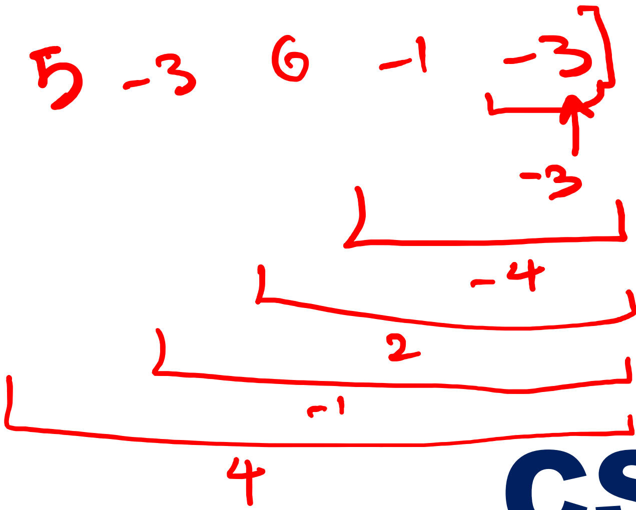


Hw6 difficult
DP.
Start early

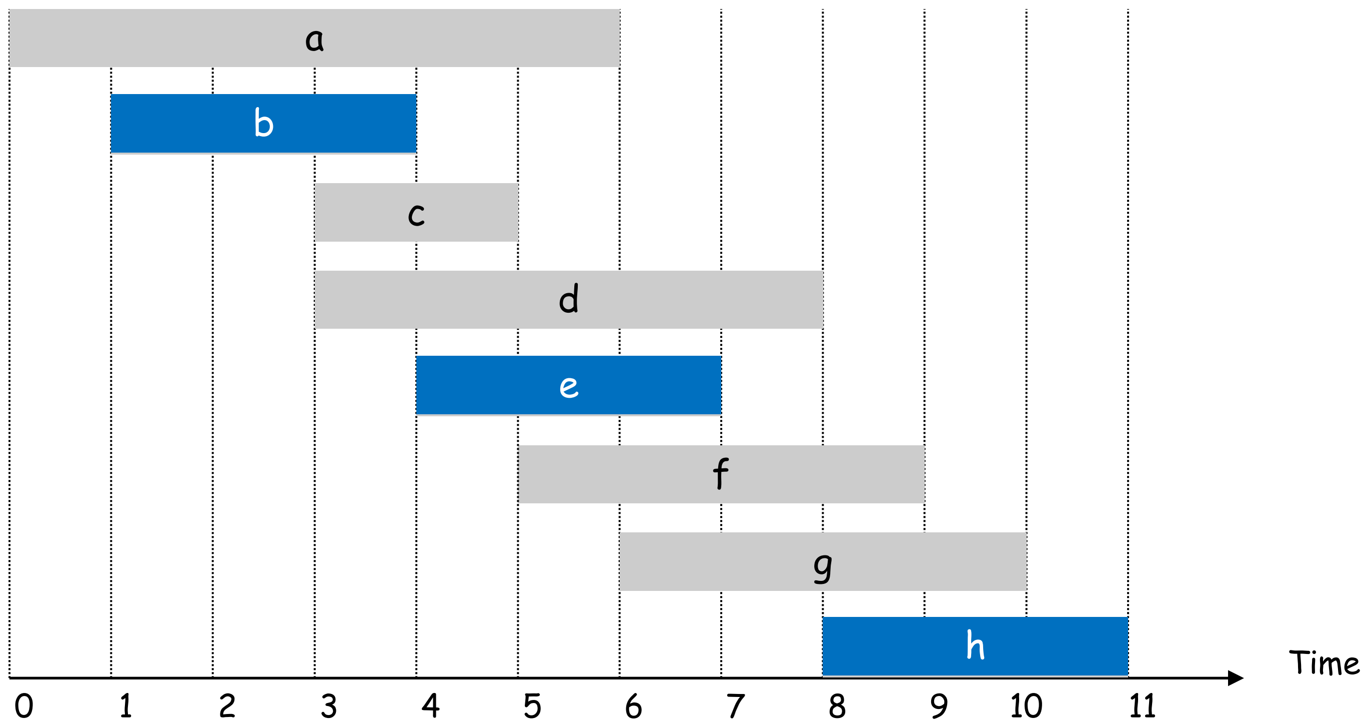
CSE 421

Alg Design by Induction, Dynamic Programming

Shayan Oveis Gharan

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.



Sorting to reduce Subproblems

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

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

Guessing

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 $1, \ldots, p(n)$

Case 2: OPT does not select job n.

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

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 $1, \ldots, i$ 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 $f(1) \le \cdots \le f(n)$

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

Case 2: OP…

- 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.

Weighted Job Scheduling by Induction

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

Let OPT(j) denote the OPT solution of $1, \ldots, j$ → Induction predicate
↳ immedi gives # subproblems

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 $OPT(j) = OPT\big(p(j)\big) \cup \{j\}$.

This is the most important step in design DP algorithms

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

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

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

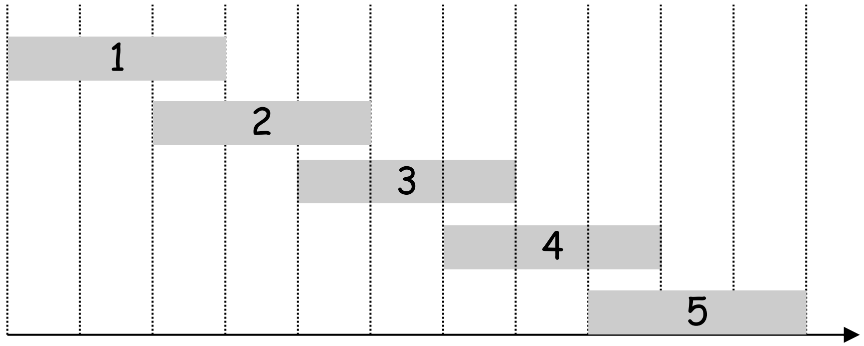
```
Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(w_j + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

Recursive Algorithm Fails

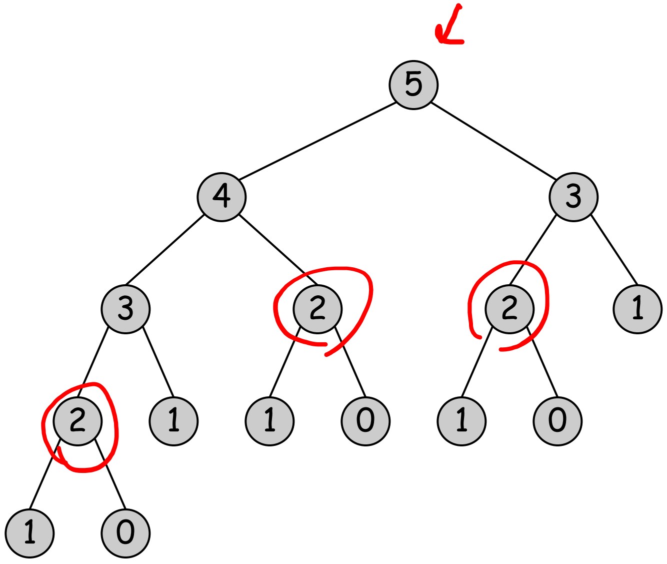
Even though we have only n subproblems, we do not store the solution to the subproblems

- 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, $s(1), \dots, s(n)$ and $f(1), \dots, f(n)$ and $w_1, \dots, w_n$.

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

Compute $p(1), p(2), \dots, p(n)$ *← can O(n log n)*

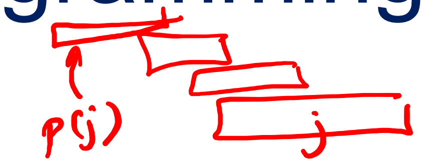
```
for j = 1 to n
   M[j] = empty
M[0] = 0
```

Base case of induction

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

n array locations to fill out; spend O(1) to fill out each.

Bottom up Dynamic Programming



You can also avoid recursion

- 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 $f(1) \leq f(2) \leq \cdots f(n)$.

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]
```

} exactly like induction

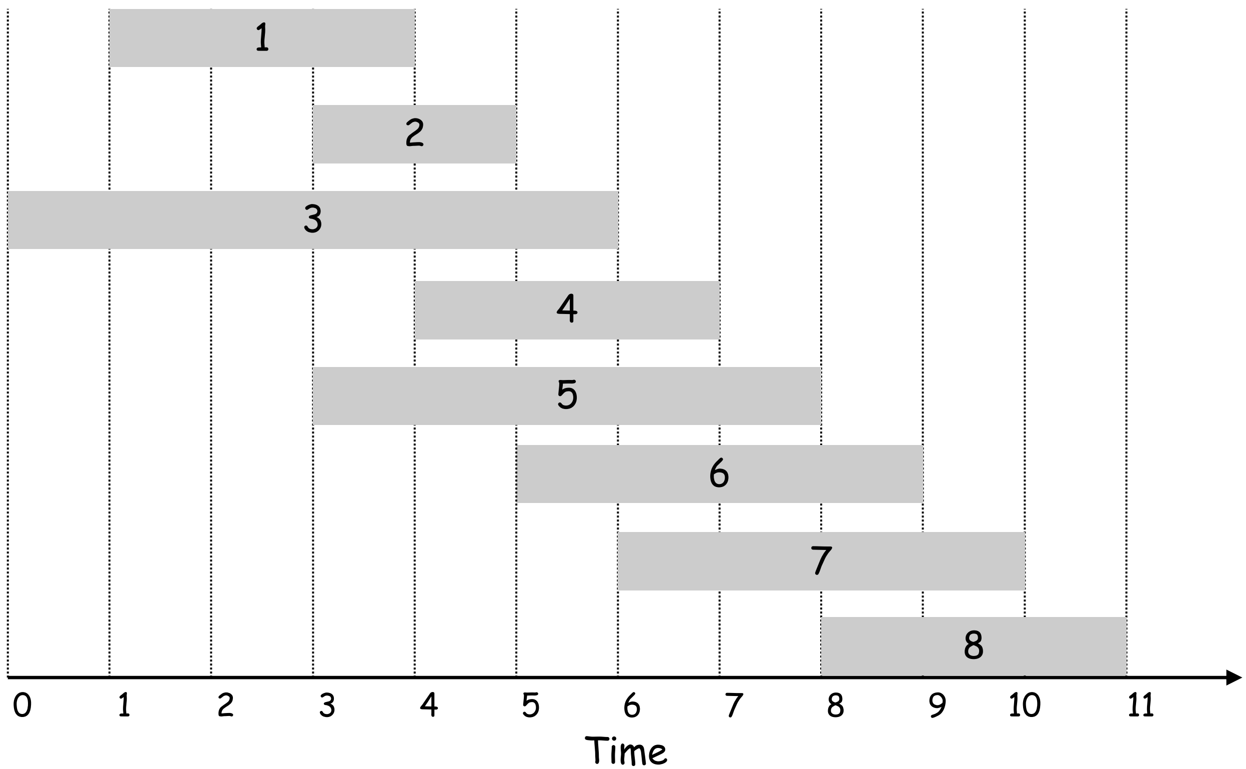
Claim: M[j] is value of OPT(j)

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.

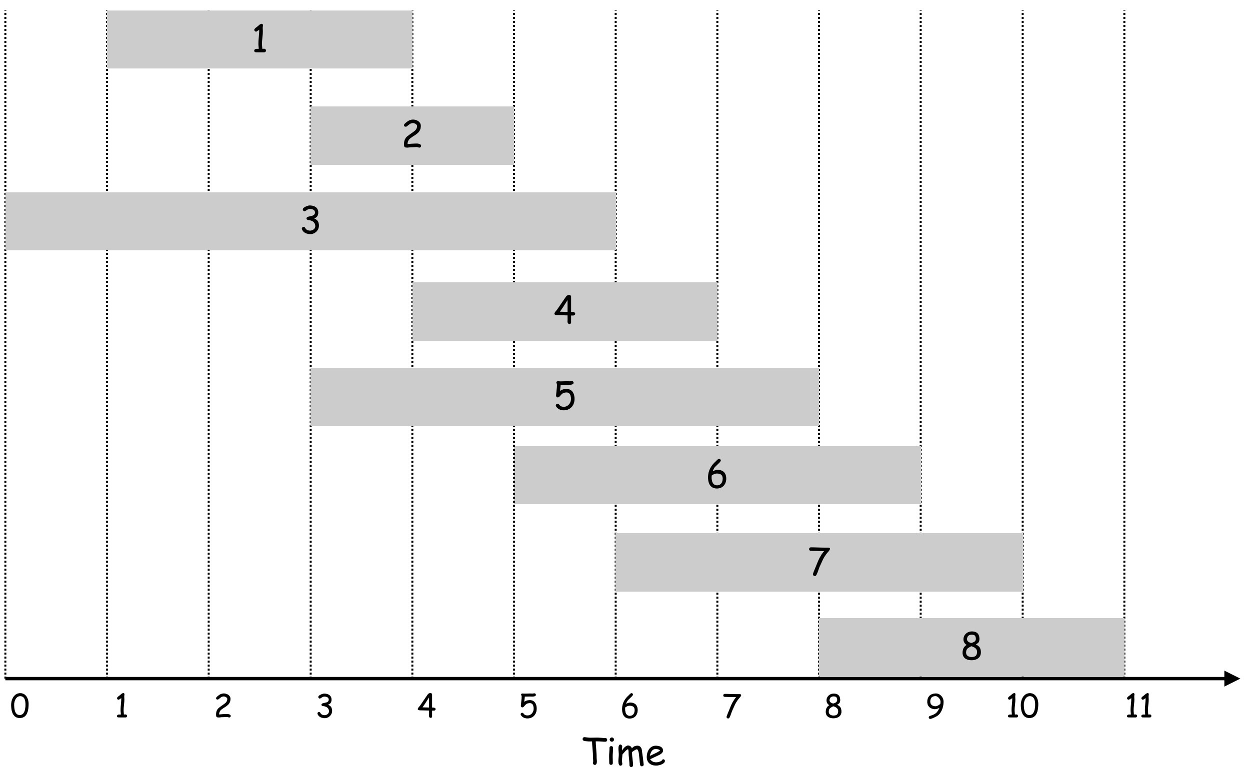


j	w_j	p(j)	OPT(j
0			0
1	3	0	
2	4	0	
3	1	0	
4	3	1	
5	4	0	
6	3	2	
7	2	3	
8	4	5	

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.

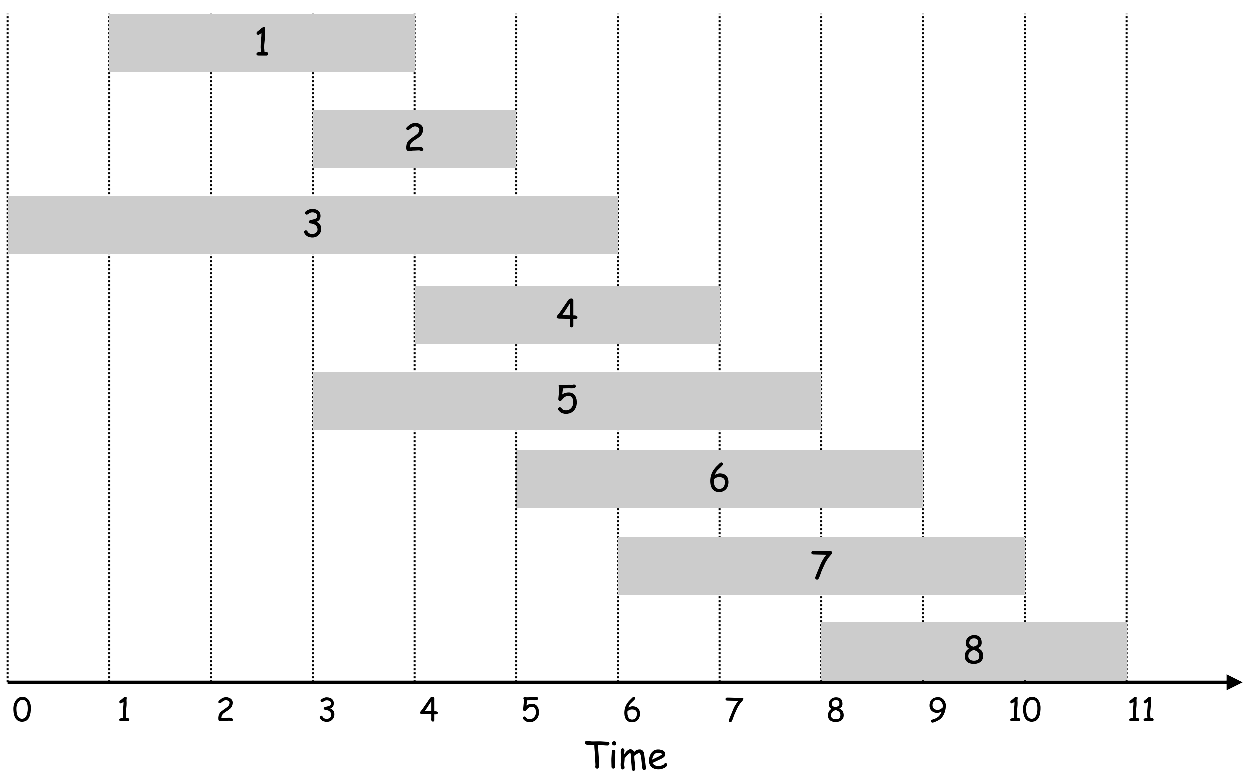


j	w_j	p(j)	OPT(j
0			0
1	3	0	3
2	4	0	
3	1	0	
4	3	1	
5	4	0	
6	3	2	
7	2	3	
8	4	5	

Example

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

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

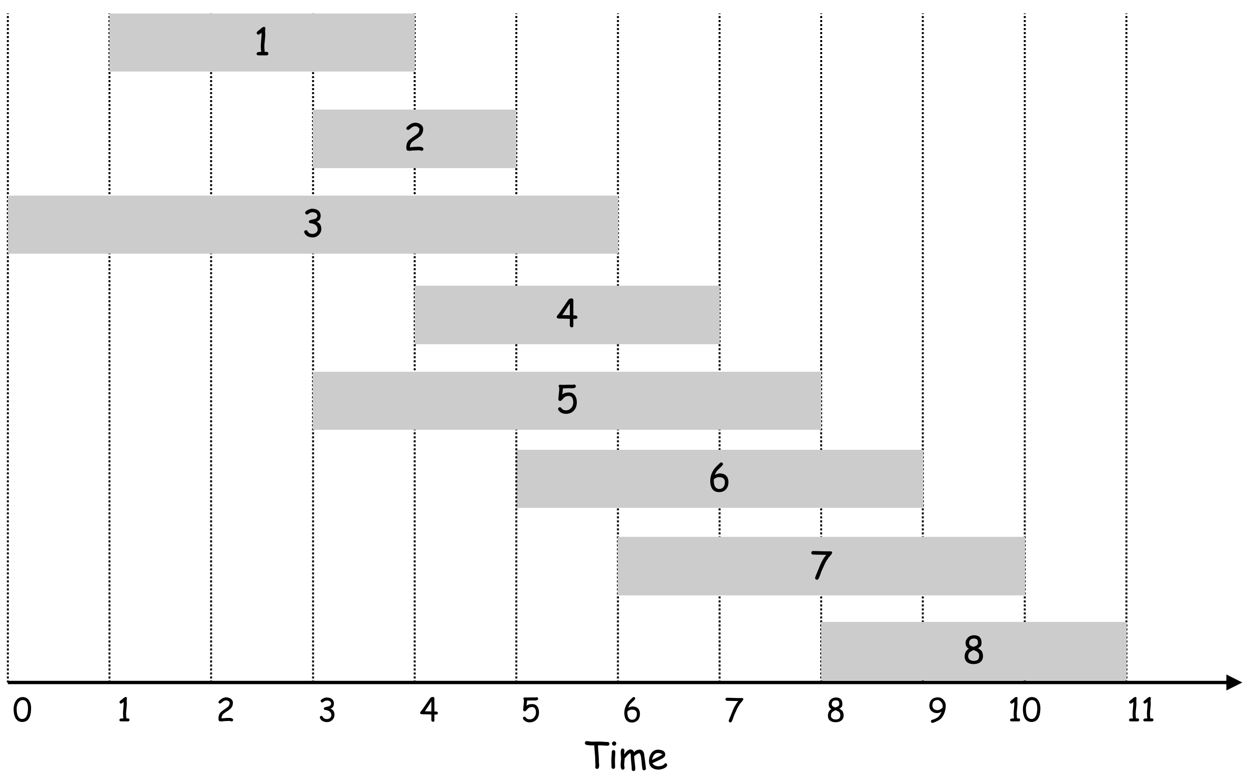


j	w_j	p(j)	OPT(j
0			0
1	3	0	3
2	4	0	**4**
3	1	0	
4	3	1	
5	4	0	
6	3	2	
7	2	3	
8	4	5	

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.

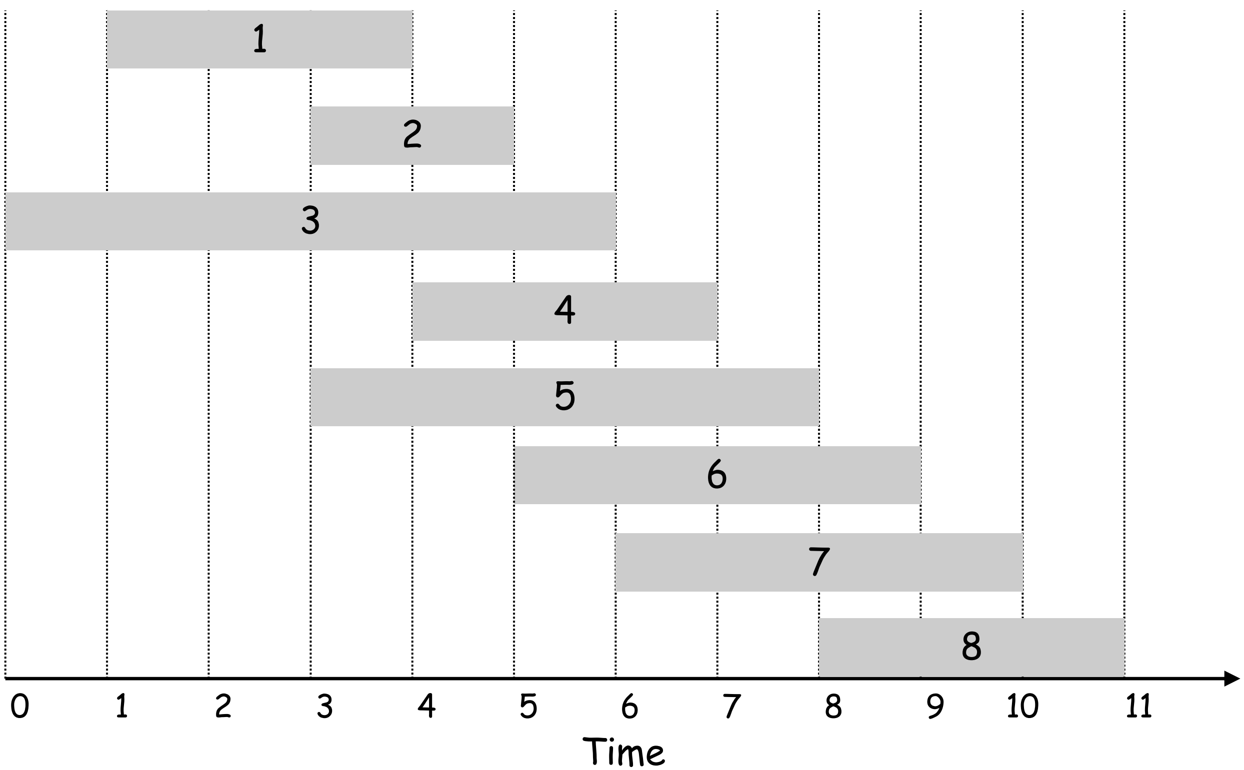


j	w_j	p(j)	OPT(j
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	
5	4	0	
6	3	2	
7	2	3	
8	4	5	

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.

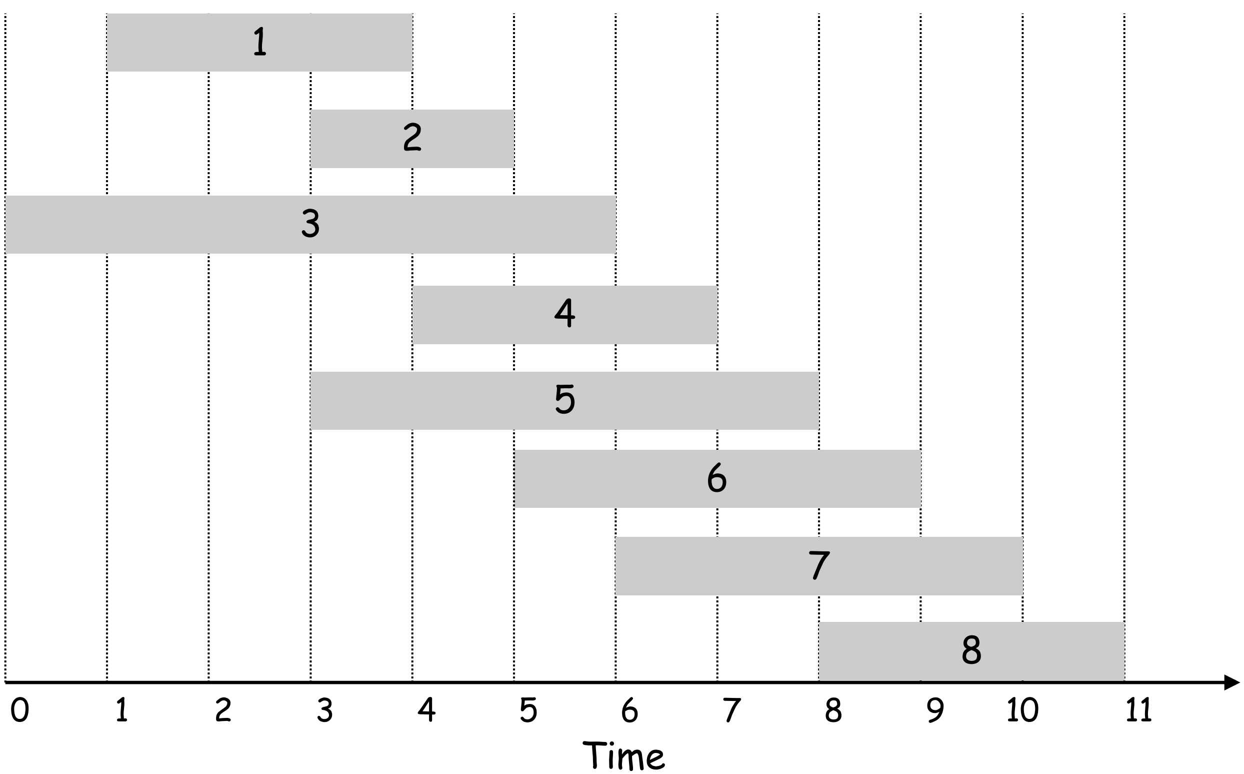


j	w_j	p(j)	OPT(j
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	
6	3	2	
7	2	3	
8	4	5	

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.

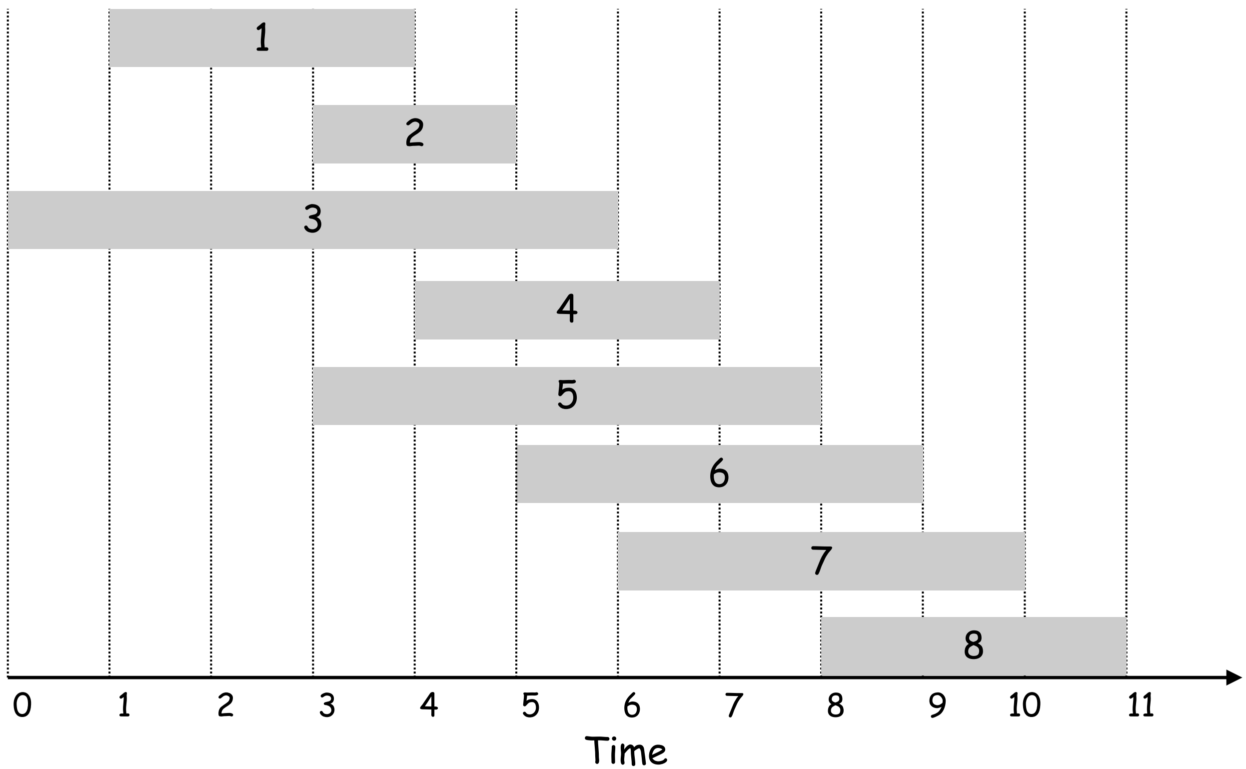


j	w_j	p(j)	OPT(j
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	6
6	3	2	
7	2	3	
8	4	5	

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.

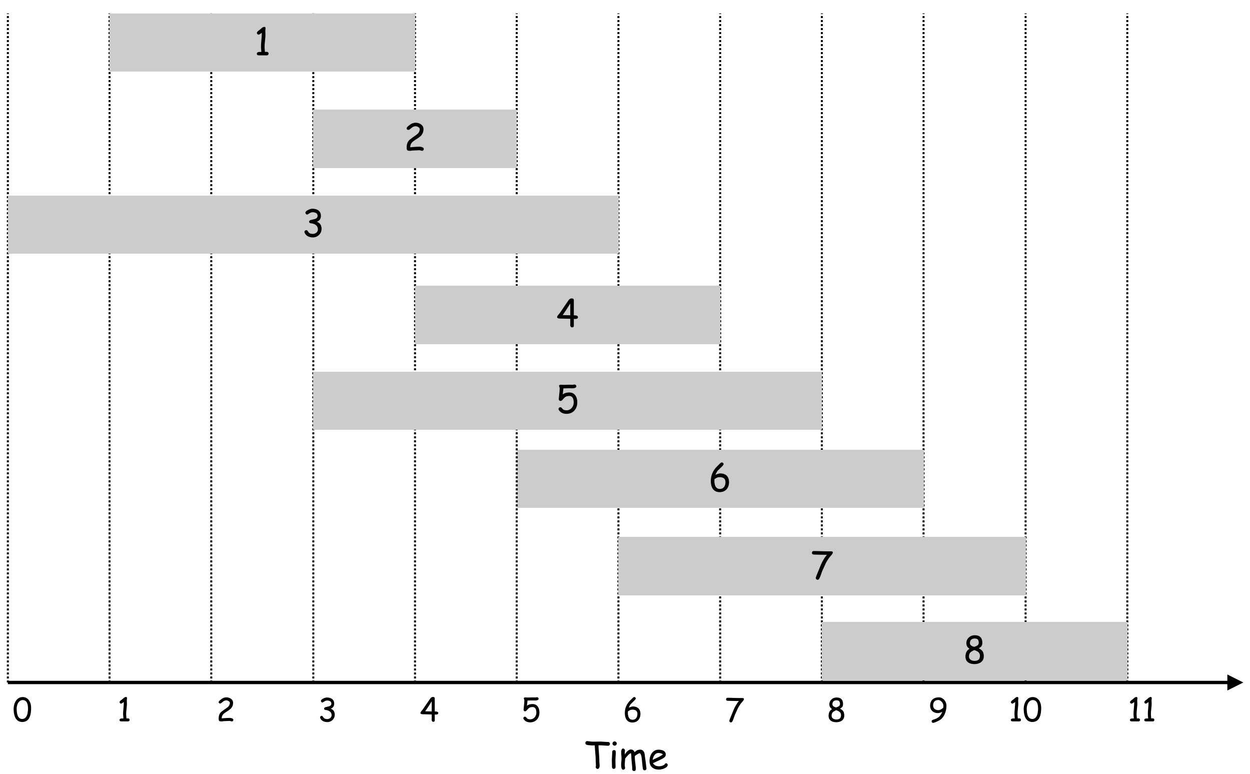


j	w_j	p(j)	OPT(j
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	6
6	3	2	7
7	2	3	
8	4	5	

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.

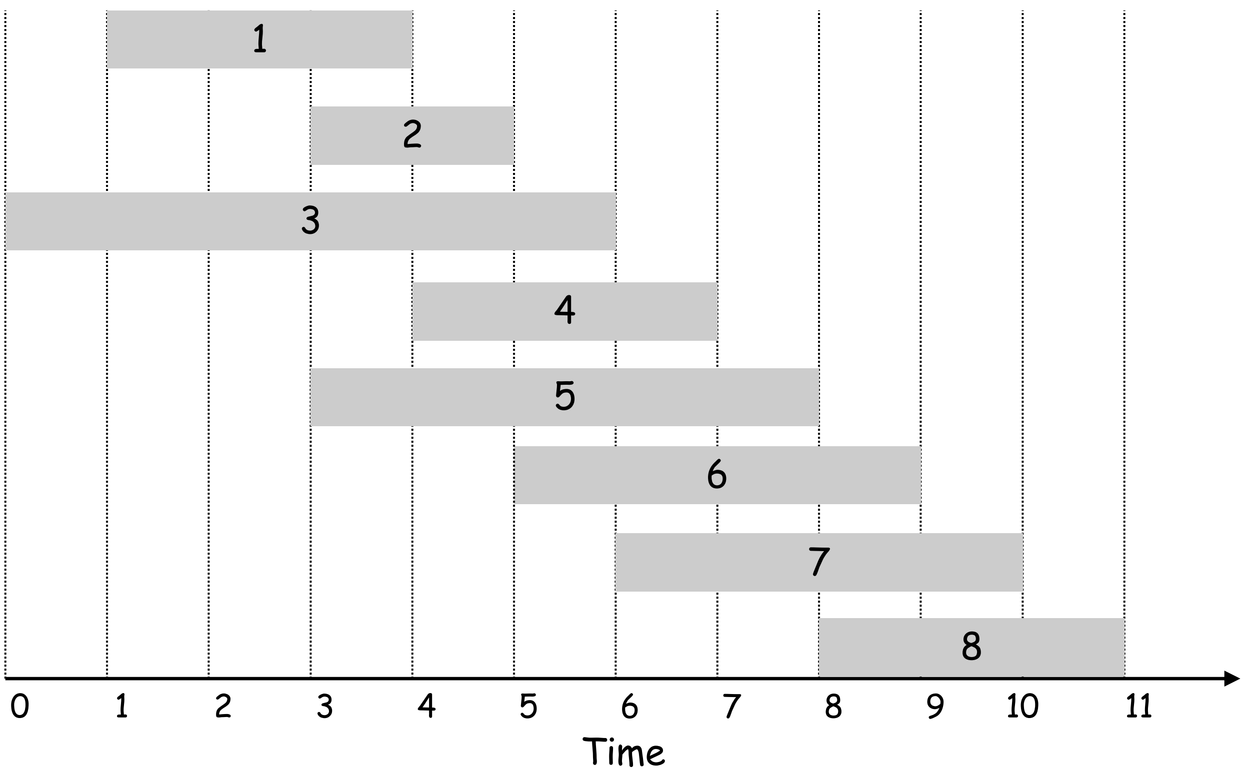


j	w_j	p(j)	OPT(j)
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	6
6	3	2	7
7	2	3	7
8	4	5	

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.

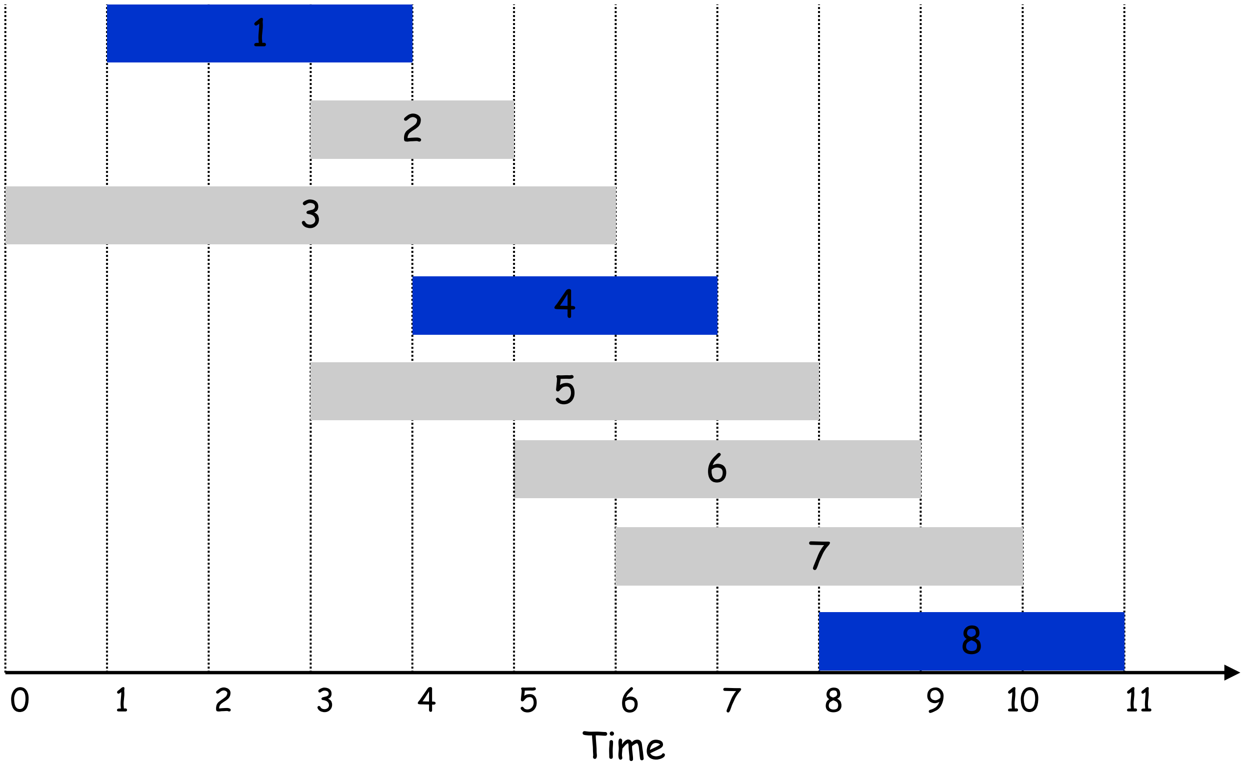


j	w_j	p(j)	OPT(j)
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	6
6	3	2	7
7	2	3	7
8	4	5	**10**

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 $v_i > 0$. (integers)

Knapsack has capacity of W kilograms.

Goal: fill knapsack so as to maximize total value.

Ex: OPT is { 3, 4 } with (weight 10) and value 36.

W = 11

Item	Value	Weight
1	1	2
2	5	3
3	14	4
4	22	6
5	30	8

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

Ex: { 5, 2 } 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)

OPT(n, W) is solution to problem.

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

We have n.W many subproblems

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

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

Take best of the two

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

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

Therefore,

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

(Base Case)

DP for Knapsack

$\Theta(n \cdot W)$

recursive

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

Non-recursive

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

return M[n, W]
```

make sure you have computed $M[j, w']$ for all $j < i$ and $w' \le w$

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

```
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 ]}
```

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, 2, 3 }	0											
{ 1, 2, 3, 4 }	0											
{ 1, 2, 3, 4, 5 }	0											

max(OPT(1,1)) >1 item 2 cannot be used
BC $w_2 > 1$

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										

Max(6 + OPT(1,0), OPT(1,2)) = 6

7 = Max(OPT(1,3), 6 + OPT(1,1))

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					
{ 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	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, 9)

29 = max(OPT(3, 9), 22 + OPT(3, 3))

25 7

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

Knapsack Problem: Running Time

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

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

Knapsack approximation algorithm:

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

DP Ideas so far

- You may have to define an ordering to decrease #subproblems

- OPT(i,w) is exactly the predicate of induction

- 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