Dynamic Programming (Chapter 6)

Algorithm Design Techniques

- Greedy
- Divide and Conquer
- Dynamic Programming
- Network Flows

Algorithm Design

	Greedy	Divide and Conquer	Dynamic Programming
Formulate problem	?	?	?
Design algorithm	less work	more work	more work
Prove correctness	more work	less work	less work
Analyze running time	less work	more work	less work

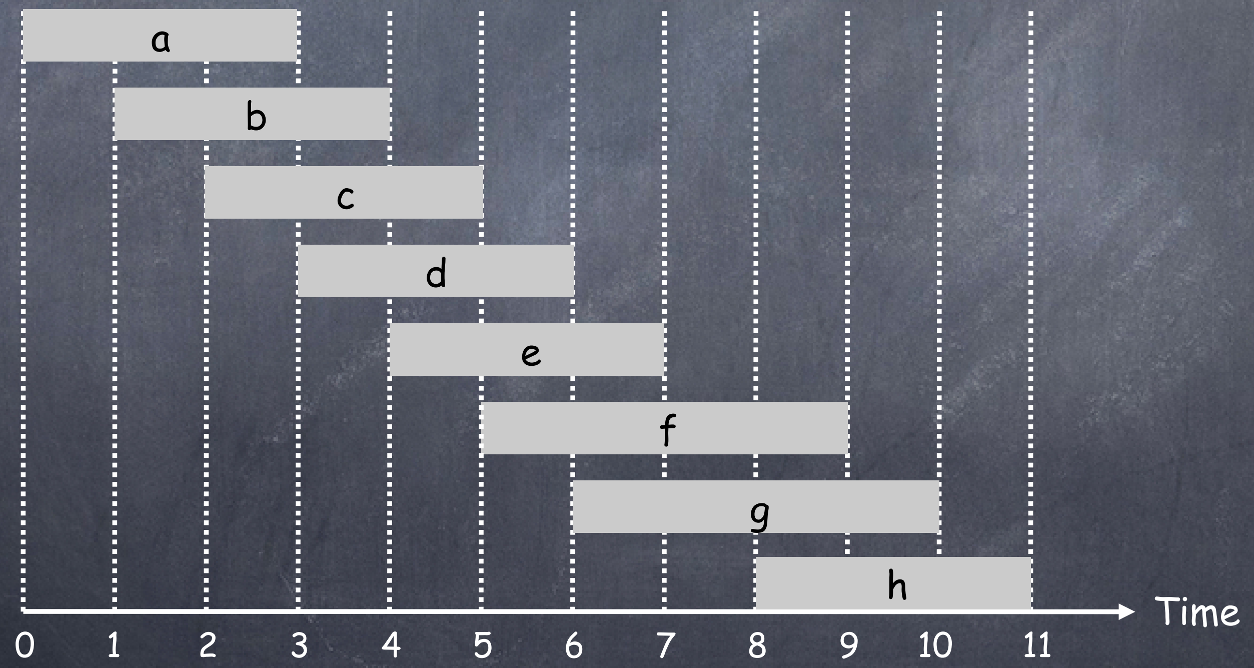
Dynamic Programming "Recipe"

- Recursive formulation of optimal solution in terms of subproblems
- Obvious implementation requires solving exponentially many subproblems
- Careful implementation to solve only polynomially many different subproblems

Interval Scheduling

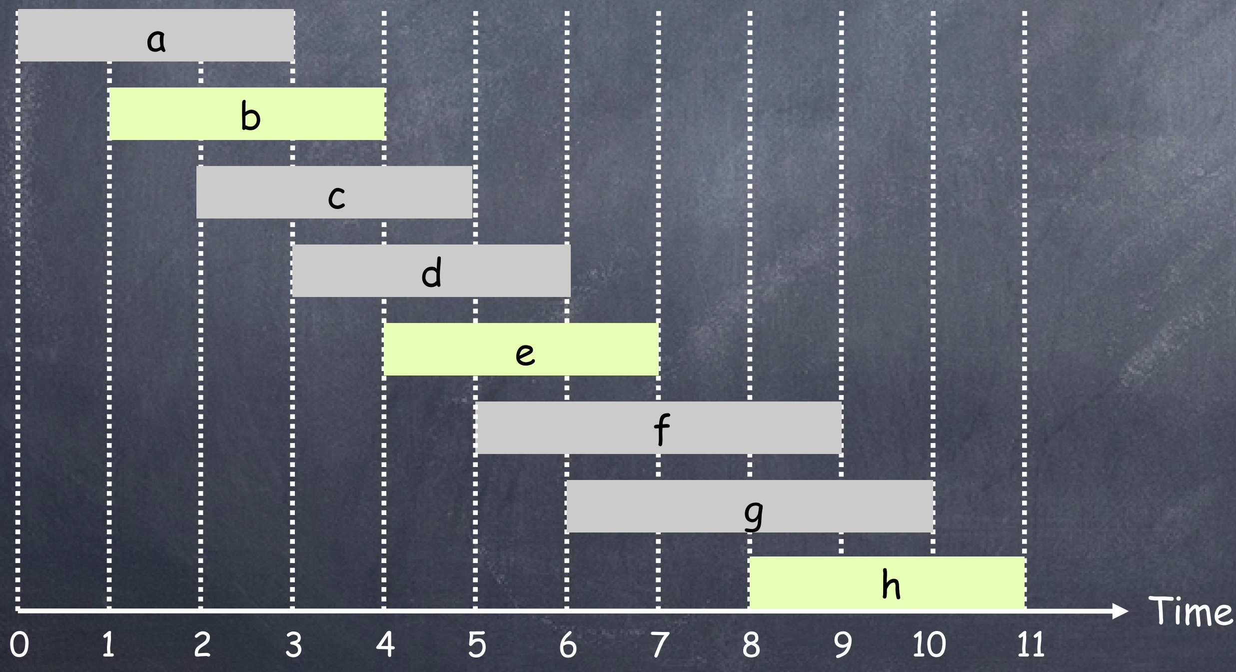
(Yes, this is an old problem!)

- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.



Interval Scheduling: Greedy Solution

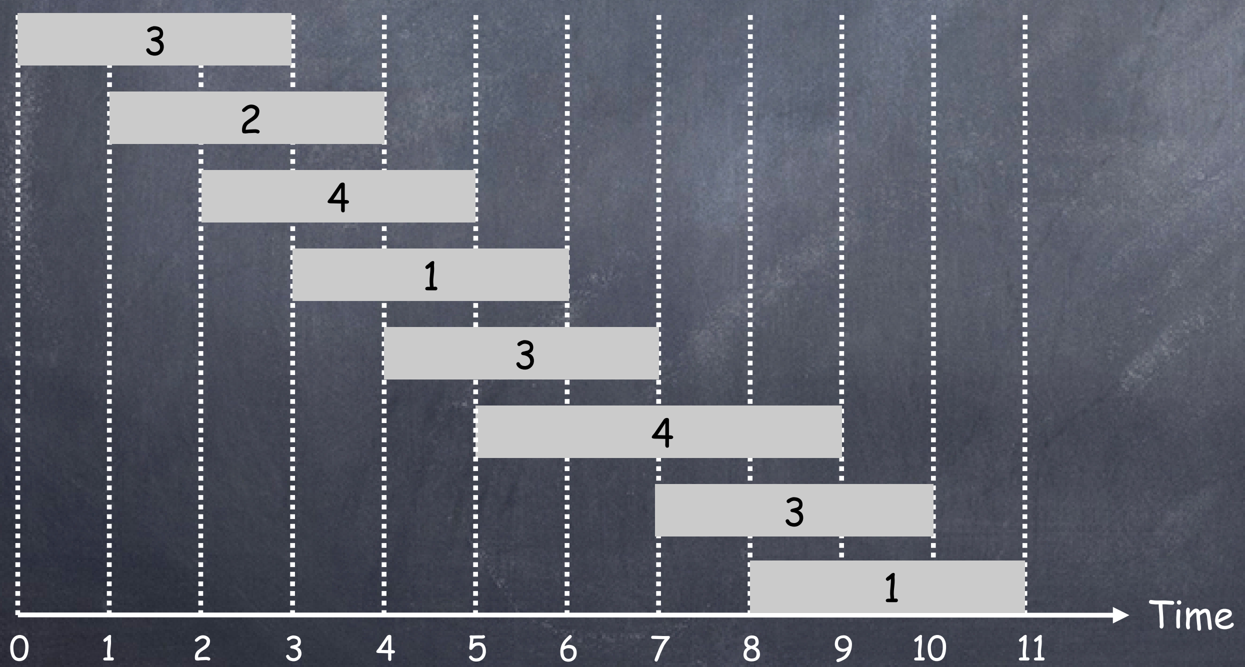
- Sort jobs by earliest finish time.
- Take each job provided it's compatible with the ones already taken.



b, e, h

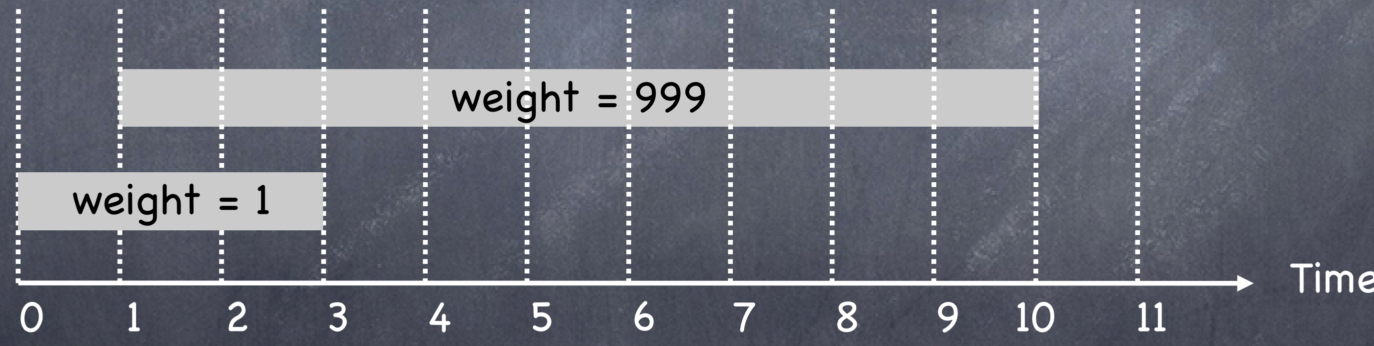
Weighted Interval Scheduling

- Job j starts at s_j, finishes at f_j, and has **weight v_j**.
- Two jobs compatible if they don't overlap.
- Goal: find **maximum weight** subset of mutually compatible jobs.



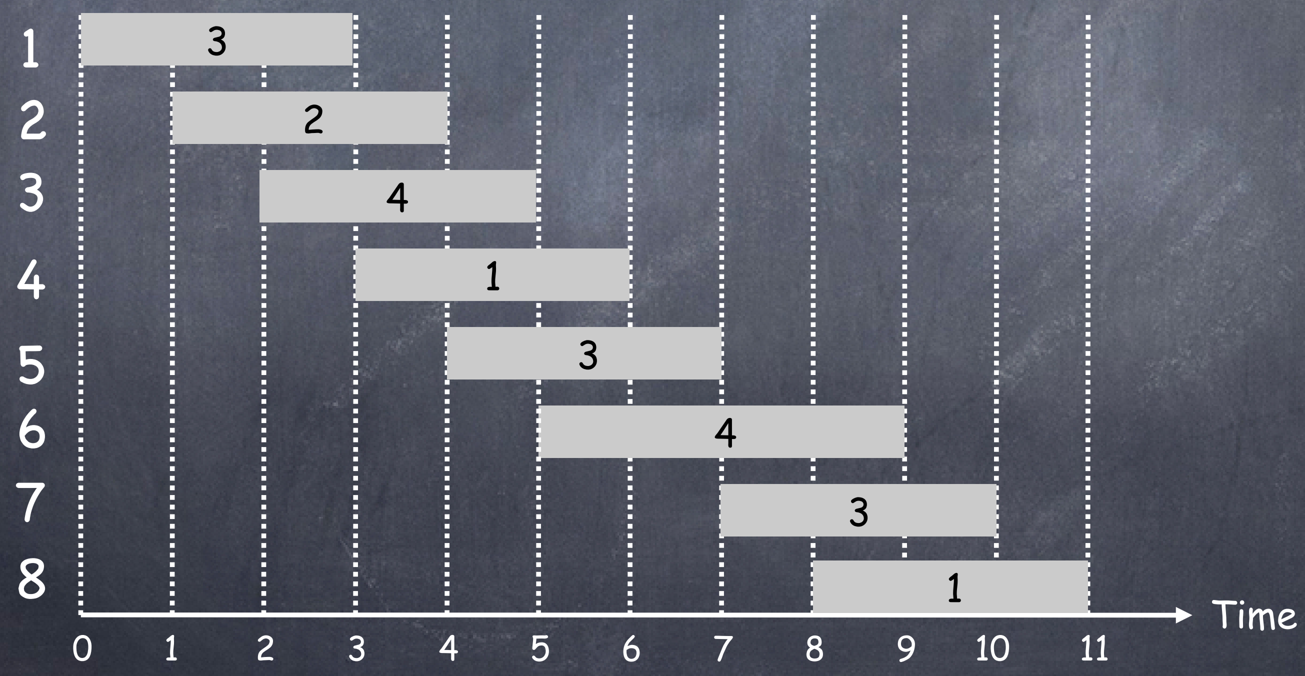
Greedy Solution?

- Observation. Greedy algorithm can be arbitrarily bad when intervals are weighted.



Weighted Interval Scheduling

- Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.
- p(j) = largest index i < j such that job i is compatible with j.
- E.g.: p(8) = 5, p(7) = 5, p(2) = 0.

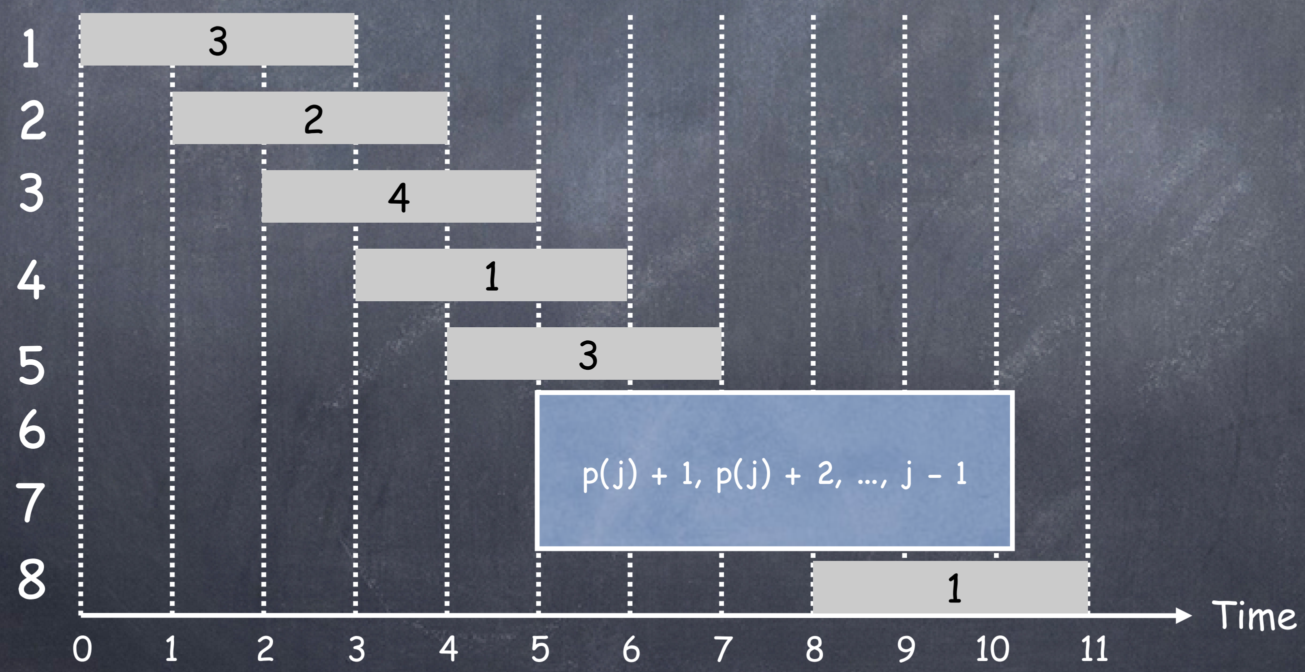


Dynamic Programming: Binary Choice

- OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.
  - Case 1: OPT selects job j.
  - Case 2: OPT does not select job j.

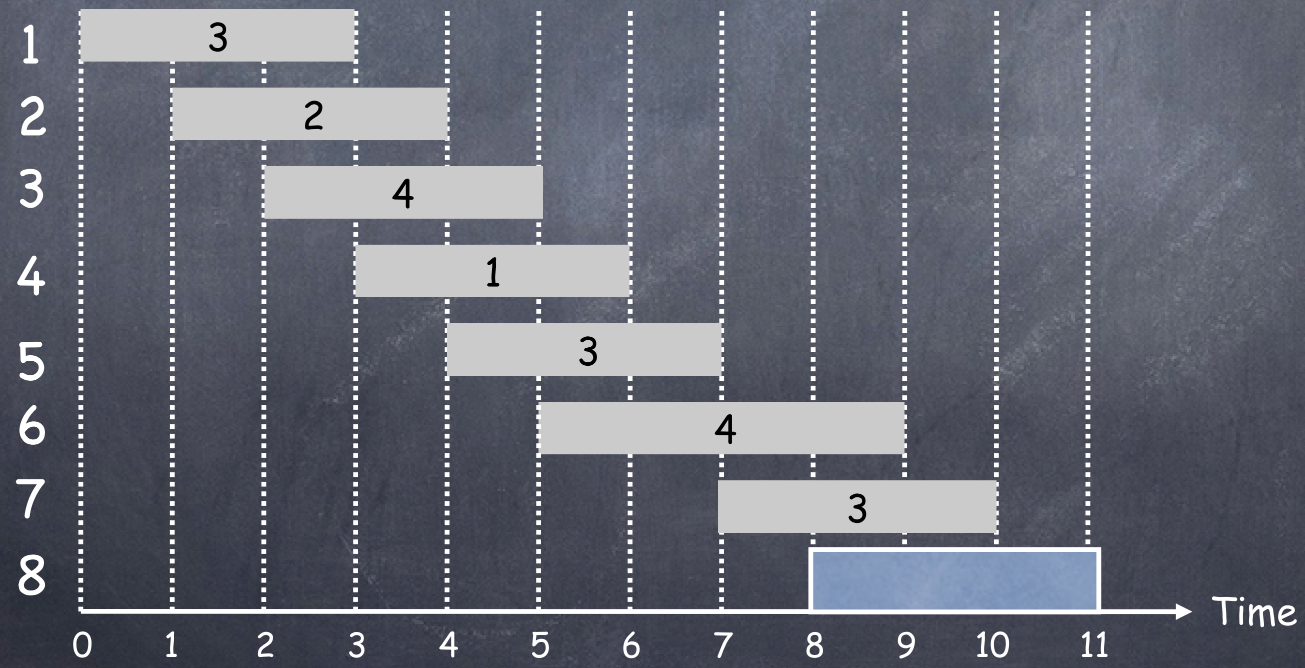
If OPT selects job j...

- can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j – 1 }
- must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)



If OPT does not select job j...

- must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1



Optimal Substructure

OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j

- Case 1: OPT selects job j
- Case 2: OPT does not select job j

Recurrence for OPT(j)

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{ \underbrace{\boxed{v_j + OPT(p(j))}}_{\text{Case 1}},\ \underbrace{\boxed{OPT(j-1)}}_{\text{Case 2}} \} & \text{otherwise} \end{cases}$$

Straightforward Recursive Algorithm

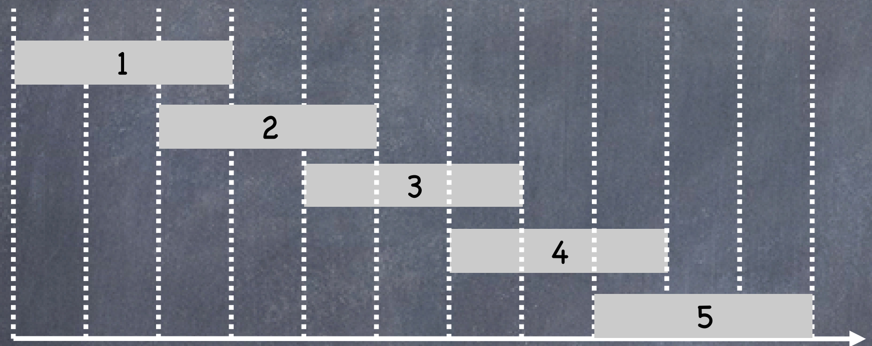
Sort jobs by finish time: $f_1 \le f_2 \le \ldots \le f_n$.

Compute p(1), p(2), ..., p(n)

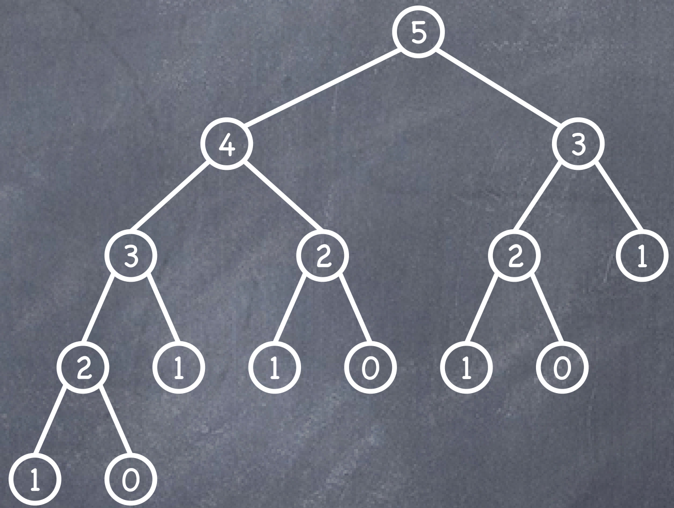
```
Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

Running time?

Worst Case Running Time



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



Worst-case is exponential
How can we do better?

Memoization

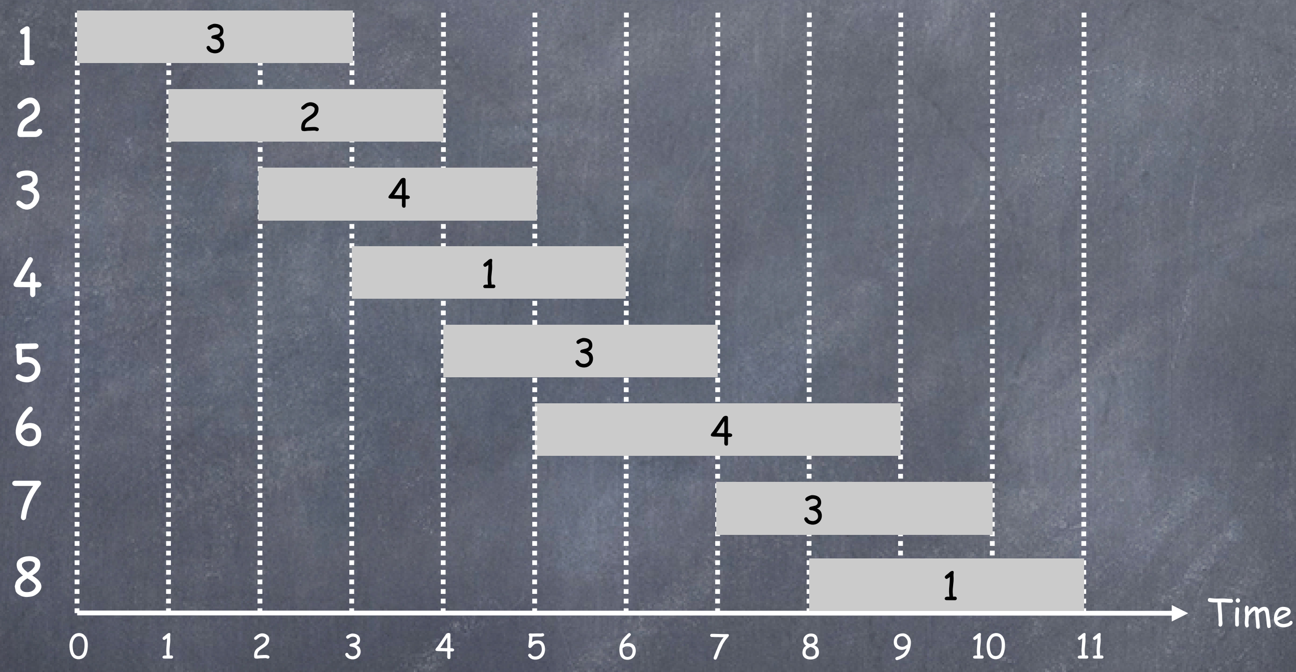
Memoization. Store results of each sub-problem in an array; lookup as needed.

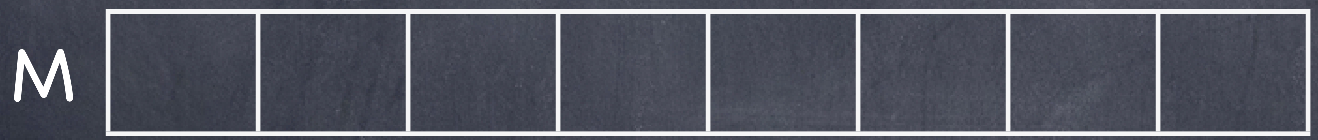
Sort jobs by finish times so that $f_1 \le f_2 \le \dots \le f_n$.
Compute p(1), p(2), …, p(n)

```
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]
}
```

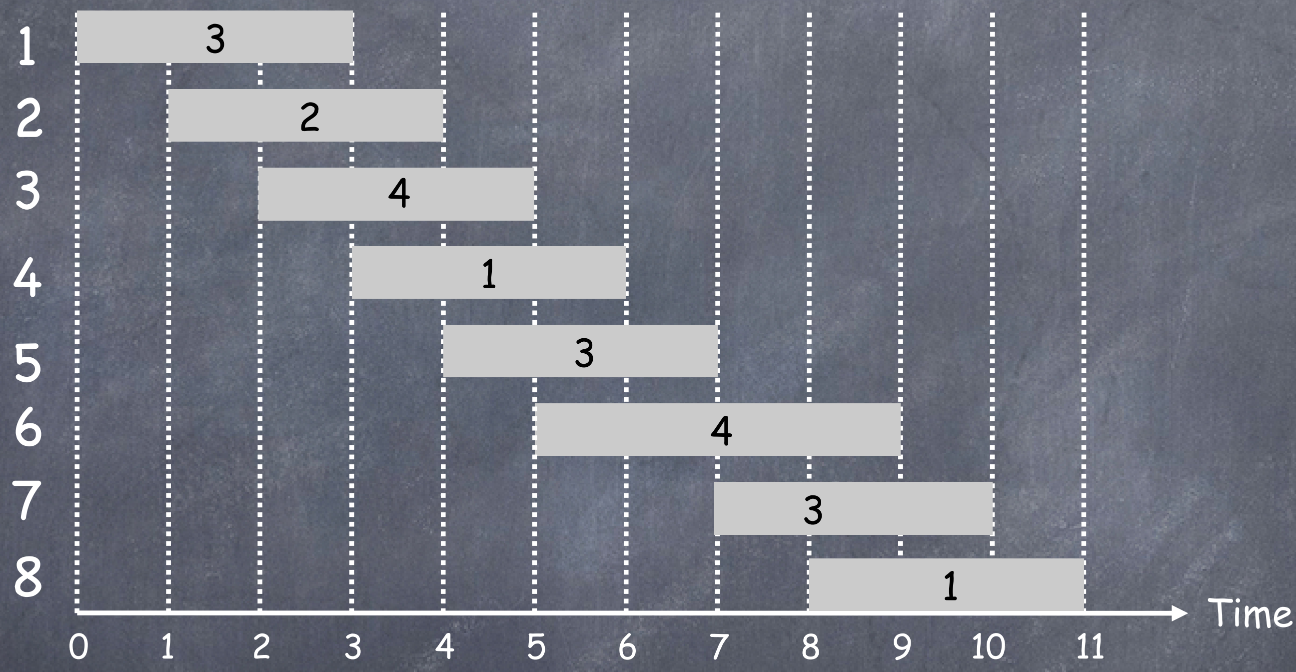
Memoization





M[8] = max(1 + M-Compute-Opt(5), M-Compute-Opt(7))

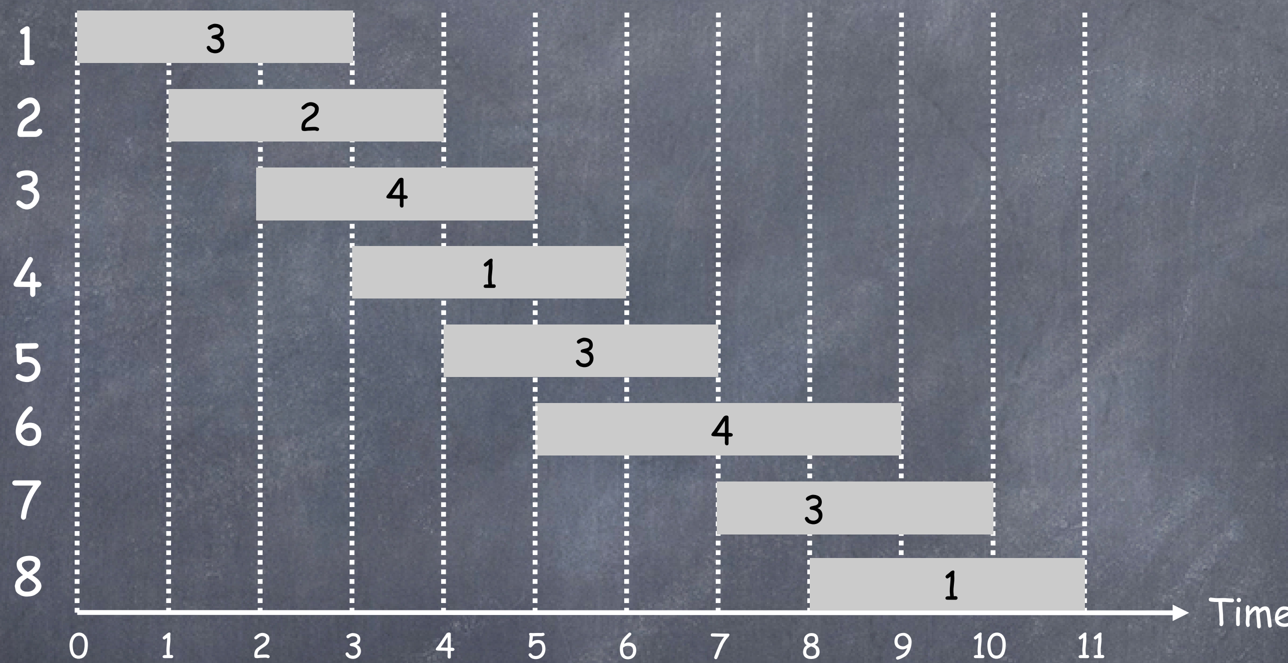
Memoization



M

M[5] = max(3 + M-Compute-Opt(2), M-Compute-Opt(4)

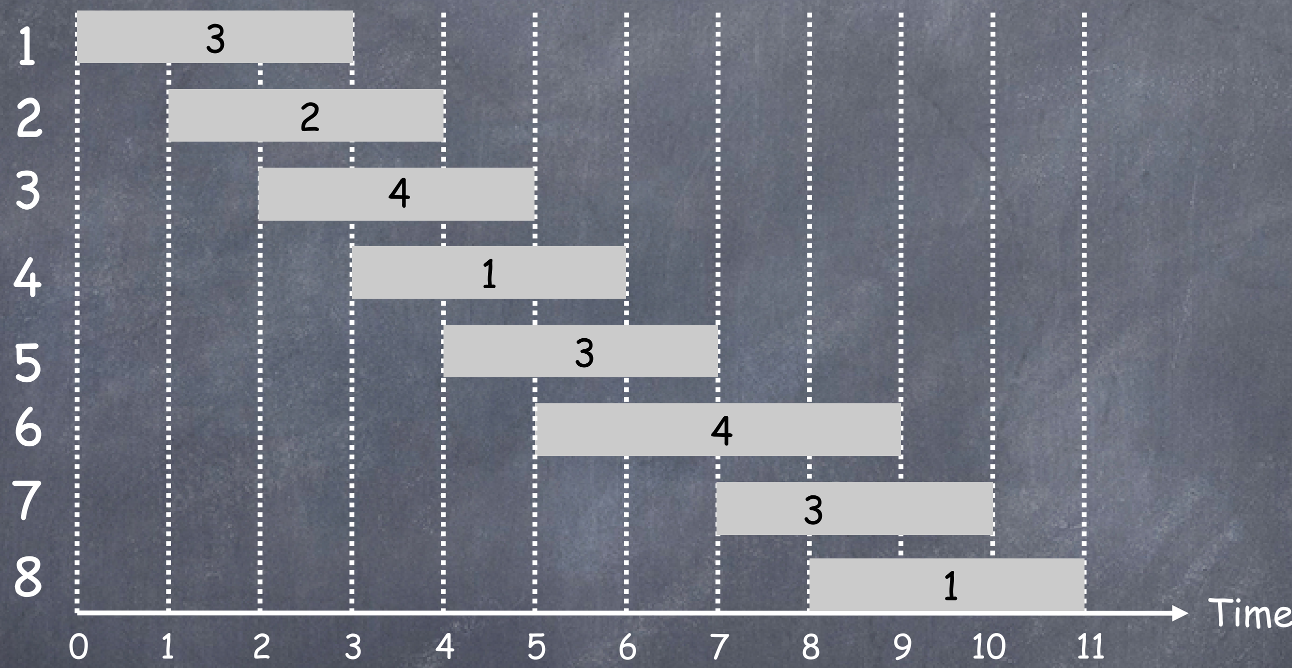
Memoization



M

M[2] = max(2 + 0, M-Compute-Opt(1))

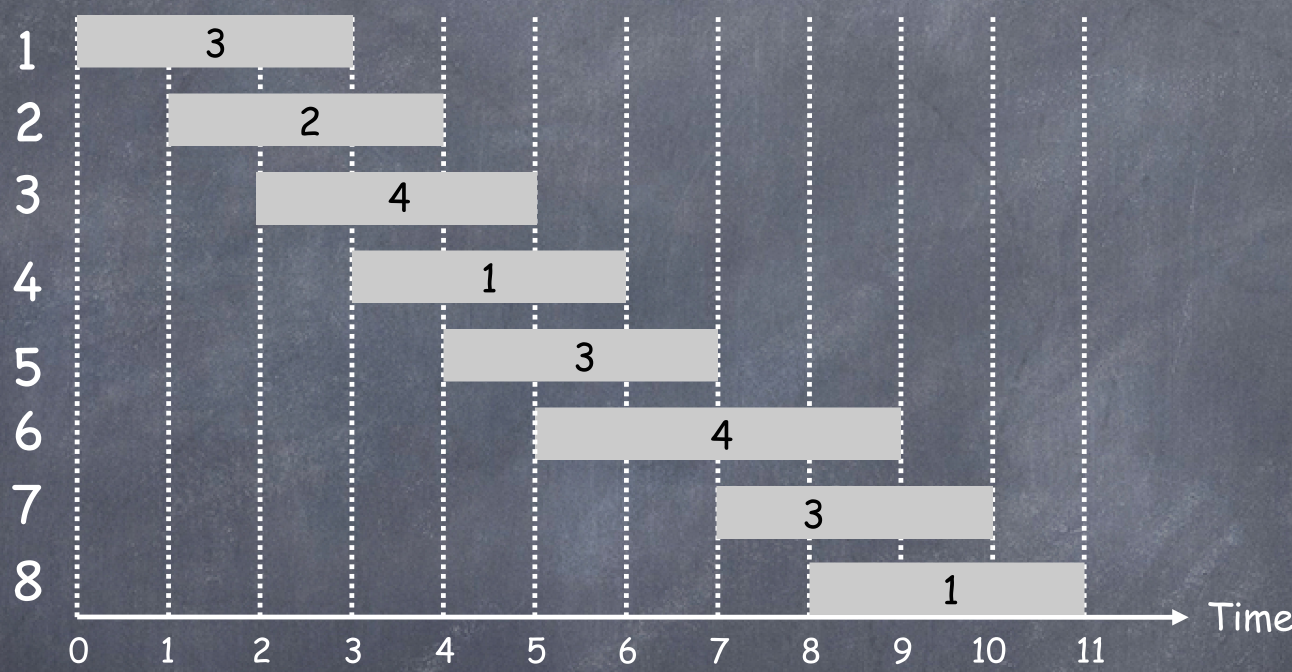
Memoization



M								
	3							

$$M[1] = \max(3 + 0, 0)$$

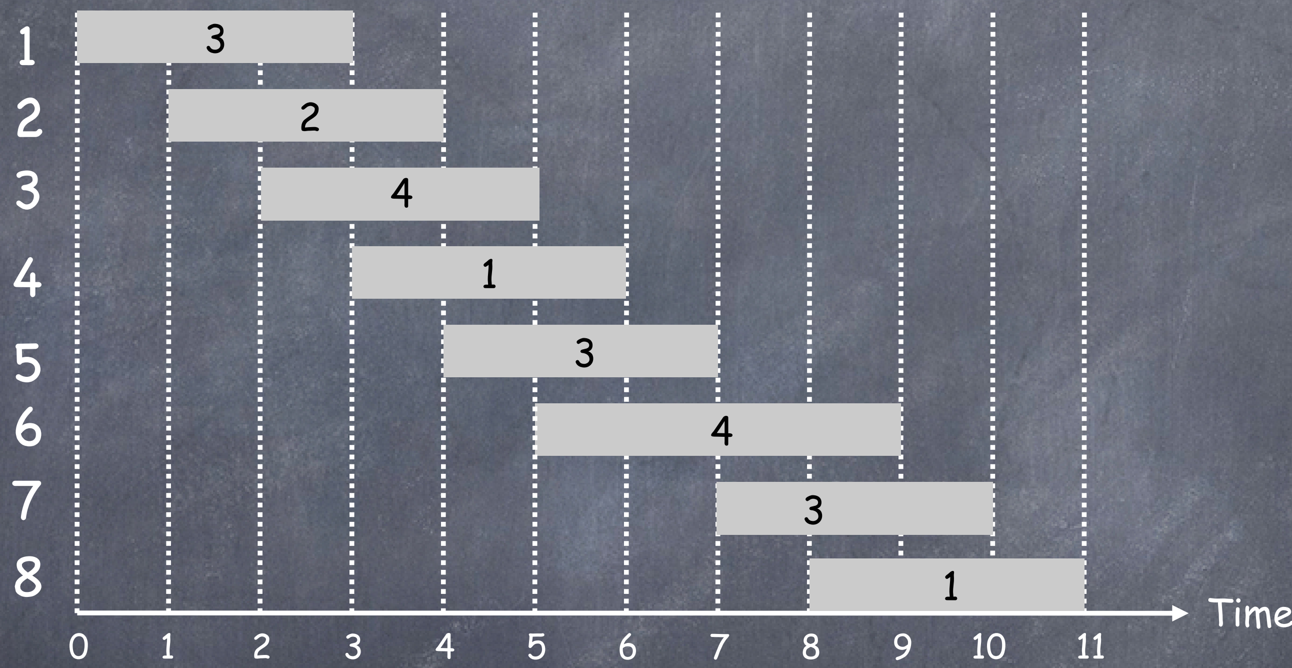
Memoization



M	3	3						

$$M[2] = \max(2 + 0, 3)$$

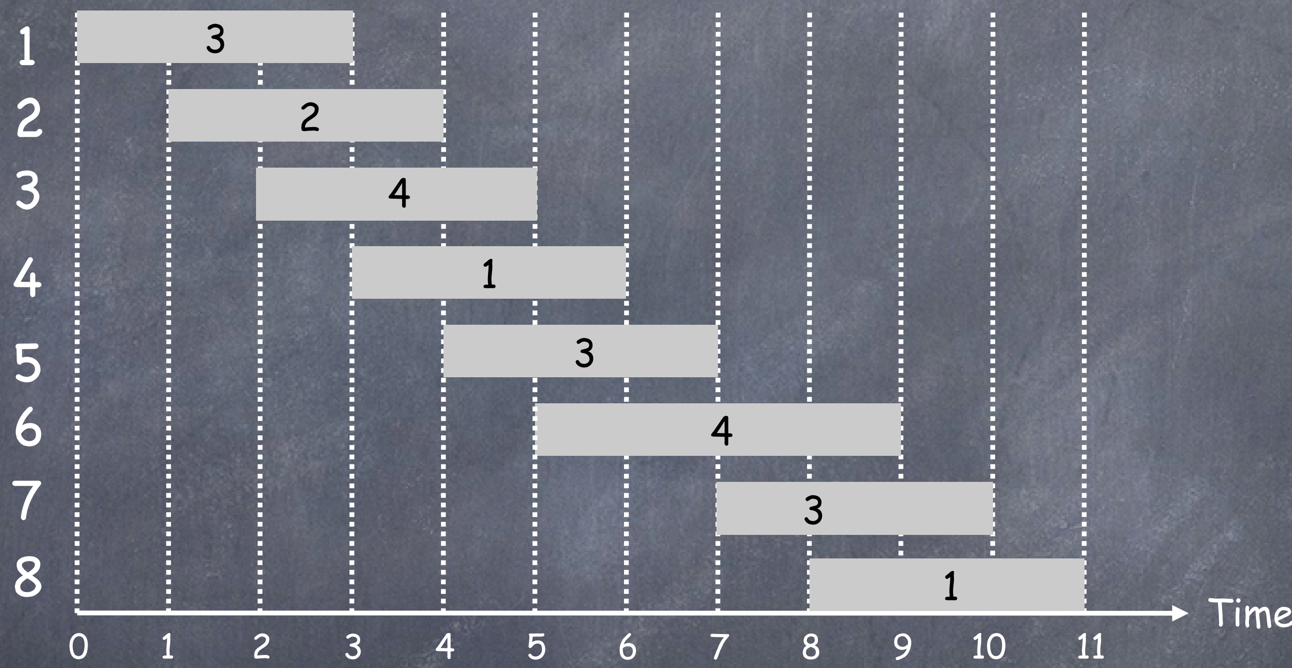
Memoization



M	3	3						

$$M[5] = \max(3 + 3, \text{M-Compute-Opt}(4)$$

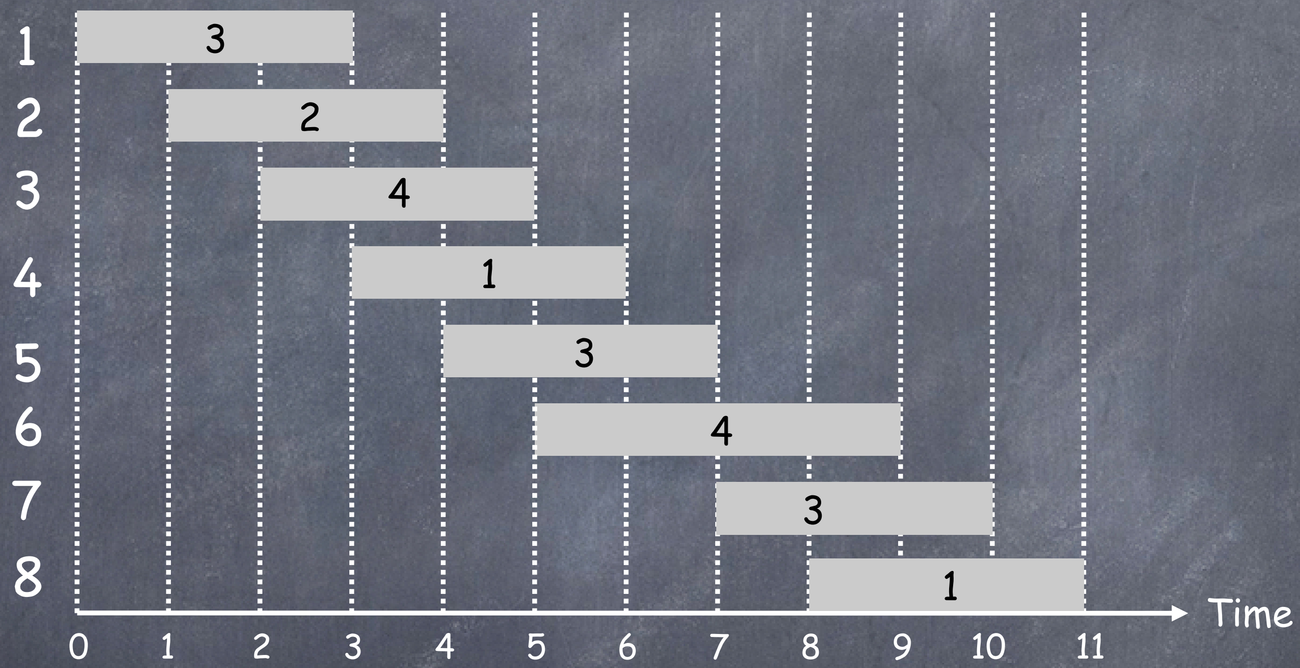
Memoization



M								
M	3	3						

M[4] = max(1 + 3, M-Compute-Opt(3)

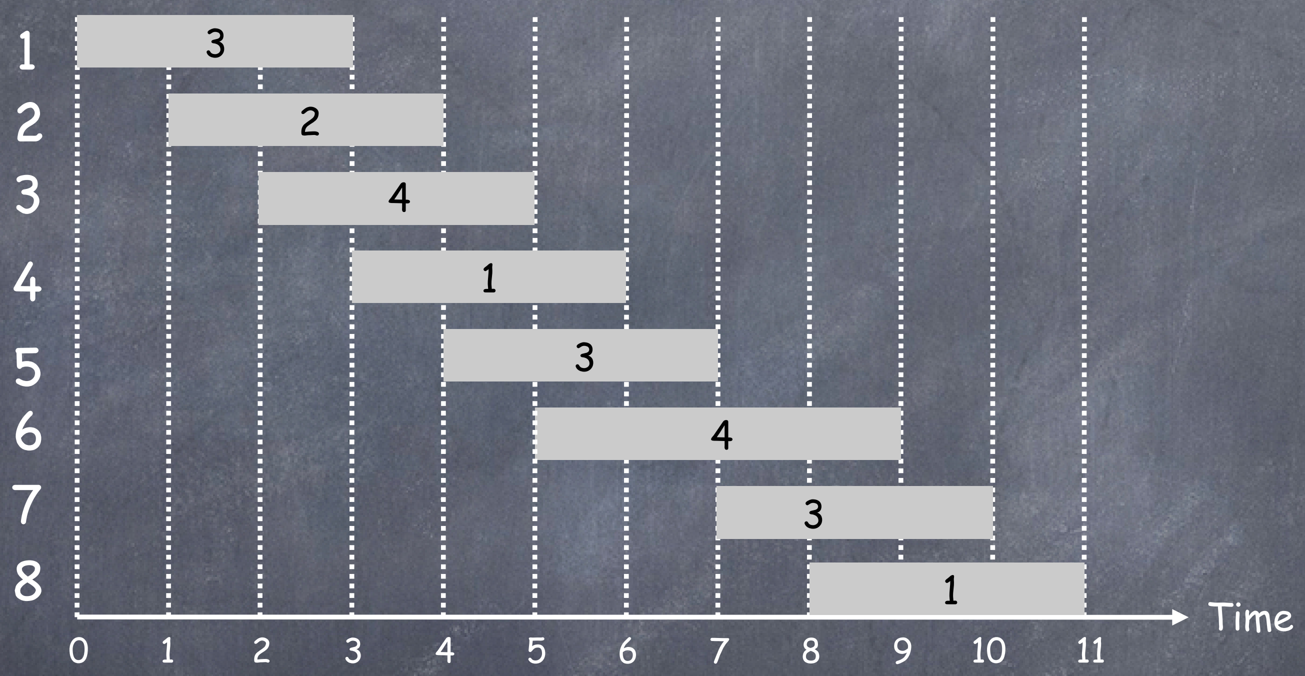
Memoization



M	3	3	4					

M[3] = max(4 + 0, 3)

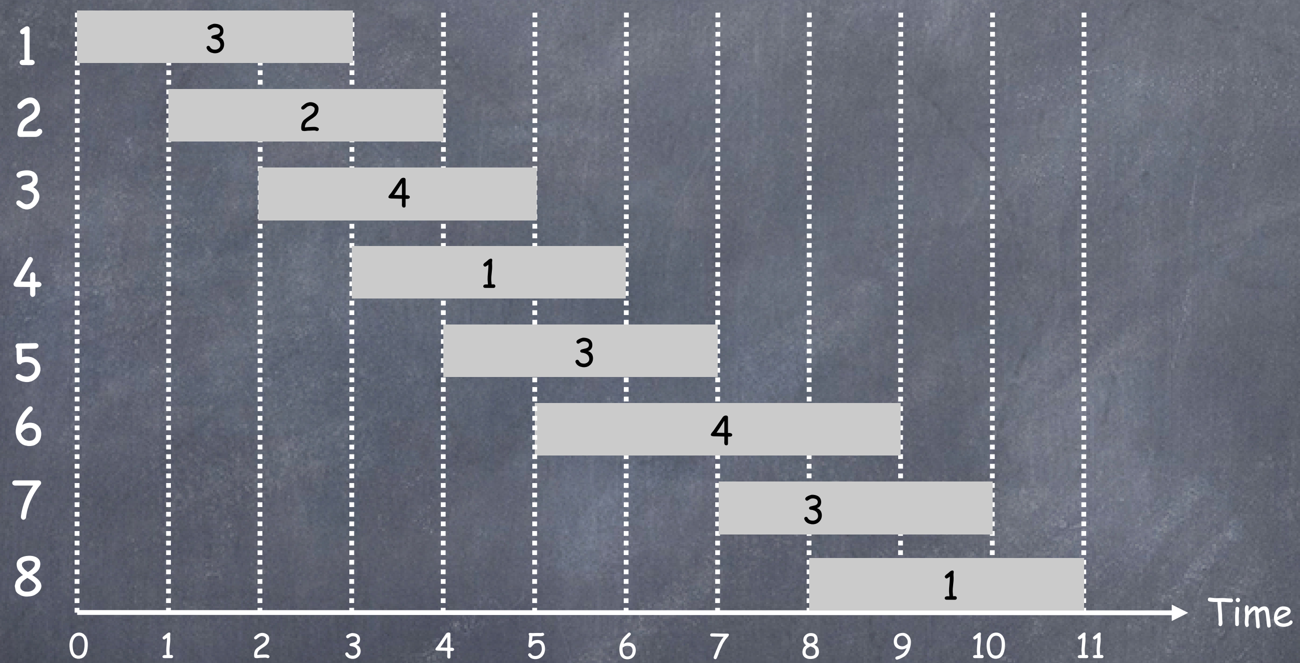
Memoization



M	3	3	4	4				

M[4] = max(1 + 3, 4)

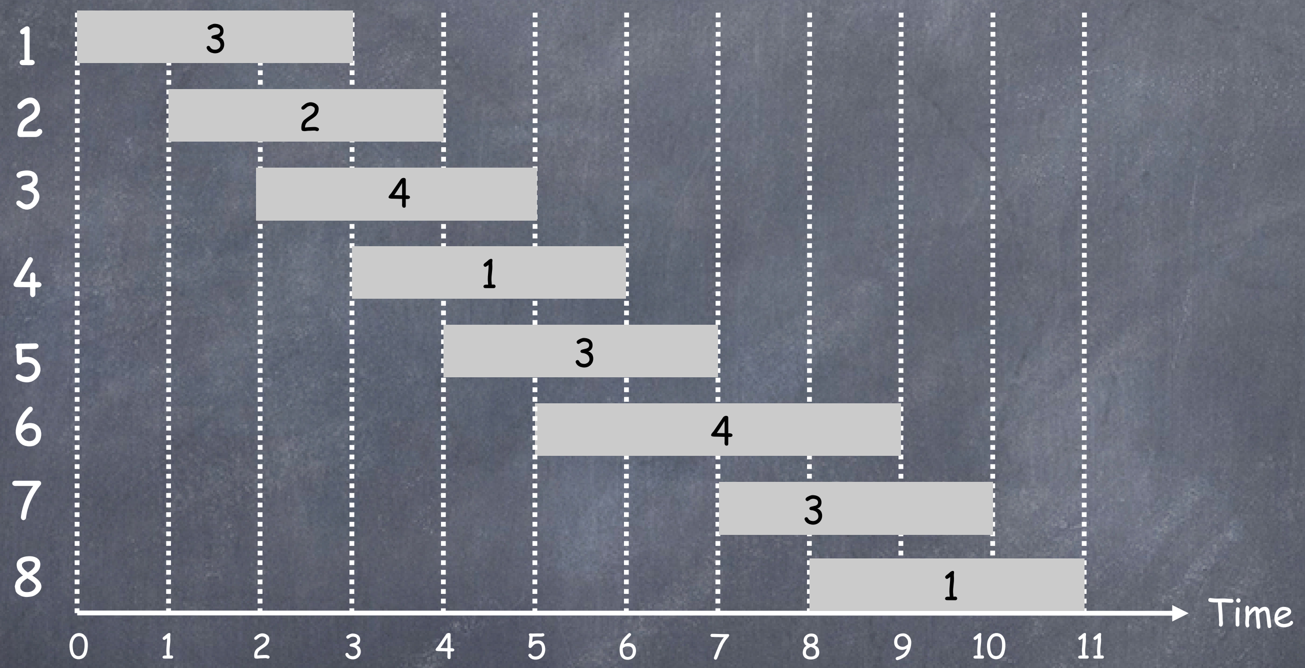
Memoization



M	3	3	4	4	6			

$M[5] = \max(3 + 3, 4)$

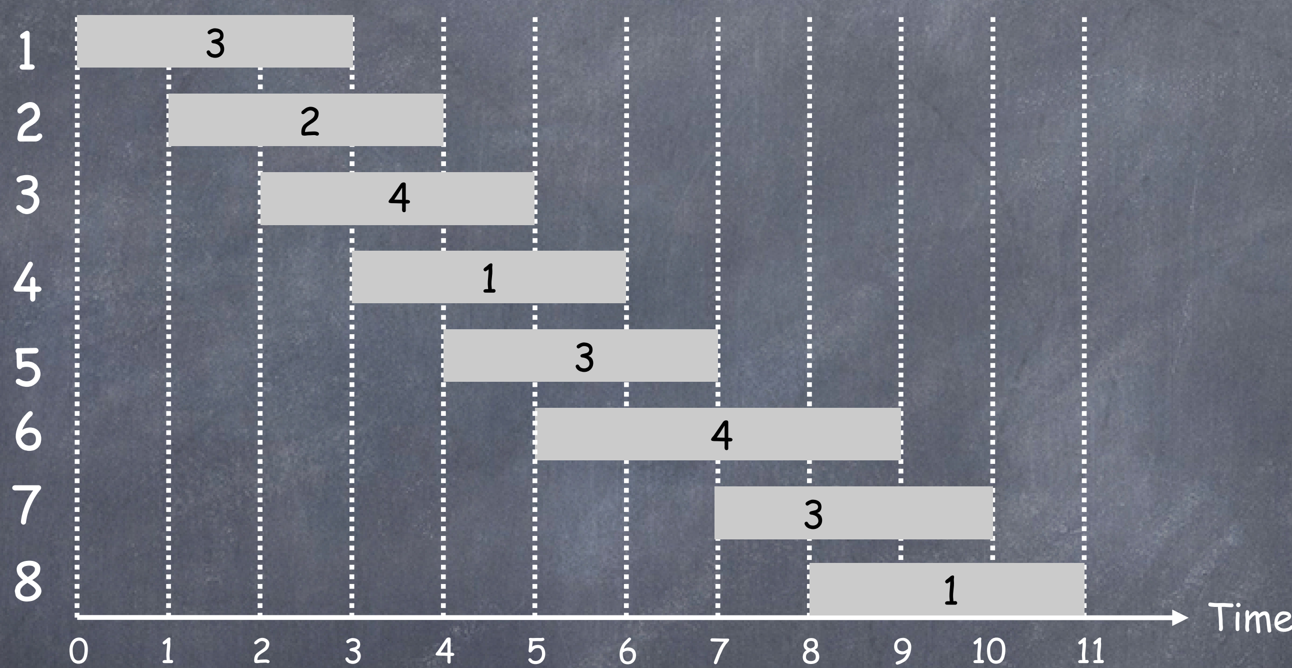
Memoization



M								
	3	3	4	4	6			

M[8] = max(1 + 6, M-Compute-Opt(7)

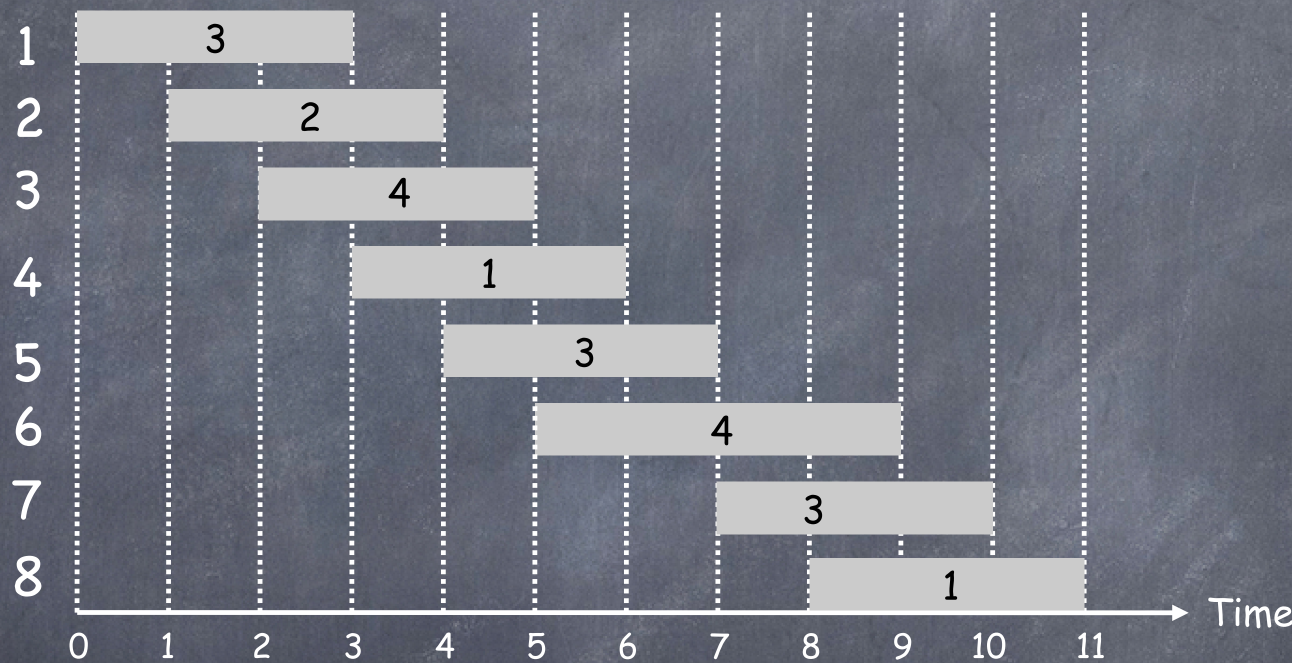
Memoization



M	3	3	4	4	6			

M[7] = max(3 + 6, M-Compute-Opt(6))

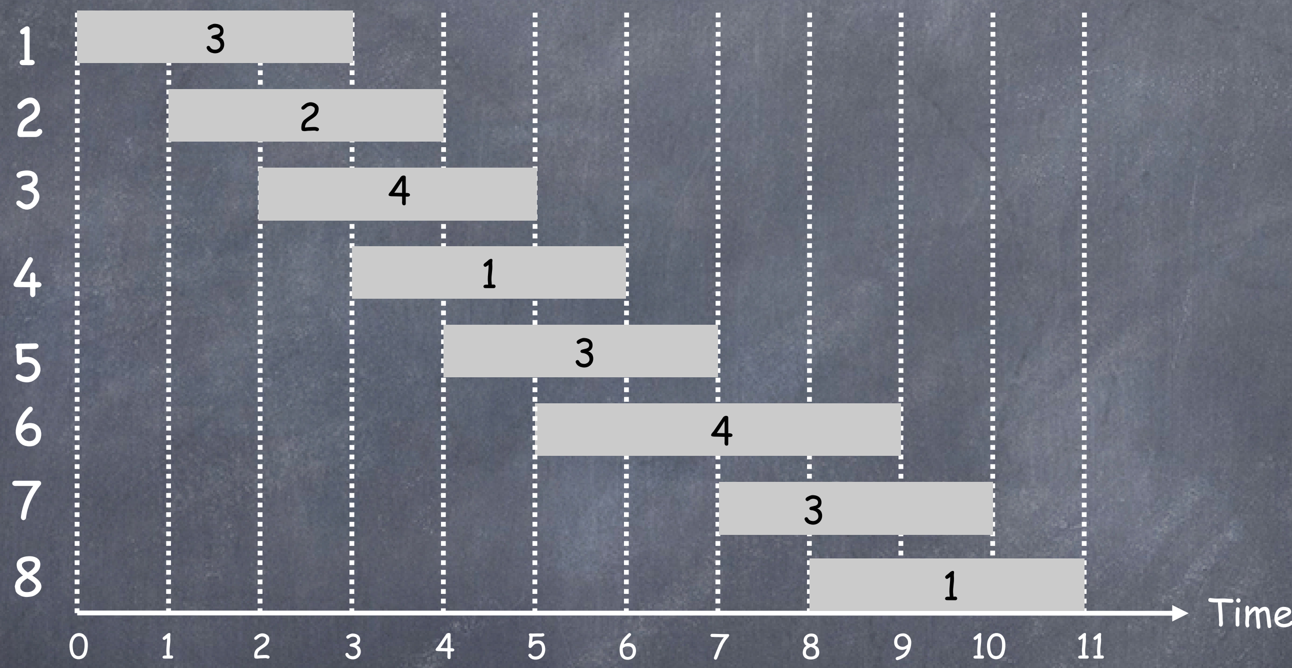
Memoization



M								
M	3	3	4	4	6	8		

$$M[6] = \max(4 + 4, 6)$$

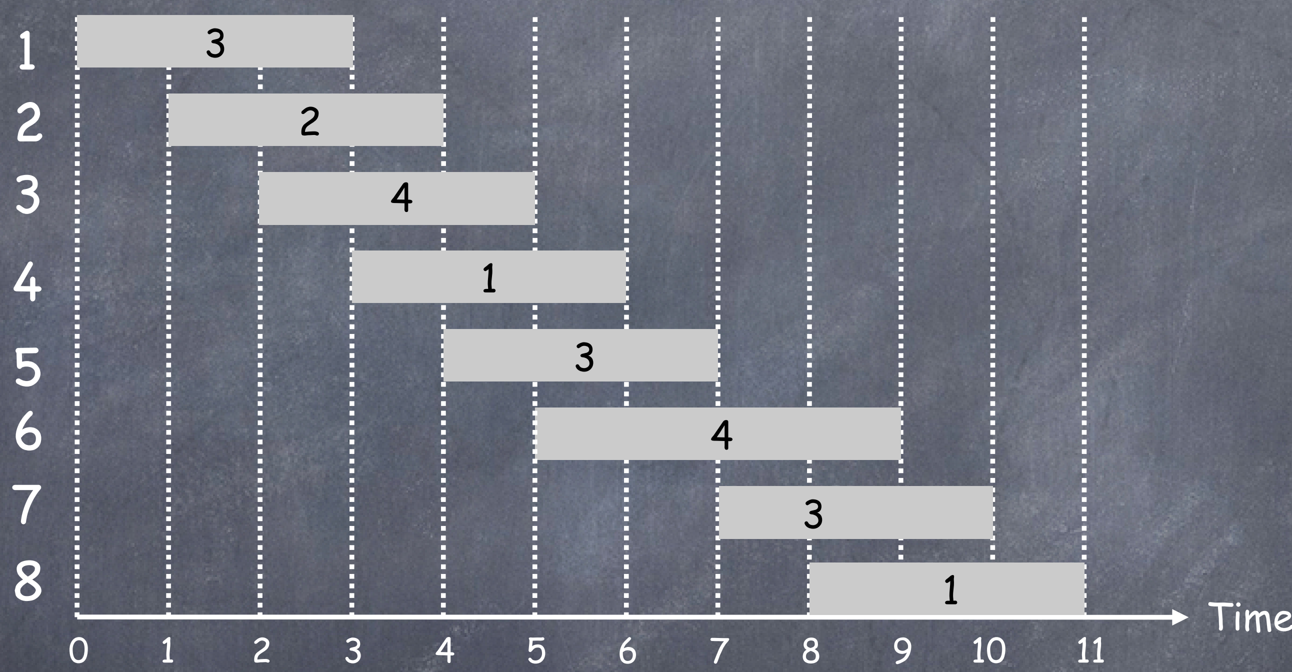
Memoization



M	3	3	4	4	6	8	9	

$$M[7] = \max(3 + 6, 8)$$

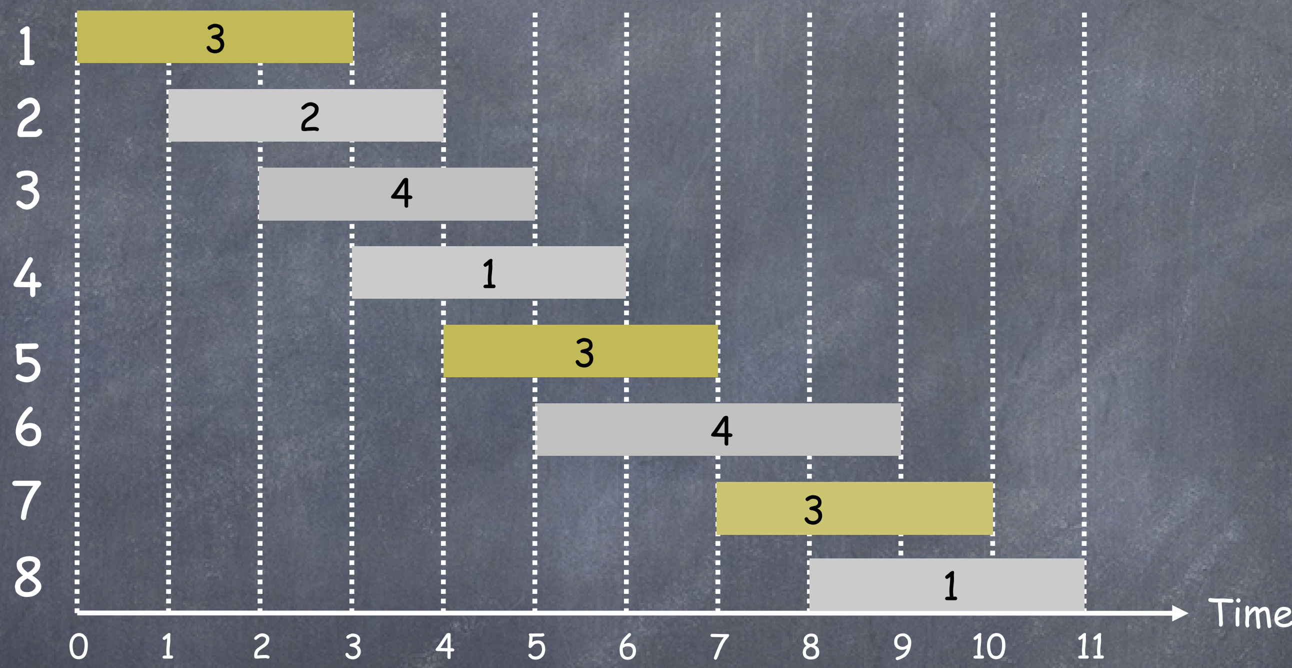
Memoization



M	3	3	4	4	6	8	9	9

$$M[8] = \max(1 + 6, 9)$$

Memoization



M	3	3	4	4	6	8	9	9

Running Time?

Sort jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

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]
}
```

Iterative Solution

Bottom-up dynamic programming. Solve subproblems in ascending order.

Sort jobs by finish time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute p(1), p(2), ..., p(n)

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

Compute the Solution (Not Just Its Value)

- Exercise: suppose you know the value OPT(j) for all j.
- How can you produce the set of intervals in the optimal solution?

Dynamic Programming "Recipe"

- Recursive formulation of optimal solution in terms of subproblems
- Obvious implementation requires solving exponentially many subproblems
- Careful implementation to solve only polynomially many different subproblems