CSE 421
Introduction to Algorithms

Lecture 11: Dynamic Programming

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.*

Dynamic Programming Applications

Areas.

- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, compilers, systems, ….

Some famous dynamic programming algorithms.

- Unix **`diff`** for comparing two files.
- Viterbi for hidden Markov models.
- Smith-Waterman for genetic sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

Three Steps to Dynamic Programming

1. Formulate the answer as a recurrence relation or recursive algorithm

2. Figure out the possible values of parameters in the recursive calls.
    - This should be “small”, i.e., bounded by a low-degree polynomial
    - Can use memoization to store a cache of previously computing values

3. Specify an order of evaluation for the recurrence so that you already have the partial results stored in memory when you need them.

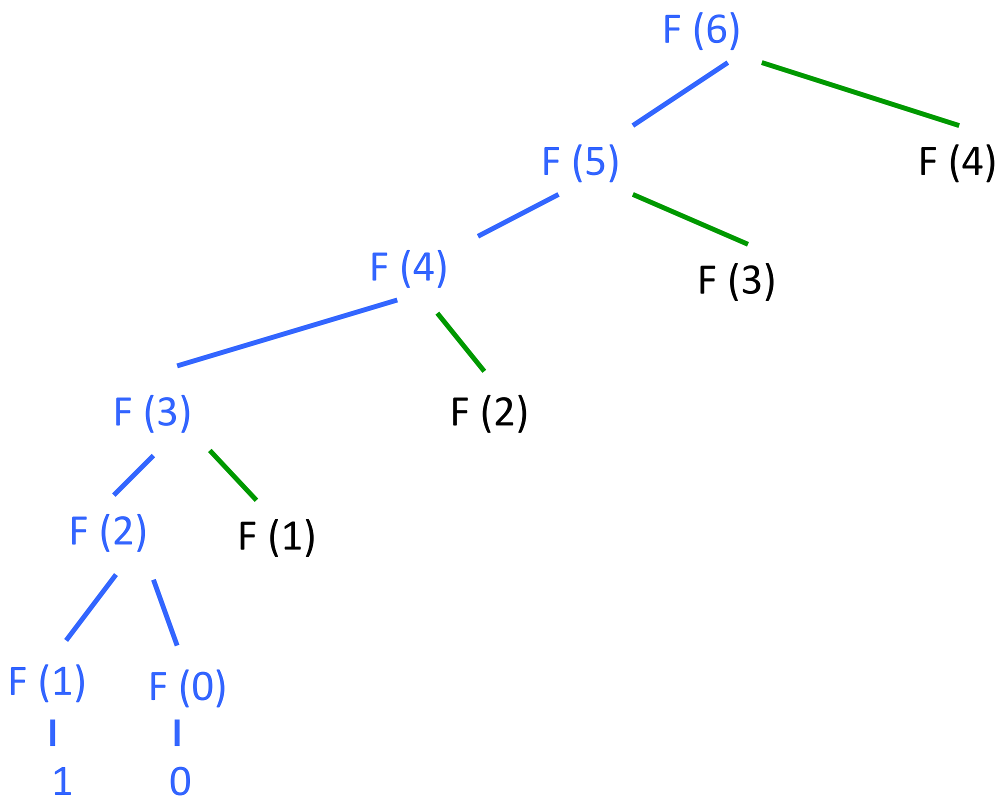
A Simple Case: Computing Fibonacci Numbers

Recall $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$ and $F_0 = 0, F_1 = 1$

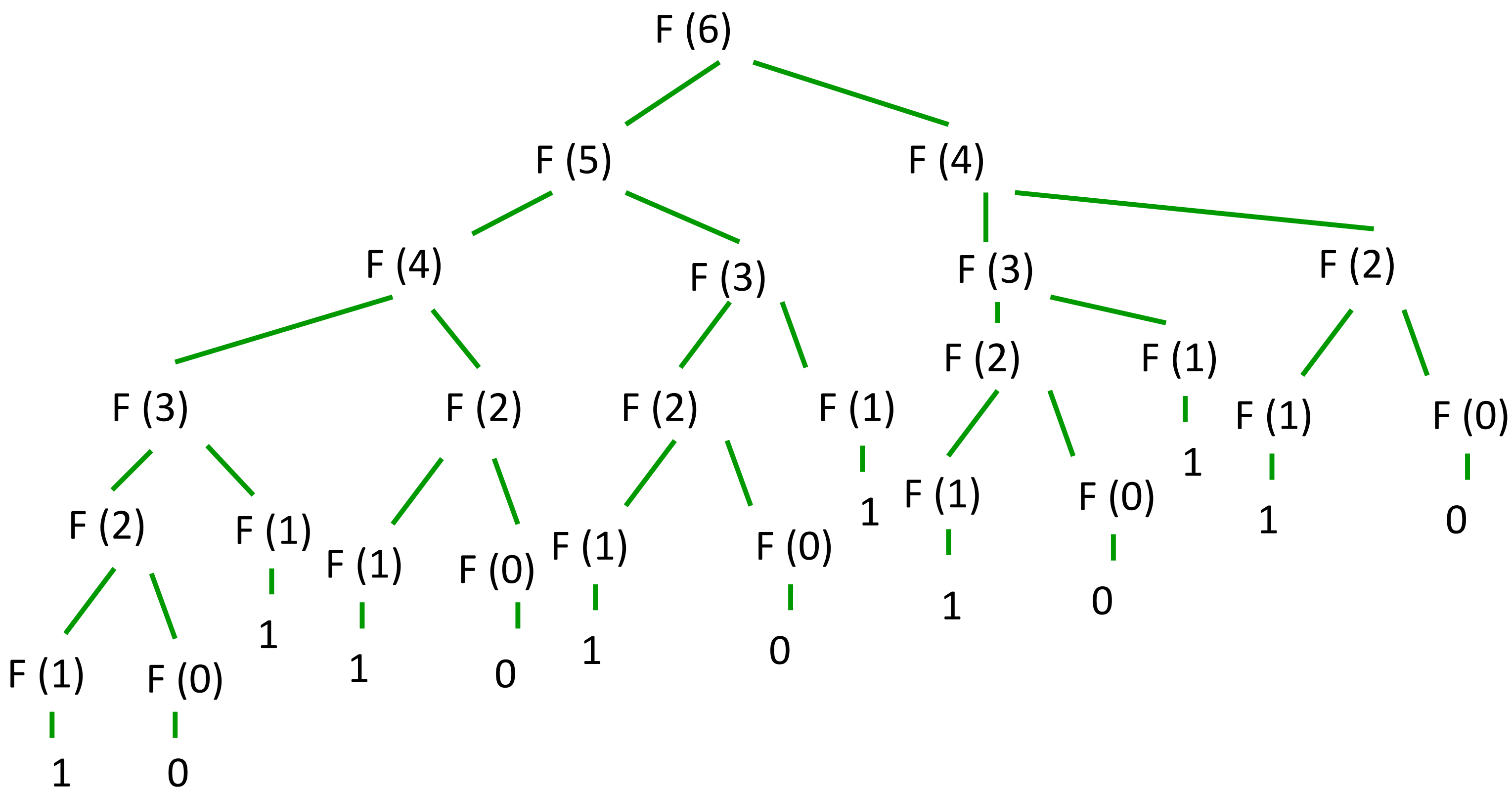
The obvious recursive algorithm direct from this recurrence is

```
F(n){
      if n=0 return(0)
      else if n=1 return(1)
      else return(F(n-1)+F(n-2))
}
```

Let's start tracking the call tree...



The full call tree has $> F_n$ leaves (exponential in n)



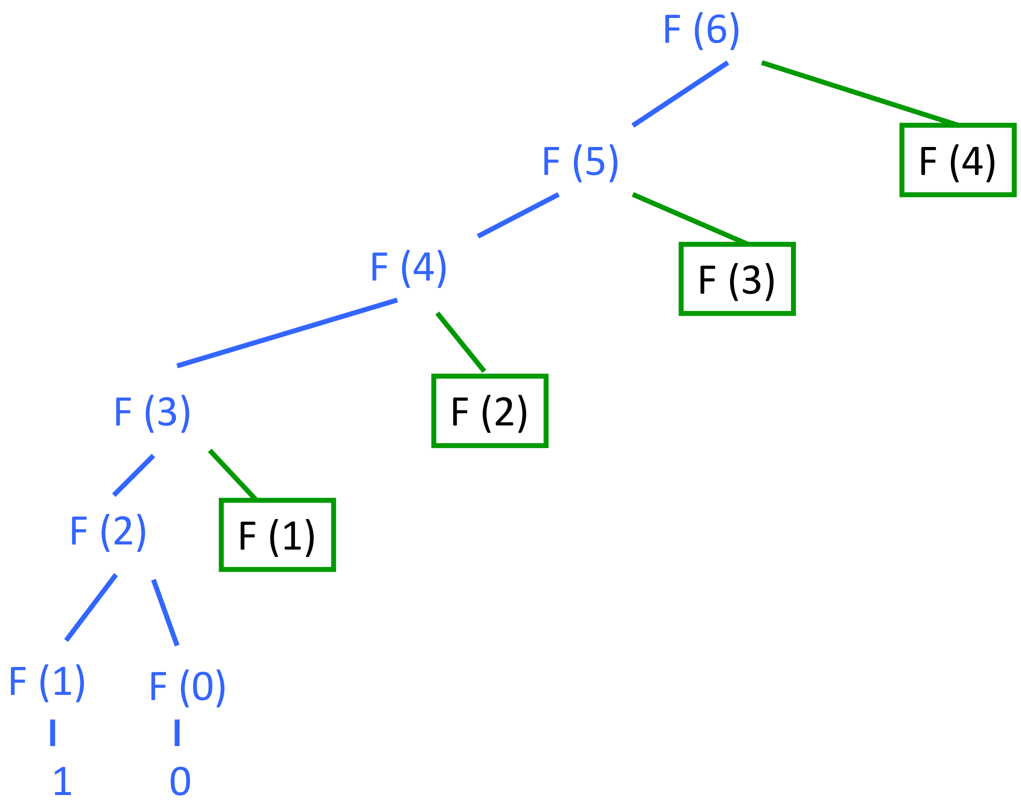
Memoization (Caching)

Remember all values from previous recursive calls in a cache

- the parameters and
- The values returned on those parameters

Before each recursive call, test to see if value has already been computed for those parameters

Memoization



Three Steps to Dynamic Programming

1. Formulate the answer as a recurrence relation or recursive algorithm

2. Figure out the possible values of parameters in the recursive calls.
   - This should be "small", i.e., bounded by a low-degree polynomial
   - Can use memoization to store a cache of previously computing values

3. Specify an order of evaluation for the recurrence so that you already have the partial results stored in memory when you need them.
   - Produces iterative code

Three Steps to Dynamic Programming

1. Formulate the answer as a recurrence relation or recursive algorithm

2. Figure out the possible values of parameters in the recursive calls.
    - This should be “small”, i.e., bounded by a low-degree polynomial
    - Can use memoization to store a cache of previously computing values

3. Specify an order of evaluation for the recurrence so that you already have the partial results stored in memory when you need them.
    - Produces iterative code

Fibonacci: Dynamic Programming Version

```
FiboDP(n):
      F[0]←0
      F[1]←1
      for i←2 to n {
            F[i]←F[i-1]+F[i-2]
      }
      return(F[n])
```

PAUL G. ALLEN SCHOOL OF COMPUTER SCIENCE & ENGINEERING

Three Steps to Dynamic Programming

1. Formulate the answer as a recurrence relation or recursive algorithm

2. Figure out the possible values of parameters in the recursive calls.
    - This should be "small", i.e., bounded by a low-degree polynomial
    - Can use memoization to store a cache of previously computing values

3. Specify an order of evaluation for the recurrence so that you already have the partial results stored in memory when you need them.
    - Produces iterative code

Once you have an iterative DP solution: see if you can save space...

Fibonacci: Space-Saving Dynamic Programming

```
FiboDP(n):
      prev←0
      curr←1
      for i←2 to n {
            temp←curr
            curr←curr+prev
            prev←temp
      }
      return(curr)
```

Dynamic Programming

When is dynamic programming useful?

- For optimization problems this typically requires that the "Principle of optimality" hold for the problem

  "Optimal solutions to the sub-problems suffice for optimal solution to the whole 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

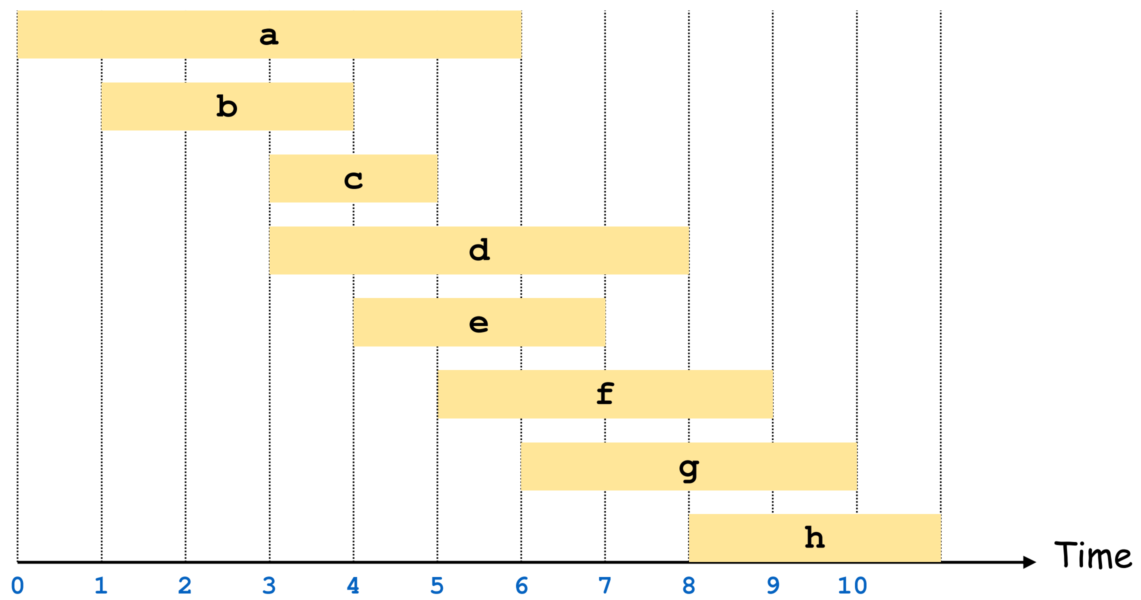
- the amount of money we get from renting out the resource
- the amount of time the resource is being used ($v_i = f_i - s_i$)

Find: A maximum-weight compatible subset of requests.

Weighted Interval Scheduling

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

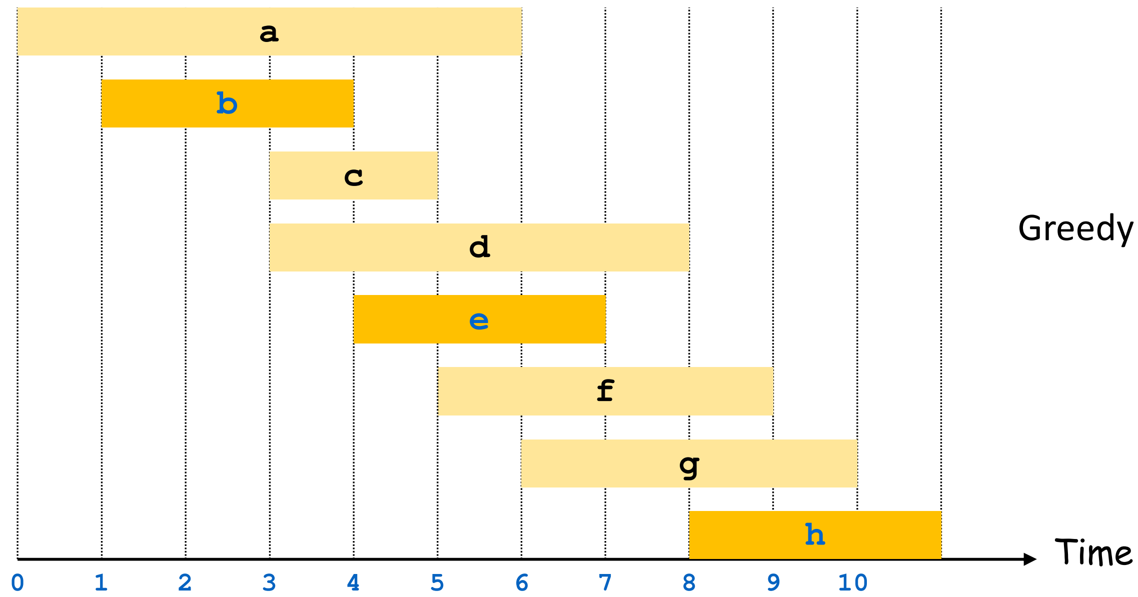
Goal: Find maximum weight subset of mutually compatible jobs.



Weighted Interval Scheduling

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

Goal: Find maximum weight subset of mutually compatible jobs.

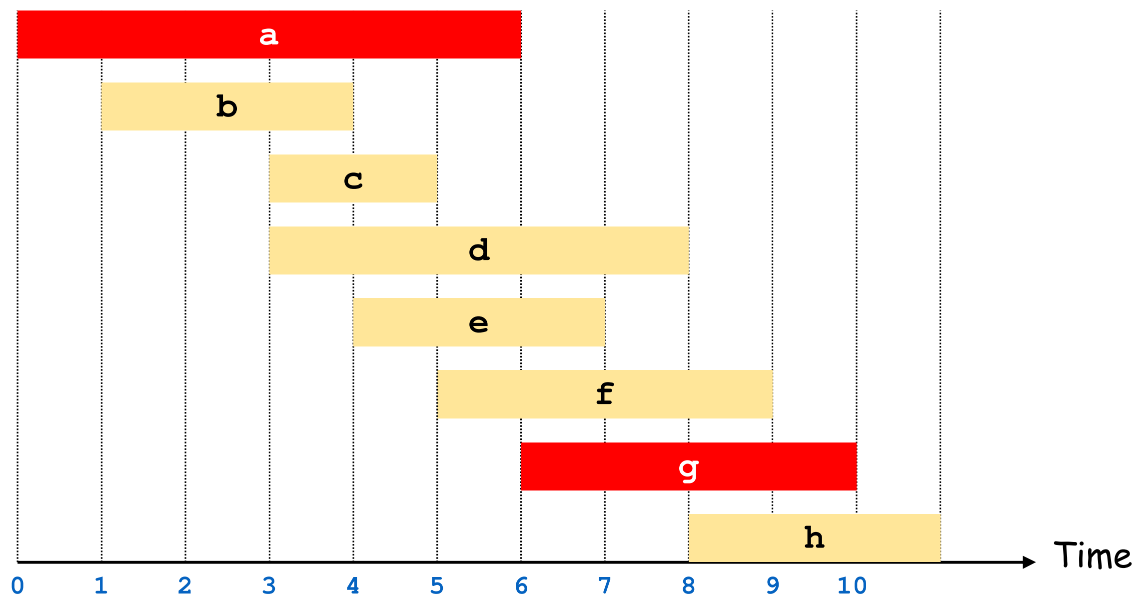


Greedy by finish times would give 9

Weighted Interval Scheduling

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

Goal: Find maximum weight subset of mutually compatible jobs.



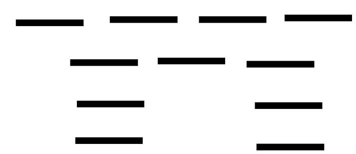
Optimal yields 10

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

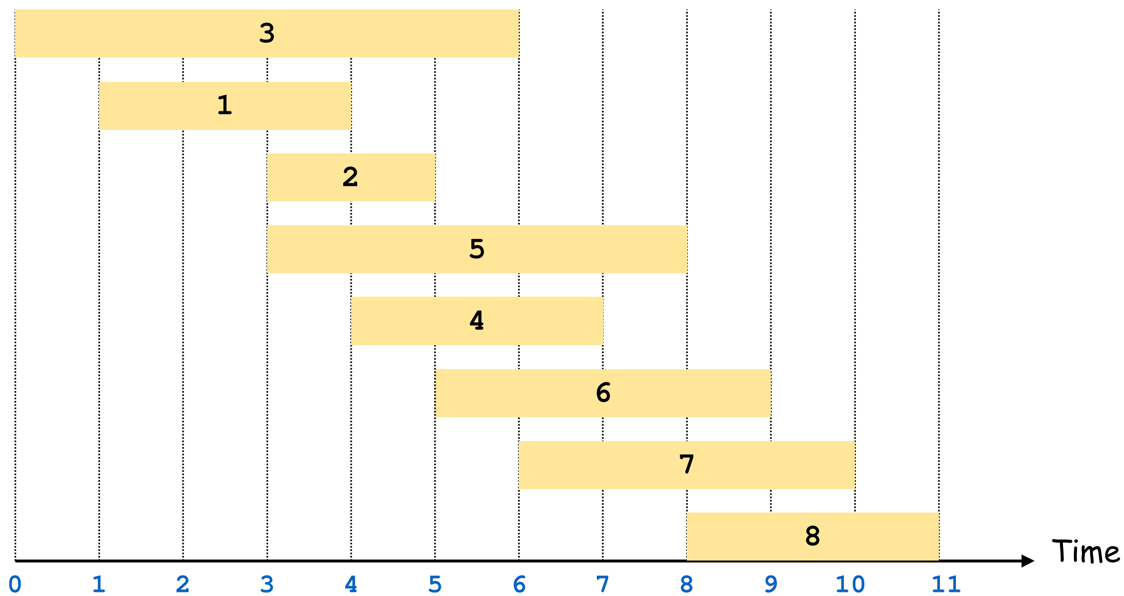


  - Earliest finish time f_i
    - Doesn't work

  - Largest value/weight v_i
    - Doesn't work

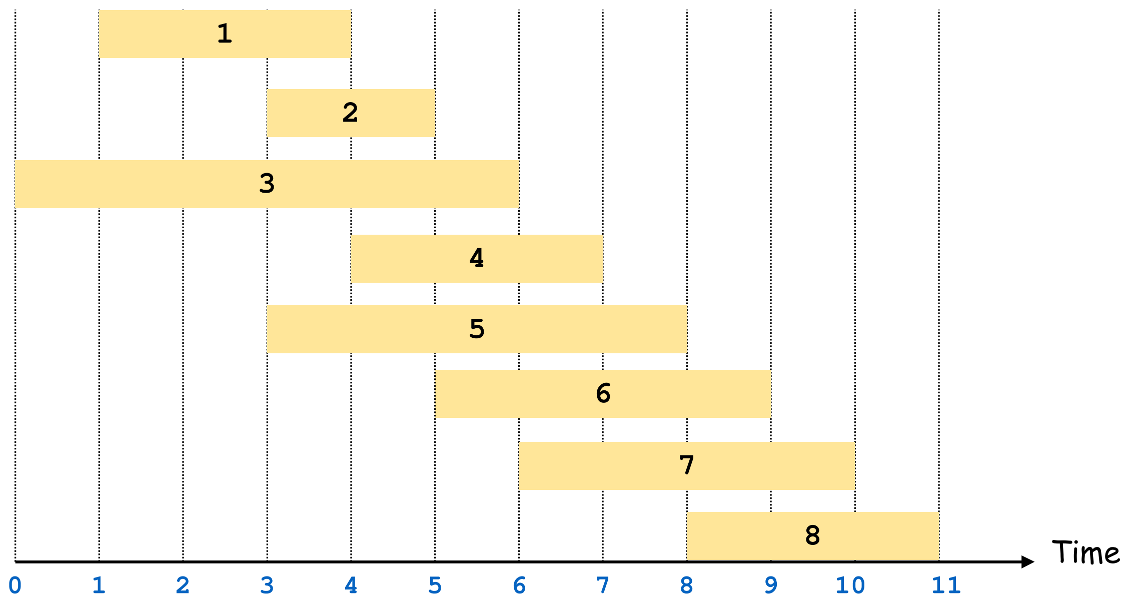
Weighted Interval Scheduling

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



Weighted Interval Scheduling

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



Towards Dynamic Programming: Step 1 – Recursive Algorithm

Suppose that we first sort the requests by finish time $\boldsymbol{f_i}$ so $\boldsymbol{f_1} \leq \boldsymbol{f_2} \leq \ldots \leq \boldsymbol{f_n}$.

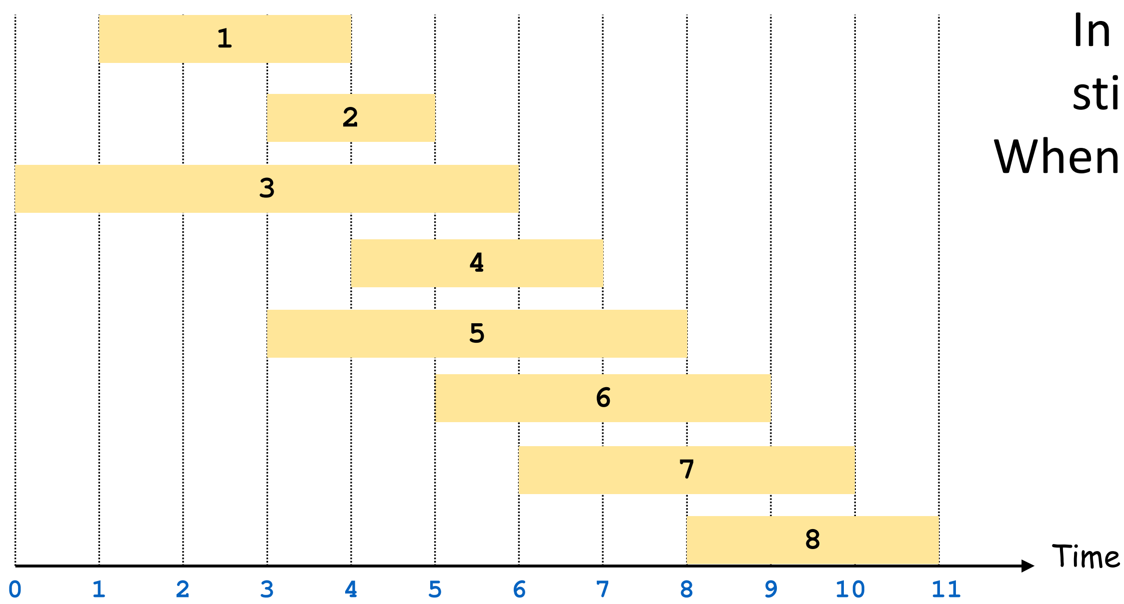
We now want

- 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 $\boldsymbol{n}$.

Weighted Interval Scheduling

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

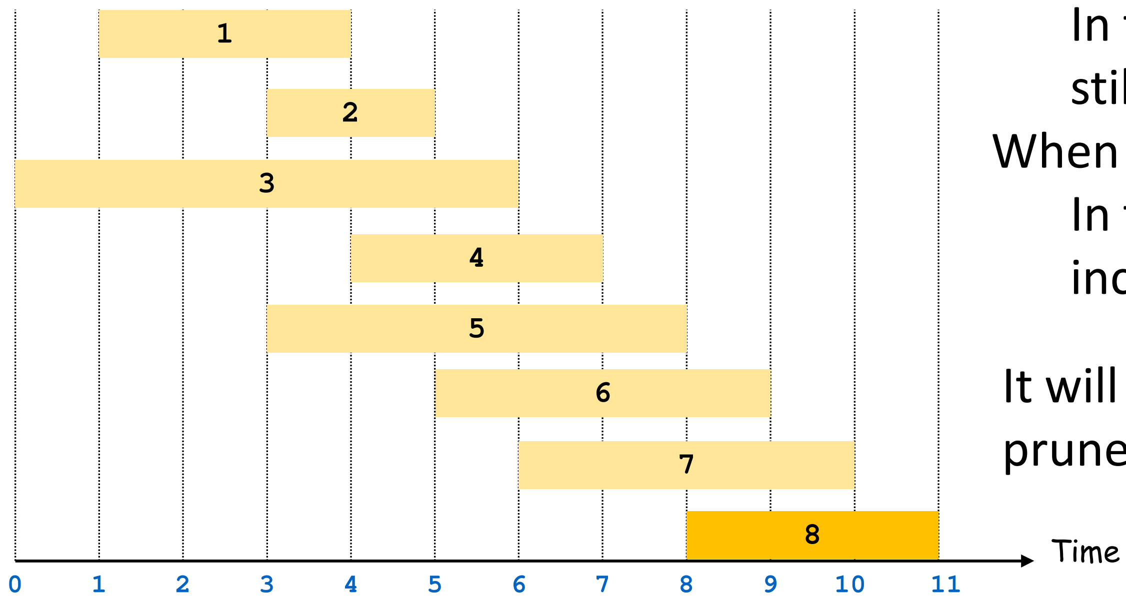


There are two cases we need to compare:

- When we don't include request n.
  - In this case all the other requests are still fair game
- When we do include request n.

Weighted Interval Scheduling

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



There are two cases we need to compare:

- When we don't include request n.
  - In this case all the other requests are still fair game
- When we do include request n.
  - In this case we need to rule out some incompatible requests.

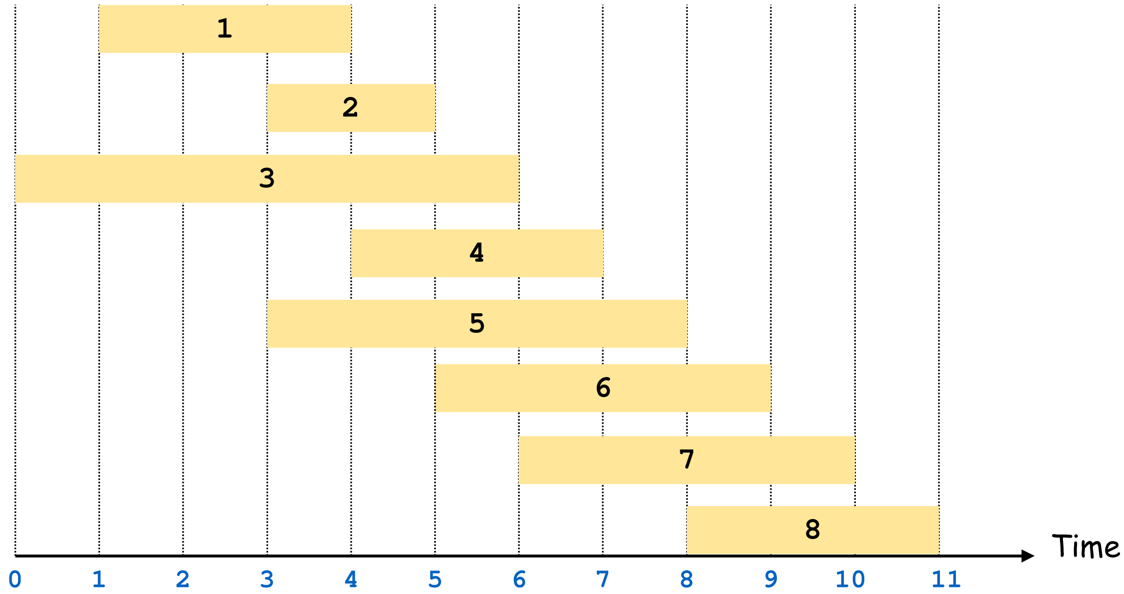
It will be convenient to be able to prune incompatible requests quickly…

Weighted Interval Scheduling

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

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

Structure of the subproblems

Notation: OPT(j) = value of optimal solution to the problem consisting of job requests $1, 2, \dots, j$.

Case 1: OPT selects job j

- It can't use incompatible jobs $p(j) + 1, \dots, j - 1$
- It must include an optimal solution to problem consisting of remaining compatible jobs $1, \dots, p(j)$.

Optimal substructure

Case 2: OPT doesn't select job

- It must include an optimal solution to problem consisting of remaining compatible jobs $1, \dots, j - 1$

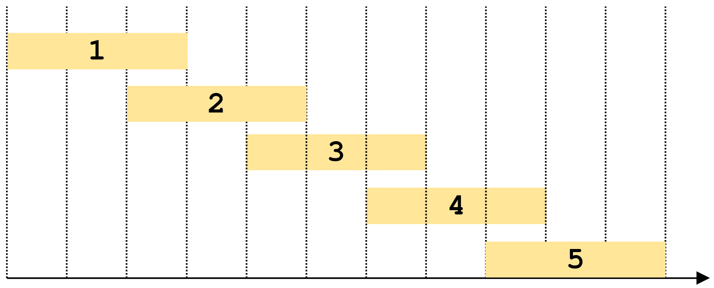
$$\text{OPT}(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{v_j + \text{OPT}(p(j)), \text{OPT}(j-1)\} & \text{otherwise} \end{cases}$$

Weighted Interval Scheduling: Recursive Solution

```
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))
}
```

Weighted Interval Scheduling: Recursive Solution

This recursive algorithm can be very bad…



Suppose that $p(j) = j - 2$ for every $j \geq 2$.

- Then **Compute-Opt**(j) calls **Compute-Opt**($j - 1$) and **Compute-Opt**($j - 2$)
- This is the same exponential run-time as the recursive Fibonacci code!

Weighted Interval Scheduling: Step 2 Memoization

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

```
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)

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(v_j + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

Weighted Interval Scheduling: Step 3

1. Formulate the answer as a recurrence relation or recursive algorithm

2. Figure out the possible values of parameters in the recursive calls.
   - This should be "small", i.e., bounded by a low-degree polynomial
   - Can use memoization to store a cache of previously computing values

3. Specify an order of evaluation for the recurrence so that you already have the partial results stored in memory when you need them.
   - Produces iterative code

Recursion for $\mathbf{OPT}[j]$ only needs values of $\mathbf{OPT}[i]$ for $0 \leq i < j$.

- So we can evaluate them in order $j = 0, 1, 2, \ldots, n$

Weighted Interval Scheduling: Iterative Solution

```
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)

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

$O(n \log n)$

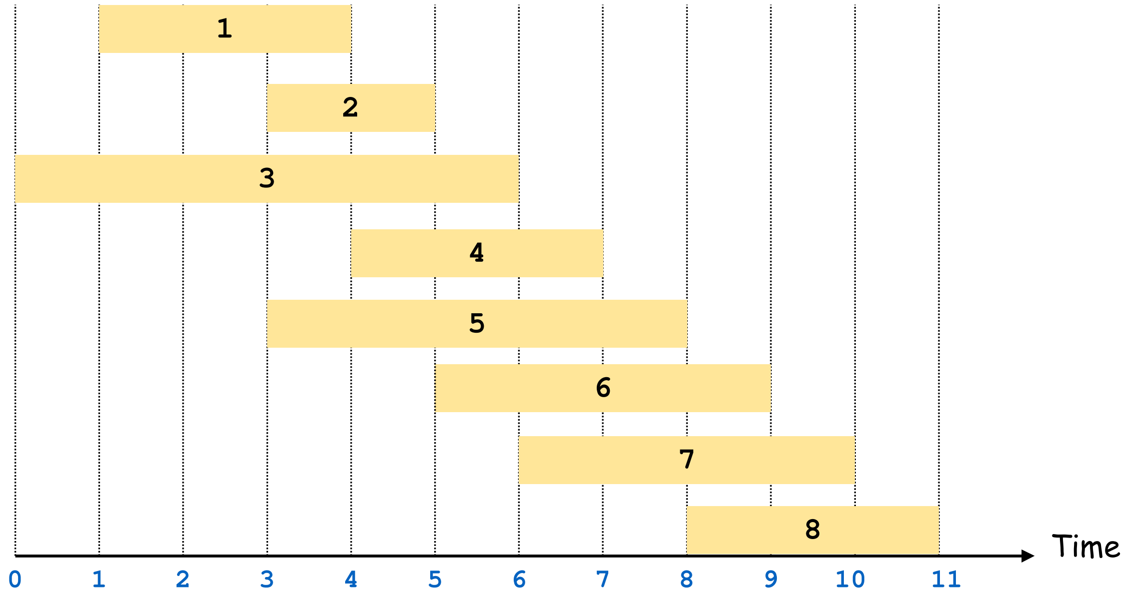
$O(n)$

Weighted Interval Scheduling: Iterative Solution

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

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

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



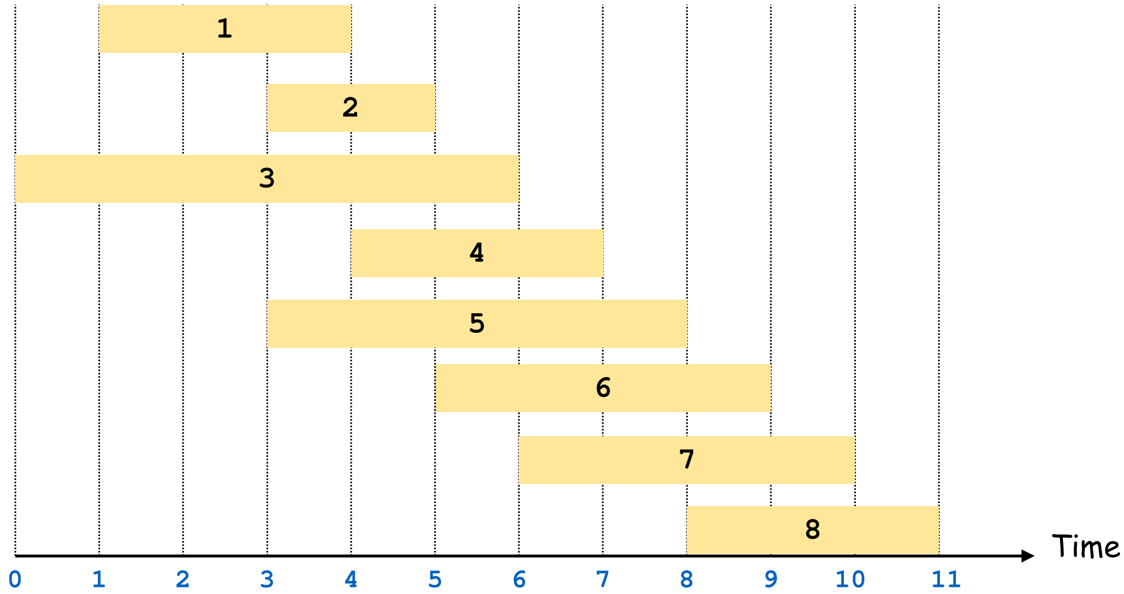
j	v_j	$p(j)$	OPT[j]
0	-	-	0
1	3	0	
2	2	0	
3	6	0	
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

Weighted Interval Scheduling: Iterative Solution

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

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

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



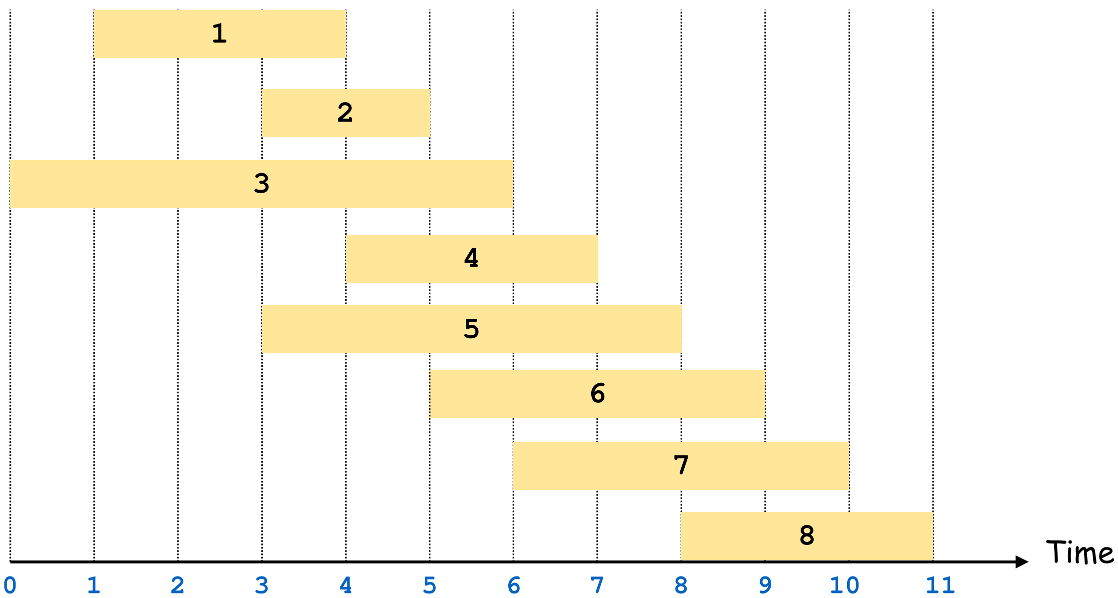
j	v_j	$p(j)$	OPT$[j]$
0	-	-	0
1	3	0	3
2	2	0	
3	6	0	
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

Weighted Interval Scheduling: Iterative Solution

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

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

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



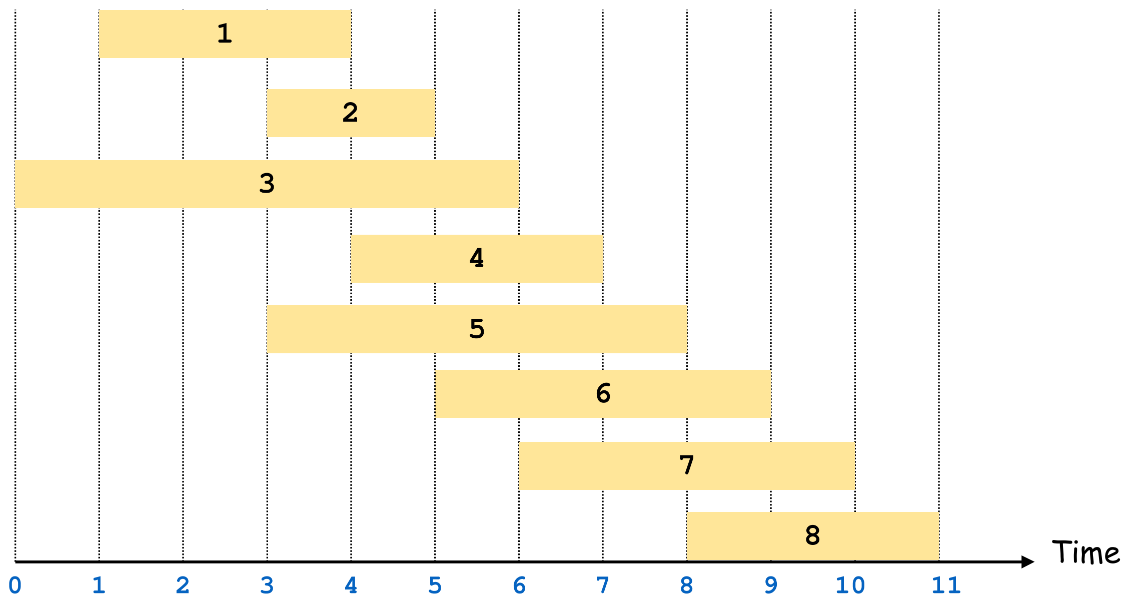
j	v_j	$p(j)$	OPT$[j]$
0	-	-	0
1	3	0	3
2	2	0	
3	6	0	
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

Weighted Interval Scheduling: Iterative Solution

Notation: Label jobs by finishing time: $f_1 \le f_2 \le \cdots \le f_n$.

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

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



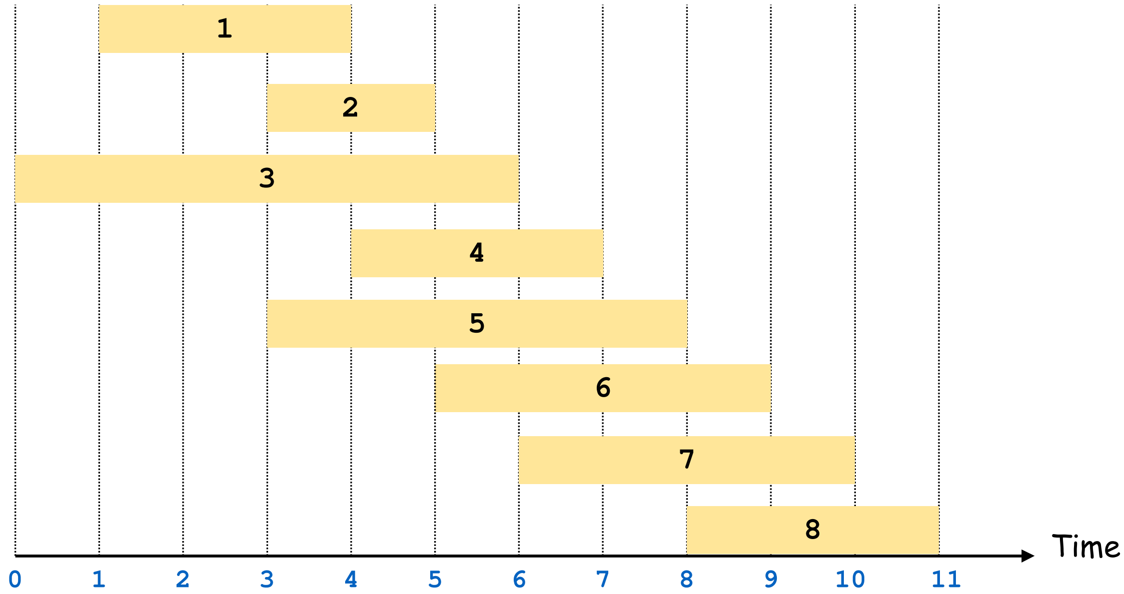
j	v_j	$p(j)$	OPT$[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

Weighted Interval Scheduling: Iterative Solution

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

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

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



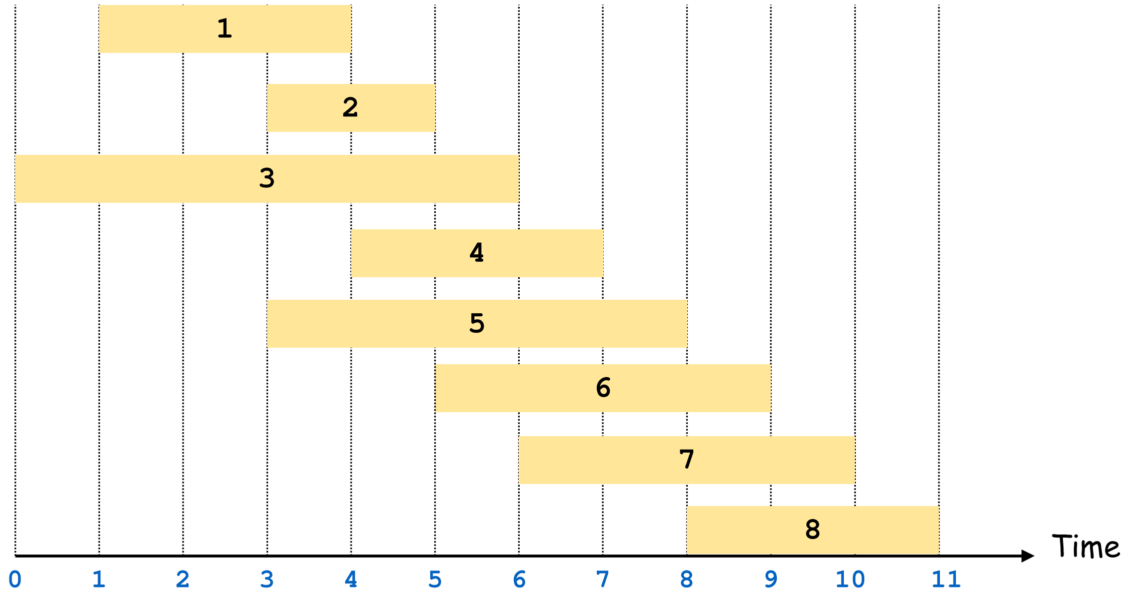
j	v_j	$p(j)$	OPT[j]
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

Weighted Interval Scheduling: Iterative Solution

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

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

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



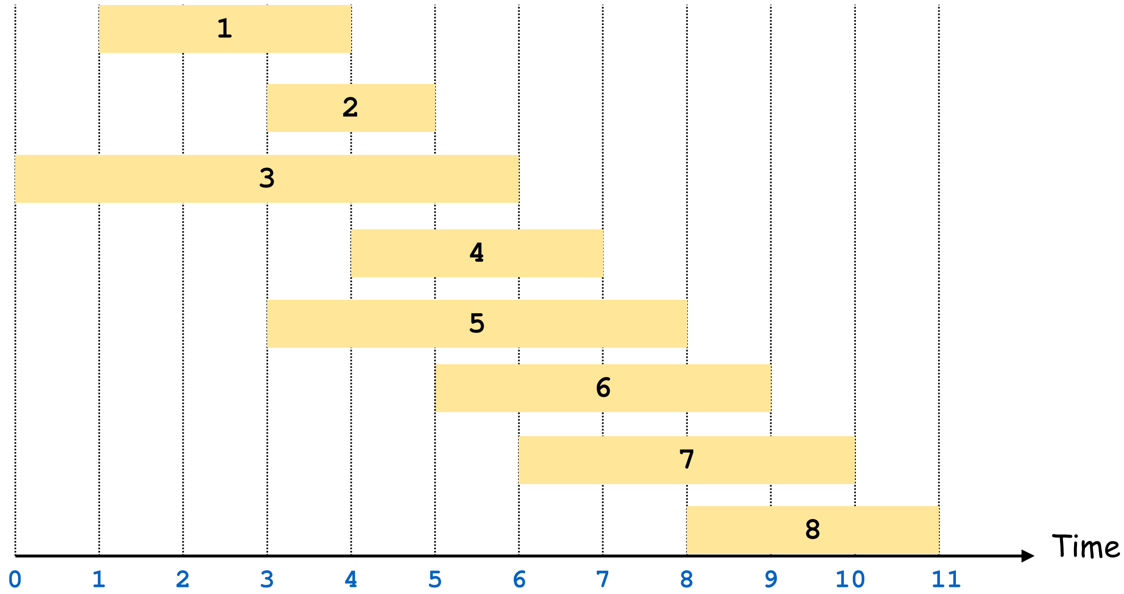
j	v_j	$p(j)$	OPT[j]
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

Weighted Interval Scheduling: Iterative Solution

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

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

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



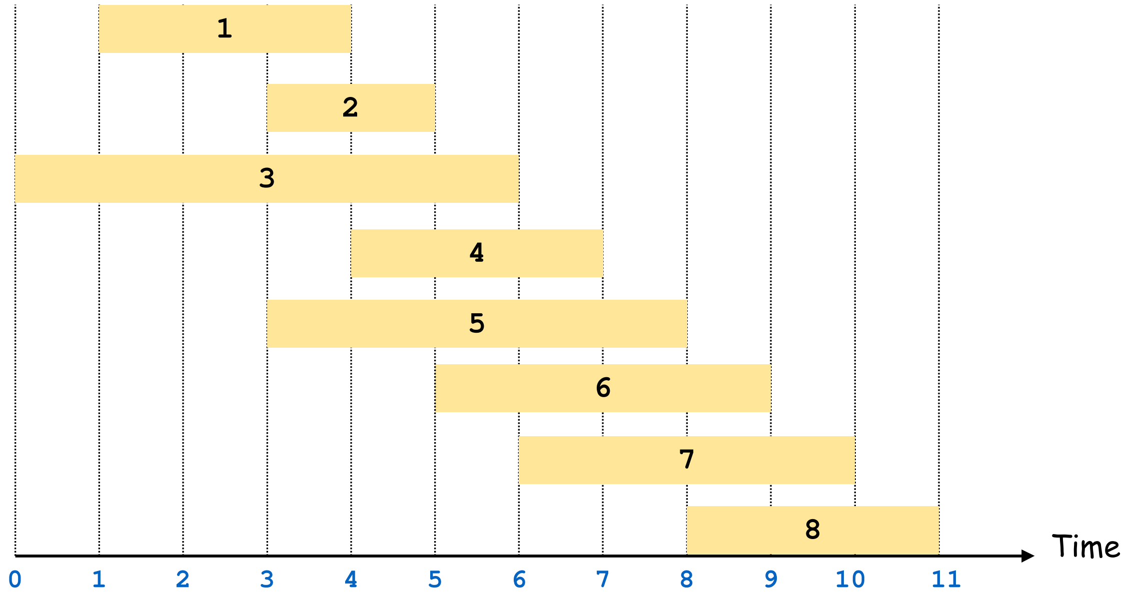
j	v_j	$p(j)$	OPT$[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	
5	5	0	
6	4	2	
7	4	3	
8	3	5	

Weighted Interval Scheduling: Iterative Solution

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

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

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



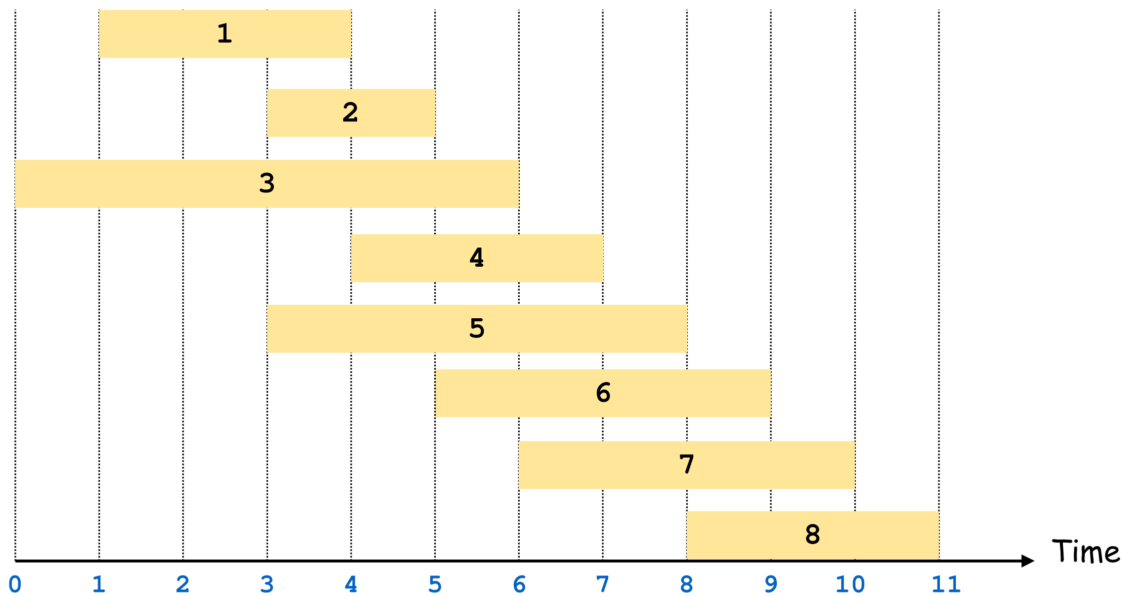
j	v_j	$p(j)$	OPT$[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	
6	4	2	
7	4	3	
8	3	5	

Weighted Interval Scheduling: Iterative Solution

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

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

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



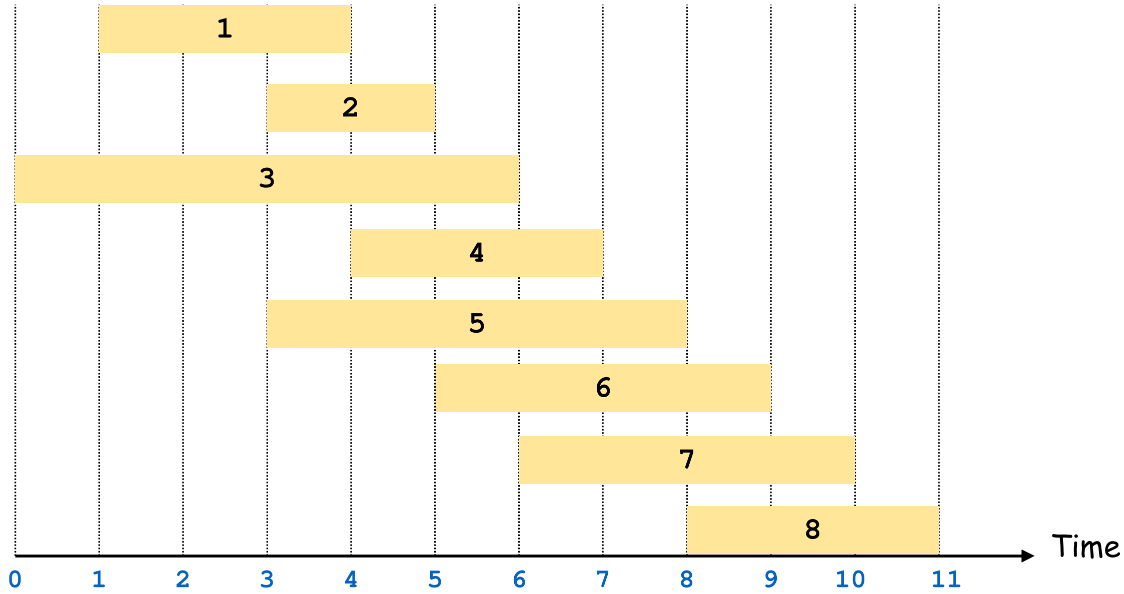
j	v_j	$p(j)$	OPT[j]
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	5
6	4	2	
7	4	3	
8	3	5	

Weighted Interval Scheduling: Iterative Solution

Notation: Label jobs by finishing time: $f_1 \le f_2 \le \cdots \le f_n$.

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

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



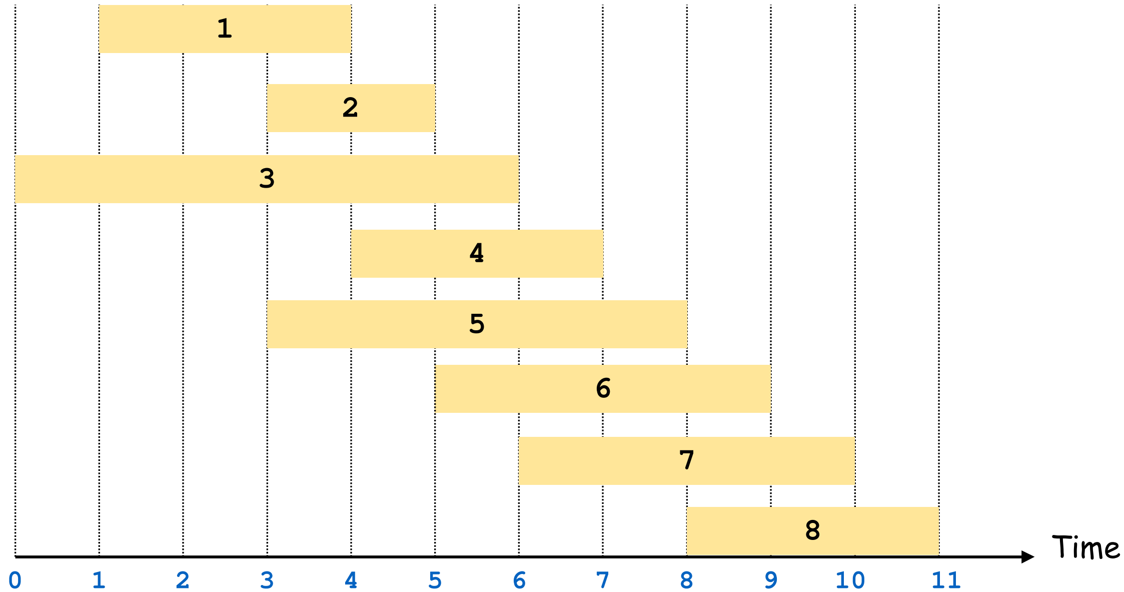
j	v_j	$p(j)$	OPT[j]
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	6
6	4	2	
7	4	3	
8	3	5	

Weighted Interval Scheduling: Iterative Solution

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

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

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



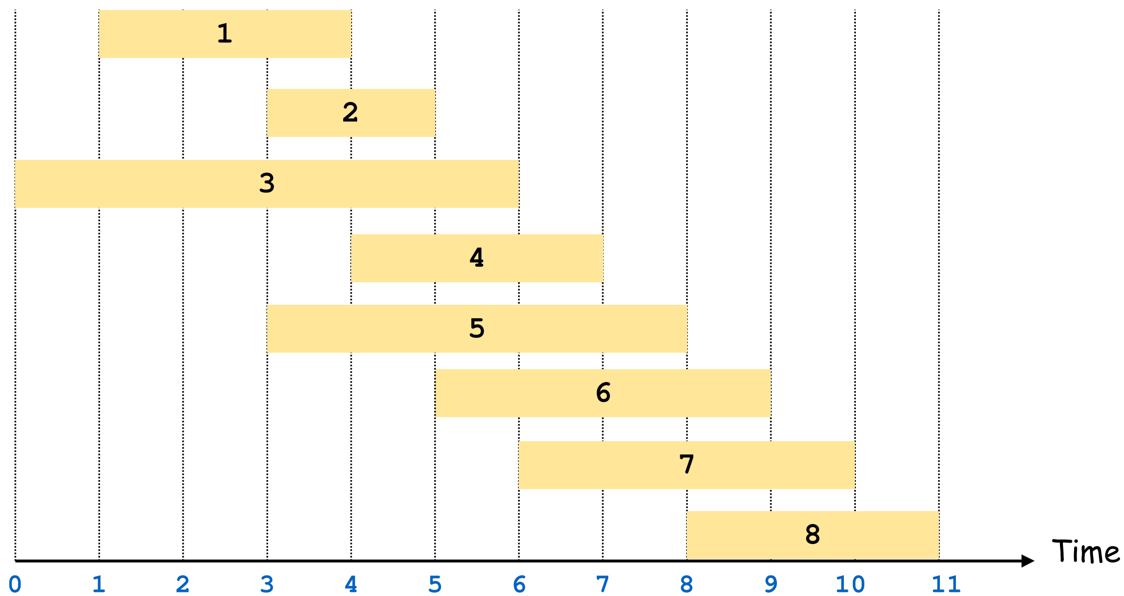
j	v_j	$p(j)$	OPT$[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	6
6	4	2	7
7	4	3	
8	3	5	

Weighted Interval Scheduling: Iterative Solution

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

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

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



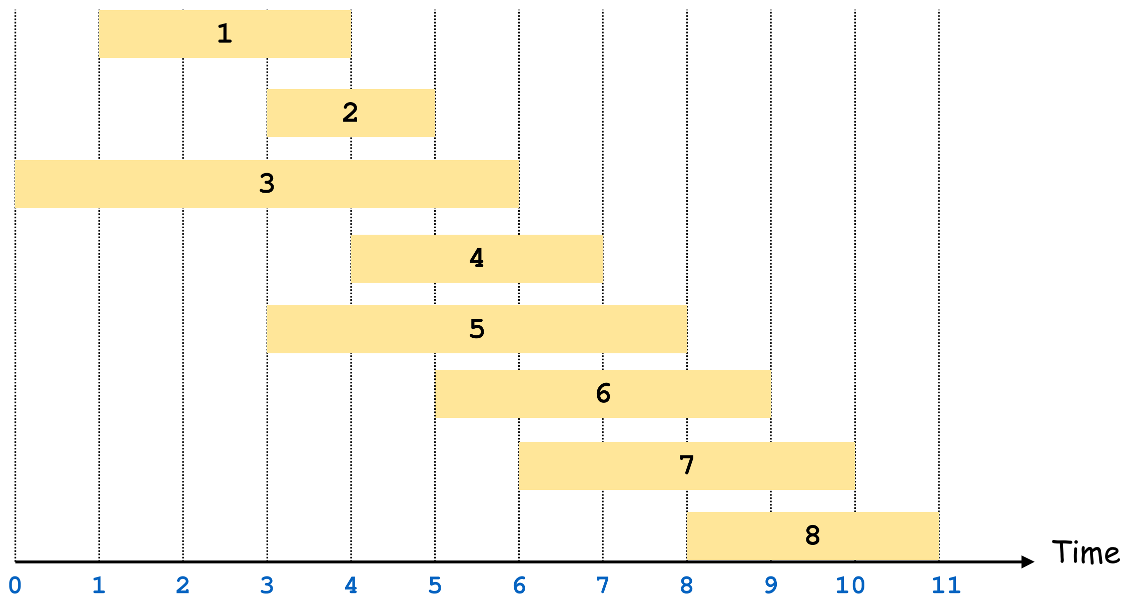
j	v_j	$p(j)$	OPT[j]
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	6
6	4	2	7
7	4	3	
8	3	5	

Weighted Interval Scheduling: Iterative Solution

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

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

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



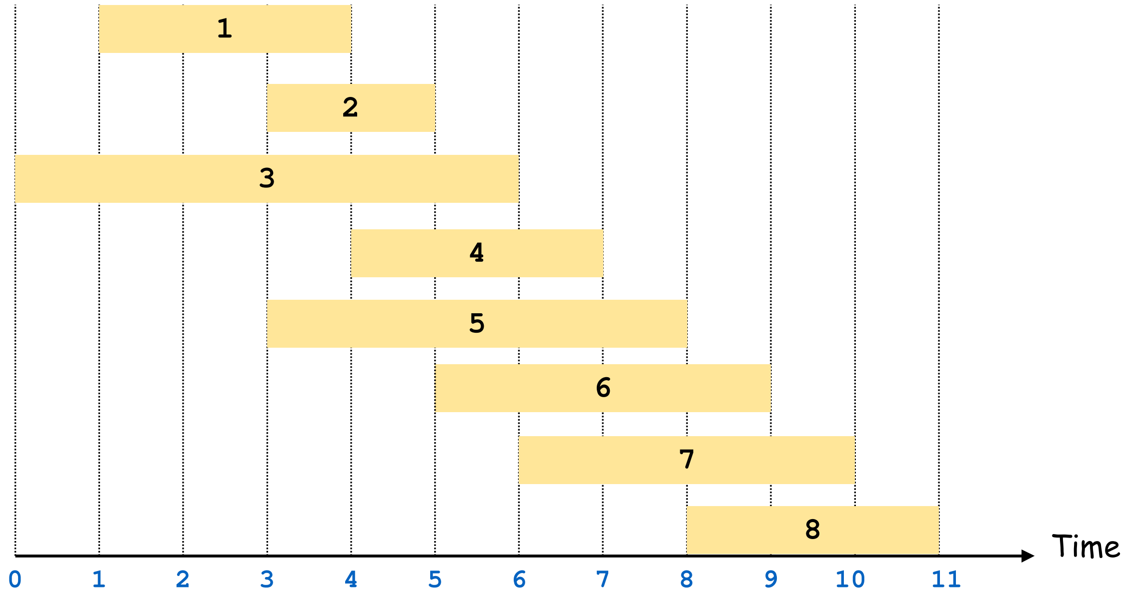
j	v_j	$p(j)$	OPT[j]
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	6
6	4	2	7
7	4	3	10
8	3	5	

Weighted Interval Scheduling: Iterative Solution

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

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

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



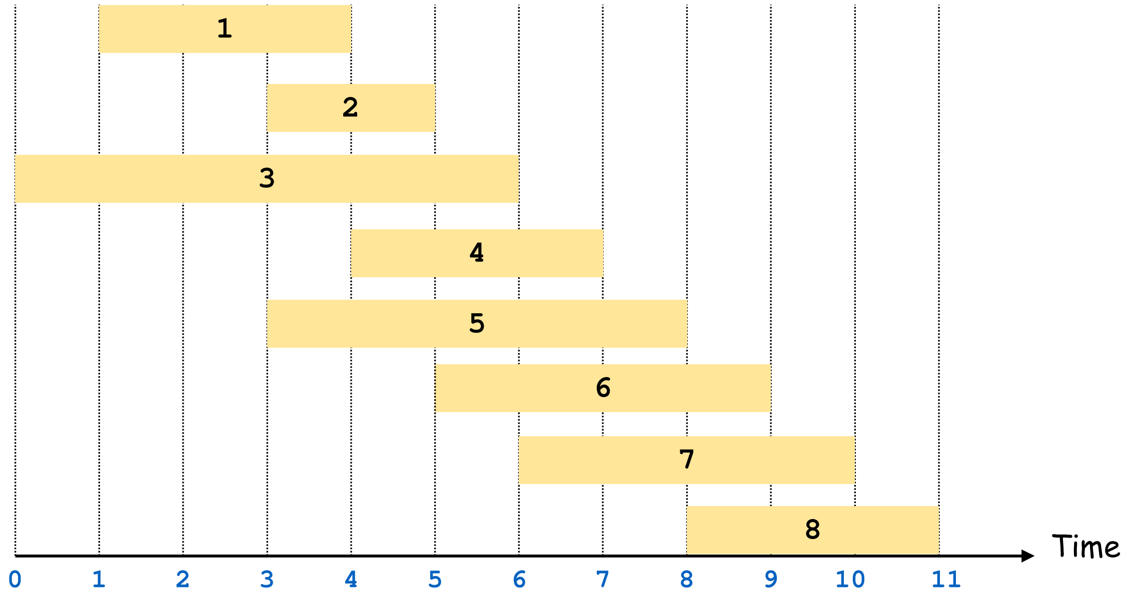
j	v_j	$p(j)$	OPT$[j]$
0	-	-	0
1	3	0	3
2	2	0	3
3	6	0	6
4	3	1	6
5	5	0	6
6	4	2	7
7	4	3	10
8	3	5	

Weighted Interval Scheduling: Iterative Solution

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

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	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 **OPT**(n) but we probably want to know what that solution **OPT** actually is!

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

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

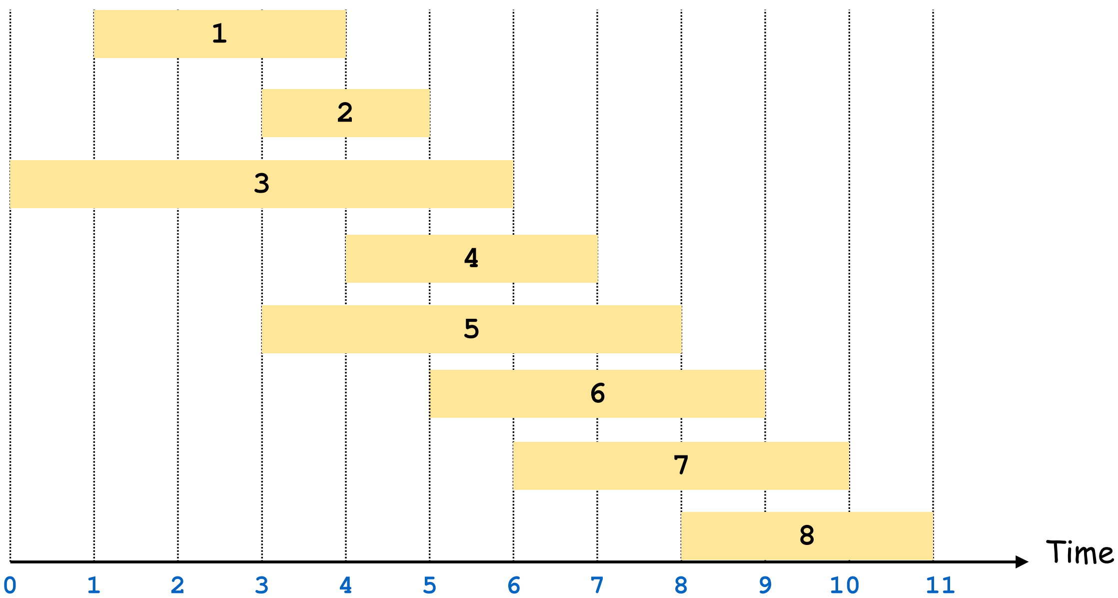
This gives a "pointer" that leads the way along a path to the optimal solution…

Weighted Interval Scheduling: Iterative Solution

Notation: Label jobs by finishing time: $f_1 \le f_2 \le \cdots \le f_n$.

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

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



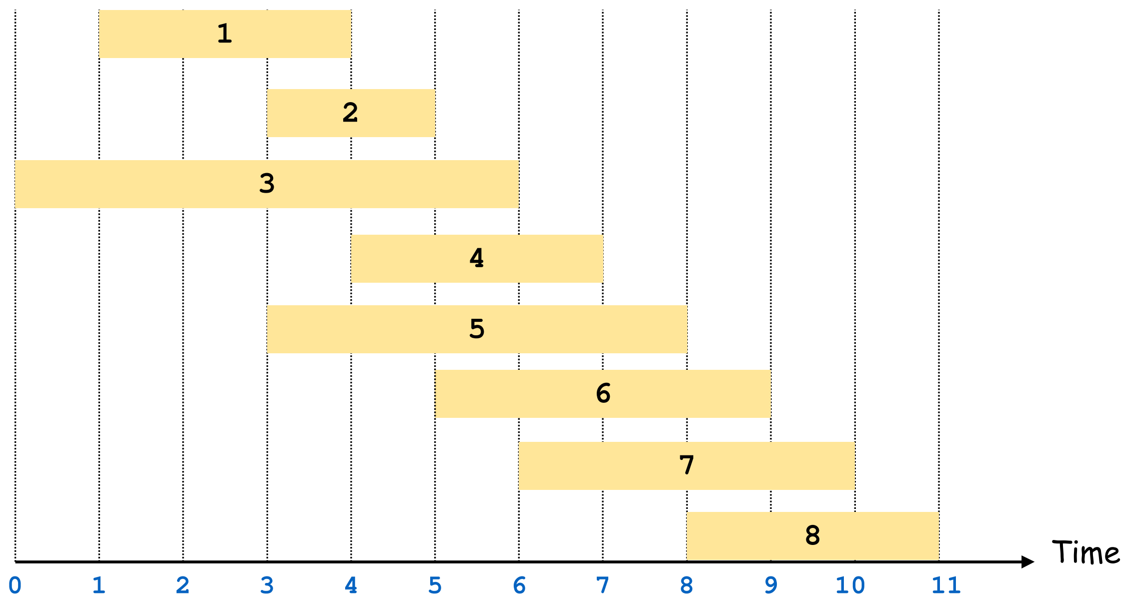
j	v_j	$p(j)$	OPT[j]	Used[j]
0	-	-	0	-
1	3	0	3	1
2	2	0	3	0
3	6	0	6	1
4	3	1	6	1
5	5	0	6	0
6	4	2	7	1
7	4	3	10	1
8	3	5	10	0

Weighted Interval Scheduling: Iterative Solution

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

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	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: Finding the Solution

```
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-Compute-Opt {
   OPT[0] = 0
   for j = 1 to n
      if vj + OPT[p(j)] > OPT[j-1] {
           OPT[j] = vj + OPT[p(j)]
           Used[j] = 1
      } else {
           OPT[j] = OPT[j-1]
           Used[j] = 0
      }
}
```

```
Find-Opt {
   j = n
   OPTSol = ∅
   while j > 0
      if Used[j] == 0 {
         j = j-1
      } else {
         OPTSol = OPTSol ∪{j}
         j = p(j)
      }
}
```

Three Steps to Dynamic Programming

1. Formulate the answer as a recurrence relation or recursive algorithm

2. Figure out the possible values of parameters in the recursive calls.
   - This should be "small", i.e., bounded by a low-degree polynomial
   - Can use memoization to store a cache of previously computing values

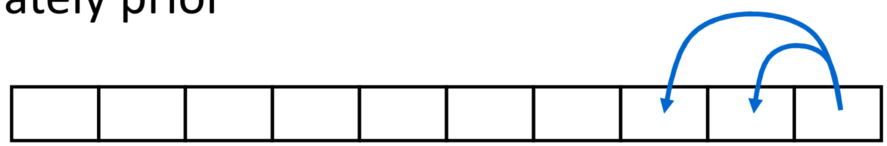
3. Specify an order of evaluation for the recurrence so that you already have the partial results stored in memory when you need them.
   - Produces iterative code

Once you have an iterative DP solution: see if you can save space...

Dynamic Programming Patterns

Fibonacci pattern:

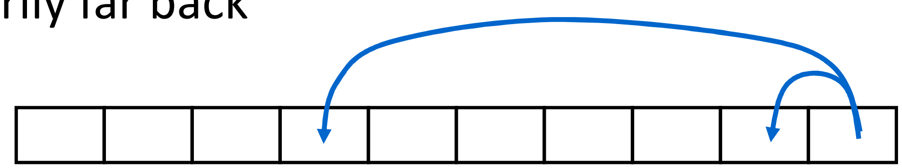
- 1-dimensional, $O(1)$ values immediately prior



- Space saving possible

Weighted interval scheduling pattern:

- 1-dimensional, $O(1)$ values arbitrarily far back



- No space saving possible