# Plan

Dynamic Programming

# Algorithmic Paradigms

**Greed.** 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.

# 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"

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, systems, ….

Some famous dynamic programming algorithms.

- Viterbi for hidden Markov models.
- 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 Mantra

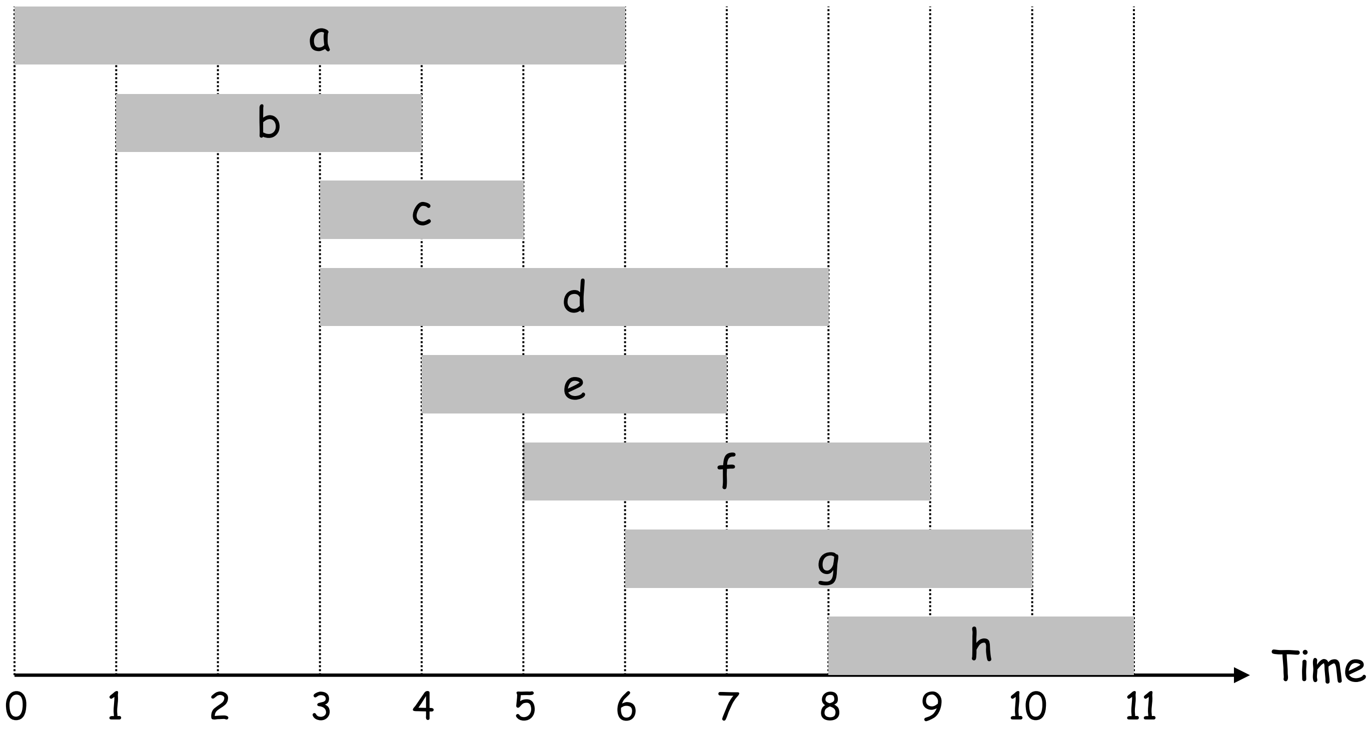
- Express OPT in terms of OPT for smaller problems [just like divide and conquer]
- Figure out a clever order to evaluate all sub-problems to minimize redundancy [pictures help!]

# 6.1 Weighted Interval Scheduling

# Weighted Interval Scheduling

Weighted interval scheduling problem.

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

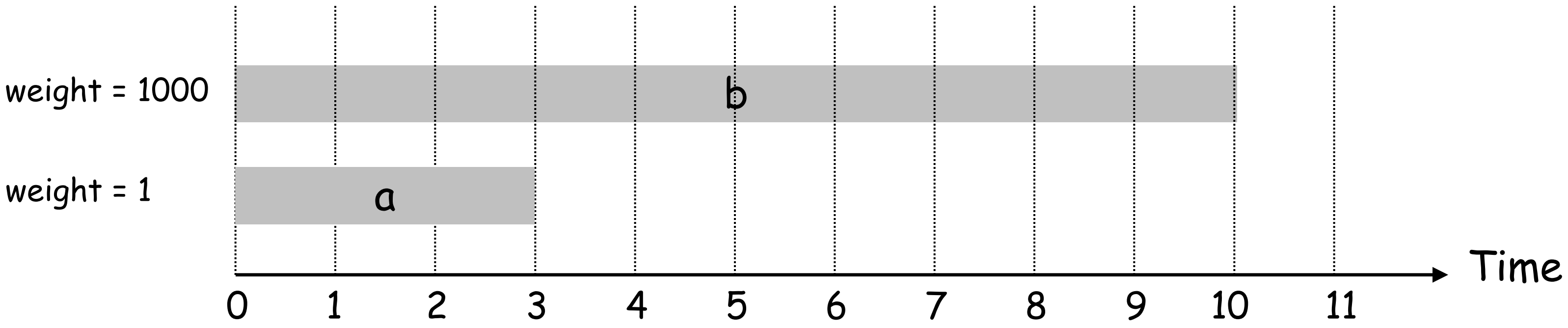


# Unweighted Interval Scheduling Review

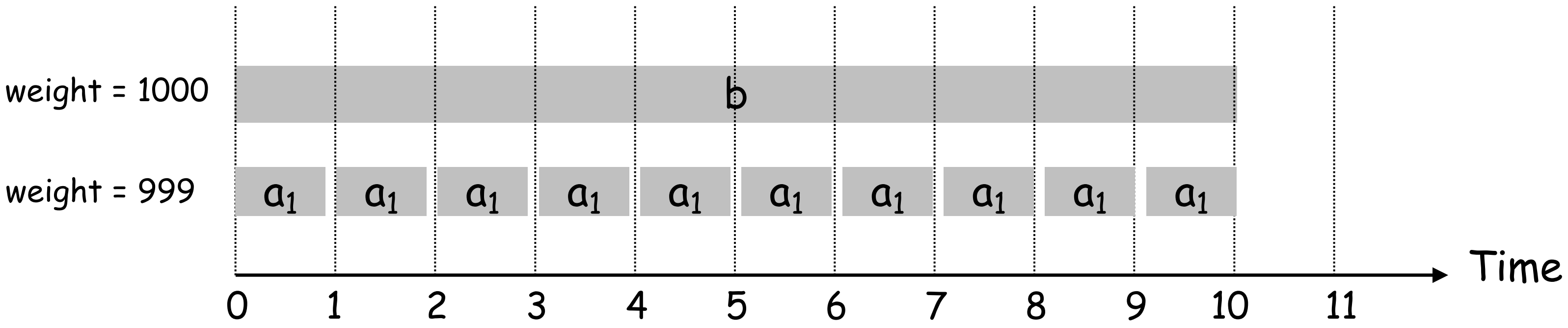
**Recall.** Greedy algorithm works if all weights are 1.

- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

**Observation.** Greedy algorithm can fail spectacularly if arbitrary weights are allowed.



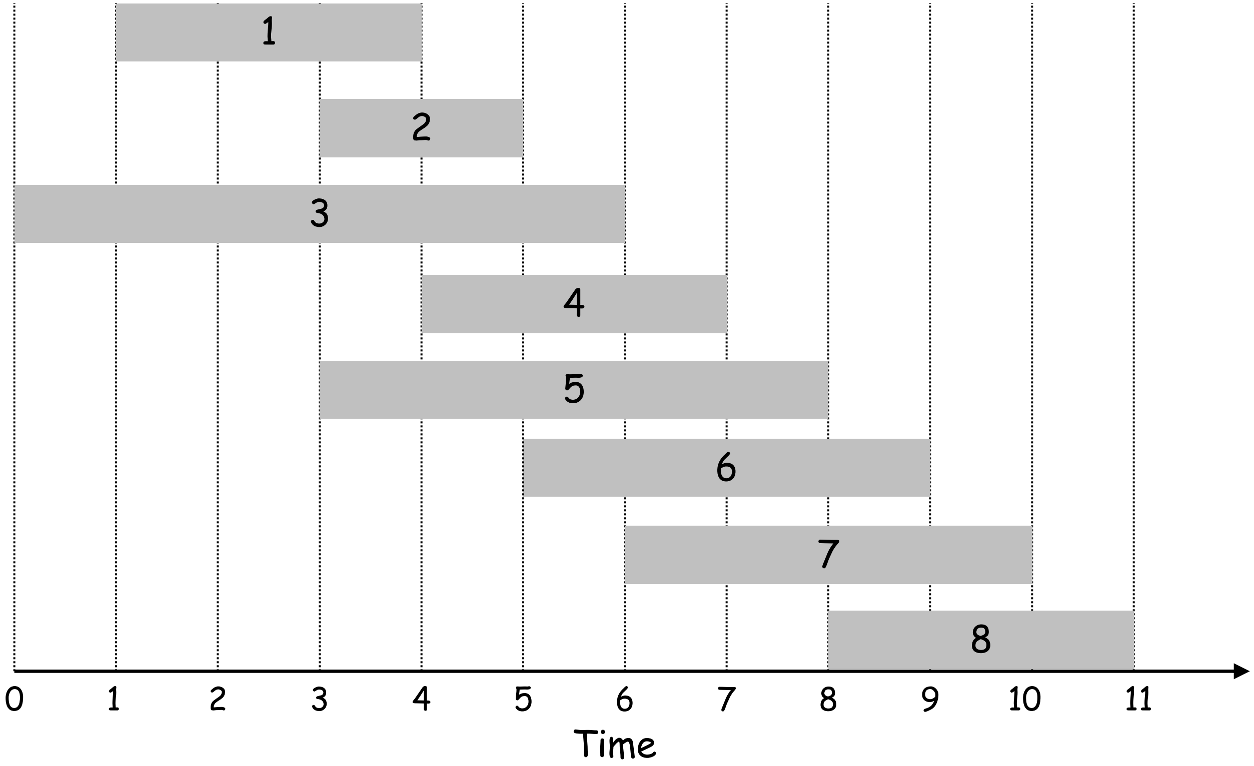
by finish



by weight

# Weighted Interval Scheduling

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

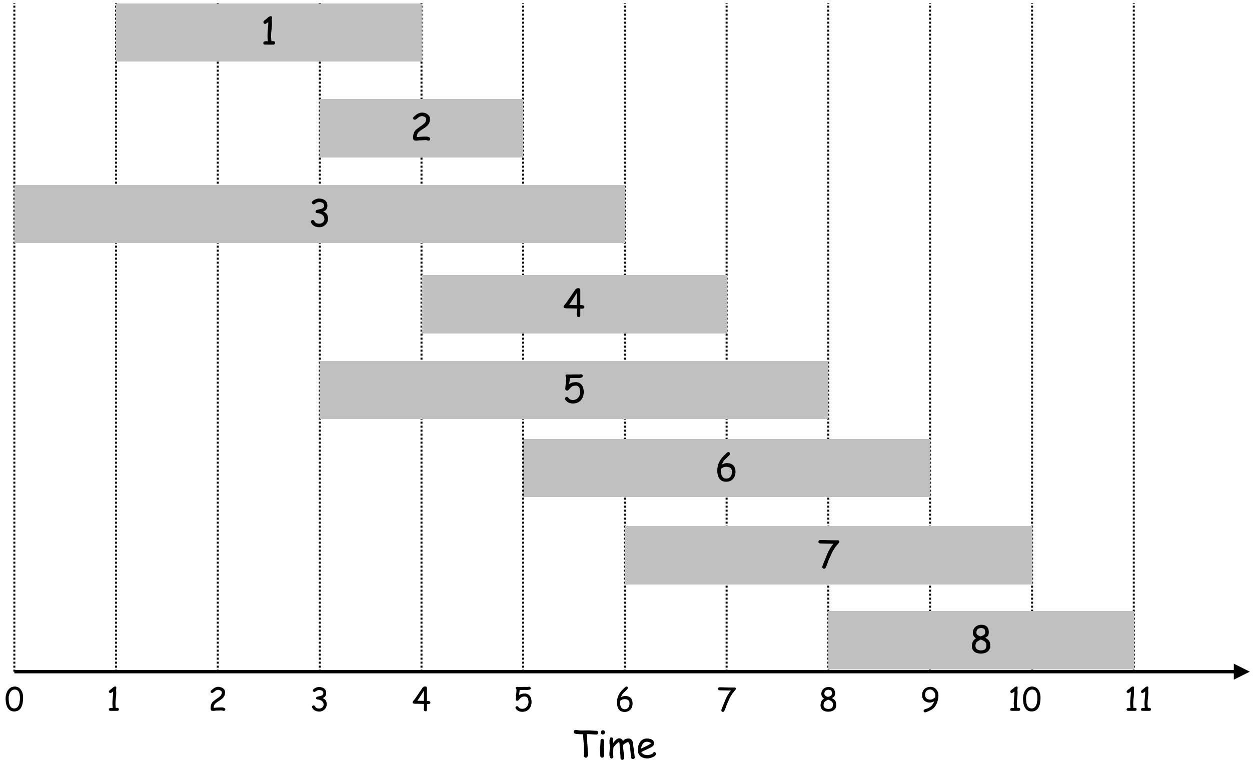


# Weighted Interval Scheduling

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

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

**Ex:** p(8) = 5, p(7) = 3, p(2) = 0.



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

# Using Subproblems

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

- Case 1: 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)

optimal substructure

- Case 2: OPT does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

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

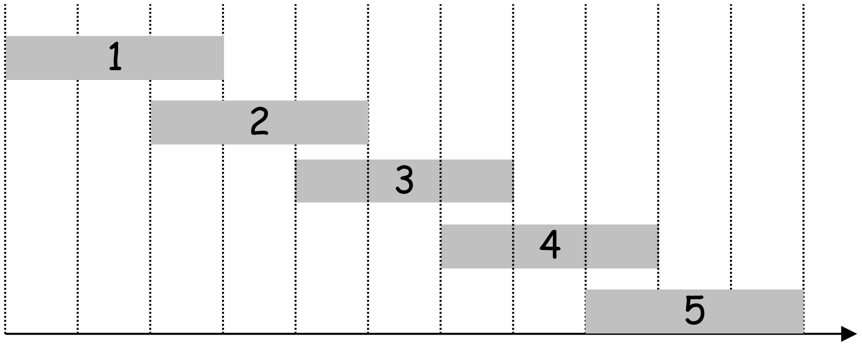
# Recursive Algorithm

```
Input: n, $s_1,…,s_n$ , $f_1,…,f_n$ , $v_1,…,v_n$

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

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))
}
```

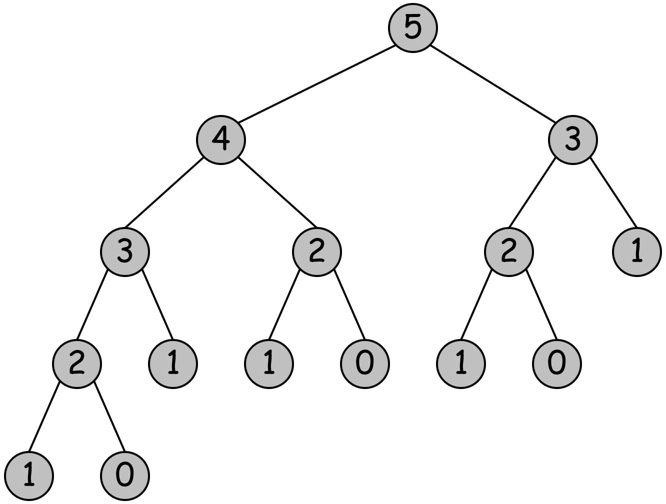
# Weighted Interval Scheduling: Recursive Algorithm

**Observation.** Recursive algorithm fails spectacularly because of redundant sub-problems.

**Ex.** Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



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



# Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

```
Input: n, $s_1,…,s_n$ , $f_1,…,f_n$ , $v_1,…,v_n$

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

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])
}

Output M[n]
```

Claim: M[j] is value of optimal solution for jobs 1..j

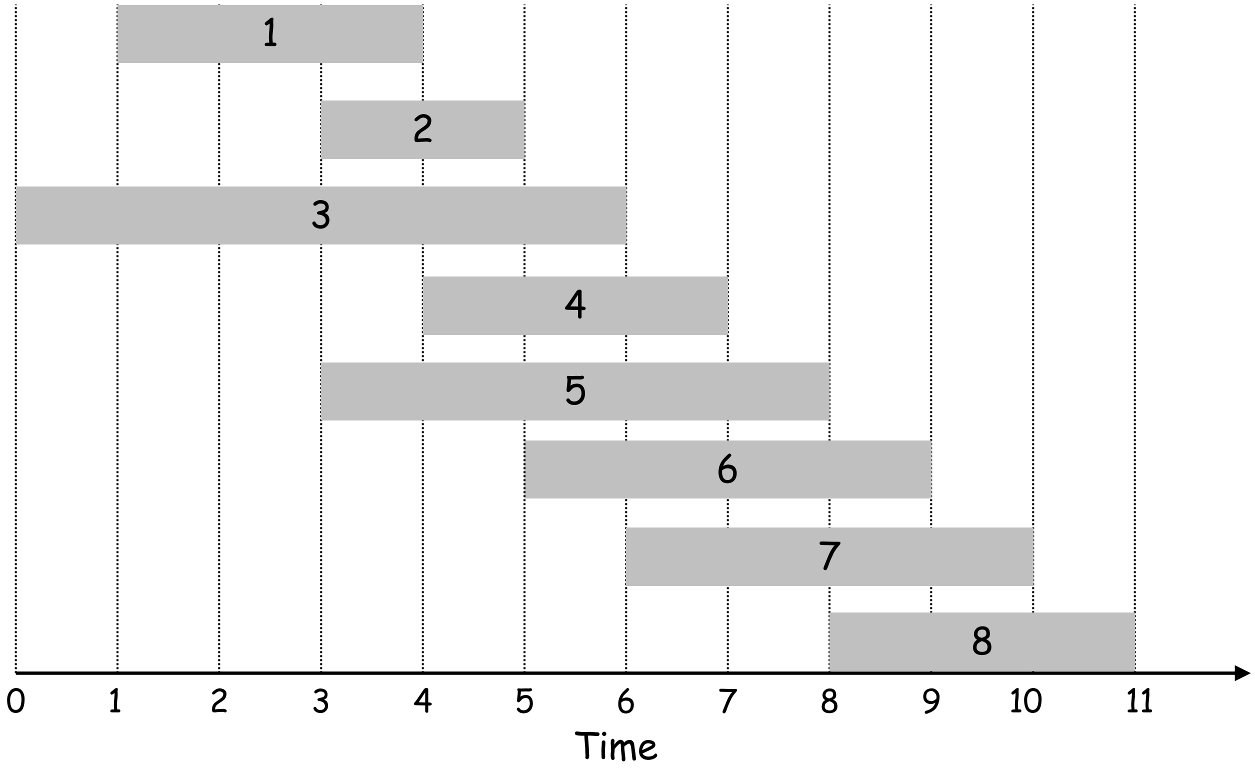
Timing: p(j)'s can be computed in O(n log n) time. Main loop is O(n); sorting is O(n log n)

# Weighted Interval Scheduling

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

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

**Ex:** p(8) = 5, p(7) = 3, p(2) = 0.

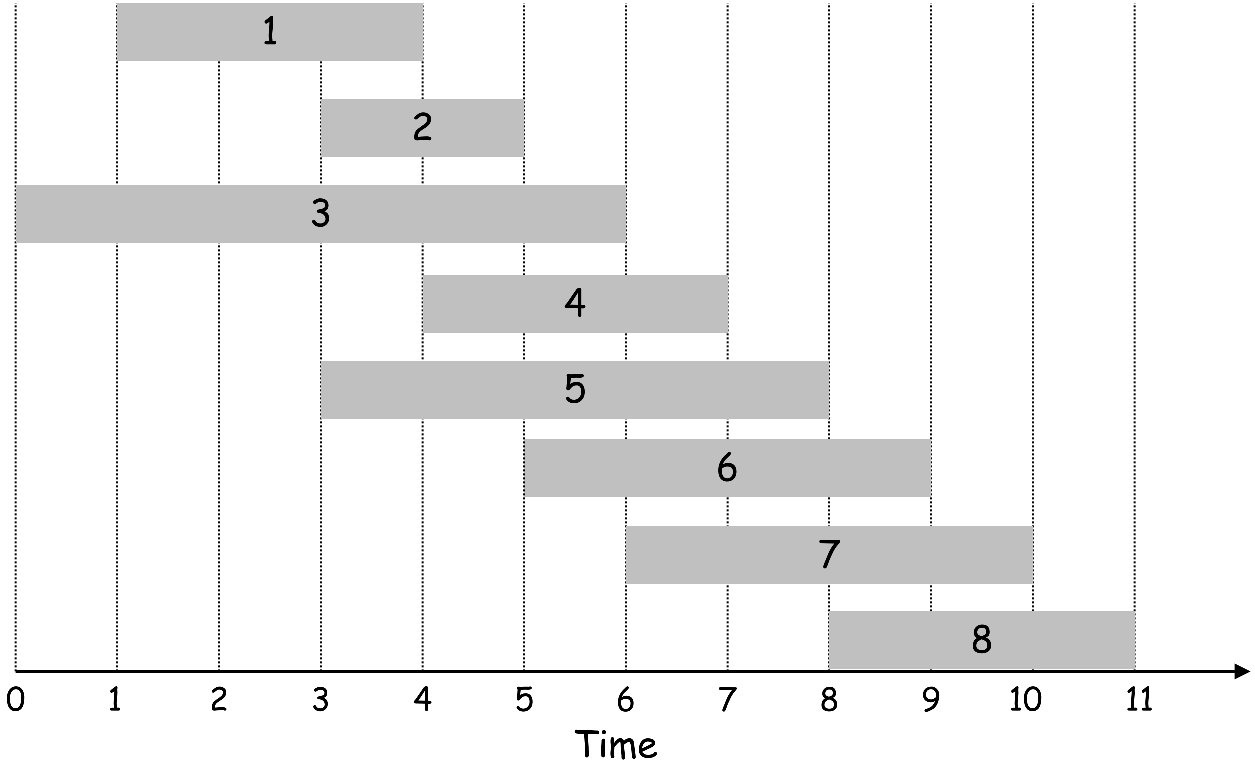


| j | vj | pj | optj |
|---|---|---|---|
| 0 |  | - | 0 |
| 1 | 3 | 0 |  |
| 2 | 4 | 0 |  |
| 3 | 1 | 0 |  |
| 4 | 3 | 1 |  |
| 5 | 4 | 0 |  |
| 6 | 3 | 2 |  |
| 7 | 2 | 3 |  |
| 8 | 4 | 5 |  |

# Weighted Interval Scheduling

**Notation.** Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.
**Def.** p(j) = largest index i < j such that job i is compatible with j.

**Ex:** p(8) = 5, p(7) = 3, p(2) = 0.



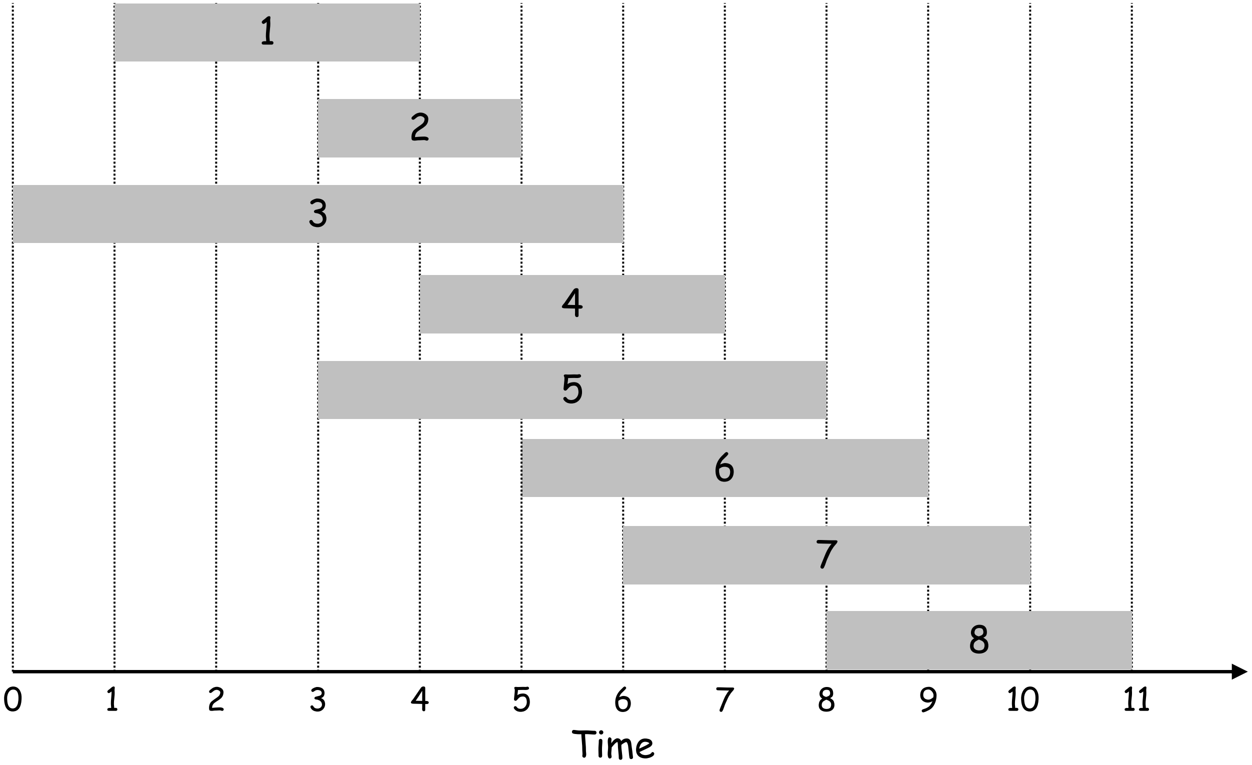
| j | vj | pj | optj |
|---|---|---|---|
| 0 | 0 | - | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | |
| 3 | 1 | 0 | |
| 4 | 3 | 1 | |
| 5 | 4 | 0 | |
| 6 | 3 | 2 | |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Weighted Interval Scheduling

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

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

**Ex:** p(8) = 5, p(7) = 3, p(2) = 0.



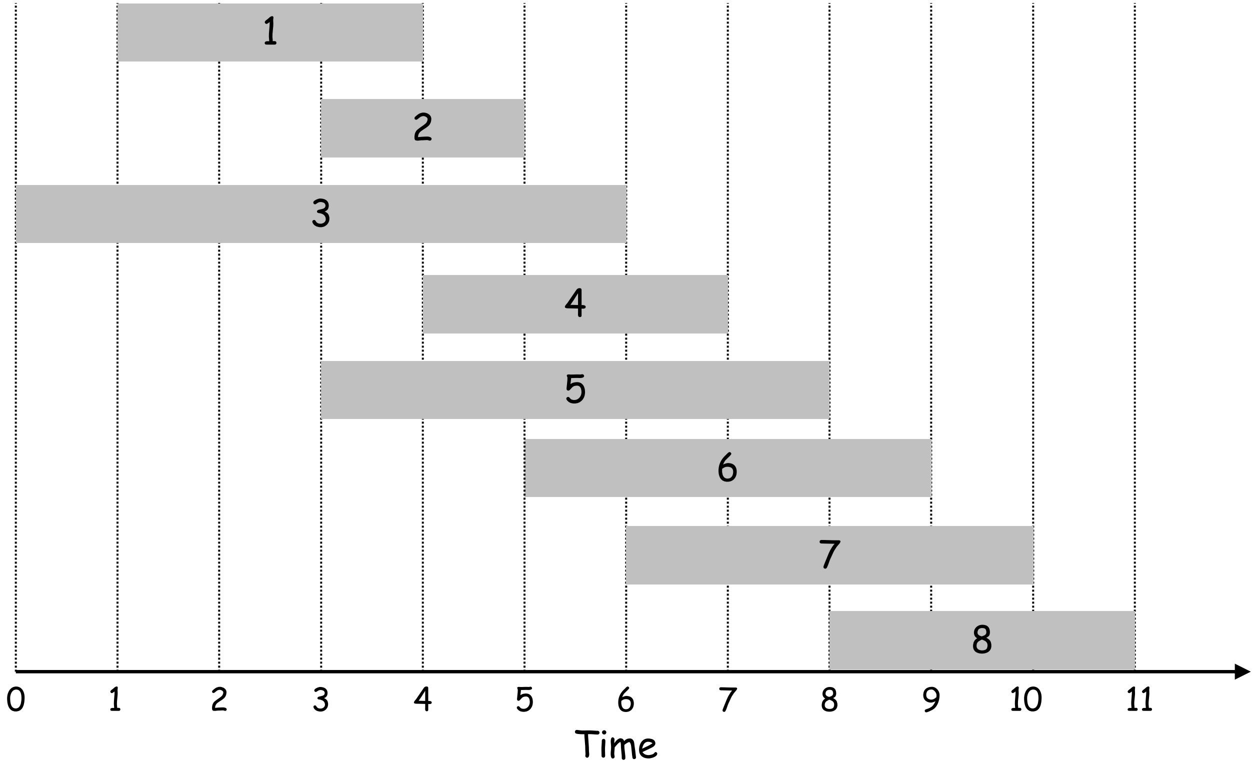
| j | vj | pj | optj |
|---|---|---|---|
| 0 | 0 | - | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 |  |
| 3 | 1 | 0 |  |
| 4 | 3 | 1 |  |
| 5 | 4 | 0 |  |
| 6 | 3 | 2 |  |
| 7 | 2 | 3 |  |
| 8 | 4 | 5 |  |

# Weighted Interval Scheduling

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

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

**Ex:** p(8) = 5, p(7) = 3, p(2) = 0.



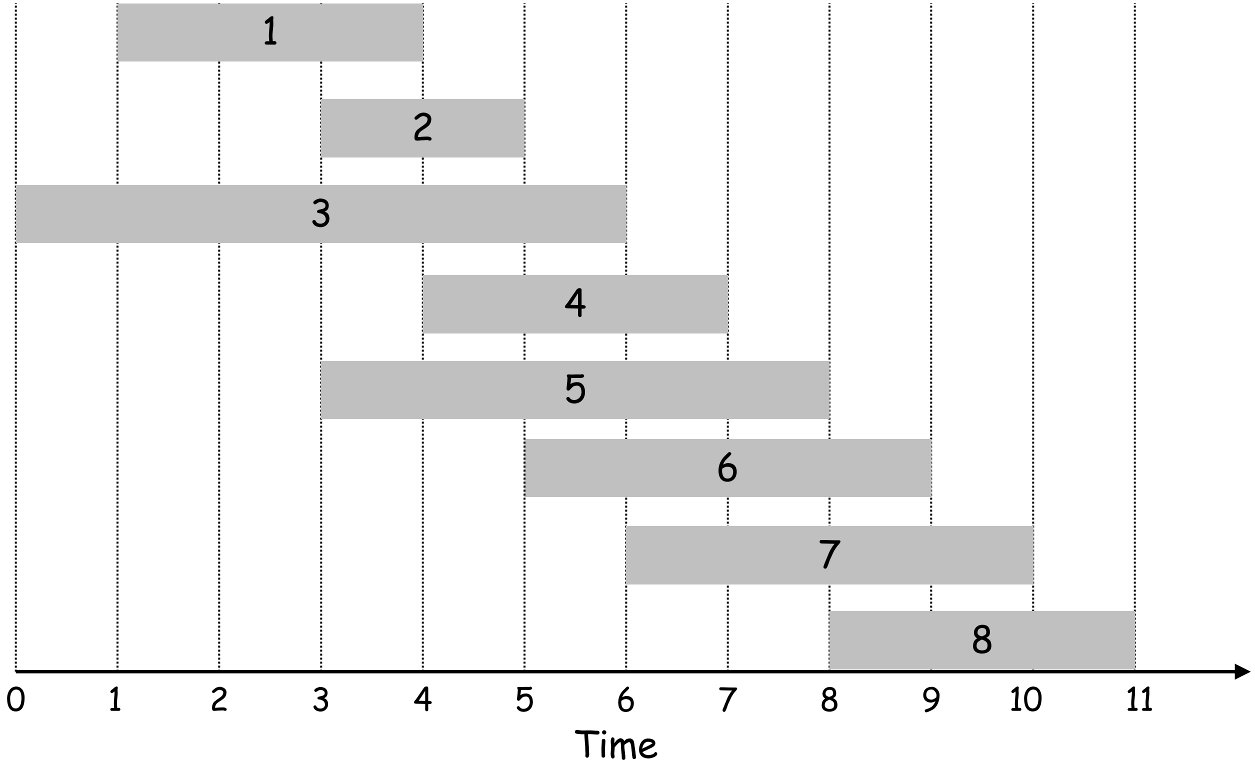
| j | vj | pj | optj |
|---|---|---|---|
| 0 | 0 | - | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | |
| 4 | 3 | 1 | |
| 5 | 4 | 0 | |
| 6 | 3 | 2 | |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Weighted Interval Scheduling

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

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

**Ex:** p(8) = 5, p(7) = 3, p(2) = 0.



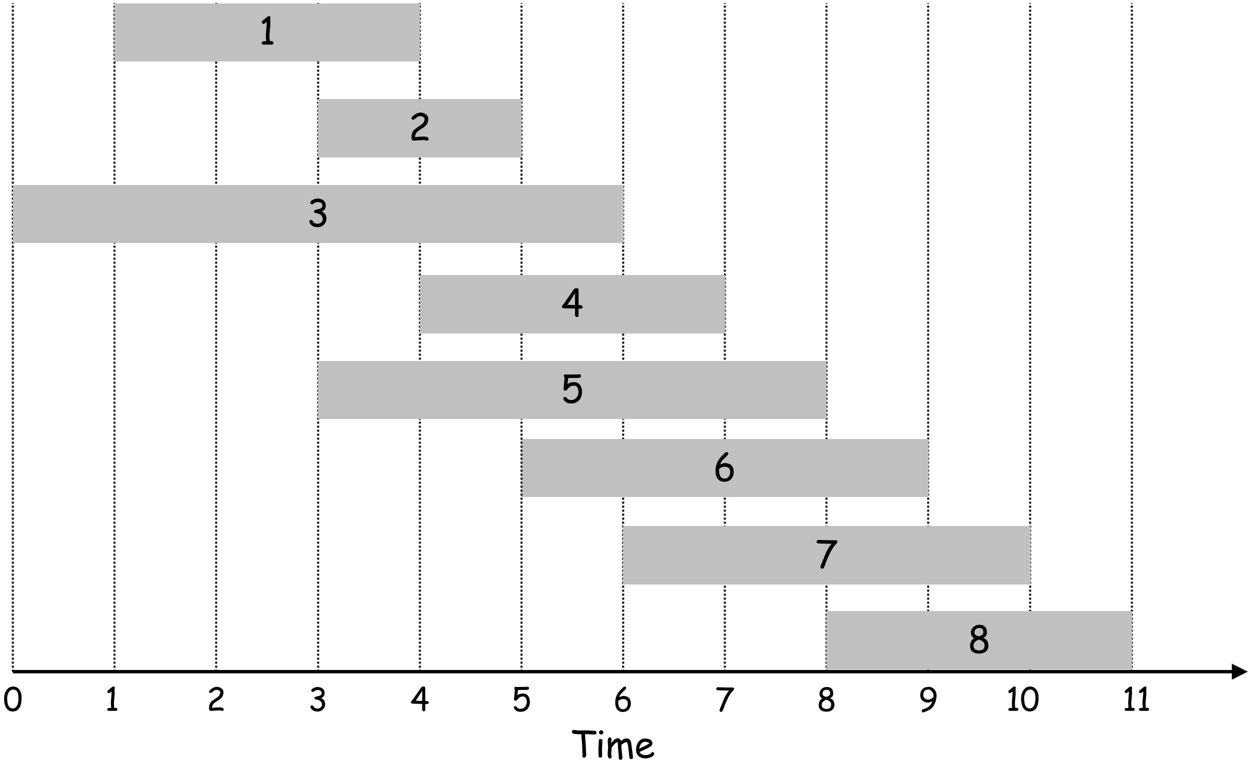
| j | vj | pj | optj |
|---|---|---|---|
| 0 | 0 | - | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | |
| 4 | 3 | 1 | |
| 5 | 4 | 0 | |
| 6 | 3 | 2 | |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Weighted Interval Scheduling

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

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

**Ex:** p(8) = 5, p(7) = 3, p(2) = 0.



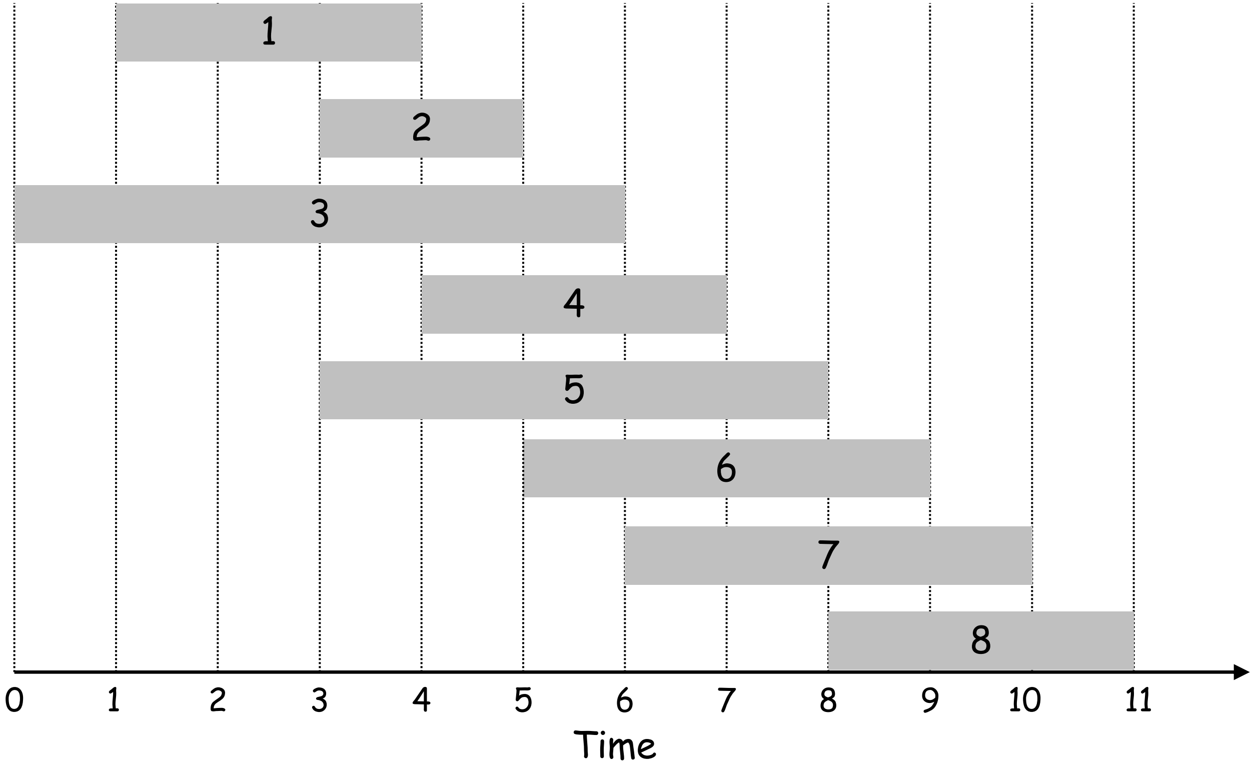
| j | vj | pj | optj |
|---|---|---|---|
| 0 | 0 | - | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | 4 |
| 4 | 3 | 1 | |
| 5 | 4 | 0 | |
| 6 | 3 | 2 | |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Weighted Interval Scheduling

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

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

**Ex:** p(8) = 5, p(7) = 3, p(2) = 0.



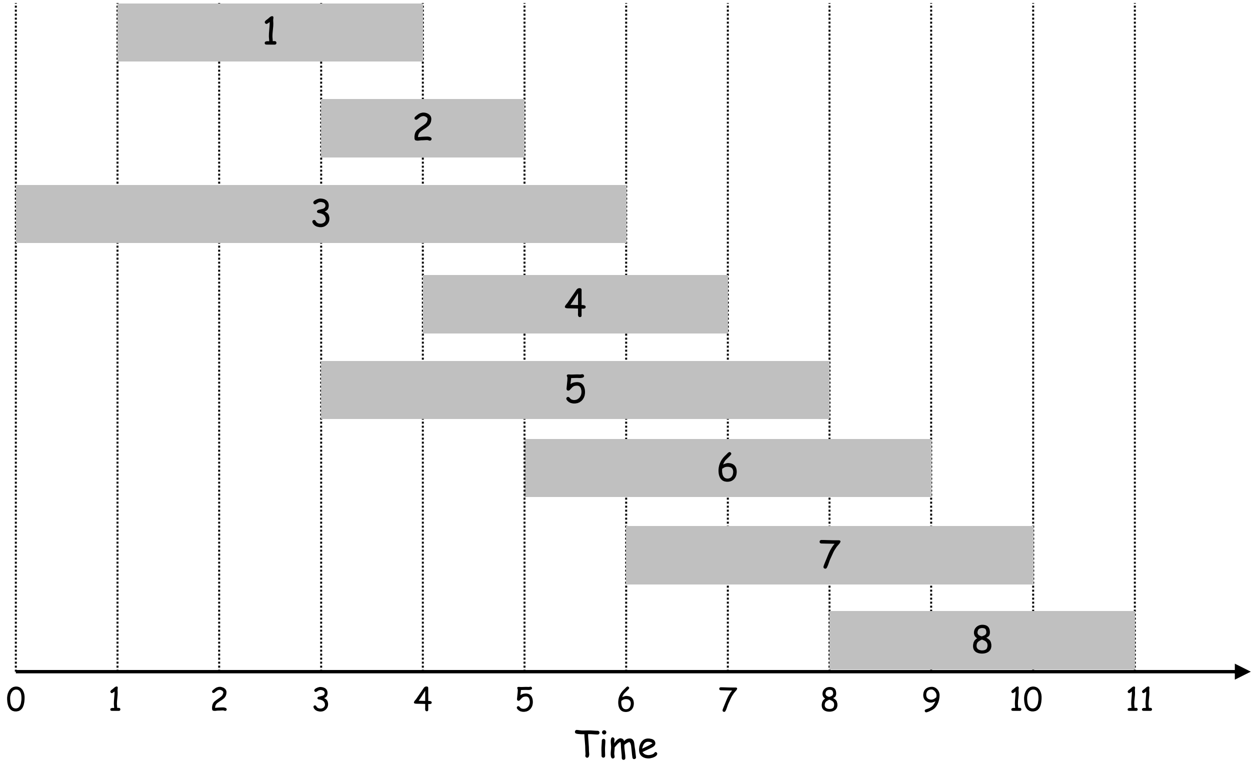
| j | vj | pj | optj |
|---|---|---|---|
| 0 | 0 | - | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | 4 |
| 4 | 3 | 1 | 6 |
| 5 | 4 | 0 | |
| 6 | 3 | 2 | |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Weighted Interval Scheduling

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

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

**Ex:** p(8) = 5, p(7) = 3, p(2) = 0.



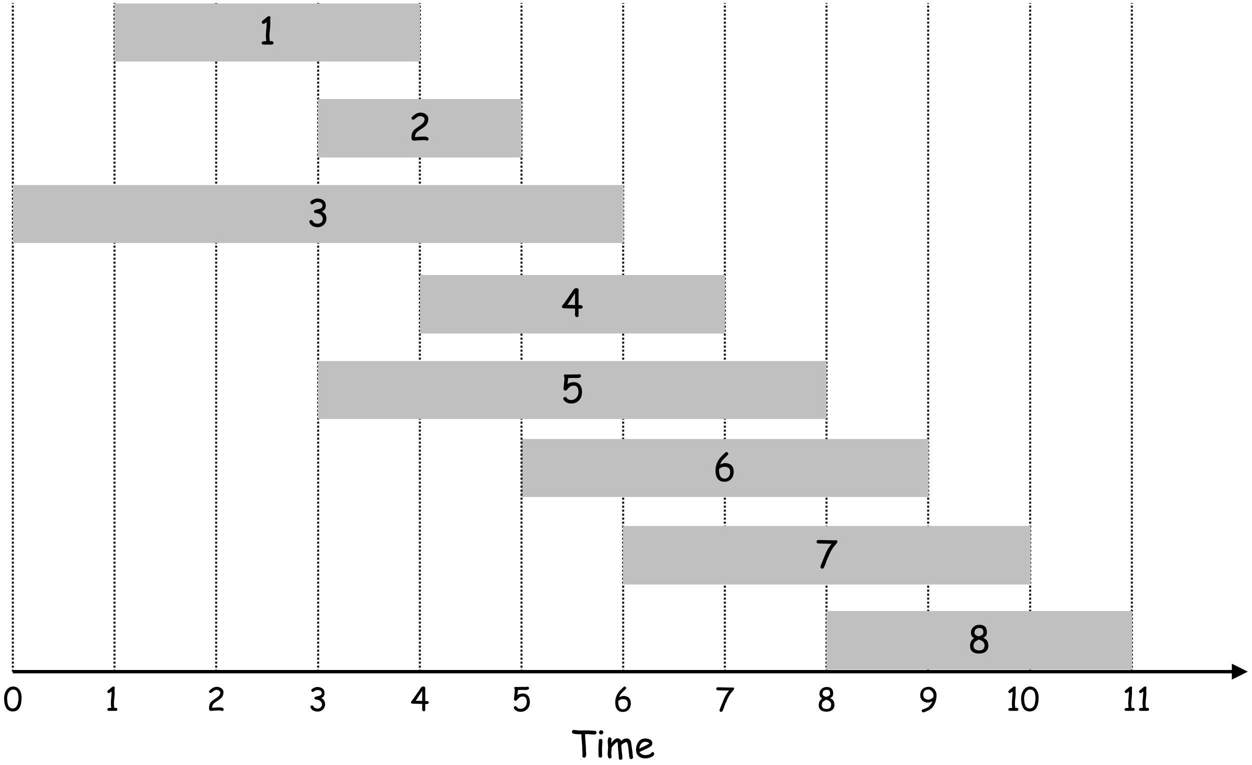
| j | vj | pj | optj |
|---|---|---|---|
| 0 | 0 | - | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | 4 |
| 4 | 3 | 1 | 6 |
| 5 | 4 | 0 | 6 |
| 6 | 3 | 2 | |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Weighted Interval Scheduling

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

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

**Ex:** p(8) = 5, p(7) = 3, p(2) = 0.



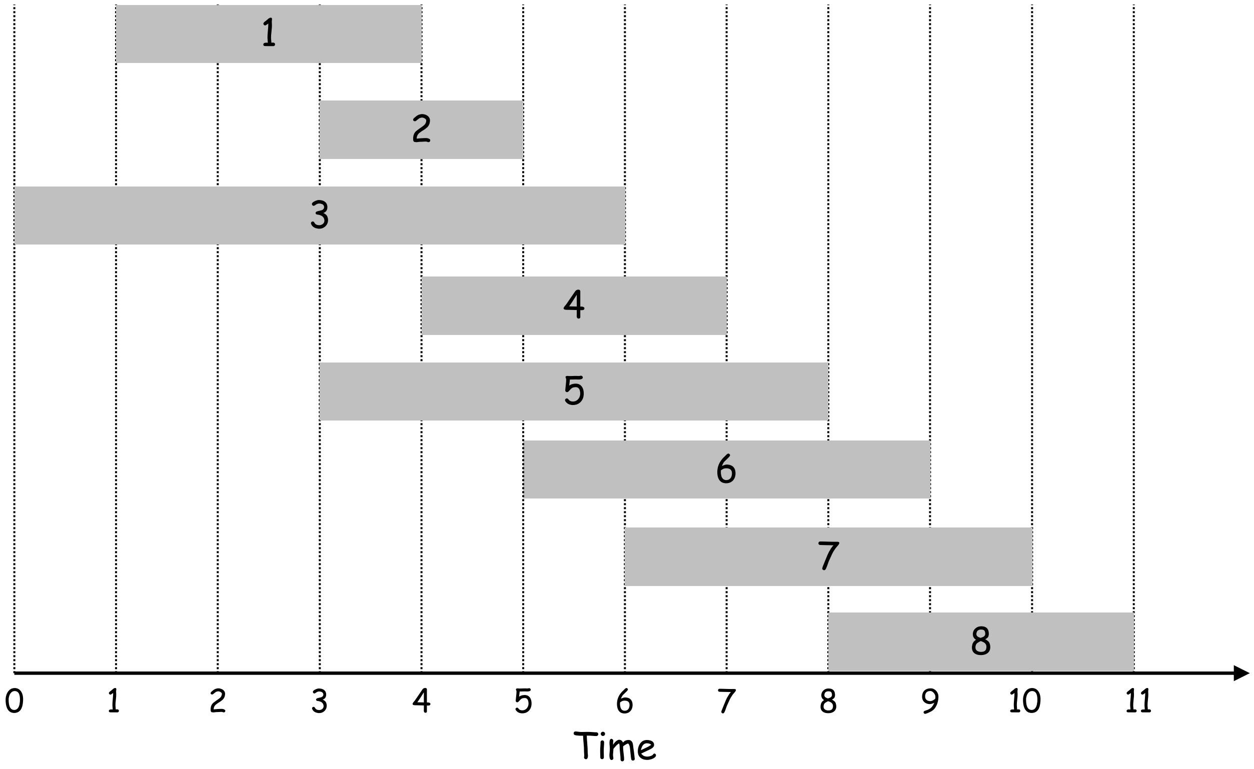
| j | vj | pj | optj |
|---|---|---|---|
| 0 | 0 | - | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | 4 |
| 4 | 3 | 1 | 6 |
| 5 | 4 | 0 | 6 |
| 6 | 3 | 2 | 7 |
| 7 | 2 | 3 | |
| 8 | 4 | 5 | |

# Weighted Interval Scheduling

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

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

**Ex:** $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.



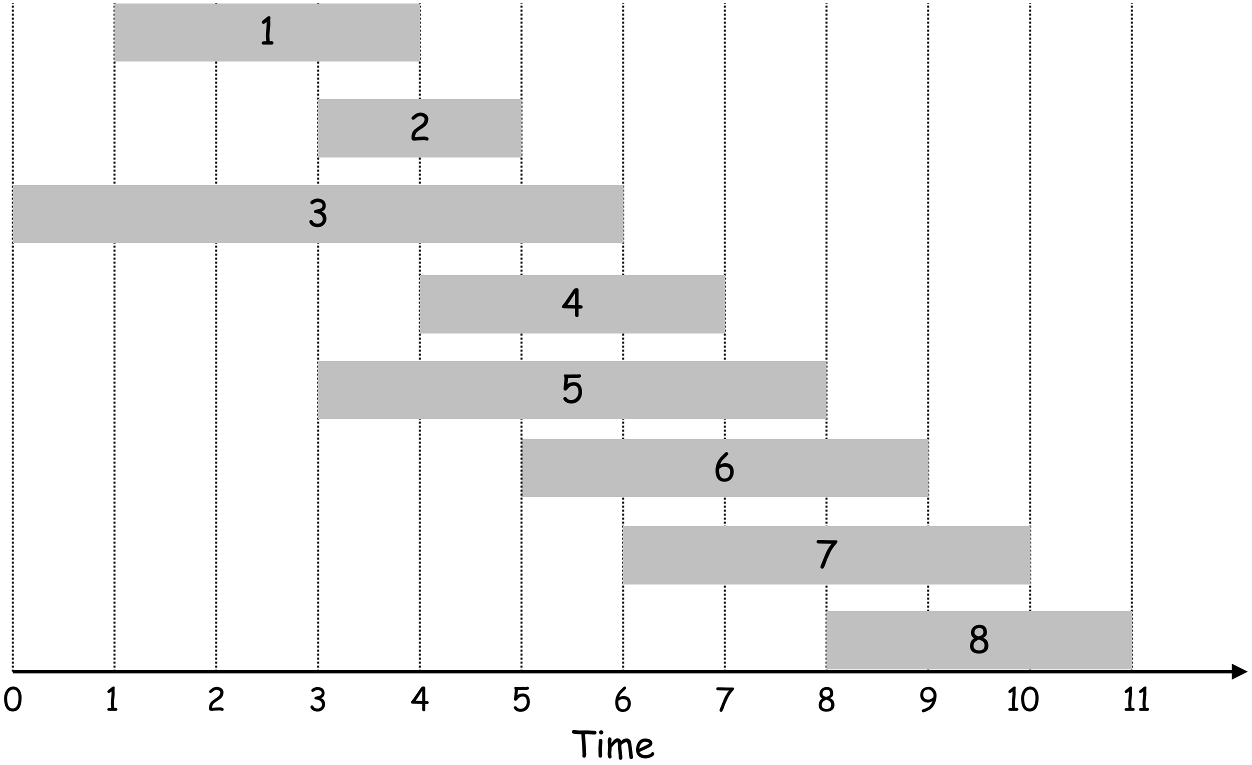
| j | vj | pj | optj |
|---|---|---|---|
| 0 | 0 | - | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | 4 |
| 4 | 3 | 1 | 6 |
| 5 | 4 | 0 | 6 |
| 6 | 3 | 2 | 7 |
| 7 | 2 | 3 | 7 |
| 8 | 4 | 5 | |

# Weighted Interval Scheduling

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

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

**Ex:** p(8) = 5, p(7) = 3, p(2) = 0.



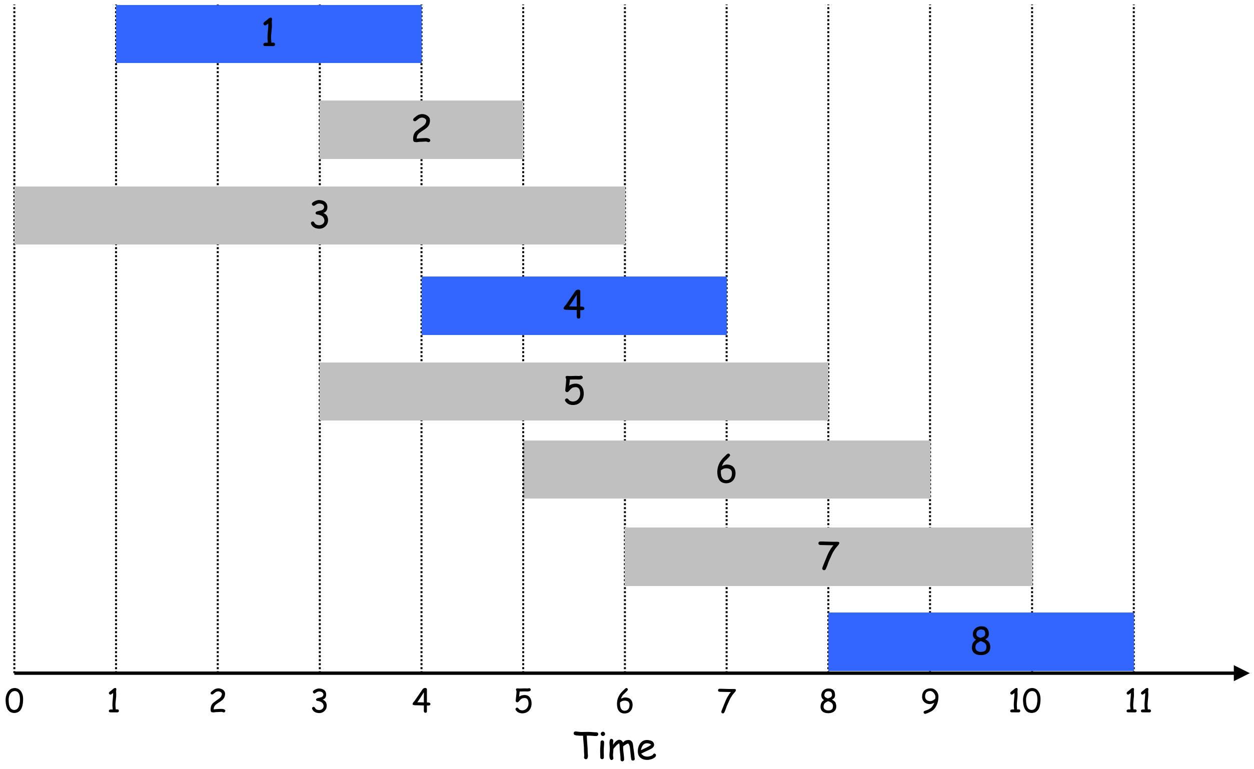
| j | vj | pj | optj |
|---|---|---|---|
| 0 | 0 | - | 0 |
| 1 | 3 | 0 | 3 |
| 2 | 4 | 0 | 4 |
| 3 | 1 | 0 | 4 |
| 4 | 3 | 1 | 6 |
| 5 | 4 | 0 | 6 |
| 6 | 3 | 2 | 7 |
| 7 | 2 | 3 | 7 |
| 8 | 4 | 5 | 10 |

# Weighted Interval Scheduling

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

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

**Ex:** p(8) = 5, p(7) = 3, p(2) = 0.



| j | vj | pj | optj |
|---|---|---|---|
| 0 | 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: { 3, 4 } has value 40.

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 ⇒ greedy not optimal.

# Dynamic Programming: False Start

**Def.** OPT(i) = max profit subset of items 1, …, i.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 }

- Case 2: OPT selects item i.
  - accepting item i does not immediately imply that we will have to reject other items
  - without knowing what other items were selected before i, we don't even know if we have enough room for i

**Conclusion.** Need more sub-problems!

# Dynamic Programming: Adding a New Variable

**Def.** OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w
- Case 2: OPT selects item i.
  - new weight limit = w – $w_i$
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

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

# Knapsack Problem: Bottom-Up

**Knapsack.** Fill up an n-by-W array.

```
Input: n, $w_1,…,w_N$, $v_1,…,v_N$

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      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-$w_i$ ]}

return M[n, W]
```

# Knapsack Algorithm

W + 1

n + 1

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ∅ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| { 1 } |  |  |  |  |  |  |  |  |  |  |  |  |
| { 1, 2 } |  |  |  |  |  |  |  |  |  |  |  |  |
| { 1, 2, 3 } |  |  |  |  |  |  |  |  |  |  |  |  |
| { 1, 2, 3, 4 } |  |  |  |  |  |  |  |  |  |  |  |  |
| { 1, 2, 3, 4, 5 } |  |  |  |  |  |  |  |  |  |  |  |  |

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 Algorithm

W + 1

n + 1

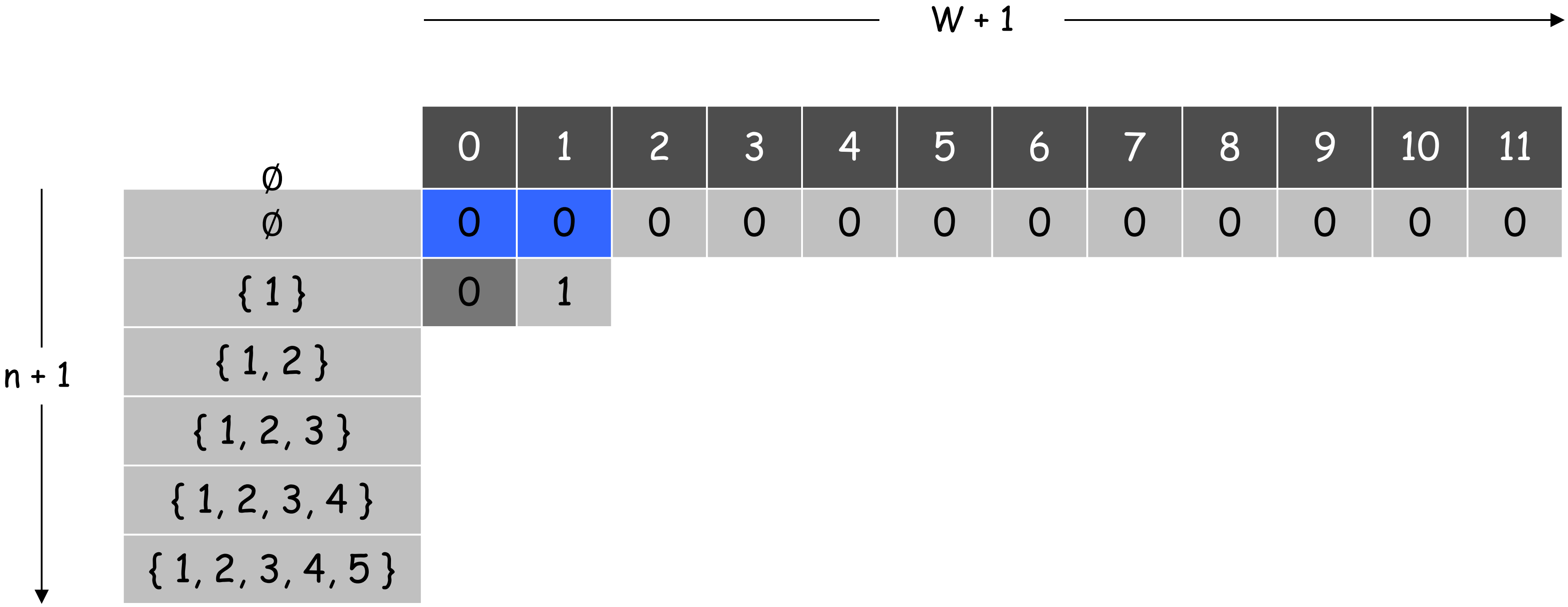
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ∅ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| { 1 } | 0 | | | | | | | | | | | |
| { 1, 2 } | | | | | | | | | | | | |
| { 1, 2, 3 } | | | | | | | | | | | | |
| { 1, 2, 3, 4 } | | | | | | | | | | | | |
| { 1, 2, 3, 4, 5 } | | | | | | | | | | | | |

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 Algorithm



| ∅ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ∅ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| { 1 } | 0 | 1 | | | | | | | | | | |
| { 1, 2 } | | | | | | | | | | | | |
| { 1, 2, 3 } | | | | | | | | | | | | |
| { 1, 2, 3, 4 } | | | | | | | | | | | | |
| { 1, 2, 3, 4, 5 } | | | | | | | | | | | | |

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 Algorithm

W + 1

n + 1

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ∅ | 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 |  |  |  |  |  |  |  |
| { 1, 2, 3, 4 } |  |  |  |  |  |  |  |  |  |  |  |  |
| { 1, 2, 3, 4, 5 } |  |  |  |  |  |  |  |  |  |  |  |  |

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 Algorithm

W + 1

n + 1

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ∅ | 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 |  |  |  |  |  |  |  |
| { 1, 2, 3, 4 } |  |  |  |  |  |  |  |  |  |  |  |  |
| { 1, 2, 3, 4, 5 } |  |  |  |  |  |  |  |  |  |  |  |  |

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 Algorithm

W + 1 →

n + 1 ↓

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ∅ | 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 | | | | | | |
| { 1, 2, 3, 4 } | | | | | | | | | | | | |
| { 1, 2, 3, 4, 5 } | | | | | | | | | | | | |

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 Algorithm

W + 1

n + 1

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ∅ | 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 | | | | | | |
| { 1, 2, 3, 4 } | | | | | | | | | | | | |
| { 1, 2, 3, 4, 5 } | | | | | | | | | | | | |

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 Algorithm

W + 1

n + 1

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ∅ | 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 } | | | | | | | | | | | | |
| { 1, 2, 3, 4, 5 } | | | | | | | | | | | | |

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 Algorithm

W + 1

n + 1

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ø | 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 | | | | |
| { 1, 2, 3, 4, 5 } | | | | | | | | | | | | |

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 Algorithm

W + 1

n + 1

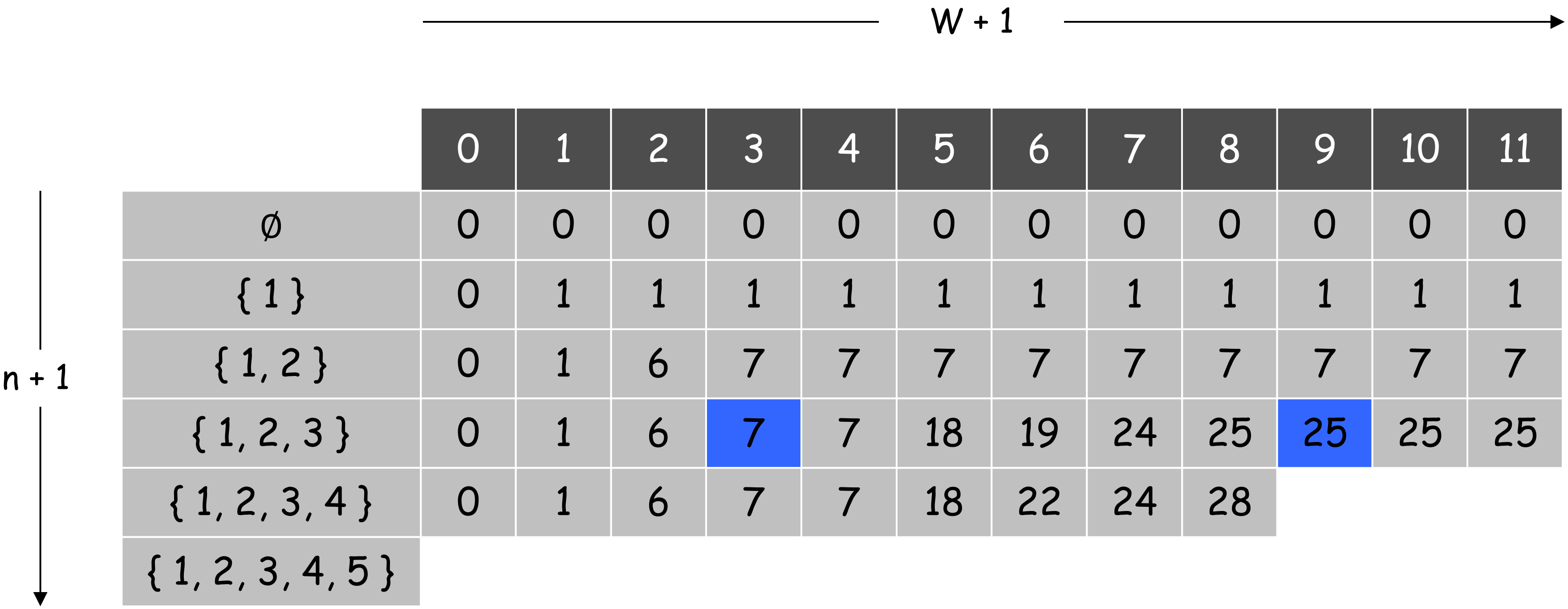
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ∅ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| { 1 } | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| { 1, 2 } | 0 | 1 | 6 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| { 1, 2, 3 } | 0 |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

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 Algorithm



W + 1

n + 1

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ∅ | 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 |  |  |  |
| { 1, 2, 3, 4, 5 } |  |  |  |  |  |  |  |  |  |  |  |  |

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 Algorithm

W + 1

n + 1

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ∅ | 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 } | | | | | | | | | | | | |

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 Algorithm

W + 1

n + 1

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ∅ | 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. O(n 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.

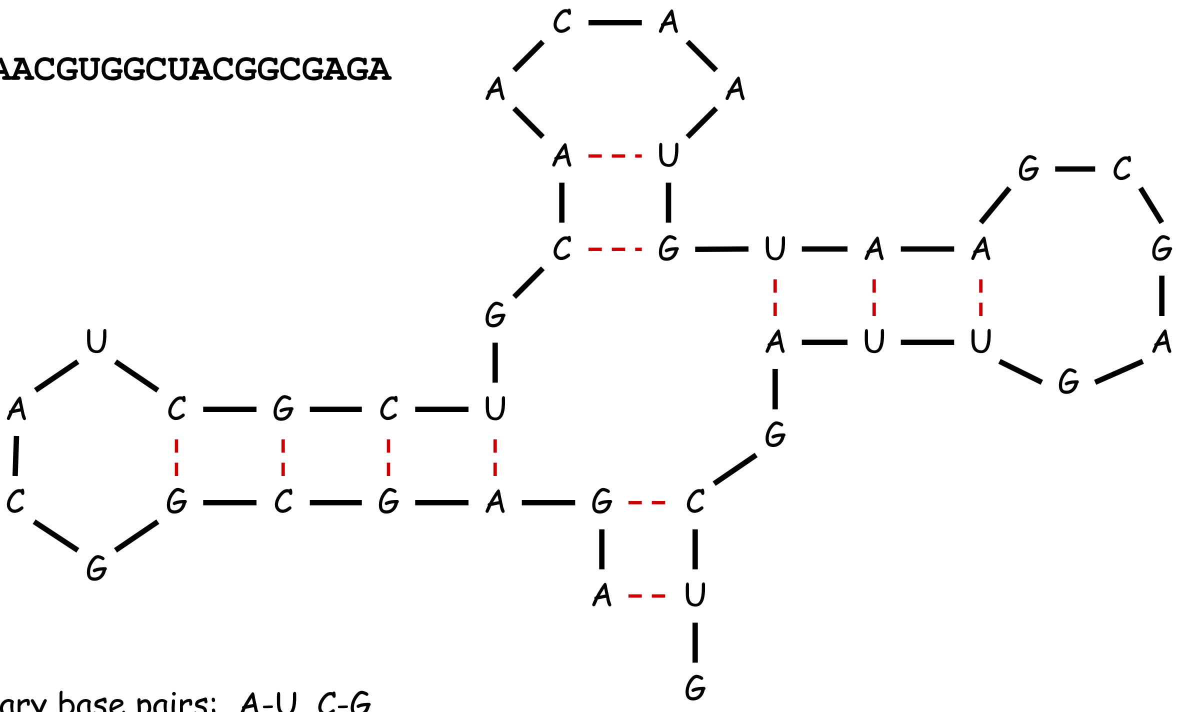
# 6.5 RNA Secondary Structure

# RNA Secondary Structure

RNA. String B = $b_1 b_2 \ldots b_n$ over alphabet { A, C, G, U }.

Secondary structure. RNA is single-stranded so it tends to loop back and form base pairs with itself. This structure is essential for understanding behavior of molecule.

Ex: **GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA**



complementary base pairs: A-U, C-G

# RNA Secondary Structure

Secondary structure. A set of pairs $S = \{ (b_i, b_j) \}$ that satisfy:

- [Watson-Crick.]
  - S is a *matching* and
  - each pair in S is a Watson-Crick pair: A-U, U-A, C-G, or G-C.
- [No sharp turns.] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j)$ in S, then $i < j - 4$.
- [Non-crossing.] If $(b_i, b_j)$ and $(b_k, b_l)$ are two pairs in S, then we cannot have $i < k < j < l$.

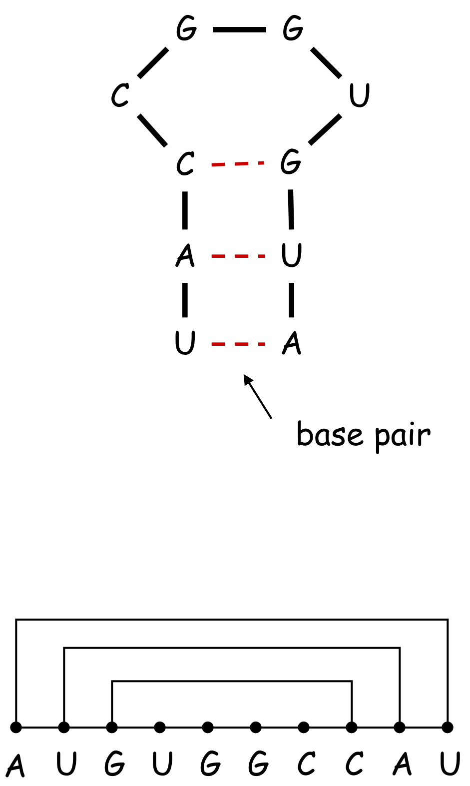
Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy.

approximate by number of base pairs

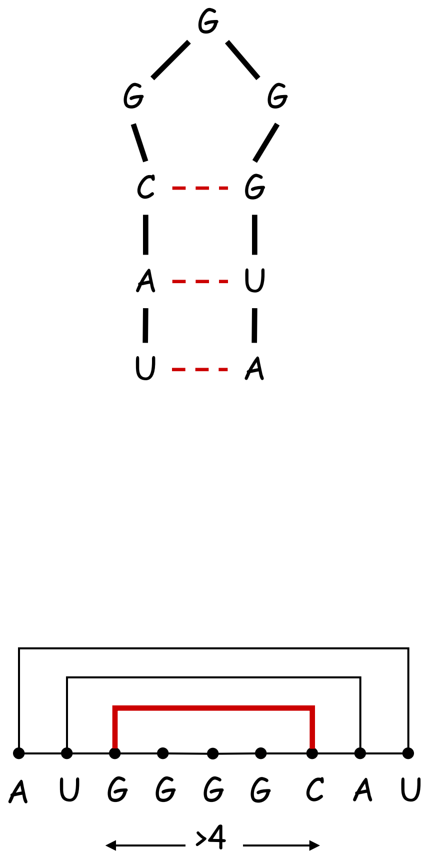
Goal. Given an RNA molecule $B = b_1 b_2 \ldots b_n$, find a secondary structure S that maximizes the number of base pairs.

# RNA Secondary Structure: Examples

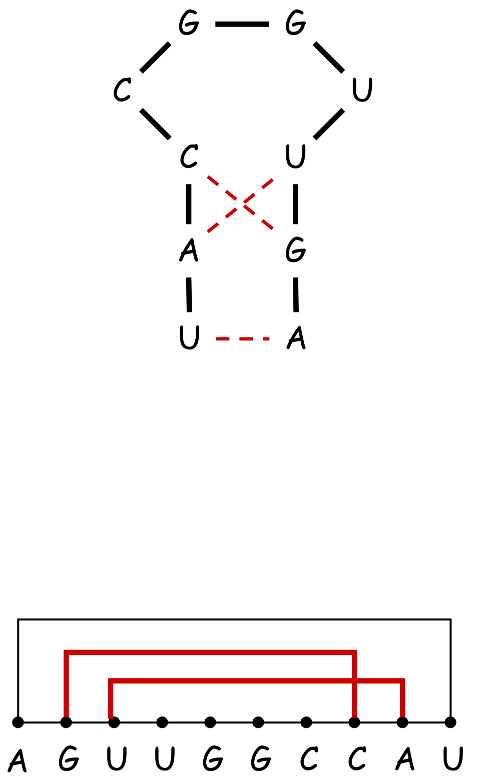
Examples.



ok



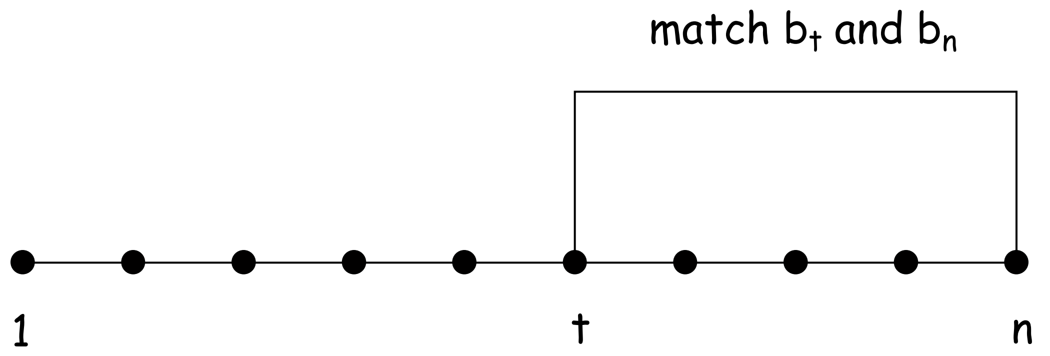
sharp turn



crossing

# RNA Secondary Structure: Subproblems

**First attempt.** OPT(j) = maximum number of base pairs in a secondary structure of the substring $b_1 b_2 \ldots b_j$.



**Difficulty.** Results in two sub-problems.

- Finding secondary structure in: $b_1 b_2 \ldots b_{t-1}$. ⟵ OPT(t-1)
- Finding secondary structure in: $b_{t+1} b_{t+2} \ldots b_{n-1}$. ⟵ need more sub-problems

# Dynamic Programming Over Intervals

**Notation.** OPT(i, j) = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$.

- Case 1. If $4 > j - i$.
  - OPT(i, j) = 0 by no-sharp turns condition.

- Case 2. Base $b_j$ is not involved in a pair.
  - OPT(i, j) = OPT(i, j-1)

- Case 3. Base $b_j$ pairs with $b_t$ for some $j - t > 4$, $i \le t$
  - non-crossing constraint decouples resulting sub-problems
  - $\text{OPT}(i, j) = 1 + \underset{\uparrow}{\max_t} \{ \text{OPT}(i, t-1) + \text{OPT}(t+1, j-1) \}$

take max over t such that $i \le t < j-4$ and $b_t$ and $b_j$ are Watson-Crick complements

**Remark.** Same core idea in CKY algorithm to parse context-free grammars.

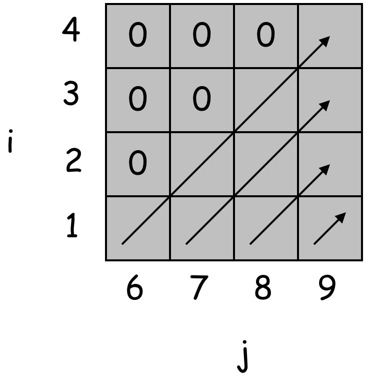
# Bottom Up Dynamic Programming Over Intervals

Q. What order to solve the sub-problems?

A. Do shortest intervals first.

```
RNA($b_1$,…,$b_n$) {
   for k = 5, 6, …, n-1
      for i = 1, 2, …, n-k
         j = i + k
         Compute M[i, j]

   return M[1, n]
}
```

using recurrence



Running time. $O(n^3)$.

# Dynamic Programming Mantra

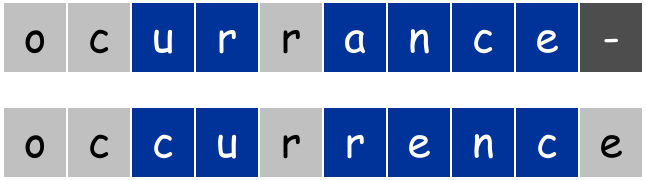
- Express OPT in terms of OPT for smaller problems [like divide and conquer]
- Figure out a clever order to evaluate all sub-problems to minimize redundancy [pictures help!]

# 6.6 Sequence Alignment

# String Similarity

## How similar are two strings?

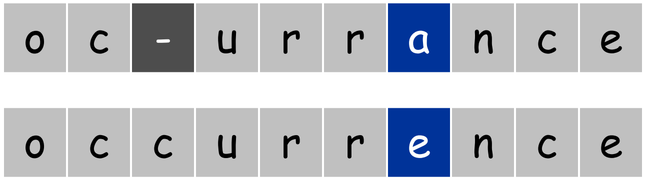
- `ocurrance`
- `occurrence`



o c u r r a n c e -

o c c u r r e n c e

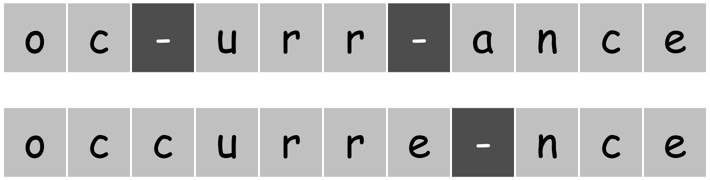
5 mismatches, 1 gap



o c - u r r a n c e

o c c u r r e n c e

1 mismatch, 1 gap



o c - u r r - a n c e

o c c u r r e - n c e

0 mismatches, 3 gaps

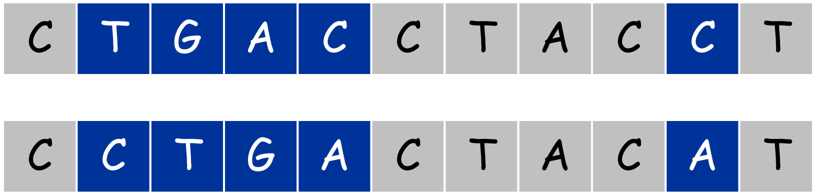
# Edit Distance

**Applications.**

- Basis for Unix diff.
- Speech recognition.
- Computational biology.

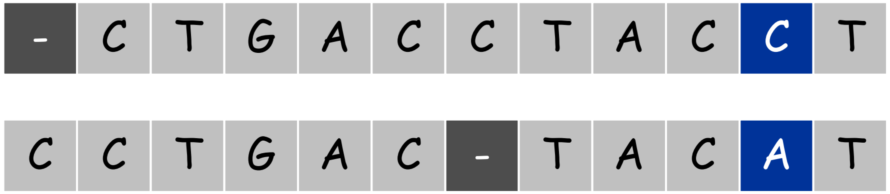
**Edit distance.** [Levenshtein 1966, Needleman-Wunsch 1970]

- Cost = # of gaps and mismatches.



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| C | T | G | A | C | C | T | A | C | C | T |
| C | C | T | G | A | C | T | A | C | A | T |

Cost: 5



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| - | C | T | G | A | C | C | T | A | C | C | T |
| C | C | T | G | A | C | - | T | A | C | A | T |

Cost: 3

# Sequence Alignment

**Goal:** Given two strings $X = x_1\, x_2\, .\, .\, .\, x_m$ and $Y = y_1\, y_2\, .\, .\, .\, y_n$ find alignment of minimum cost.

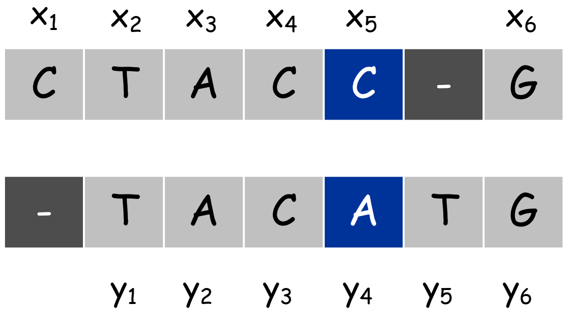
**Def.** An alignment M is a set of ordered pairs $x_i$-$y_j$ such that each item occurs in at most one pair and no crossings.

**Def.** The pair $x_i$-$y_j$ and $x_{i'}$-$y_{j'}$ cross if $i < i'$, but $j > j'$.

Cost of M: # mismatches and gaps.

**Ex:** `CTACCG` vs. `TACATG`.

**Sol:** $M = x_2\text{-}y_1, x_3\text{-}y_2, x_4\text{-}y_3, x_5\text{-}y_4, x_6\text{-}y_6$.



| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |  | $x_6$ |
|---|---|---|---|---|---|---|
| C | T | A | C | C | - | G |
| - | T | A | C | A | T | G |
|  | $y_1$ | $y_2$ | $y_3$ | $y_4$ | $y_5$ | $y_6$ |

# Sequence Alignment: Problem Structure

**Def.** OPT(i, j) = min cost of aligning strings $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$.

# Sequence Alignment: Problem Structure

**Def.** OPT(i, j) = min cost of aligning strings $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$.

- Case 1: OPT matches $x_i$-$y_j$.
  - pay mismatch for $x_i$-$y_j$ + min cost of aligning two strings $x_1 x_2 \ldots x_{i-1}$ and $y_1 y_2 \ldots y_{j-1}$
- Case 2a: OPT leaves $x_i$ unmatched.
  - pay gap for $x_i$ and min cost of aligning $x_1 x_2 \ldots x_{i-1}$ and $y_1 y_2 \ldots y_j$
- Case 2b: OPT leaves $y_j$ unmatched.
  - pay gap for $y_j$ and min cost of aligning $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_{j-1}$

$$OPT(i,\ j) = \begin{cases} j & \text{if } \ \mathrm{i} = 0 \\ \min \begin{cases} 1_{x_i y_j} + OPT(i-1,\ j-1) \\ 1 + OPT(i-1,\ j) \\ 1 + OPT(i,\ j-1) \end{cases} & \\ i & \text{if } \ \mathrm{j} = 0 \end{cases}$$

$1_{x_i y_j}$ is 0 if $x_i$=$y_j$ else 1

# Sequence Alignment: Algorithm

```
Sequence-Alignment(m, n, $x_1x_2...x_m$, $y_1y_2...y_n$) {
   for i = 0 to m
      M[0, i] = i
   for j = 0 to n
      M[j, 0] = j

   for i = 1 to m
      for j = 1 to n
         M[i, j] = min(1xi,yj + M[i-1, j-1],
                       1 + M[i-1, j],
                       1 + M[i, j-1])
   return M[m, n]
}
```

Analysis. O(mn) time and space.

English words or sentences: m, n ~ 10.

Computational biology: m = n = 100,000. 10 billions ops OK, but 10GB array?

# Dynamic Programming Summary

Recipe.

- Characterize structure of problem.
- Recursively define value of optimal solution.
- Compute value of optimal solution.
- Construct optimal solution from computed information.

Dynamic programming techniques.

- Binary choice: weighted interval scheduling.
- Adding a new variable: knapsack.
- Dynamic programming over intervals: RNA secondary structure.

CKY parsing algorithm for context-free grammar has similar structure

Top-down vs. bottom-up: different people have different intuitions.