# CSE 421

## Dynamic Programming

Shayan Oveis Gharan

# Strengthening Induction Hypothesis

We have seen examples on how to design algorithms by induction

In some cases it may help to strengthen the IH.
High-level plan: Prove $P(n) \wedge Q(n)$ inductively.

IH: Assume $P(n-1) \wedge Q(n-1)$.

IS: You may use $Q(n-1)$ to help you to prove $P(n)$
Remember you also have to prove $Q(n)$.

# Maximum Consecutive Subsequence

Problem: Given a sequence $x_1, \ldots, x_n$ of integers (not necessarily positive),

Goal: Find a subsequence of consecutive elements s.t., the sum of its numbers is maximum.

1 -3 [7 -2 -3 8] -10 1 -7

Applications: Figuring out the highest interest rate period in stock market

# Second Attempt (Strengthing Ind Hyp)

Stronger Ind Hypothesis: Given $x_1, \ldots, x_{n-1}$ we can compute the maximum-sum subsequence, and the maximum-sum suffix subsequence.

Can be empty

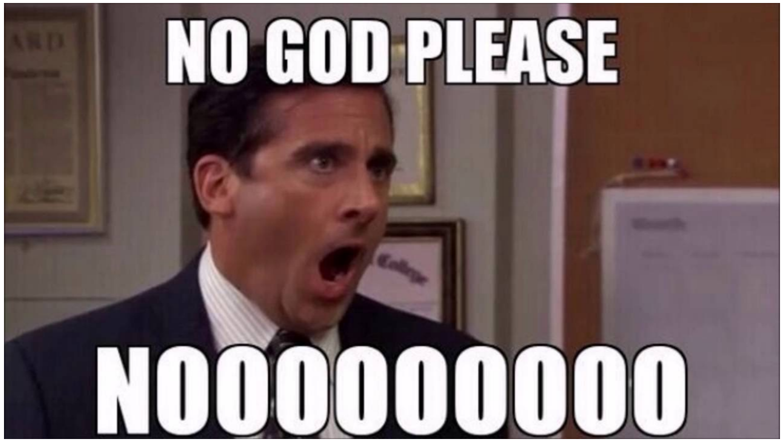
-3, [6, -1, 2,] -8, [6, -2]

$x_i$ $x_j$ $x_k$ $x_{n-1}$

Say $\boldsymbol{x_i, \ldots, x_j}$ is the maximum-sum and $x_k, \ldots, x_{n-1}$ is the maximum-sum suffix subsequences.

- If $x_k + \cdots + x_{n-1} + x_n > x_i + \cdots + x_j$ then $x_k, \ldots, x_n$ will be the new maximum-sum subsequence

# Are we done?



# Updating Max Suffix Subsequence

-3, 6, -1, 2, -8, 6, -2, | 4

$x_n$

Say $x_k, \ldots, x_{n-1}$ is the maximum-sum suffix subsequences of $x_1, \ldots, x_{n-1}$.

- If $x_k + \cdots + x_n \geq 0$ then,
  $x_k, \ldots, x_n$ is the new maximum-sum suffix subsequence

- Otherwise,
  The new maximum-sum suffix is the empty string.

# Maximum Sum Subsequence ALG

```
Initialize S=0 (Sum of numbers in Maximum Subseq)
Initialize U=0 (Sum of numbers in Maximum Suffix)
for  (i=1 to n) {
    if (x[i] + U > S)
        S = x[i] + U

    if (x[i] + U > 0)
        U = x[i] + U
    else
        U = 0
}
Output S.
```

-3 6 -1 2 -8 6 -2 4

# Pf of Correct: Maximum Sum Subseq

Ind Hypo: Suppose

- $x_i, \ldots, x_j$ is the max-sum-subseq of $x_1, \ldots, x_{n-1}$
- $x_k, \ldots, x_{n-1}$ is the max-suffix-sum-sub of $x_1, \ldots, x_{n-1}$

Ind Step: Suppose $x_a, \ldots, x_b$ is the max-sum-subseq of $x_1, \ldots, x_n$

Case 1 ($b < n$): $x_a, \ldots, x_b$ is also the max-sum-subseq of $x_1, \ldots, x_{n-1}$
So, by IH $a = i, b = j$ and the algorithm correctly outputs OPT

Case 2 ($b = n$): We must have $x_a, \ldots, x_{b-1}$ is the max-suff-sum of $x_1, \ldots, x_{n-1}$.
If not, then by IH

$$x_k + \cdots x_{n-1} > x_a + \cdots + x_{n-1}$$

So, $x_k + \cdots + x_n > x_a + \cdots + x_b$ which is a contradiction.
Therefore, $a = k$ and the algorithm correctly outputs OPT

Special Cases (You don't need to mention if follows from above):

- The max-suffix-sum is empty string
- There are multiple maximum sum subsequences.

# Pf of Correct: Max-Sum Suff Subseq

Ind Hypo: Suppose

- $x_i, \dots, x_j$ is the max-sum-subseq of $x_1, \dots, x_{n-1}$
- $x_k, \dots, x_{n-1}$ is the max-suffix-sum-sub of $x_1, \dots, x_{n-1}$

Ind Step: Suppose $x_a, \dots, x_n$ is the max-suffix-sum-subseq of $x_1, \dots, x_n$
Note that we may also have an empty sequence

Case 1 (OPT is empty): Then, we must have $x_k + \cdots + x_n < 0$. So the algorithm correctly finds max-suffix-sum subsequence.

Case 2 ($x_a, \dots, x_n$ is nonempty): We must have $x_a + \cdots + x_n \geq 0$.
Also, $x_a, \dots, x_{n-1}$ must be the max-suffix-sum of $x_1, \dots, x_{n-1}$. If not, by IH

$$x_a + \cdots + x_{n-1} < x_k + \cdots + x_{n-1}$$

which implies $x_a + \cdots + x_n < x_k + \cdots + x_n$ which is a contradiction.

Therefore, $a = k$. So, the algorithm correctly finds max-suffix-sum subsequence.

# Summary

- Before designing an algorithm study properties of optimum solution
- If ordinary induction fails, you may need to strengthen the induction hypothesis

# Dynamic Programming

# Algorithmic Paradigm

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

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

Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems. Memorize the answers to obtain polynomial time ALG.

# Dynamic Programming History

Bellman. Pioneered the systematic study of dynamic programming in the 1950s.

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"

# Dynamic Programming Applications

## Areas:

- Bioinformatics
- Control Theory
- Information Theory
- Operations Research
- Computer Science: Theory, Graphics, AI, …

## Some famous DP algorithms

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

# Dynamic Programming

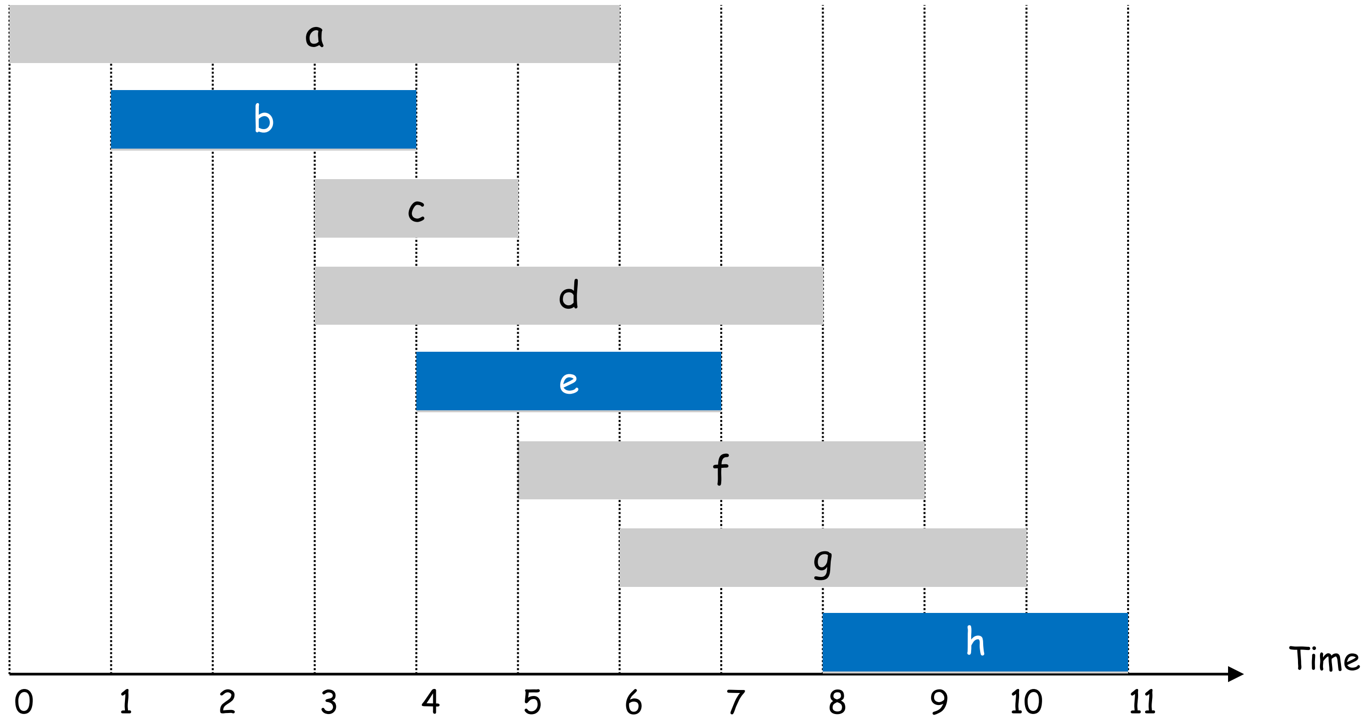
Dynamic programming is nothing but algorithm design by induction!

We just ”remember” the subproblems that we have solved so far to avoid re-solving the same sub-problem many times.

# 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.

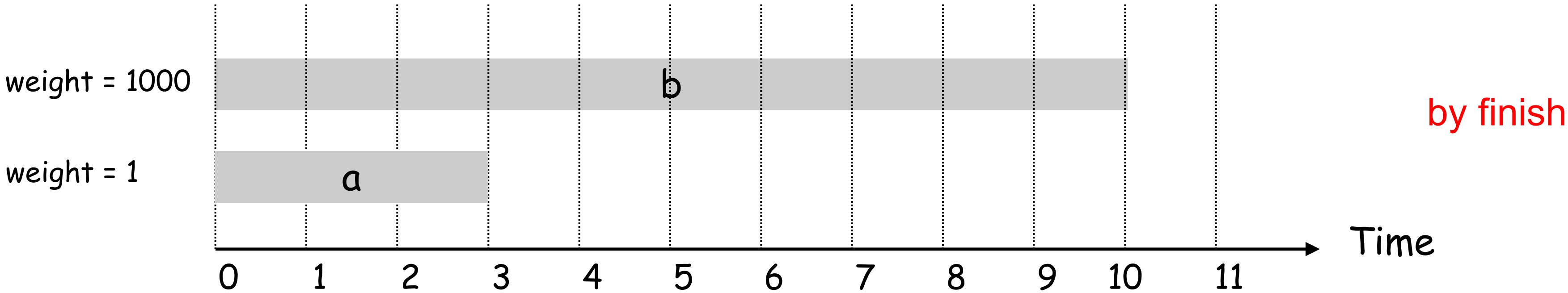


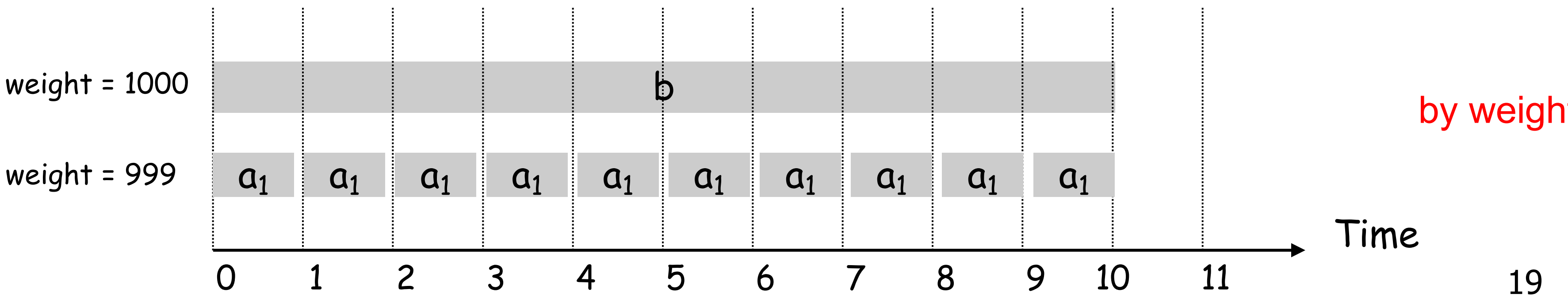
# Unweighted Interval Scheduling: Review

Recall: Greedy algorithm works if all weights are 1:

- Consider jobs in ascending order of finishing time
- Add job to a subset if it is compatible with prev added jobs.

OBS: Greedy ALG fails spectacularly (no approximation ratio) if arbitrary weights are allowed:



# Weighted Job Scheduling by Induction

Suppose $1, \dots, n$ are all jobs. Let us use induction:

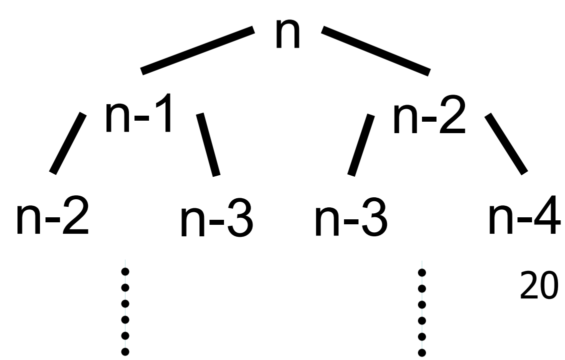
IH (strong ind): Suppose we can compute the optimum job scheduling for $< n$ jobs.

IS: Goal: For any n jobs we can compute OPT.
Case 1: Job n is not in OPT.
-- Then, just return OPT of $1, \dots, n-1$.

Take best of the two

Case 2: Job n is in OPT.
-- Then, delete all jobs not compatible with n and recurse.

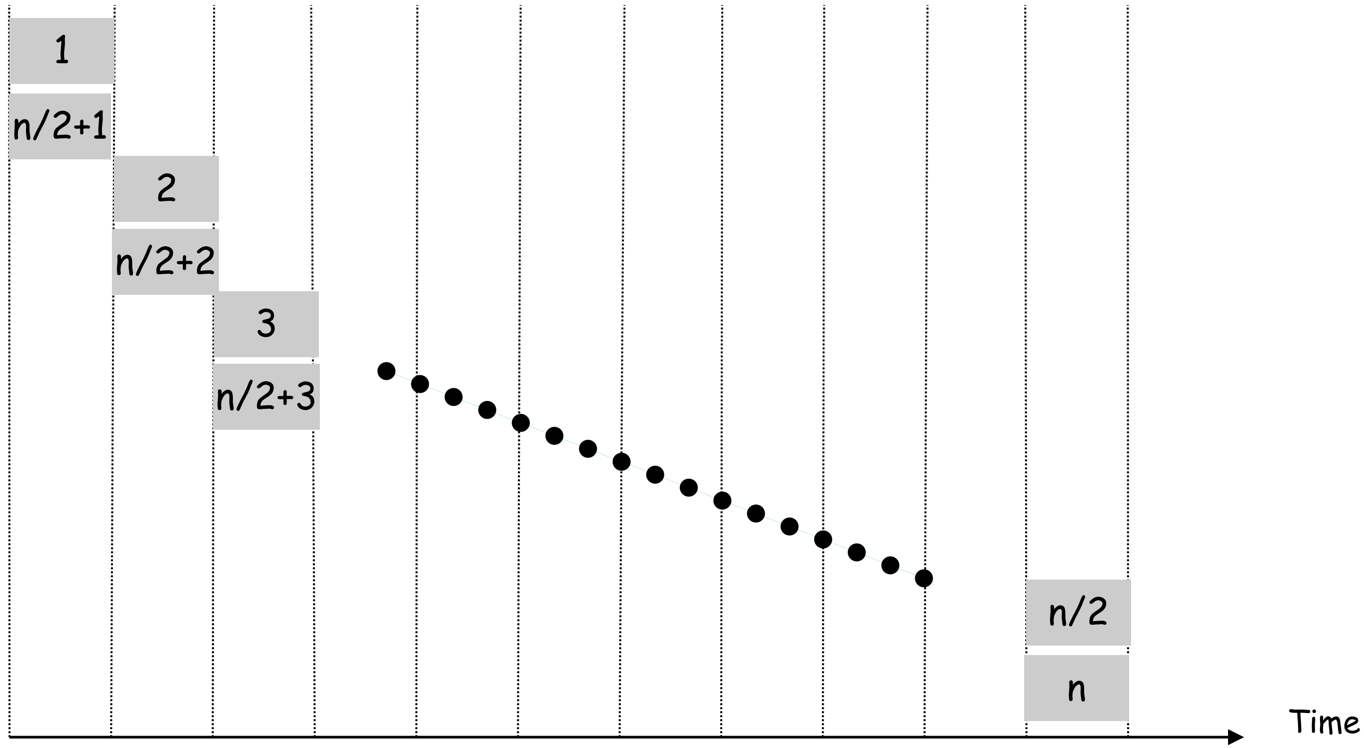
Q: Are we done?
A: No, How many subproblems are there?
Potentially $2^n$ all possible subsets of jobs.



# A Bad Example

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

How many subproblems do we get?



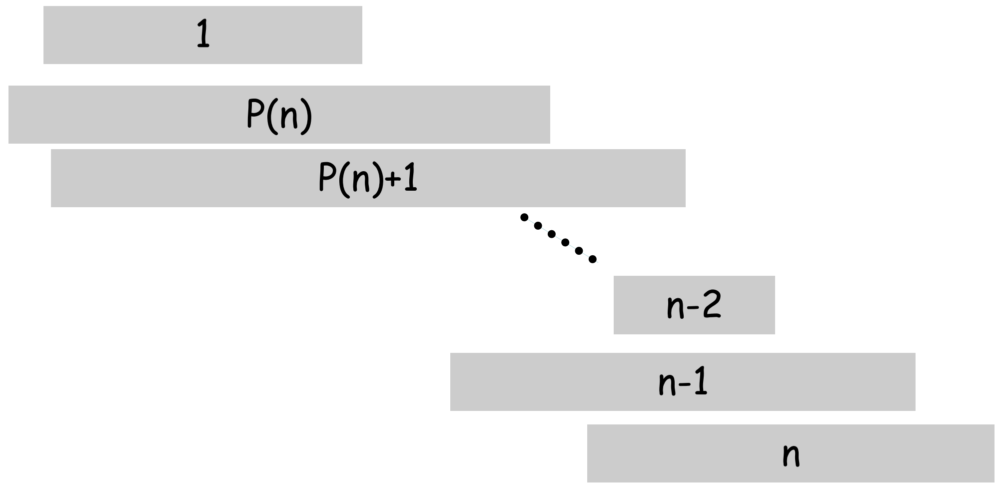
# Sorting to Reduce Subproblems

IS: For jobs 1,…,n we want to compute OPT

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

Case 1: Suppose OPT has job n.

- So, all jobs i that are not compatible with n are not OPT
- Let p(n) = largest index i < n such that job i is compatible with n.
- Then, we just need to find OPT of $1, \ldots, p(n)$



# Sorting to reduce Subproblems

IS: For jobs 1,…,n we want to compute OPT

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

Case 1: Suppose OPT has job n.

- So, all jobs i that are not compatible with n are not OPT
- Let p(n) = l… …atible with n.
- Then, …

This is how we differentiate from solving Maximum Independent Set Problem

Take best of the two

Case 2: OP…

- Then, OPT is just the optimum $1, \ldots, n-1$

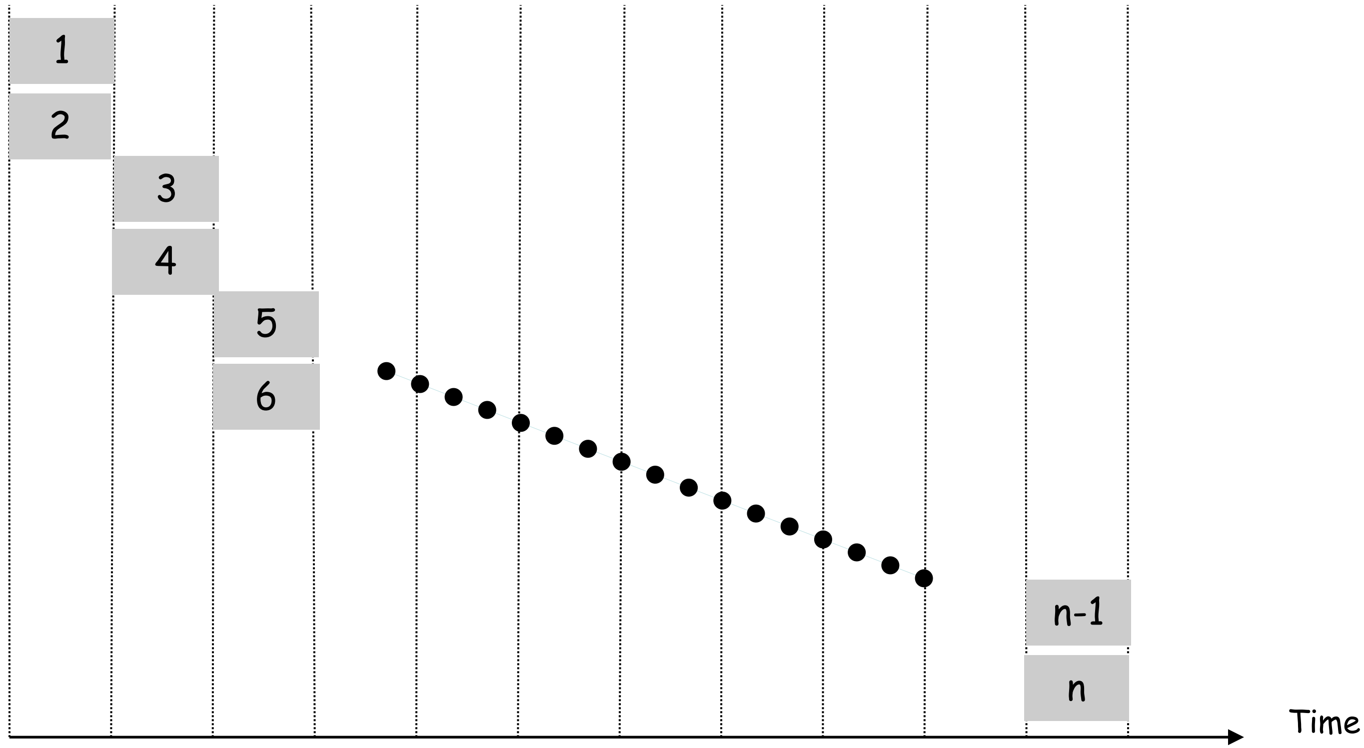
Q: Have we made any progress (still reducing to two subproblems)?

A: Yes! This time every subproblem is of the form $1, \ldots, i$ for some $i$

So, at most $n$ possible subproblems.

# Bad Example Review

How many subproblems do we get in this sorted order?



# Weighted Job Scheduling by Induction

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

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

This is the most important step in design DP algorithms

To solve OPT(j):

Case 1: OPT(j) has job j,

- So, all jobs i that are n
- Let p(j) = largest index i j.
- So $OPT(j) = OPT\big(p(j)\big) \cup \{j\}$.

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

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

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

# Algorithm

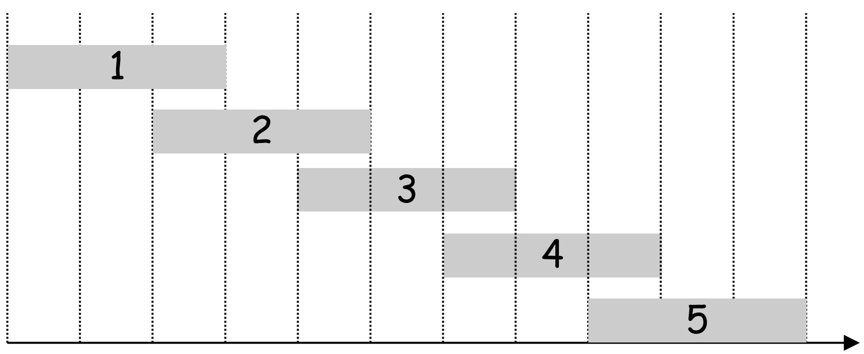
```
Input: n, s(1),…,s(n) and f(1),…,f(n) and w_1,…,w_n.

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(w_j + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

# Recursive Algorithm Fails

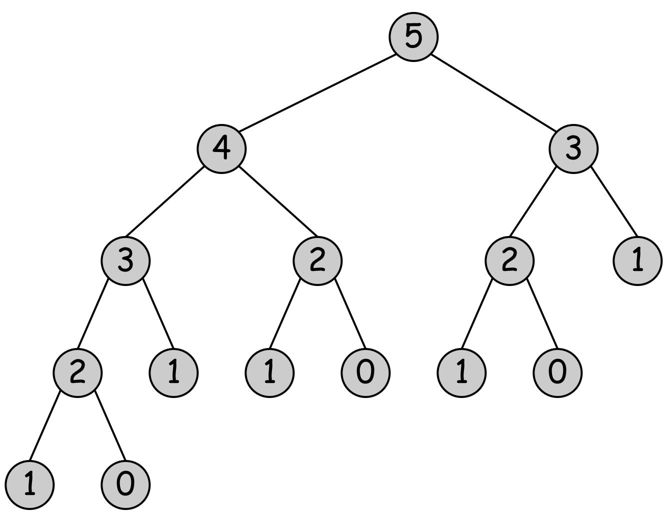
Even though we have only n subproblems, we do not store the solution to the subproblems

- 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




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



# 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),…,s(n)  and f(1),…,f(n) and w_1,…,w_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
M[0] = 0

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

# Bottom up Dynamic Programming

You can also avoid recusion
- recursion may be easier conceptually when you use induction

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

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

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

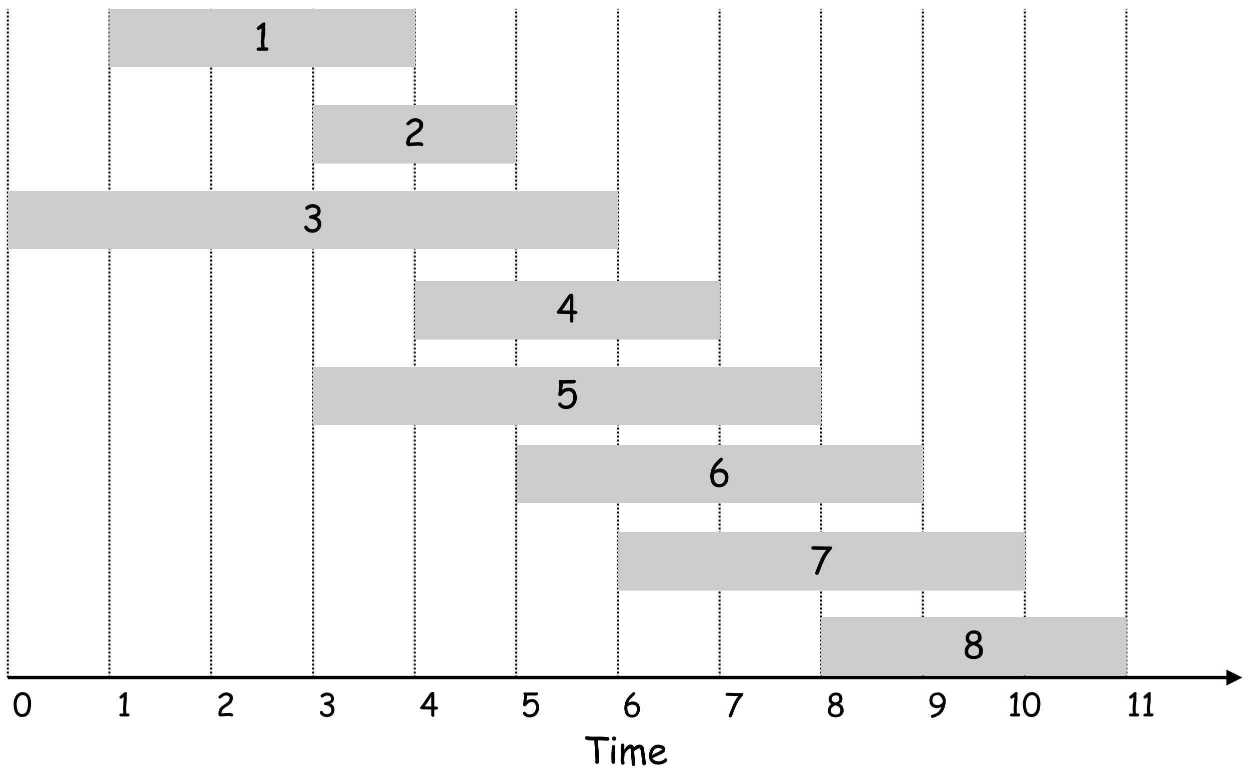
```
Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(w_j + M[p(j)], M[j-1])
}

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: $f(1) \leq \cdots \leq f(n)$.

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

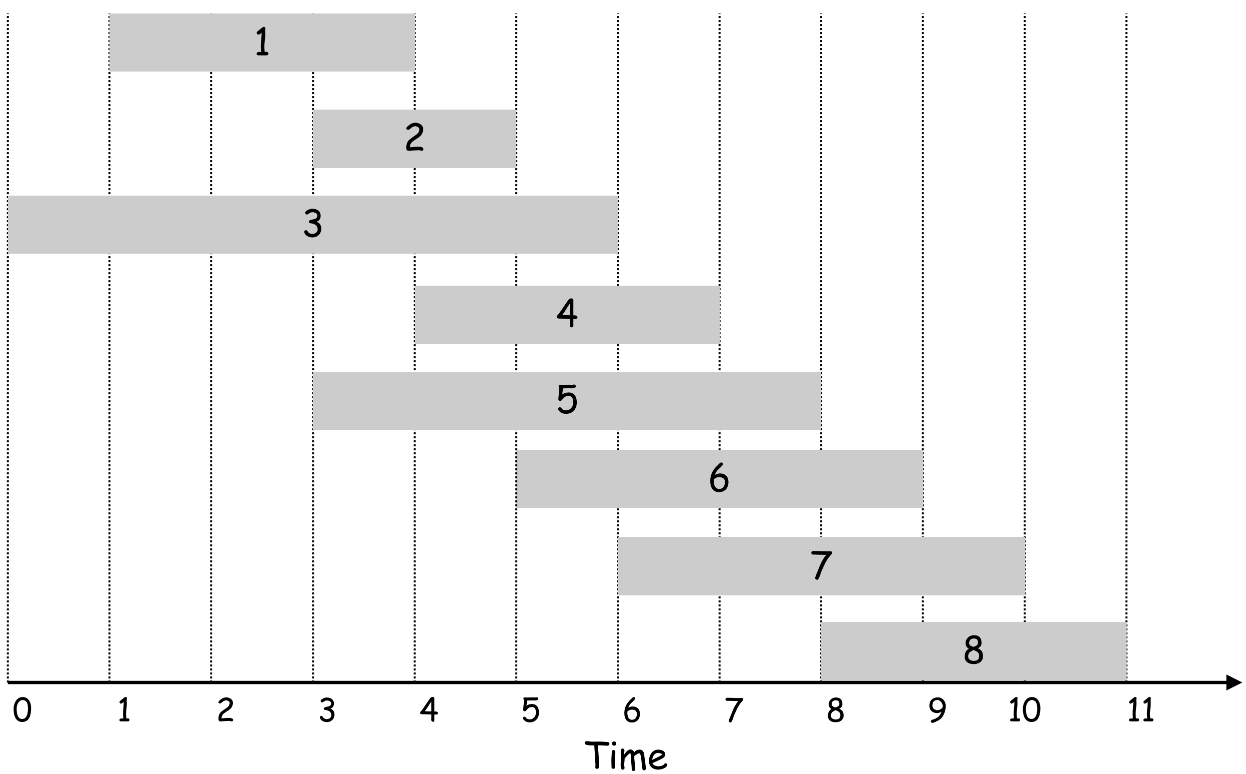



| j | $w_j$ | p(j) | OPT(j |
|---|---|---|---|
| 0 | | | 0 |
| 1 | 3 | 0 | |
| 2 | 4 | 0 | |
| 3 | 1 | 0 | |
| 4 | 3 | 1 | |
| 5 | 4 | 0 | |
| 6 | 3 | 2 | |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Example

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

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

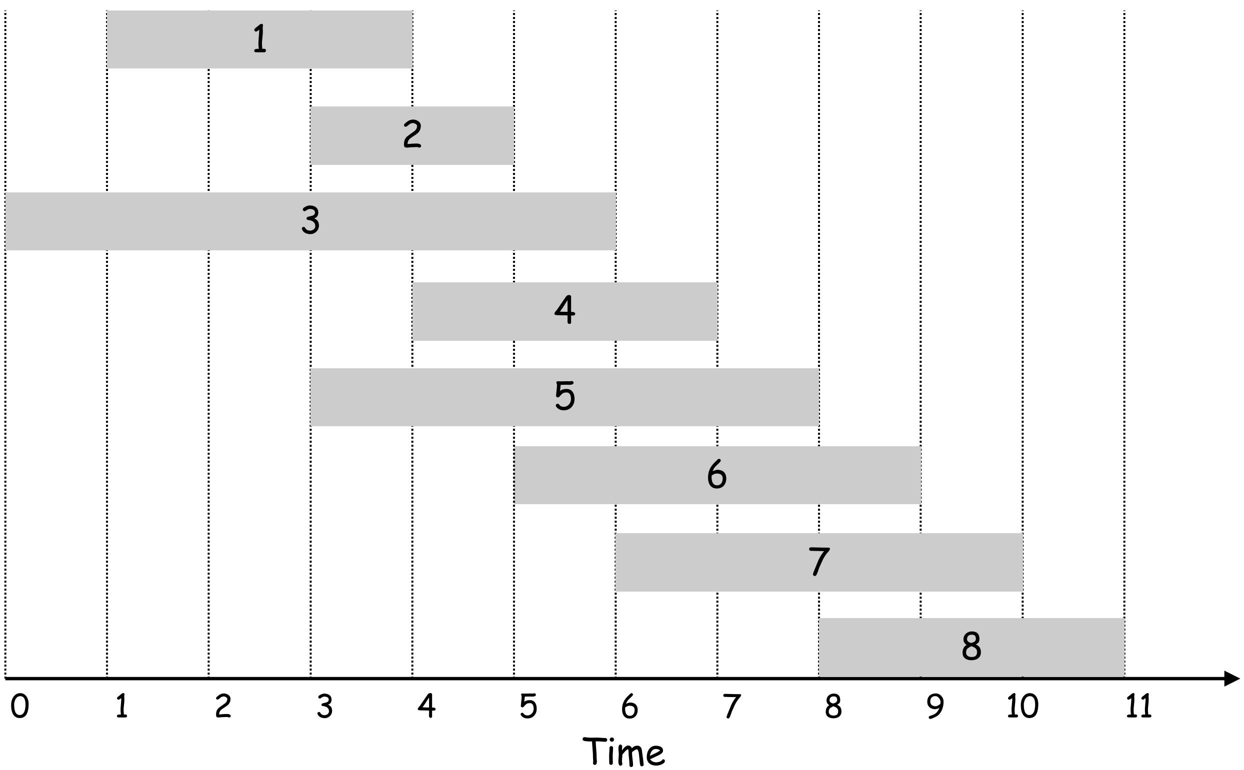



| j | $w_j$ | p(j) | OPT(j |
|---|---|---|---|
| 0 | | | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | |
| 3 | 1 | 0 | |
| 4 | 3 | 1 | |
| 5 | 4 | 0 | |
| 6 | 3 | 2 | |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Example

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

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

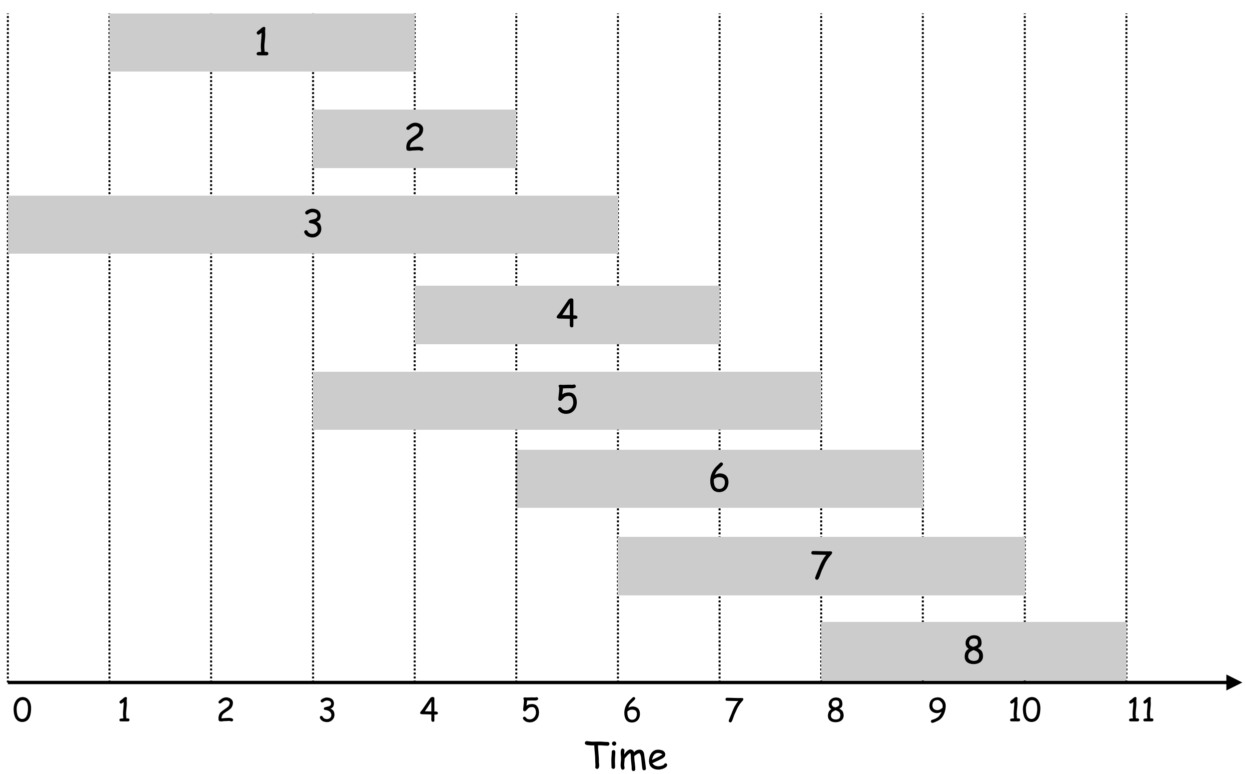



| j | $w_j$ | p(j) | OPT(j |
|---|---|---|---|
| 0 | | | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | **4** |
| 3 | 1 | 0 | |
| 4 | 3 | 1 | |
| 5 | 4 | 0 | |
| 6 | 3 | 2 | |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Example

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

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

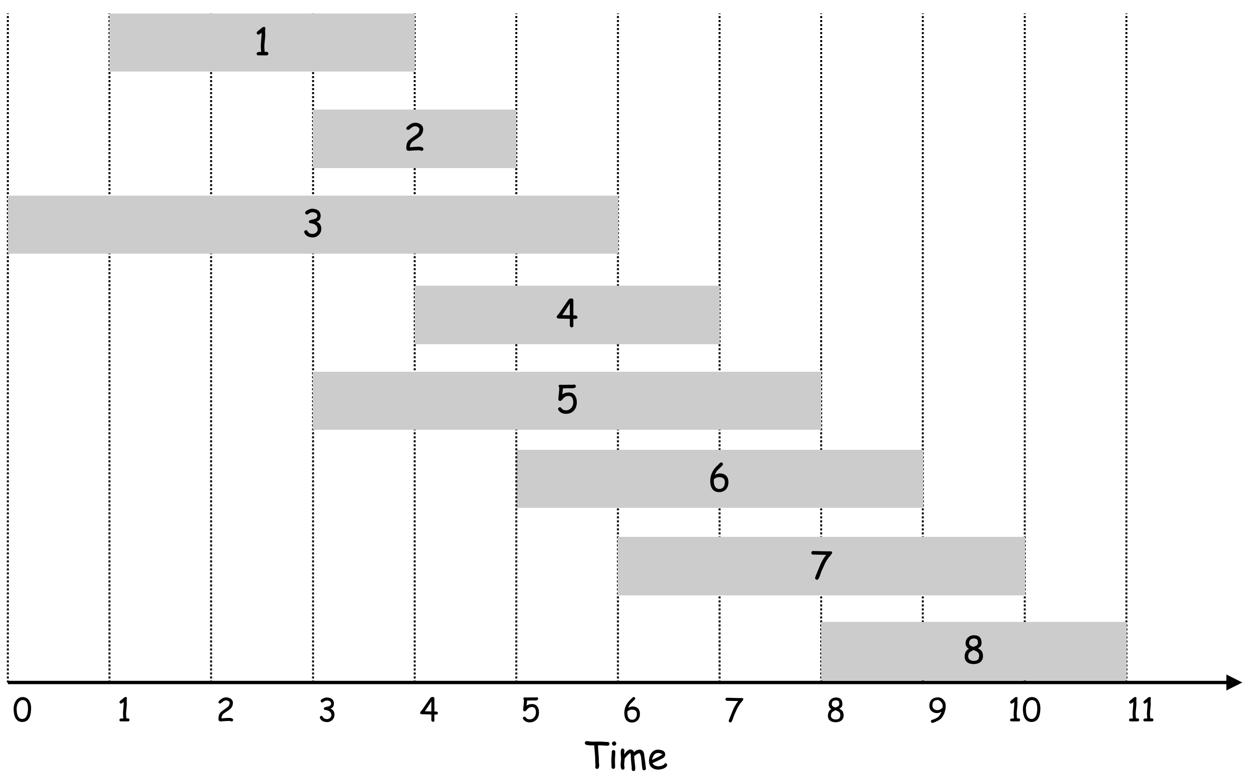



| j | $w_j$ | p(j) | OPT(j |
|---|---|---|---|
| 0 | | | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | 4 |
| 4 | 3 | 1 | |
| 5 | 4 | 0 | |
| 6 | 3 | 2 | |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Example

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

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

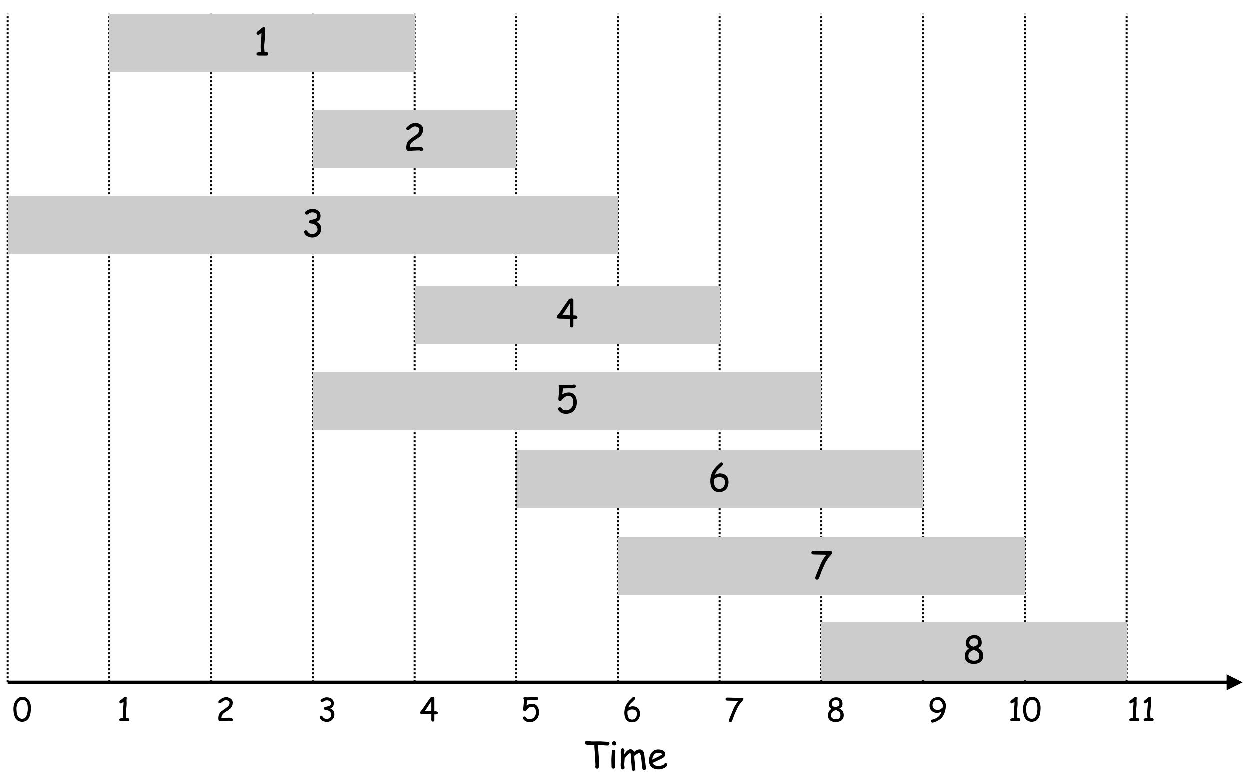



| j | $w_j$ | p(j) | OPT(j |
|---|---|---|---|
| 0 | | | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | 4 |
| 4 | 3 | 1 | 6 |
| 5 | 4 | 0 | |
| 6 | 3 | 2 | |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Example

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

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

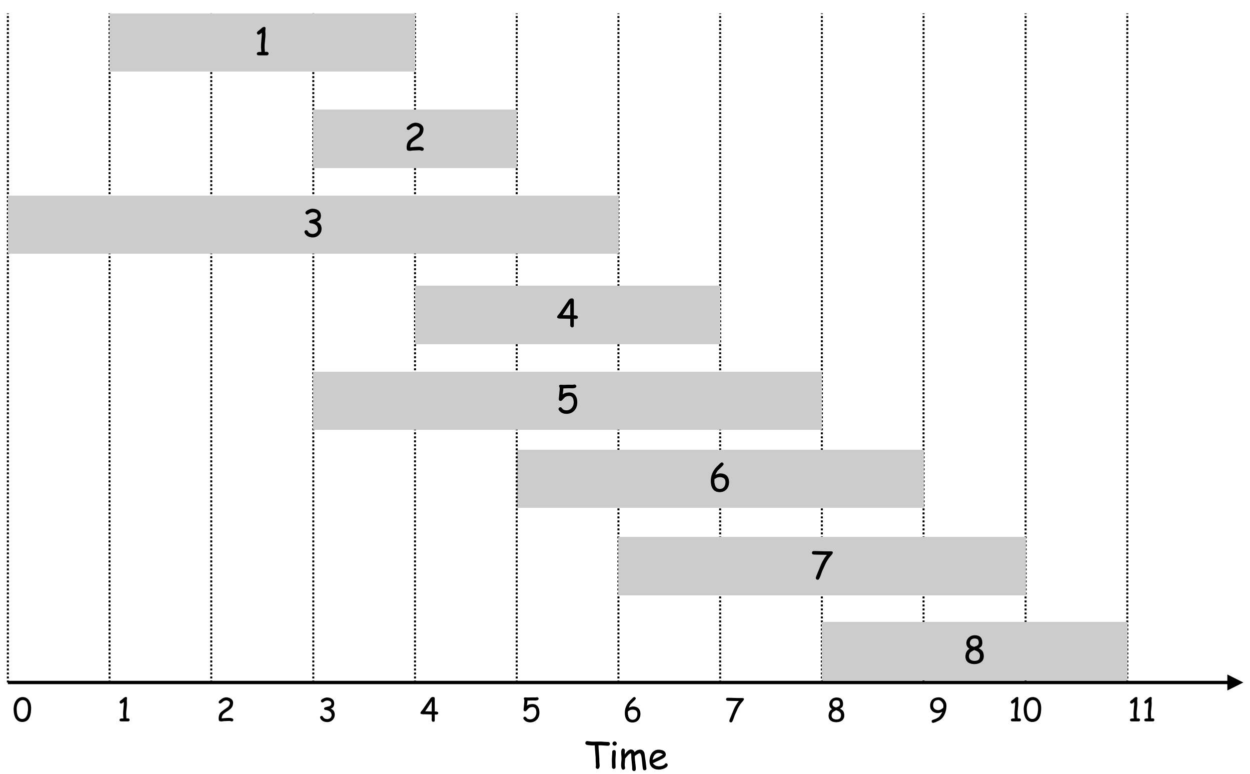



| j | $w_j$ | p(j) | OPT(j |
|---|---|---|---|
| 0 | | | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | 4 |
| 4 | 3 | 1 | 6 |
| 5 | 4 | 0 | 6 |
| 6 | 3 | 2 | |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Example

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

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

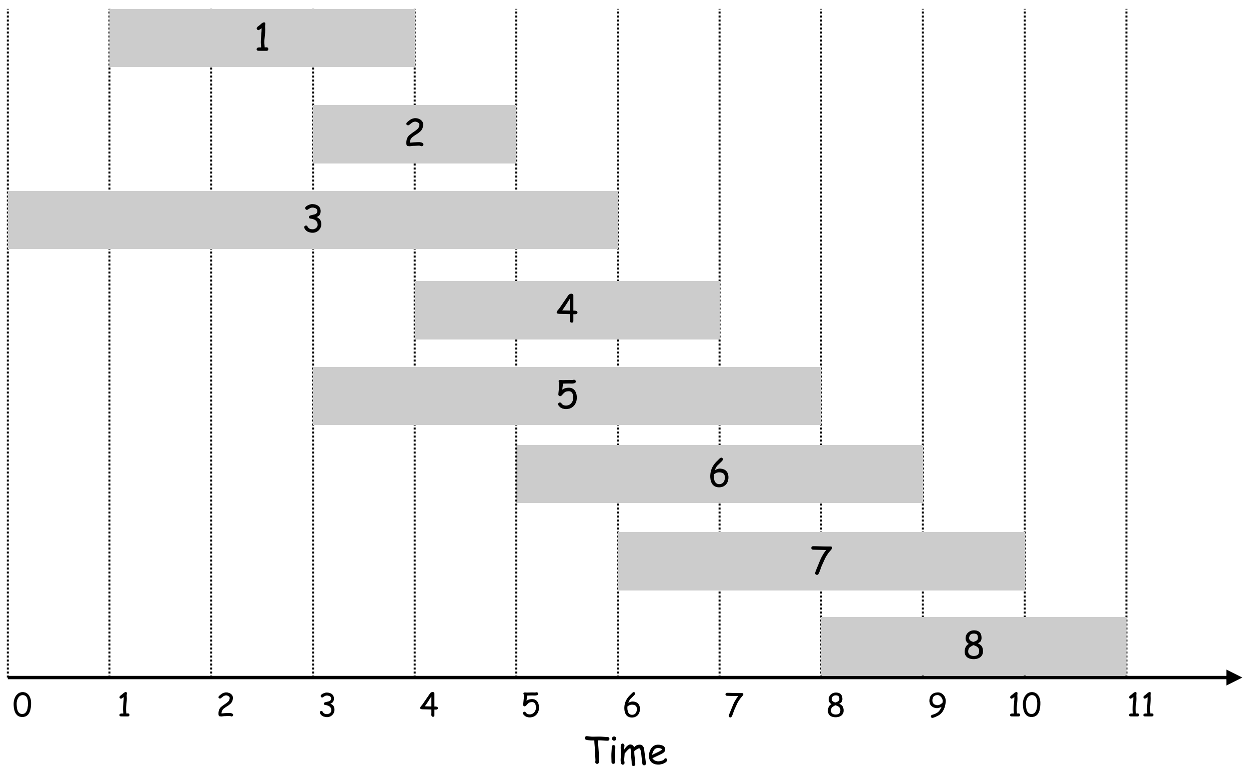



| j | $w_j$ | p(j) | OPT(j |
|---|---|---|---|
| 0 | | | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | 4 |
| 4 | 3 | 1 | 6 |
| 5 | 4 | 0 | 6 |
| 6 | 3 | 2 | 7 |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Example

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

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

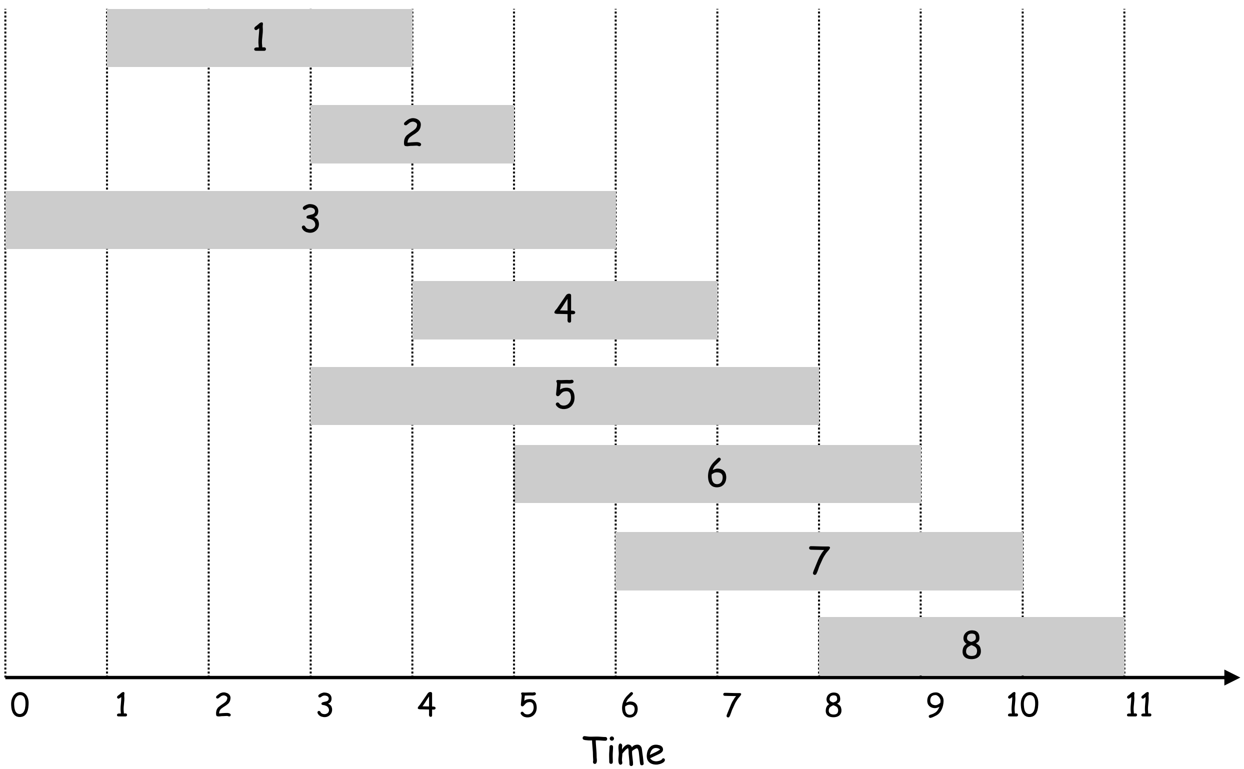



| j | $w_j$ | p(j) | OPT(j |
|---|---|---|---|
| 0 | | | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | 4 |
| 4 | 3 | 1 | 6 |
| 5 | 4 | 0 | 6 |
| 6 | 3 | 2 | 7 |
| 7 | 2 | 3 | 7 |
| 8 | 4 | 5 | |

# Example

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

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

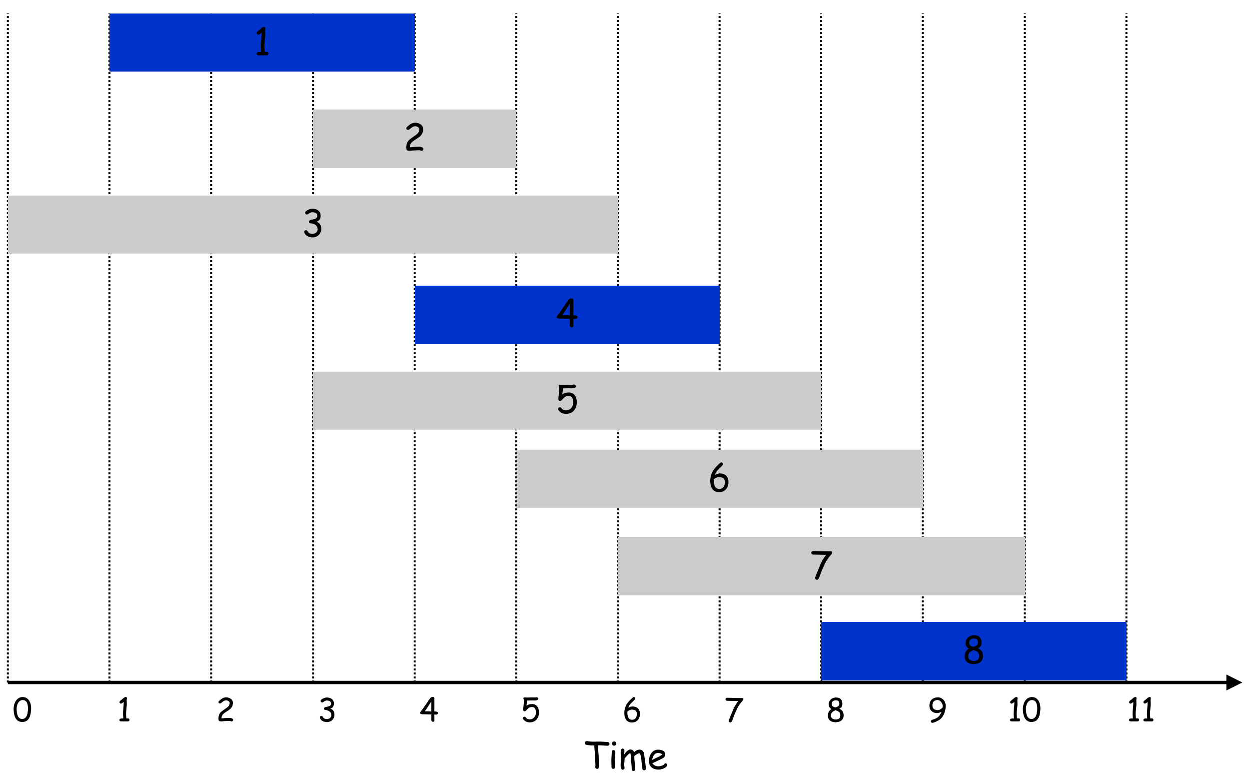



| j | $w_j$ | p(j) | OPT(j |
|---|---|---|---|
| 0 | | | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | 4 |
| 4 | 3 | 1 | 6 |
| 5 | 4 | 0 | 6 |
| 6 | 3 | 2 | 7 |
| 7 | 2 | 3 | 7 |
| 8 | 4 | 5 | **10** |

# Example

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

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** |