2

CSE 421

Alg Design by Induction, Dynamic Programming

Shayan Oveis Gharan

Q/A

- I did terrible in my midterm what can I do?
  - First see if you have fundamental problems or simple mistakes?
  - Try to spend time on your areas of weakness.
  - Try more exercises: there are lots of exercise in the book
  - See https://train.usaco.org/usacogate

- Grades are not important after you leave school
  - Make sure you learn the material so you can use it for the rest of your life

- How to think, how to write?
  - Many cases it is better to spend more time on thinking than writing.

Sample Soln of Problem 4 Midterm

Define C=A+B.
For any $i < j$ we have $C[i] < C[j]$, since

$A[i] < A[j]$ since A is sorted and distinct

$B[i] < B[j]$ since B is sorted and distinct.

So, C is sorted and distinct.

Therefore, by Problem 4 of sample midterm, we can find an element k such that C[k]=k in time $O(\log n)$.

Except, whenever we want to check the value of C[k], for some k, we check the value of $A[k] + B[k]$.

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 $\boxed{7 \;\; -2 \;\; -3 \;\; 8}$ -10 1 -7

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

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 7 -2 1 -8 6 -2 4

Pf of Correct: Maximum Sum 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 $x_a, \dots, x_b$ is the max-sum-subseq of $x_1, \dots, x_n$

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

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

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

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

- $x_i, \ldots, x_j$ 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 $x_a, \ldots, x_n$ is the max-sum-subseq of $x_1, \ldots, x_n$
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 ($x_a, \ldots, x_n$ is nonempty): We must have $x_a + \cdots + x_n \geq 0$.
Also, $x_a, \ldots, x_{n-1}$ must be the max-suffix-sum of $x_1, \ldots, x_{n-1}$. If not,

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

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 subseqence.

Summary

- Try to reduce an instance of size n to smaller instances

- Before designing the algorithm study structural 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

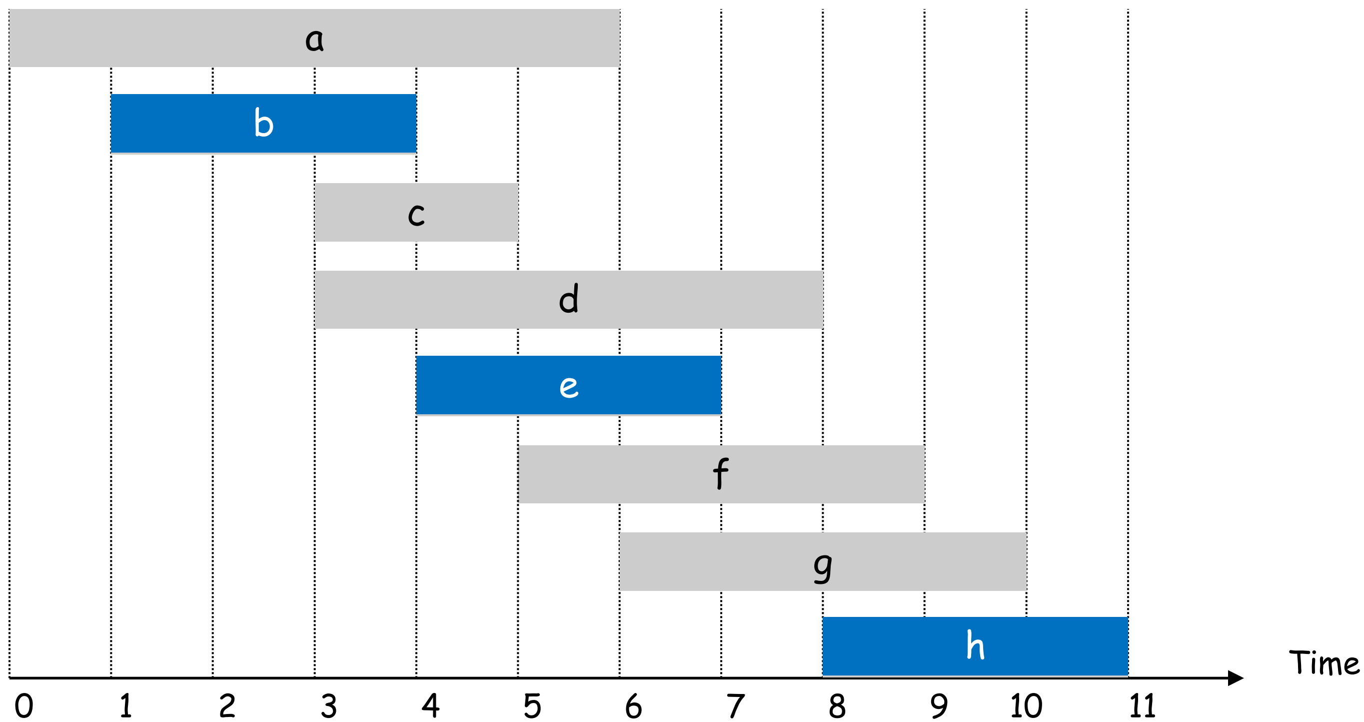
Dynamic programming is nothing but algorithm design by induction!

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.

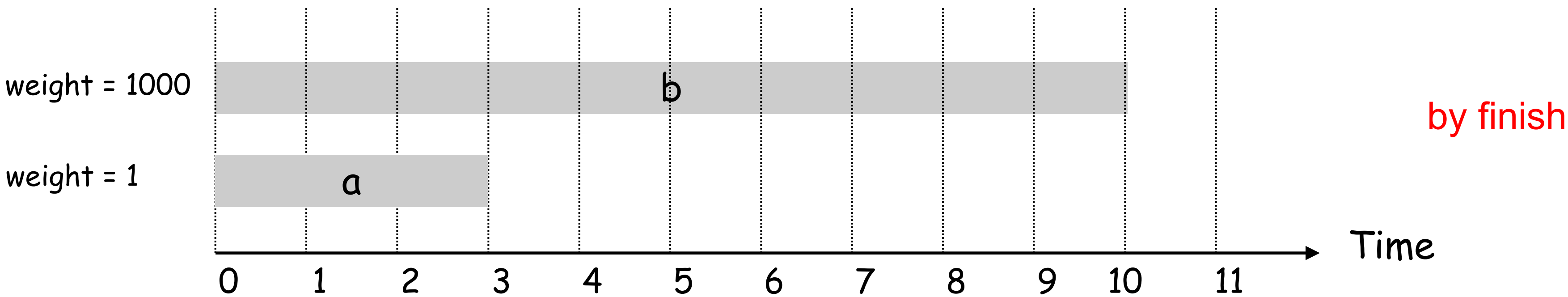


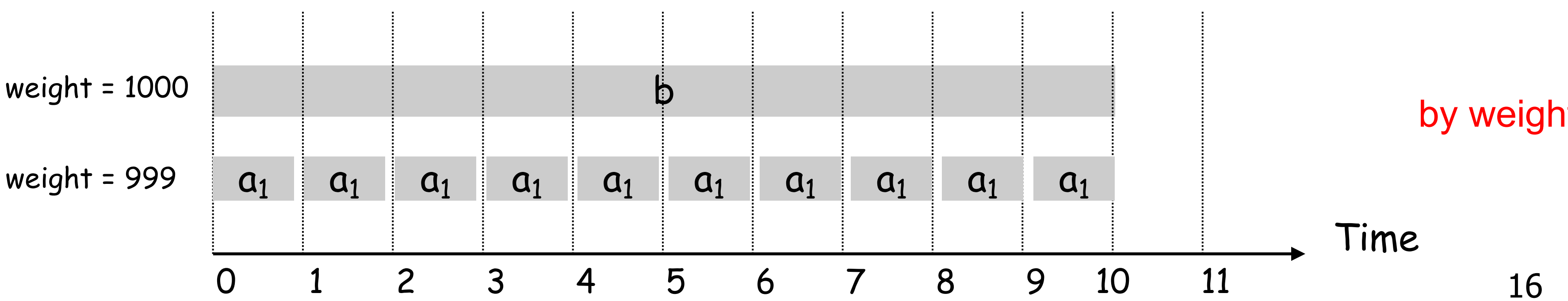
Unweighted Interval Scheduling: Review

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

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

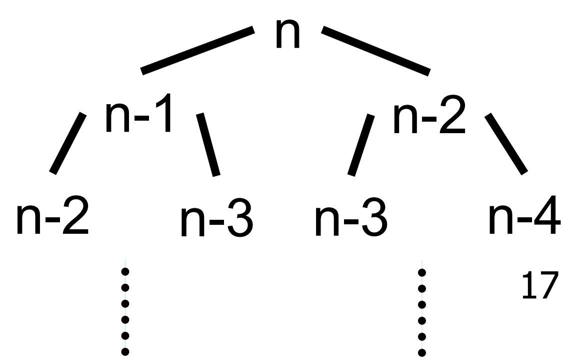
IS: Goal: For any n jobs we can compute OPT.
Case 1: Job n is not in OPT.
-- Then, just return OPT of $1, \ldots, n-1$.

$T(n) = T(n-1) + T(n-2) + O(1)$

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.

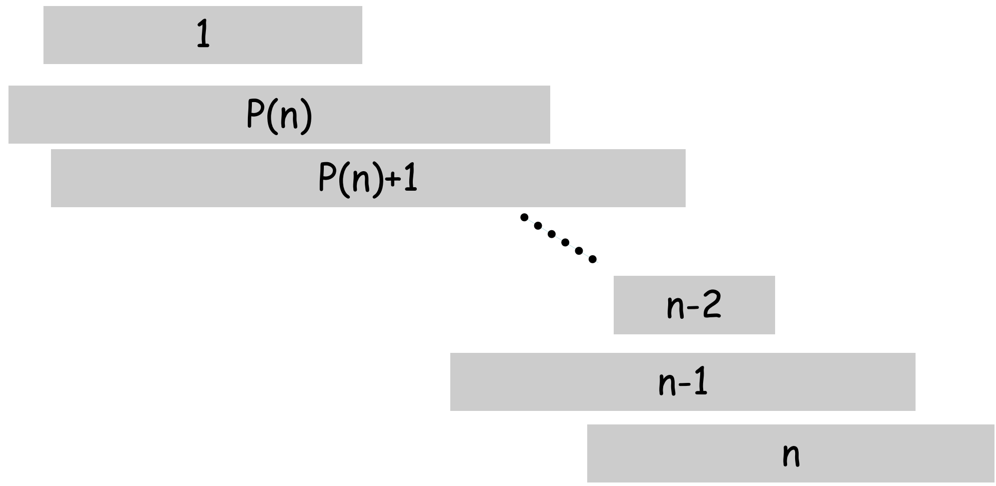


Sorting to Reduce Subproblems

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 $1, \dots, p(n)$



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) = largest index i < n such that job i is compatible with n.
- Then, we just need to find OPT of $1, \dots, p(n)$

Case 2: OPT does not select job n.

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

Take best of the two

Q: Have we made any progress (still reducing to two subproblems)?
A: Yes! This time every subproblem is of the form $1, \dots, i$ for some i
So, at most n possible subproblems.

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 … patible 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.

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 $1, \dots, j$

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

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

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))
}
```

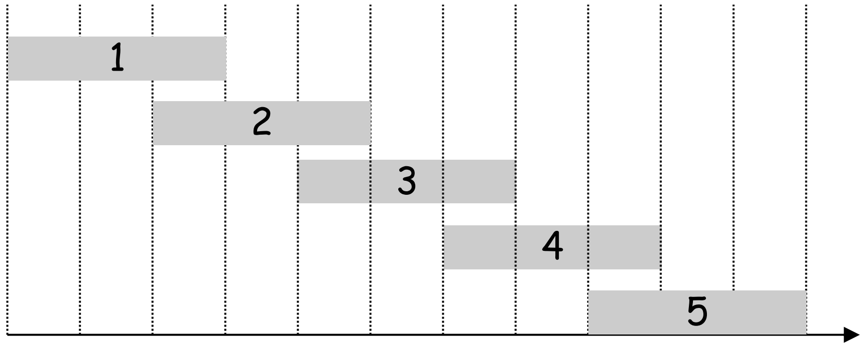
$f(n) = f(n-1) + f(n-2)$

Recursive Algorithm Fails

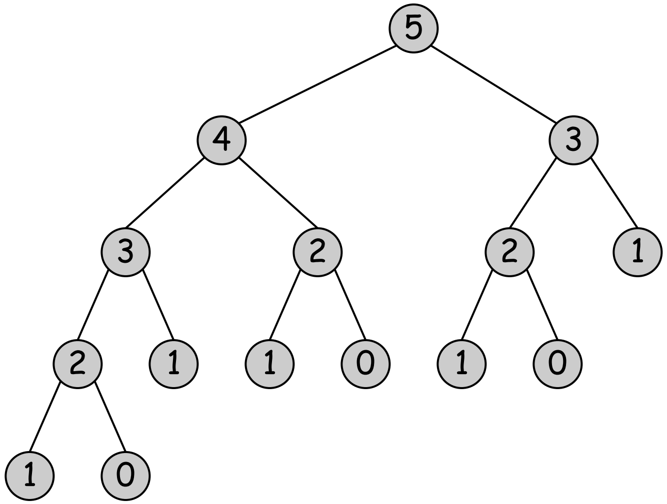
Even though we have only n subproblems, we do not store the solution to the subproblems

- 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),...,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)

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]
```

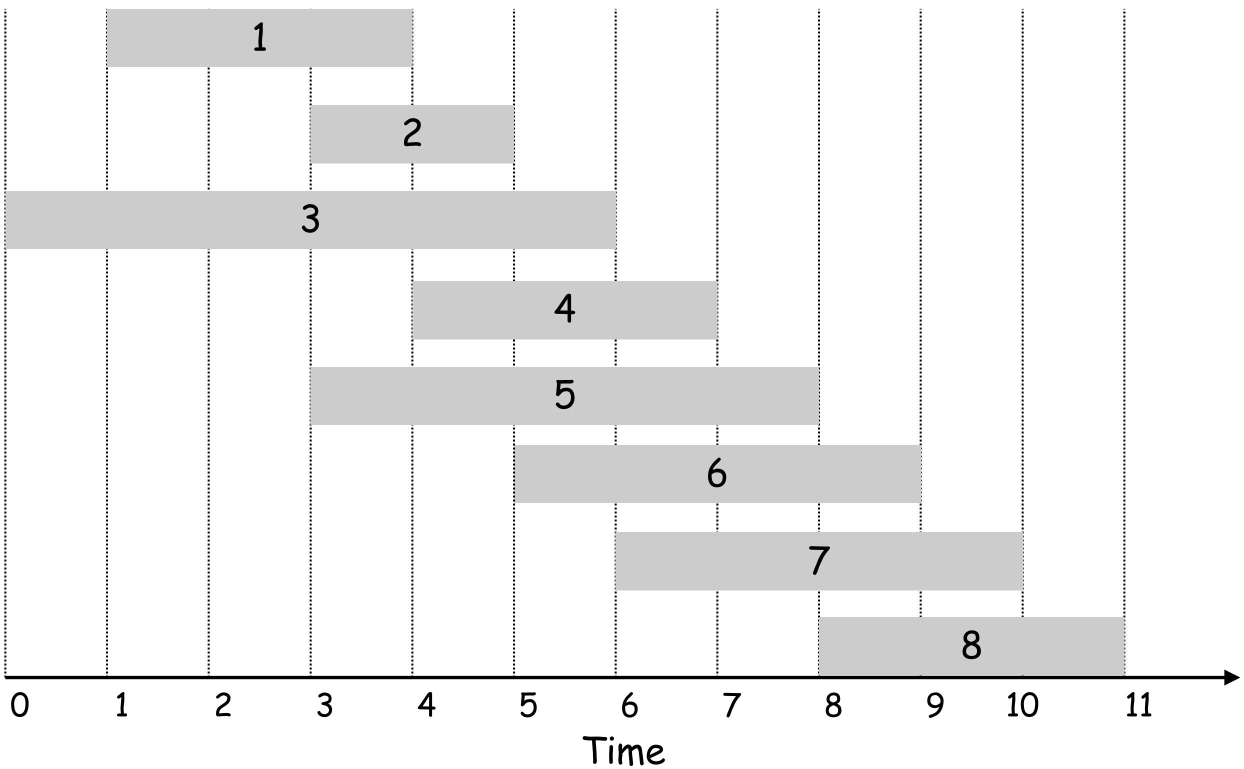
Claim: M[j] is value of OPT(j)

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.

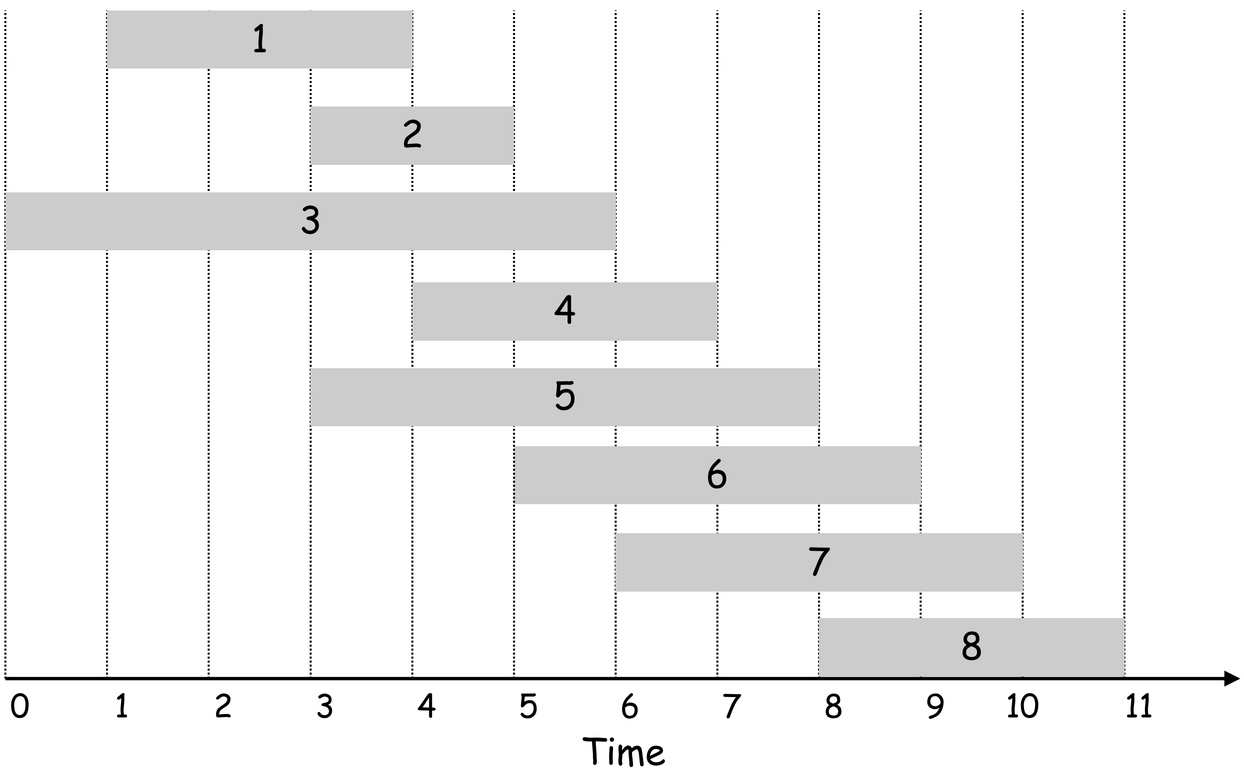


j	w_j	p(j)	OPT(j
0			0
1	3	0	
2	4	0	
3	1	0	
4	3	1	
5	4	0	
6	3	2	
7	2	3	
8	4	5	

Example

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

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

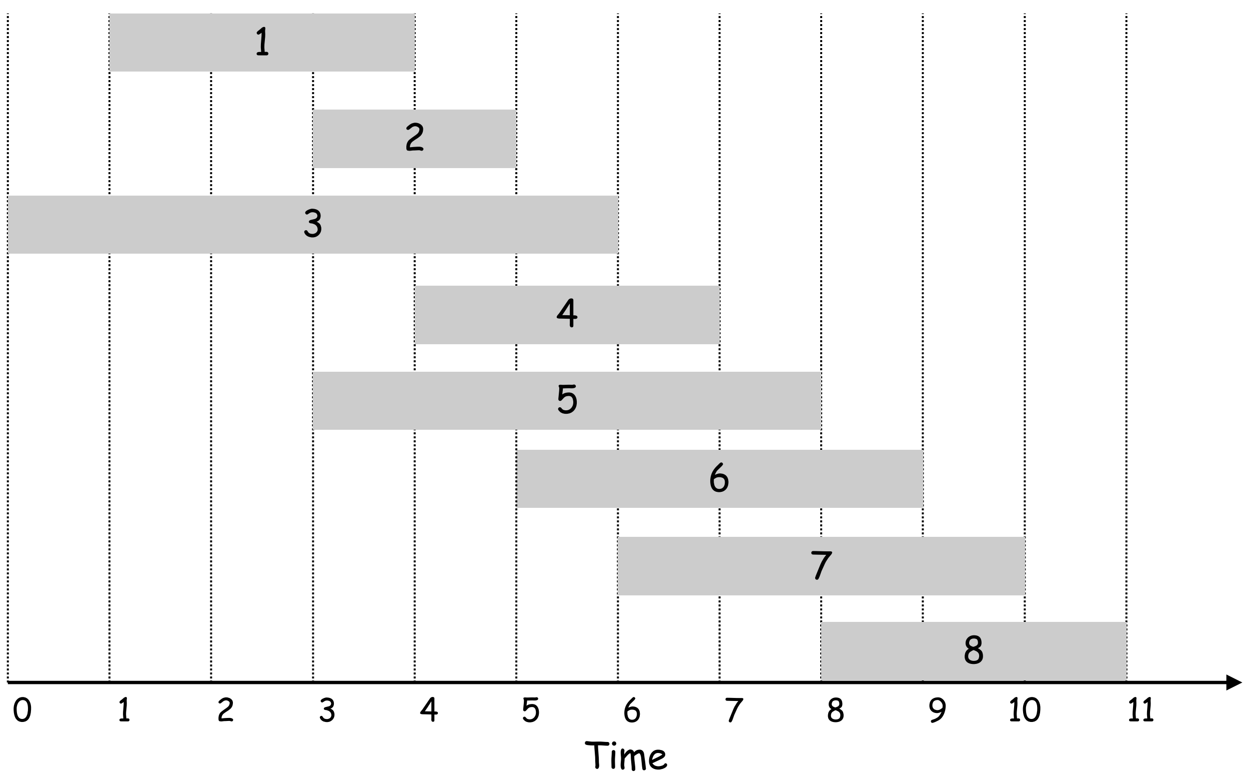


j	w_j	p(j)	OPT(j
0			0
1	3	0	3
2	4	0	
3	1	0	
4	3	1	
5	4	0	
6	3	2	
7	2	3	
8	4	5	

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.

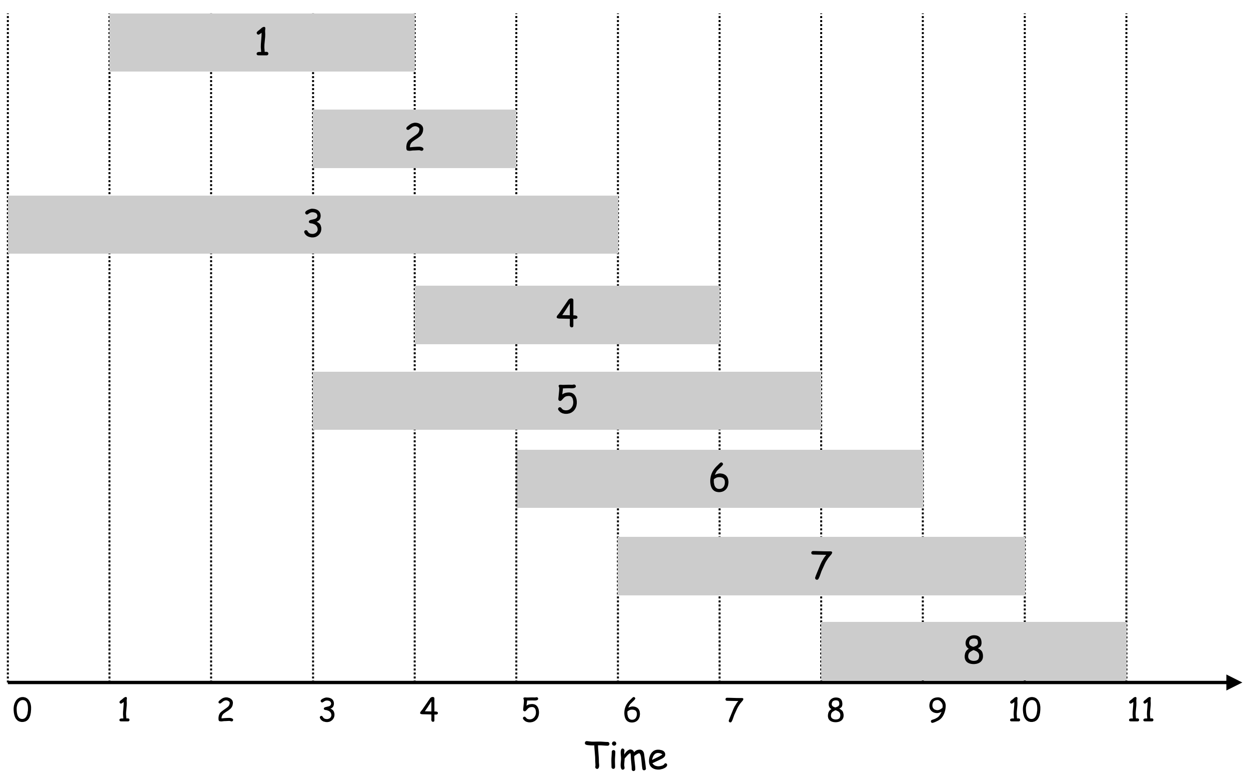


j	w_j	p(j)	OPT(j
0			0
1	3	0	3
2	4	0	**4**
3	1	0	
4	3	1	
5	4	0	
6	3	2	
7	2	3	
8	4	5	

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.

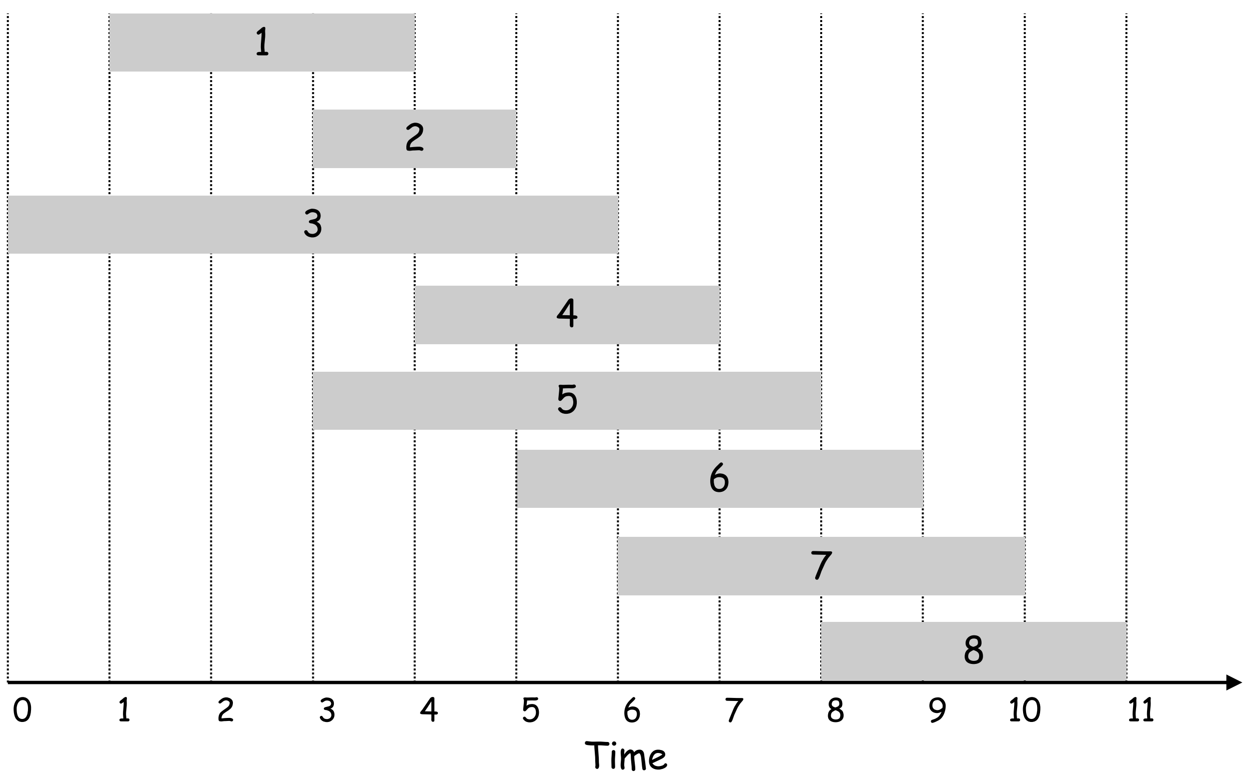


j	w_j	p(j)	OPT(j
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	
5	4	0	
6	3	2	
7	2	3	
8	4	5	

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.

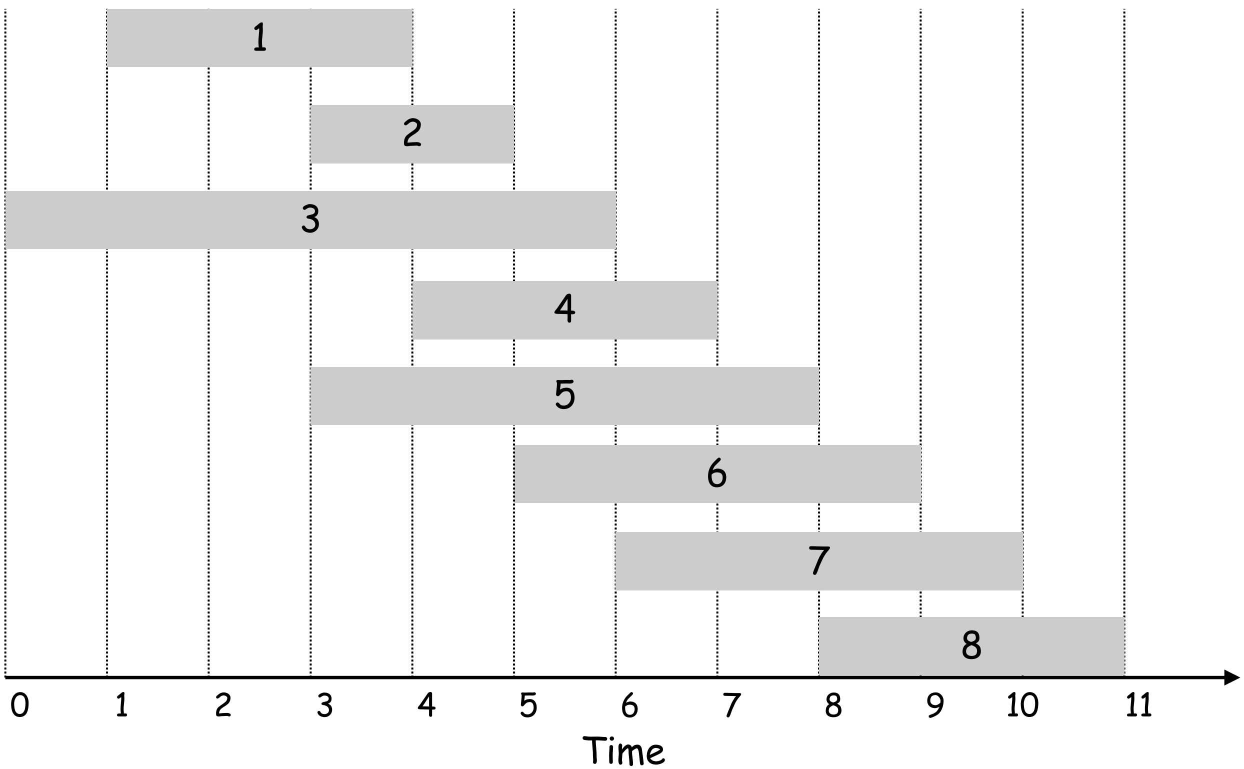


j	w_j	p(j)	OPT(j
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	
6	3	2	
7	2	3	
8	4	5	

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.

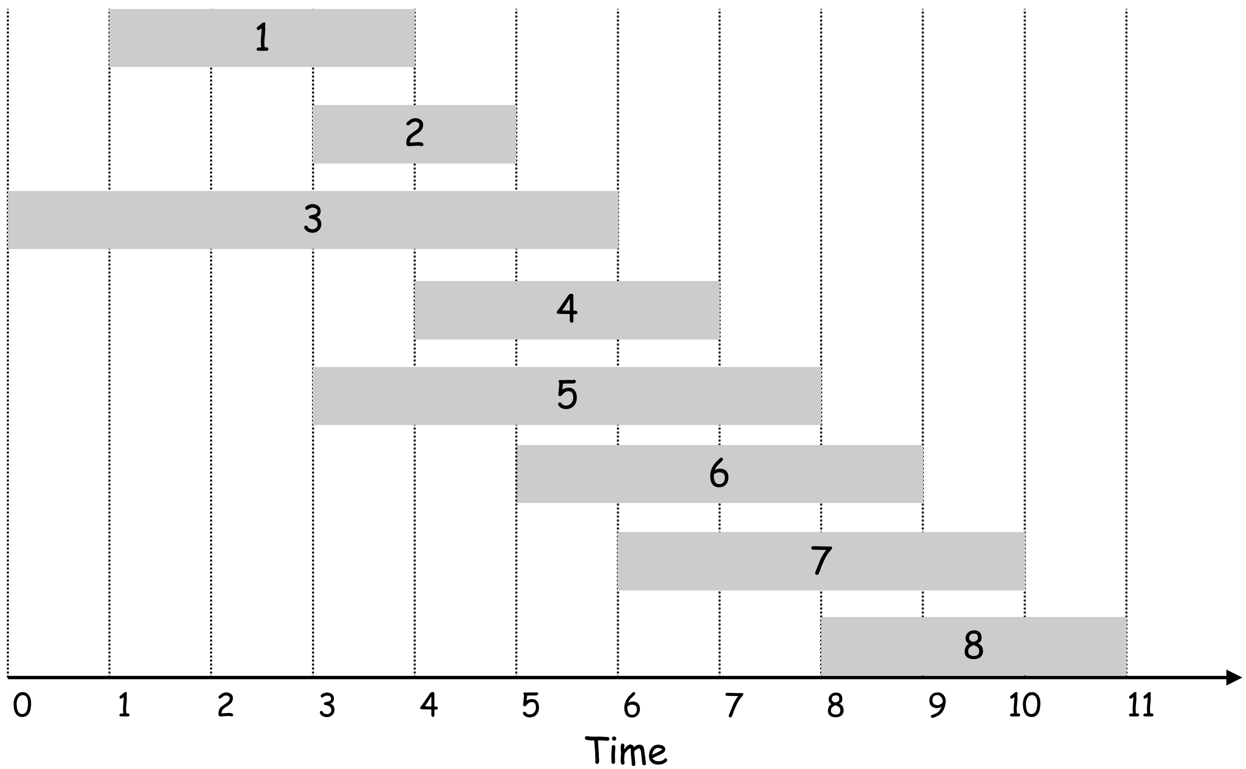


j	w_j	p(j)	OPT(j
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	6
6	3	2	
7	2	3	
8	4	5	

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.

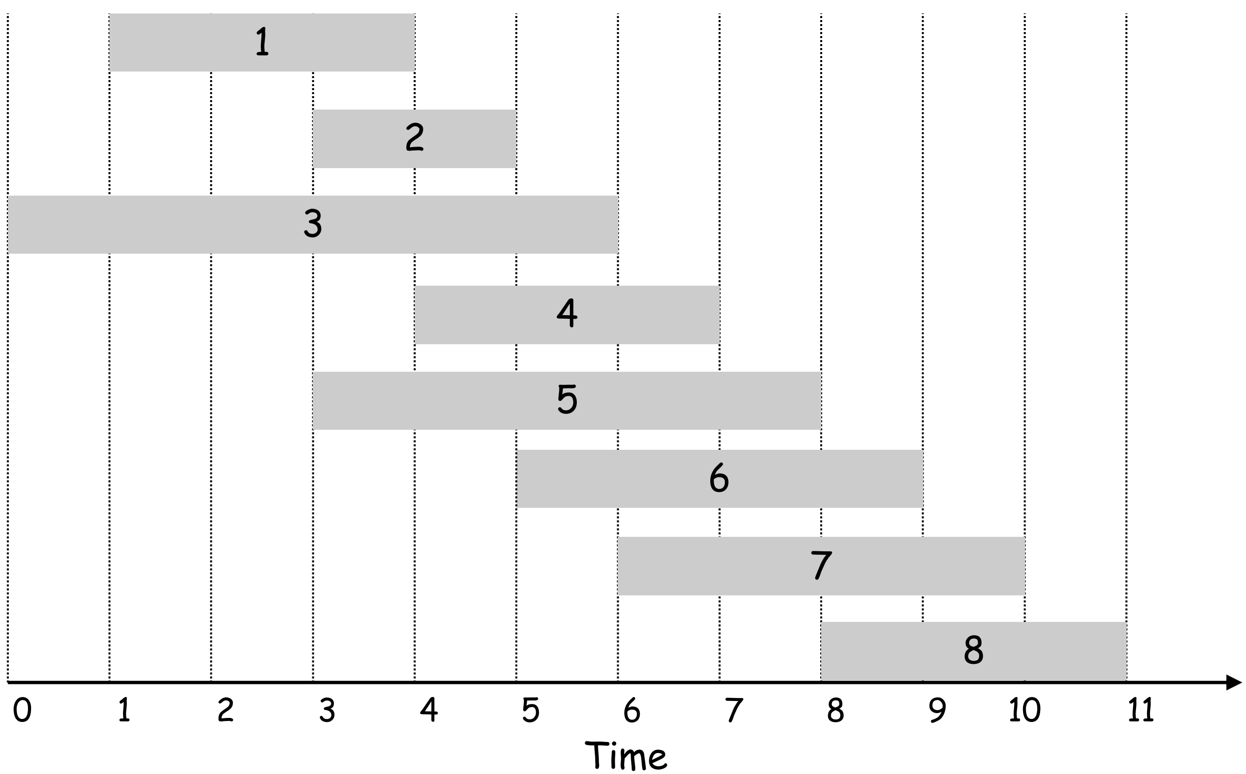


j	w_j	p(j)	OPT(j
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	6
6	3	2	7
7	2	3	
8	4	5	

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.

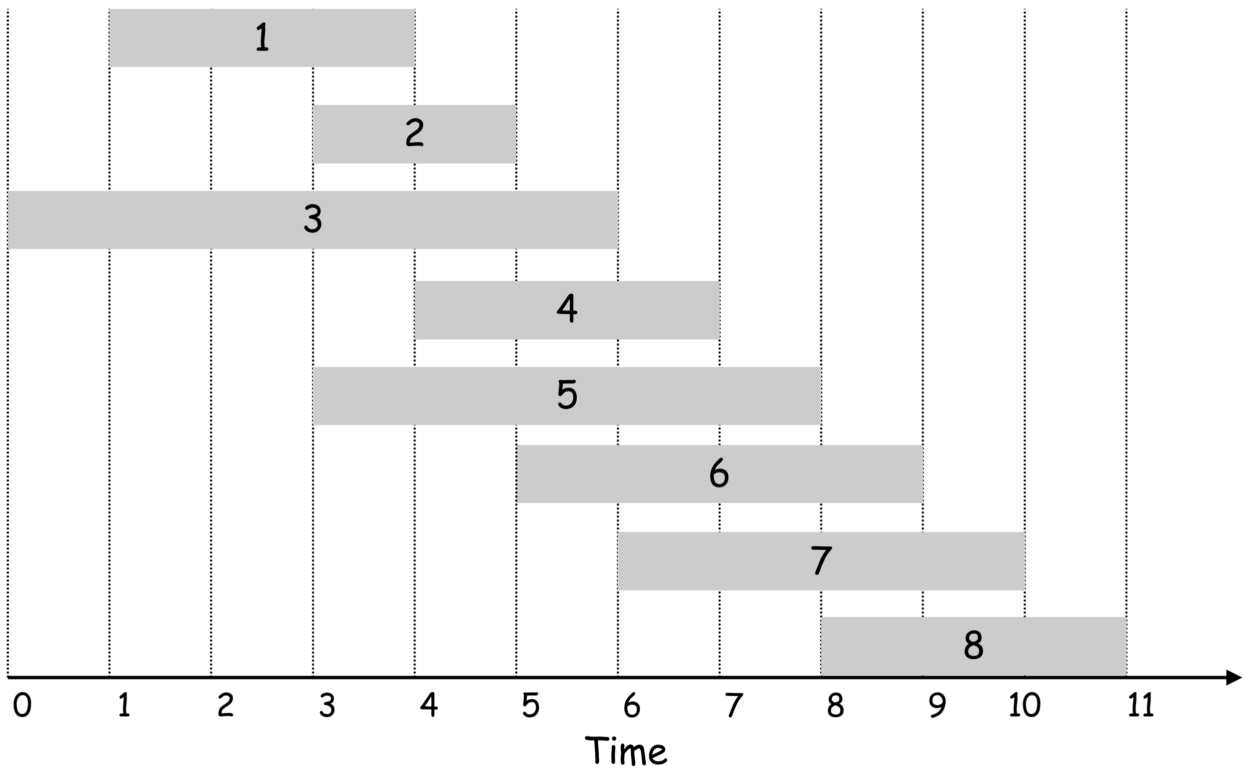


j	w_j	p(j)	OPT(j
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	6
6	3	2	7
7	2	3	7
8	4	5	

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.

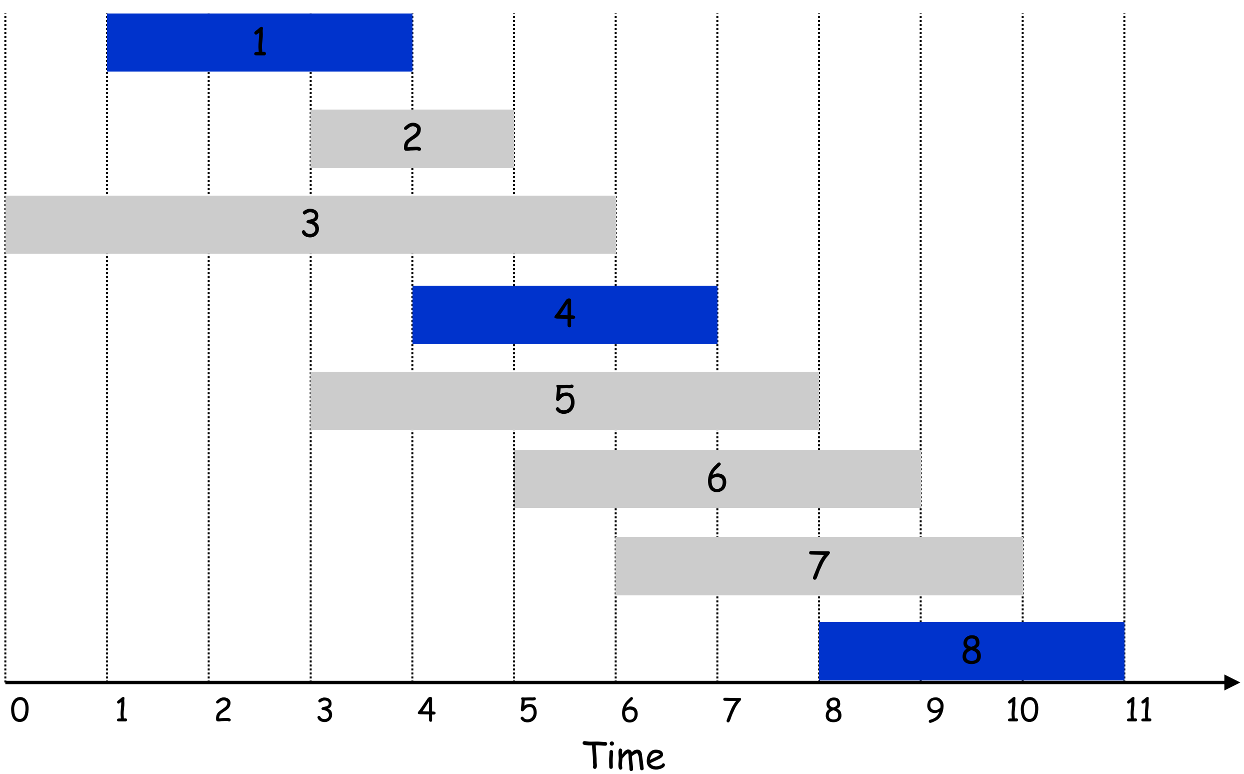


j	w_j	p(j)	OPT(j
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	6
6	3	2	7
7	2	3	7
8	4	5	**10**

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**

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.

$\boxed{W = 11}$

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 $1, \ldots, i$ of weight $\leq W$.

Case 1: $OPT(i)$ does not select item i
 - In this caes $OPT(i) = OPT(i-1)$

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 (Strengthenning Hypothesis)

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

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

```
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]
```

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

DP for Knapsack

W + 1

n + 1

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

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

n + 1

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

n + 1

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

```
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

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
φ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19					
{ 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

n + 1

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

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

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 →

n + 1 ↓

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

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 (w_i > w)
  M[i, w] = M[i-1, w]
else
  M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}
```

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 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 have 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