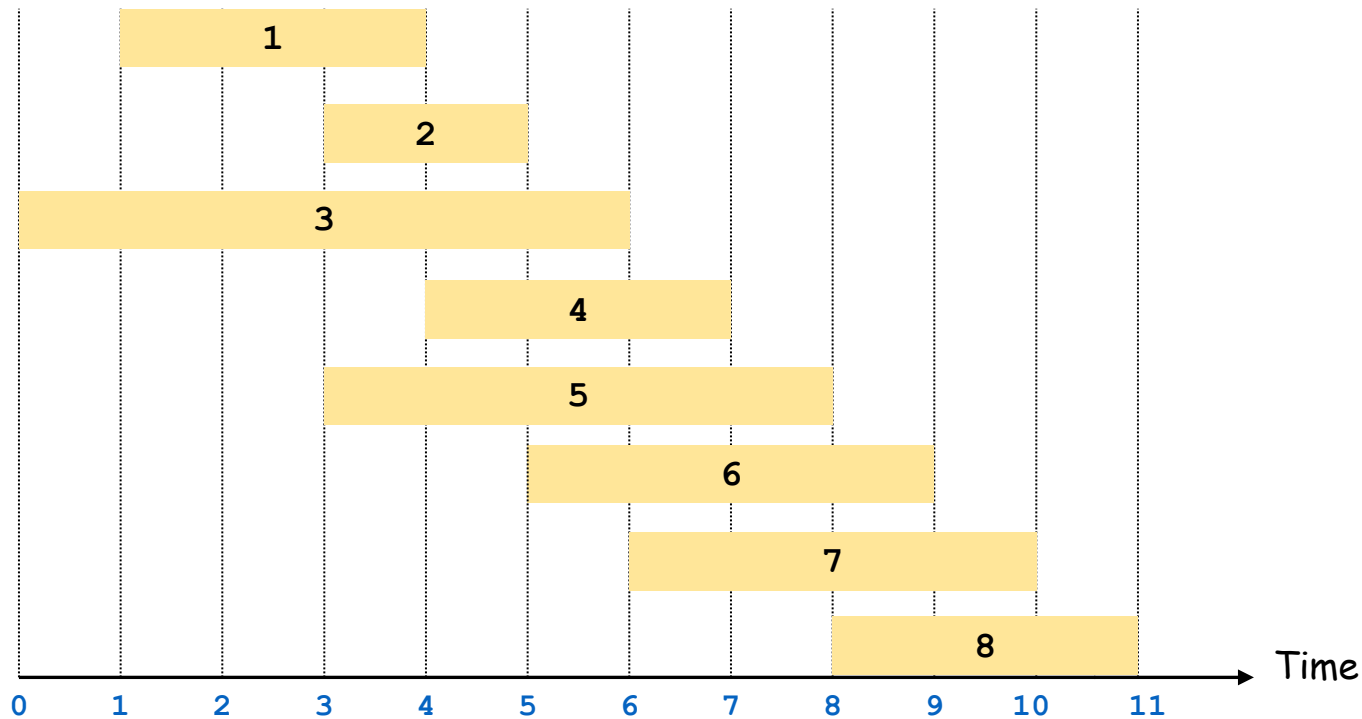
This gives a “pointer” that leads the way along a path to the optimal solution...

# Weighted Interval Scheduling: Finding the Solution

**Notation:** Label jobs by finishing time:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Defn:**  $p(j)$  = largest index  $i < j$  s.t. job  $i$  is compatible with  $j$ .

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j - 1)\}$$



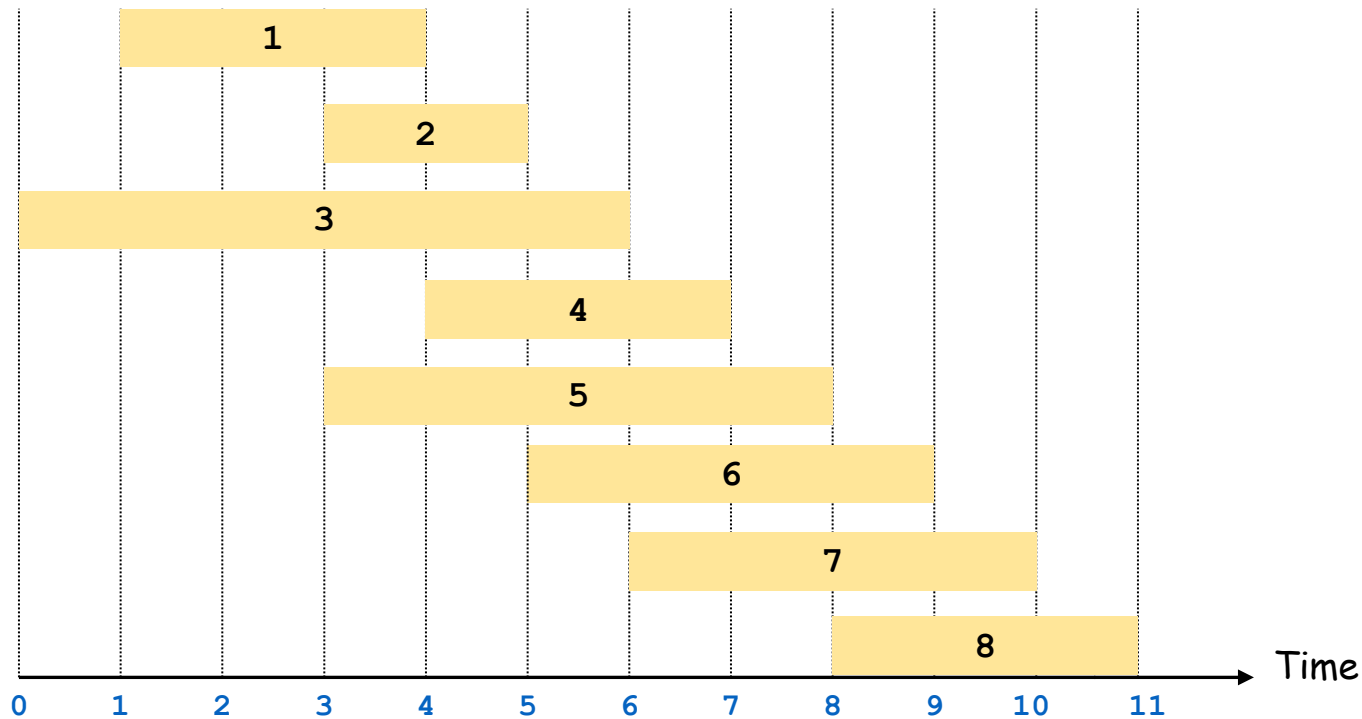
$j$	$v_j$	$p(j)$	$OPT[j]$	Used $[j]$
0	-	-	0	-
1	3	0	3	1
2	2	0	3	0
3	6	0	6	1
4	3	1	6	1
5	5	0	6	0
6	4	2	7	1
7	4	3	10	1
8	3	5	10	0

# Weighted Interval Scheduling: Iterative Solution

**Notation:** Label jobs by finishing time:  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Defn:**  $p(j)$  = largest index  $i < j$  s.t. job  $i$  is compatible with  $j$ .

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j - 1)\}$$



$j$	$v_j$	$p(j)$	$OPT[j]$	Used $[j]$
0	-	-	0	-
1	3	0	3	1
2	2	0	3	0
3	6	0	6	1
4	3	1	6	1
5	5	0	6	0
6	4	2	7	1
7	4	3	10	1
8	3	5	10	0

# Weighted Interval Scheduling - Complete

**Sort requests by finish time**

**Compute each  $p(i)$**

**WIS( $j$ ):**

OPT[0] = 0

for each  $i = 1$  up to  $j$ :

include $_i$  = OPT[ $p(i)$ ]+value[ $i$ ]

exclude $_i$  = OPT[ $i - 1$ ]

if include $_i$  > exclude $_i$ :

OPT[ $i$ ] = include $_i$

used[ $i$ ] = 1

else:

OPT[ $i$ ] = exclude $_i$

used[ $i$ ] = 0

return find\_opt(used);

find\_opt(used):

$j = n$

intervals = {}

while  $j > 0$ :

if used[ $j$ ]==0:

$j = j - 1$

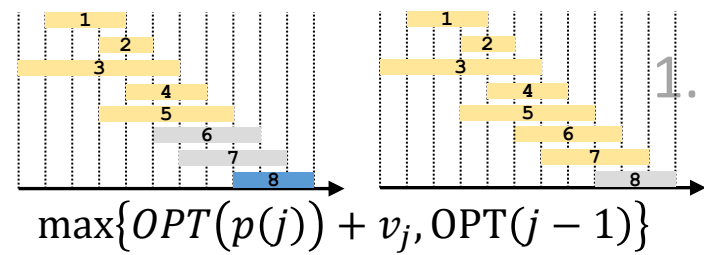
else:

intervals.add( $j$ )

$j = p(j)$

return intervals

# Weighted Interval Scheduling – Four Steps



$j$	$OPT[j]$
0	0
1	
2	
3	
4	
5	
6	
7	
8	

$j$	$OPT[j]$
0	0
1	
2	
3	
4	
5	
6	
7	
8	

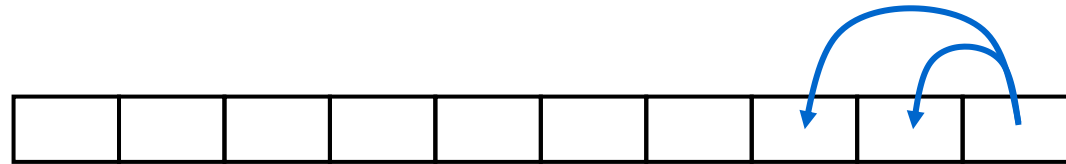
1. Formulate the answer with a recursive structure
  - What are the options for the last choice?
  - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
  - Figure out the possible values of all parameters in the recursive calls.
  - How many subproblems (options for last choice) are there?
  - What are the parameters needed to identify each?
  - How many different values could there be per parameter?
3. Specify an order of evaluation.
  - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
  - With this step: a “Bottom-up” (iterative) algorithm
  - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
  - Is it possible to reuse some memory locations?



# Dynamic Programming Patterns

Fibonacci pattern:

- 1-dimensional,  $O(1)$  values immediately prior
- Space saving possible



Weighted interval scheduling pattern:

- 1-dimensional,  $O(1)$  values arbitrarily far back
- No space saving possible

