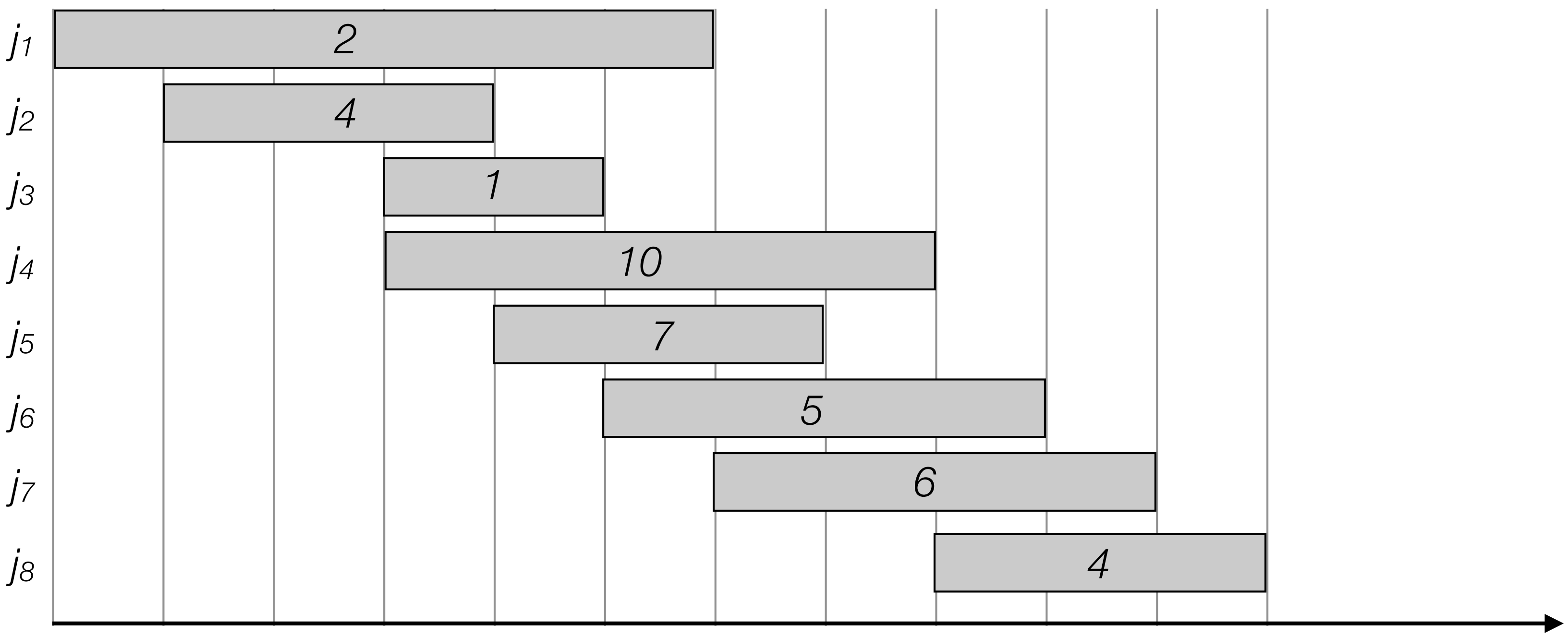
Dynamic Programming

Algorithm Design 6.1, 6.2, 6.3

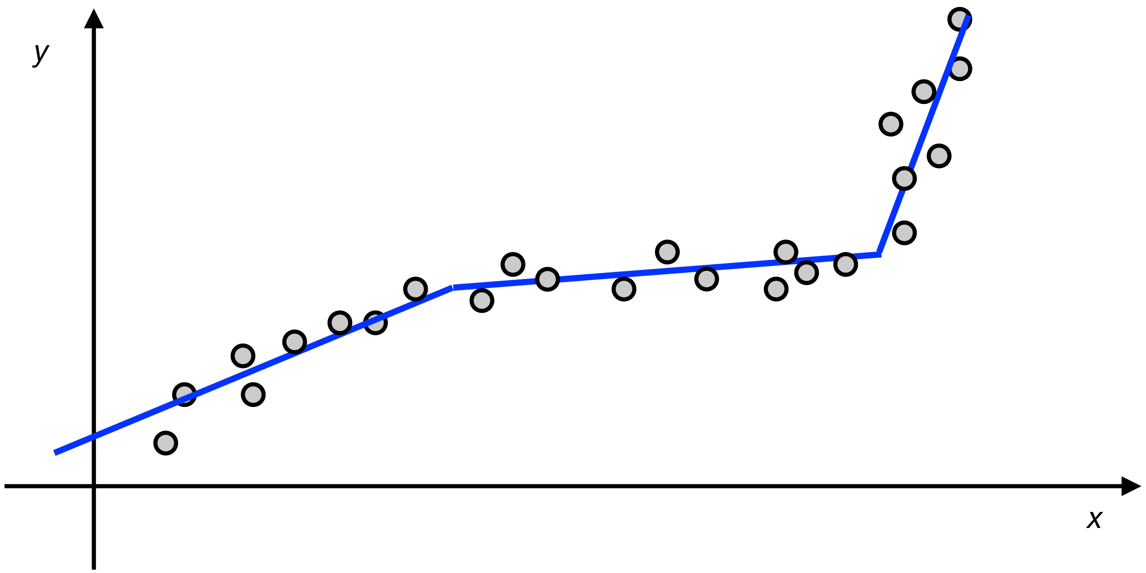
Applications

- In class (today and next time)
  - Weighted interval scheduling
    - Set of weighted intervals with start and finishing times
    - Goal: find maximum weight subset of non-overlapping intervals



Applications

- In class (today and next time)
  - Weighted interval scheduling
  - Segmented least squares
    - Given n points in the plane find a small sequence of lines that minimizes the squared error.



Applications

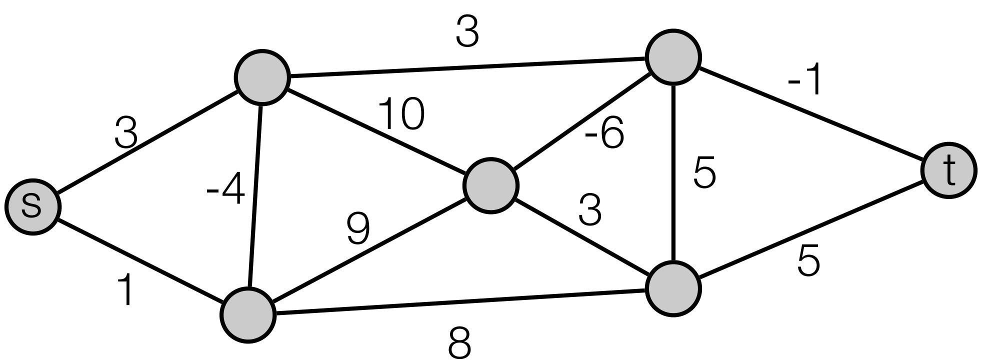
- In class (today and next time)
  - Weighted interval scheduling
  - Segmented least squares
  - Sequence alignment
    - Given two strings A and B how many edits (insertions, deletions, relabelings) is needed to turn A into B?

```
A C A A G T C        A C A A - G T C
- C A T G T -        - C A - T G T -

1 mismatch, 2 gaps    0 mismatches, 4 gaps
```

Applications

- In class (today and next time)
  - Weighted interval scheduling
  - Segmented least squares
  - RNA Secondary structure
  - Sequence alignment
  - Shortest paths with negative weights
    - Given a weighted graph, where edge weights can be negative, find the shortest path between two given vertices.



Applications

- In class (today and next time)
  - Weighted interval scheduling
  - Segmented least squares
  - RNA Secondary structure
  - Sequence alignment
  - Shortest paths with negative weights

- Some other famous applications
  - Unix diff for comparing 2 files
  - Vovke-Kasami-Younger for parsing context-free grammars
  - Viterbi for hidden Markov models
  - ….

Dynamic Programming

- Greedy. Build solution incrementally, optimizing some local criterion.

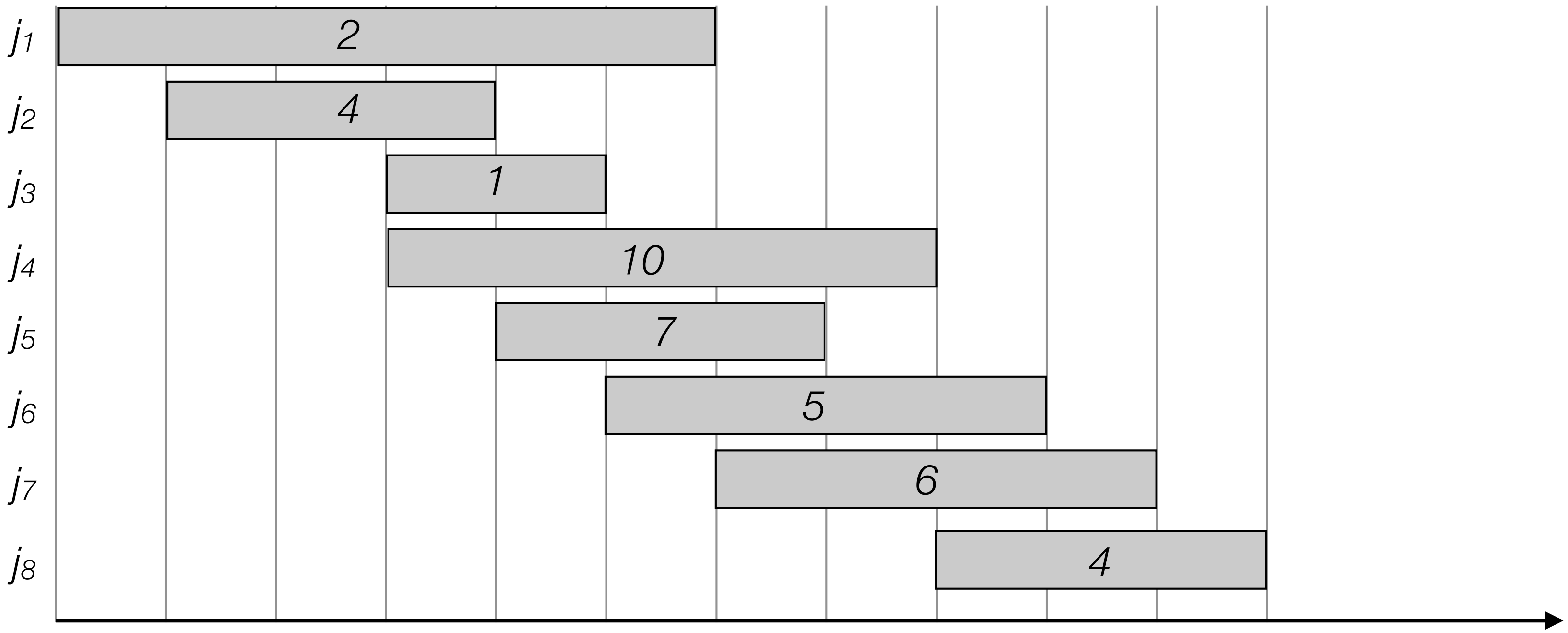
- Divide-and-conquer. Break up problem into independent subproblems, solve each subproblem, and combine to get solution to original problem.

- Dynamic programming. Break up problem into overlapping subproblems, and build up solutions to larger and larger subproblems.
  - Can be used when the problem have “optimal substructure”:
    - ✦ *Solution can be constructed from optimal solutions to subproblems*
    - ✦ *Use dynamic programming when subproblems overlap.*

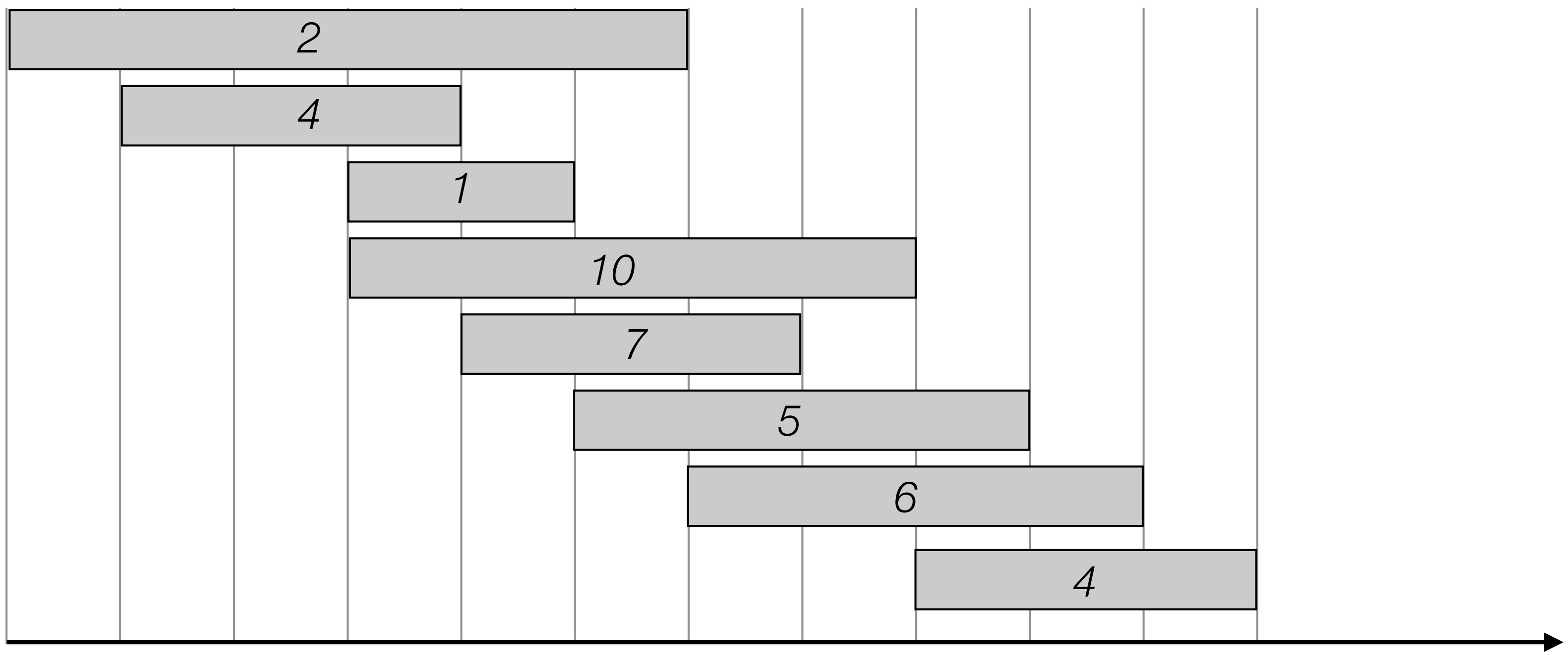
Weighted Interval Scheduling

Weighted interval scheduling

- Weighted interval scheduling problem
  - n jobs (intervals)
  - Job i starts at s_i, finishes at f_i and has weight/value v_i.
  - Goal: Find maximum weight subset of non-overlapping (compatible) jobs.

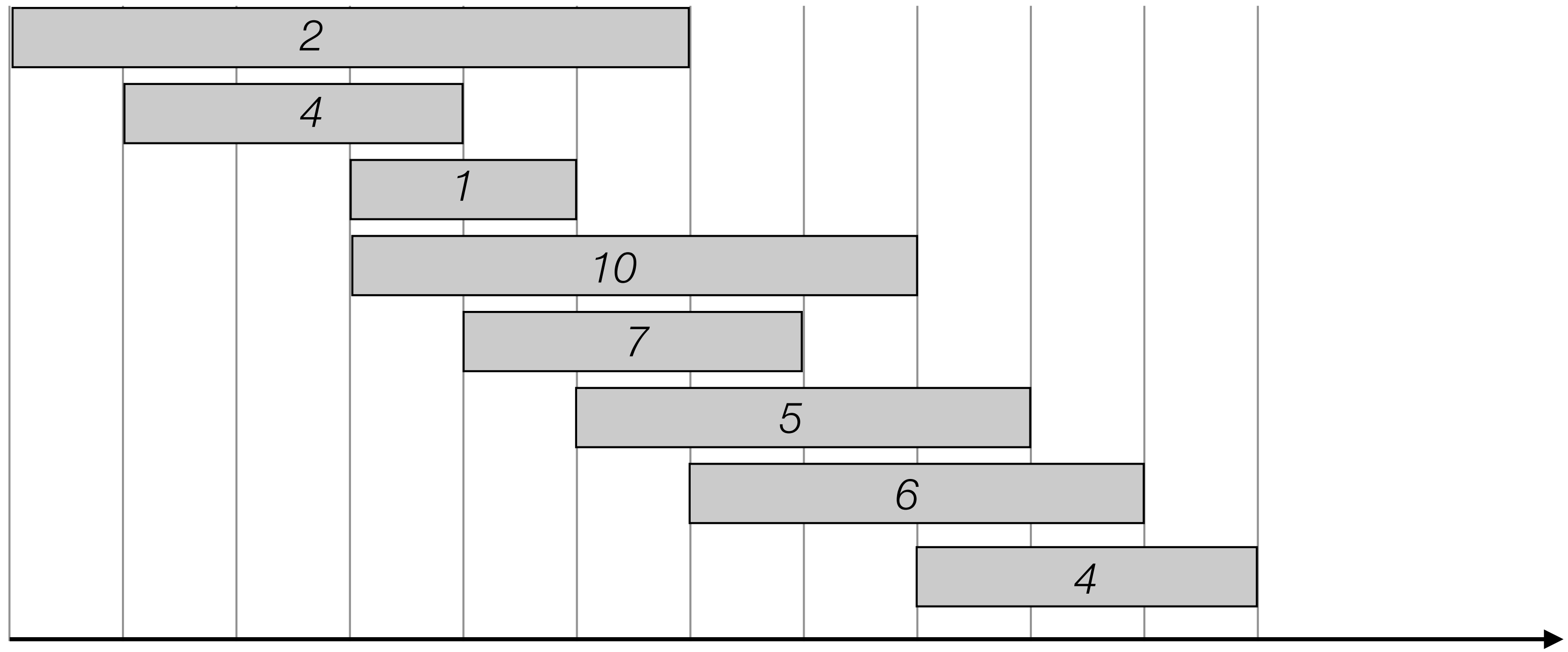


Weighted interval scheduling



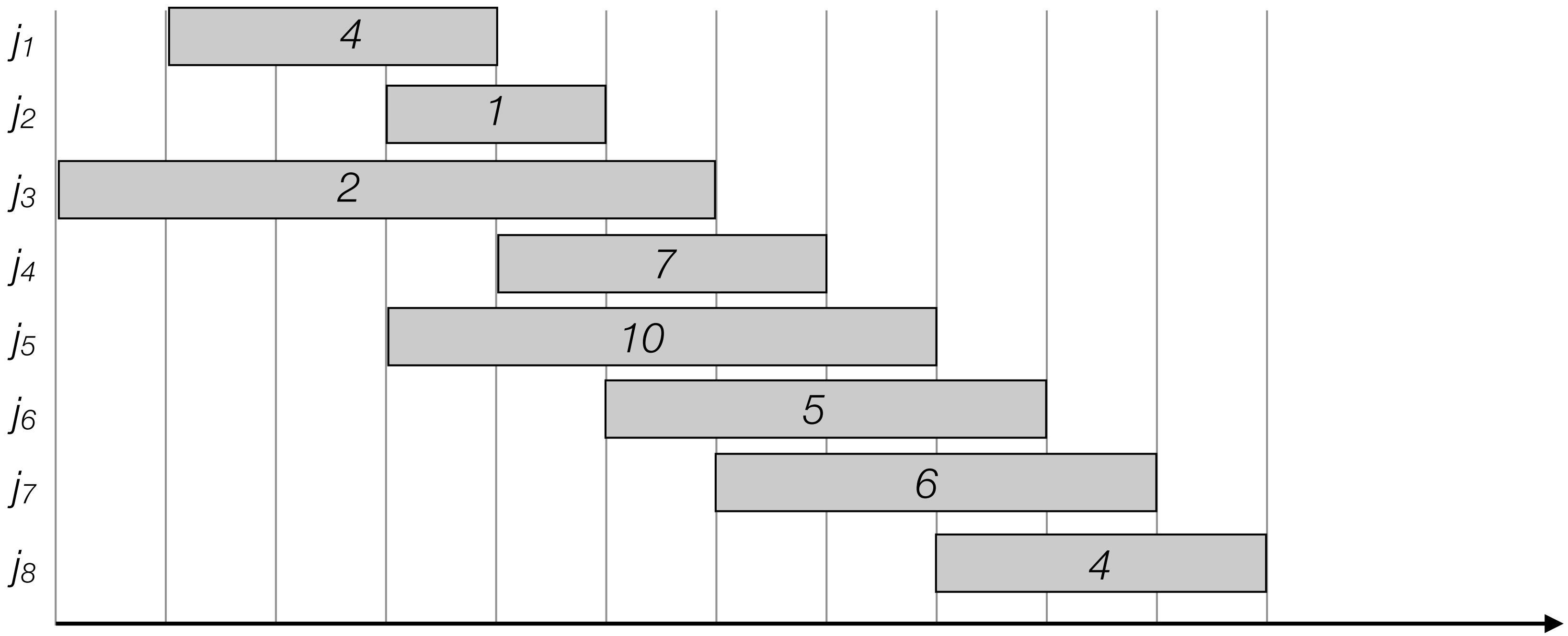
Weighted interval scheduling

- Label/sort jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$



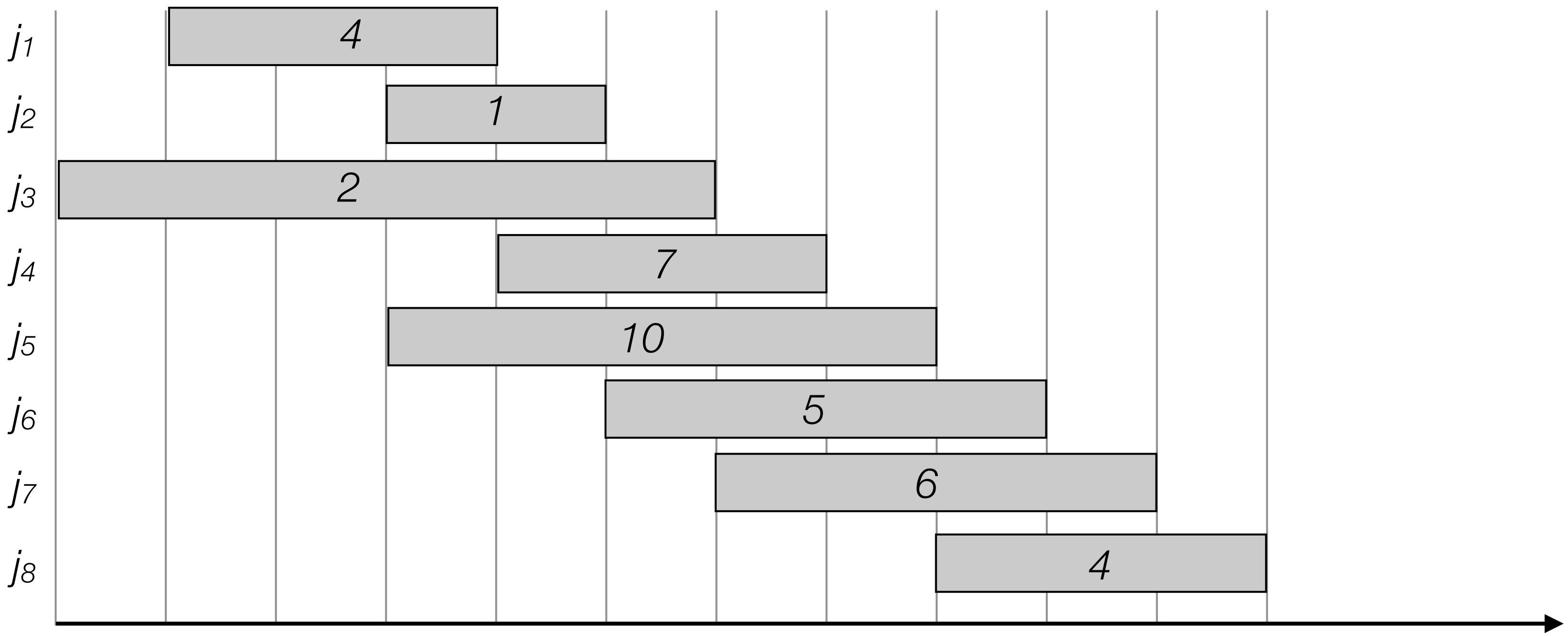
Weighted interval scheduling

- Label/sort jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$



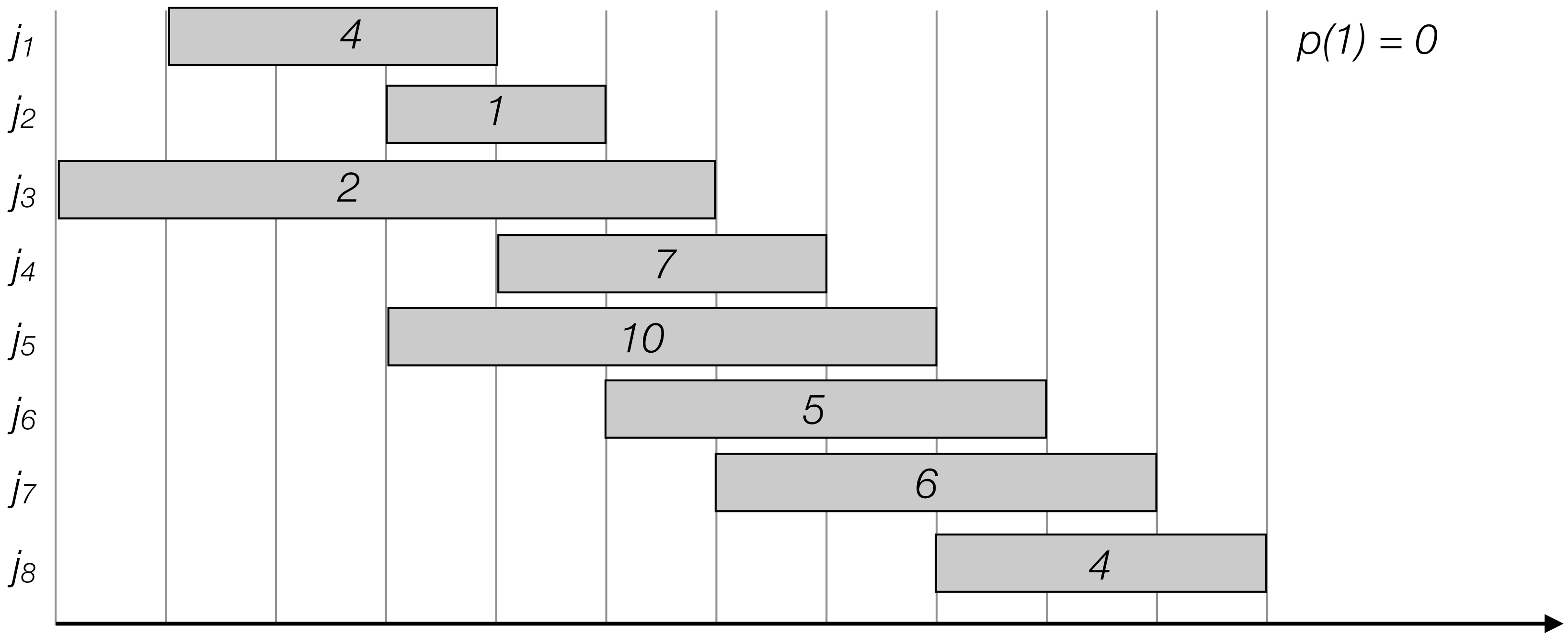
Weighted interval scheduling

- Label/sort 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.



Weighted interval scheduling

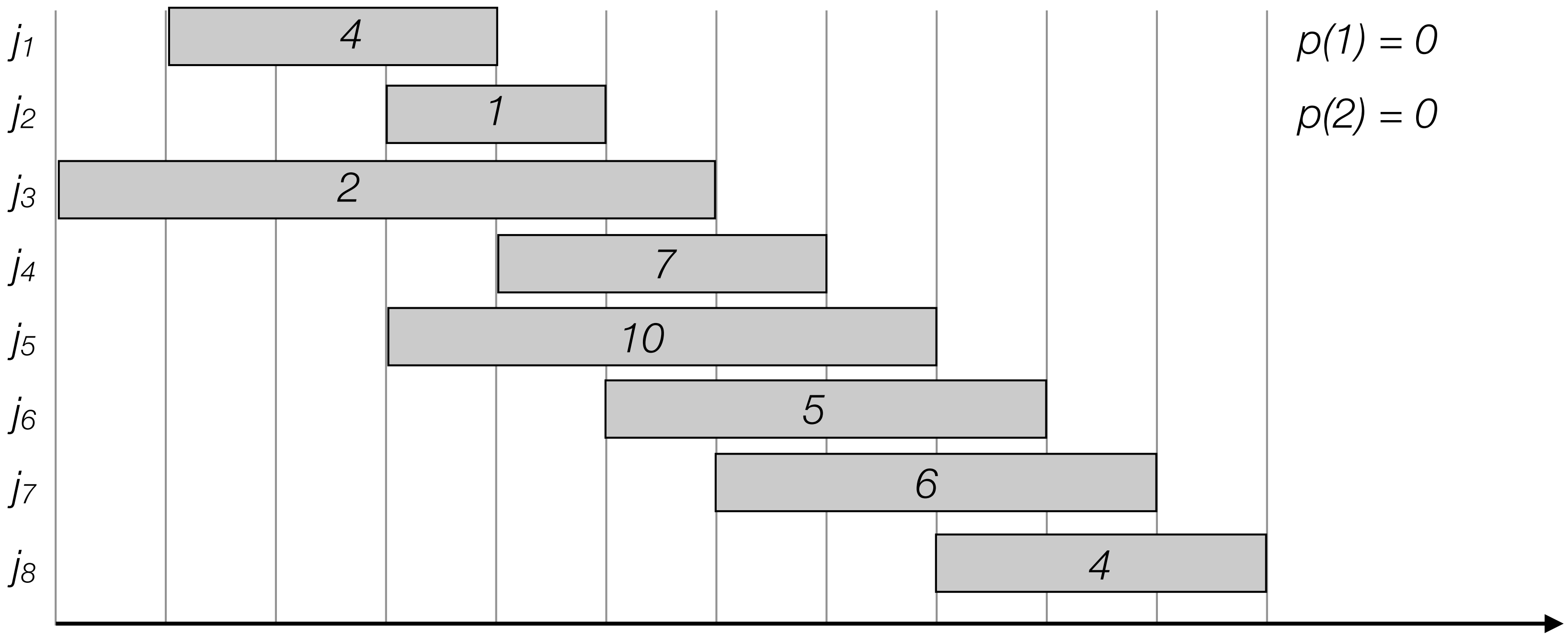
- Label/sort jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$
- $p(j)$ = largest index $i < j$ such that job i is compatible with j.



$p(1) = 0$

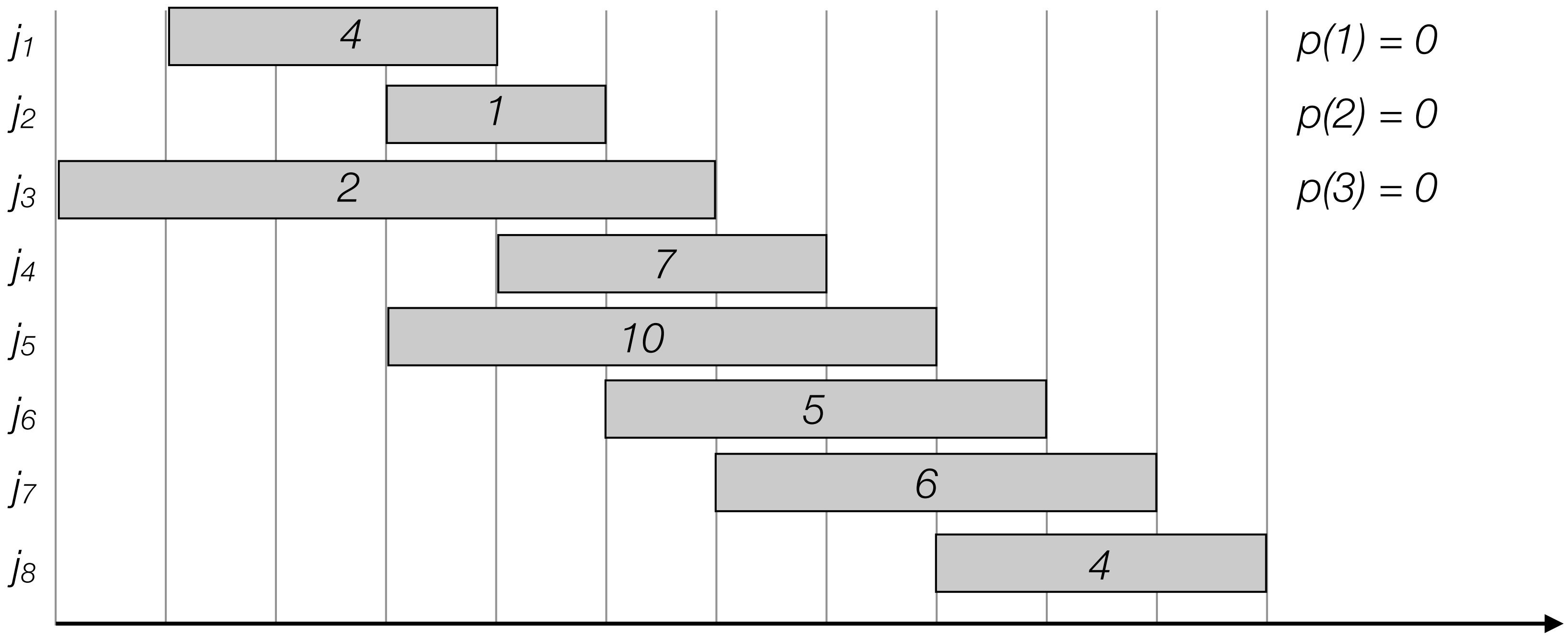
Weighted interval scheduling

- Label/sort 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.



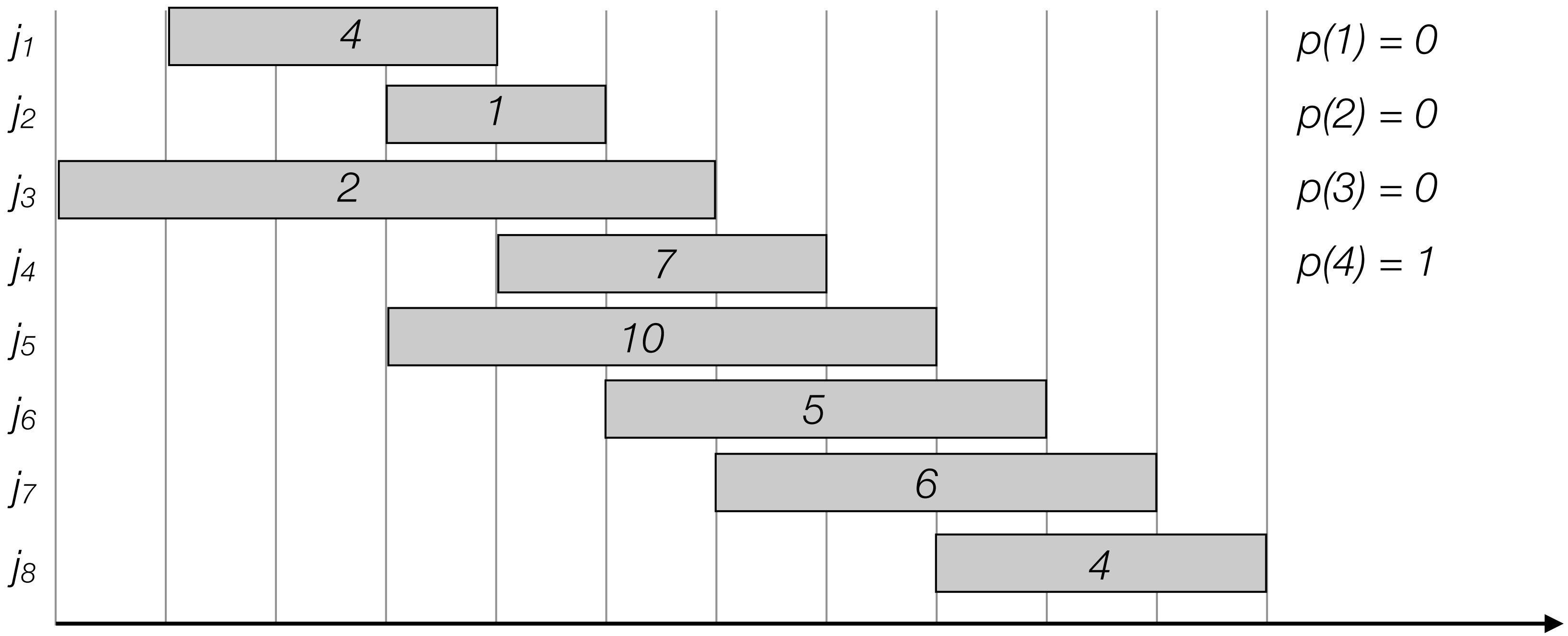
Weighted interval scheduling

- Label/sort 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.



Weighted interval scheduling

- Label/sort 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.



$p(1) = 0$

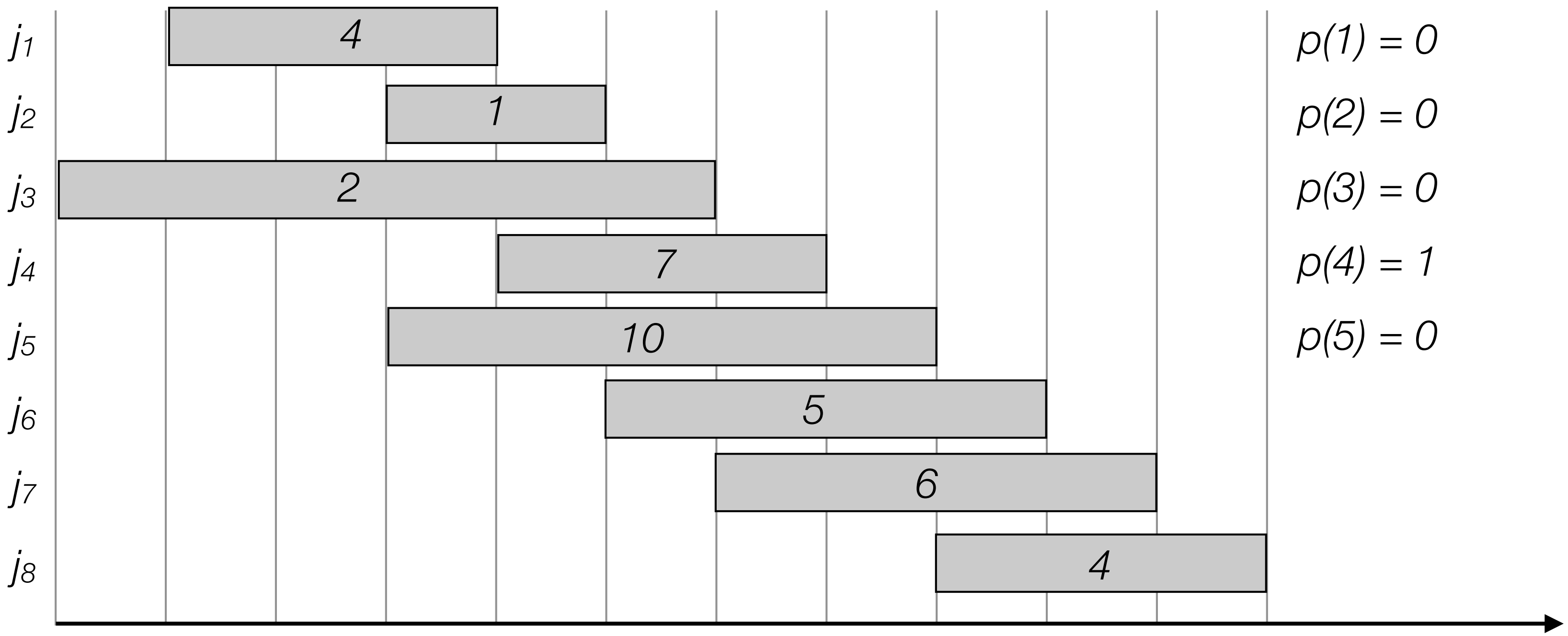
$p(2) = 0$

$p(3) = 0$

$p(4) = 1$

Weighted interval scheduling

- Label/sort 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.



$p(1) = 0$

$p(2) = 0$

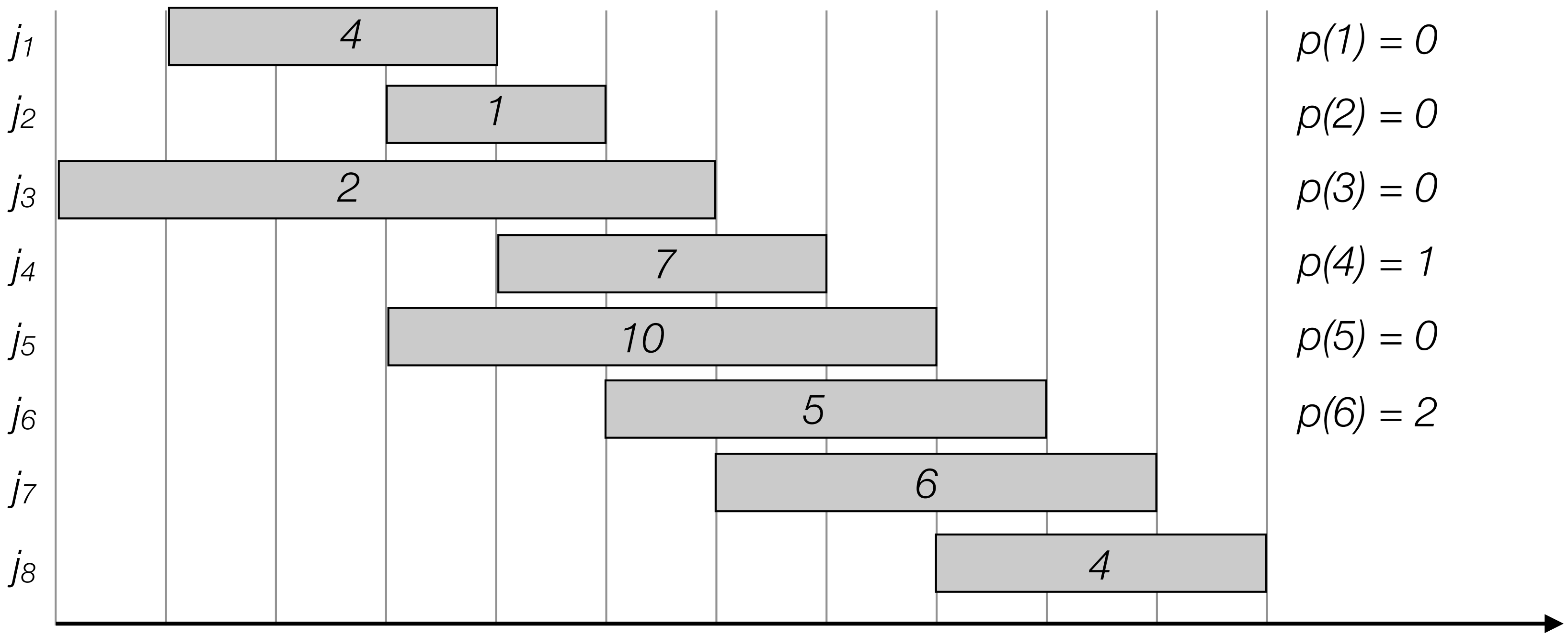
$p(3) = 0$

$p(4) = 1$

$p(5) = 0$

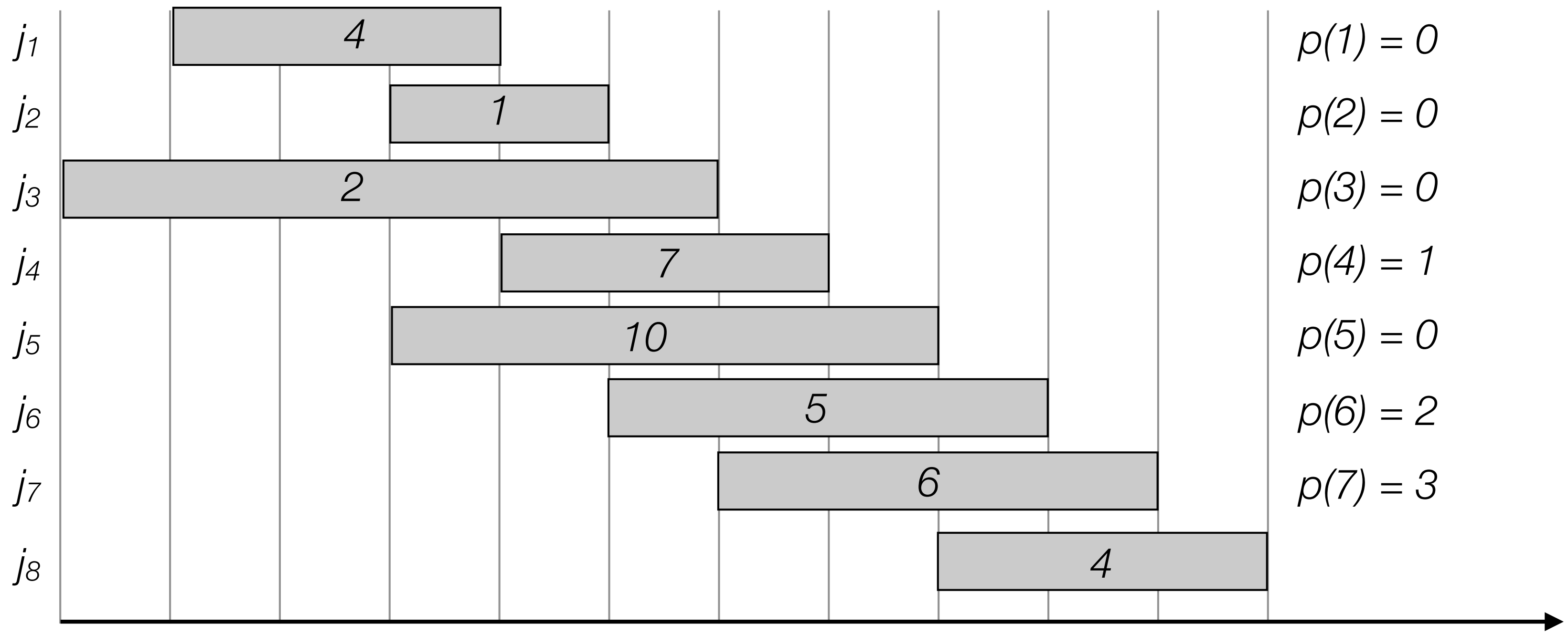
Weighted interval scheduling

- Label/sort 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.



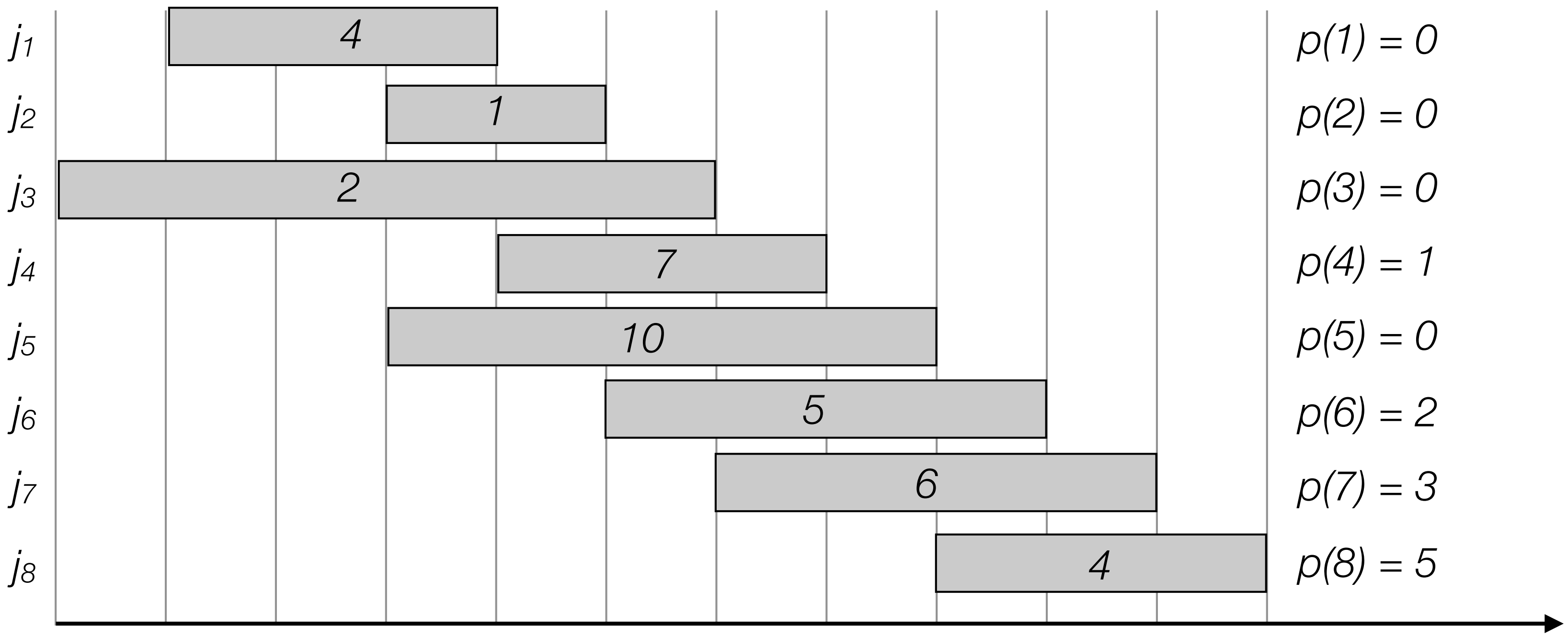
Weighted interval scheduling

- Label/sort 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.



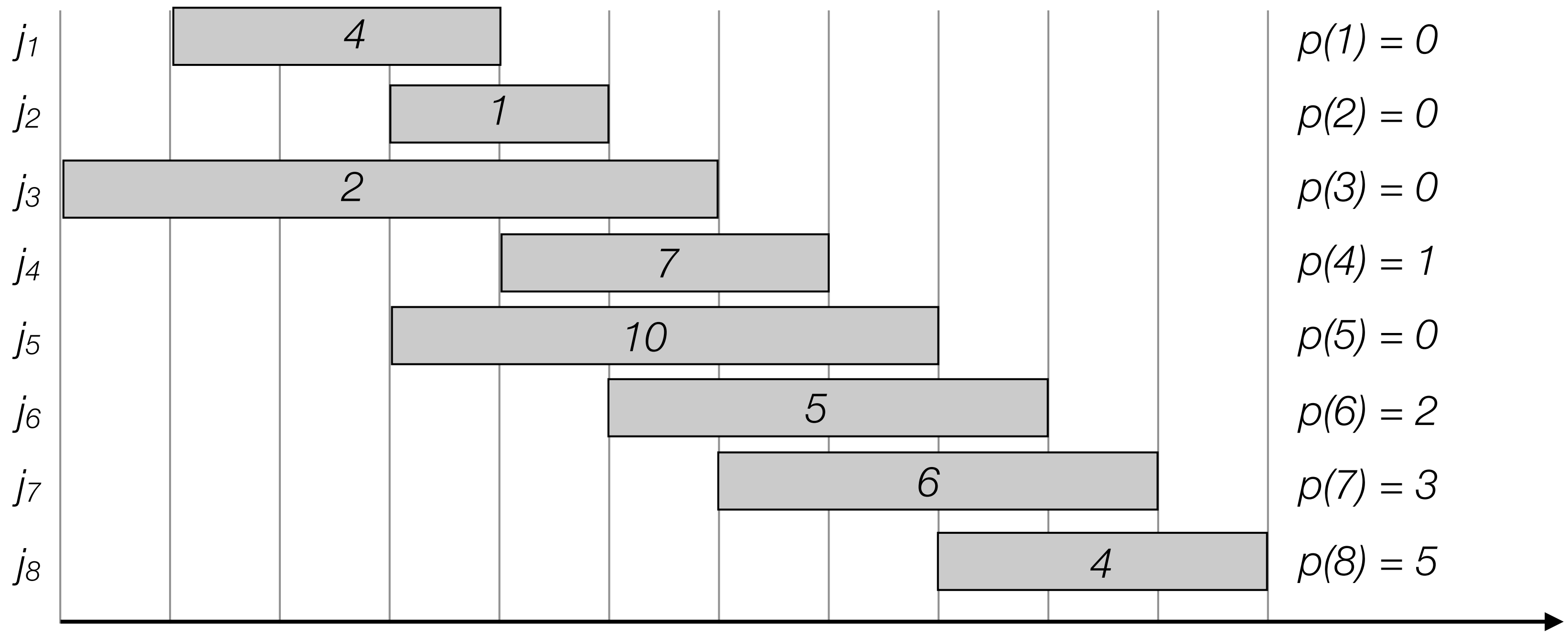
Weighted interval scheduling

- Label/sort 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.



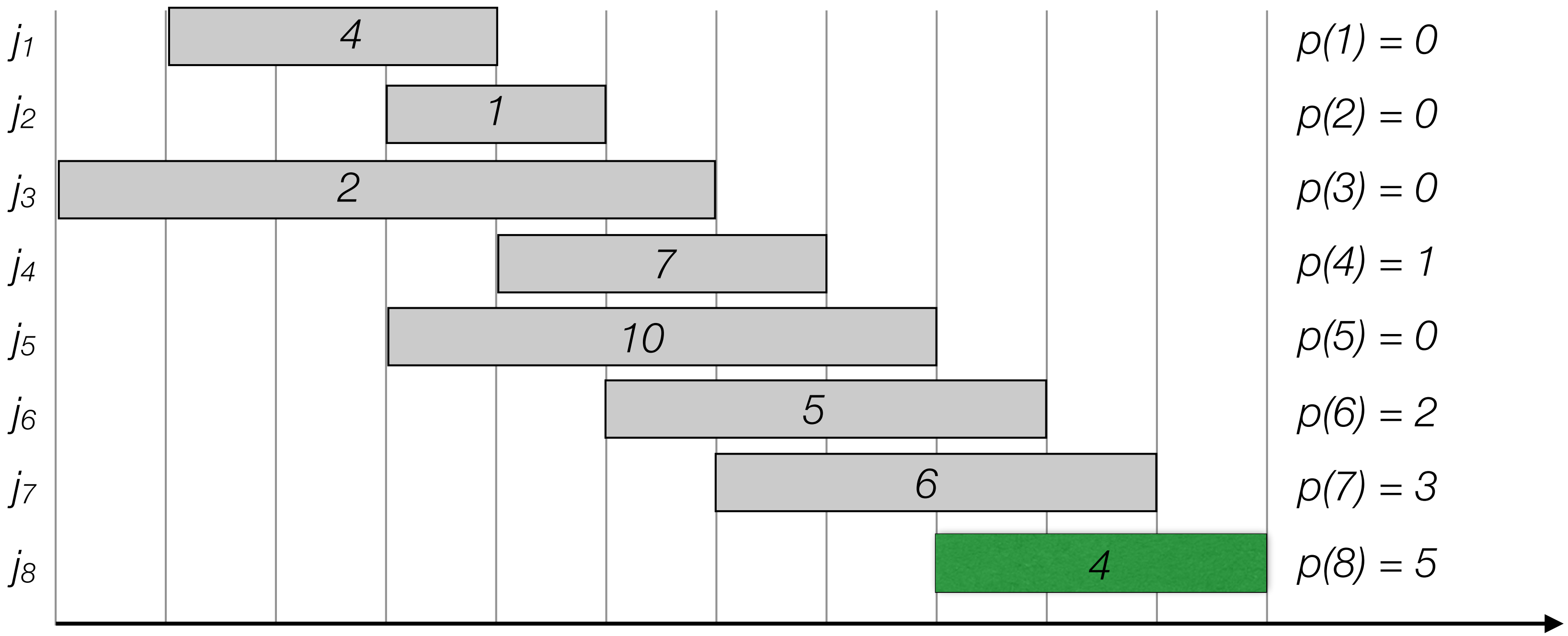
Weighted interval scheduling

- Label/sort 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.
- Optimal solution OPT:



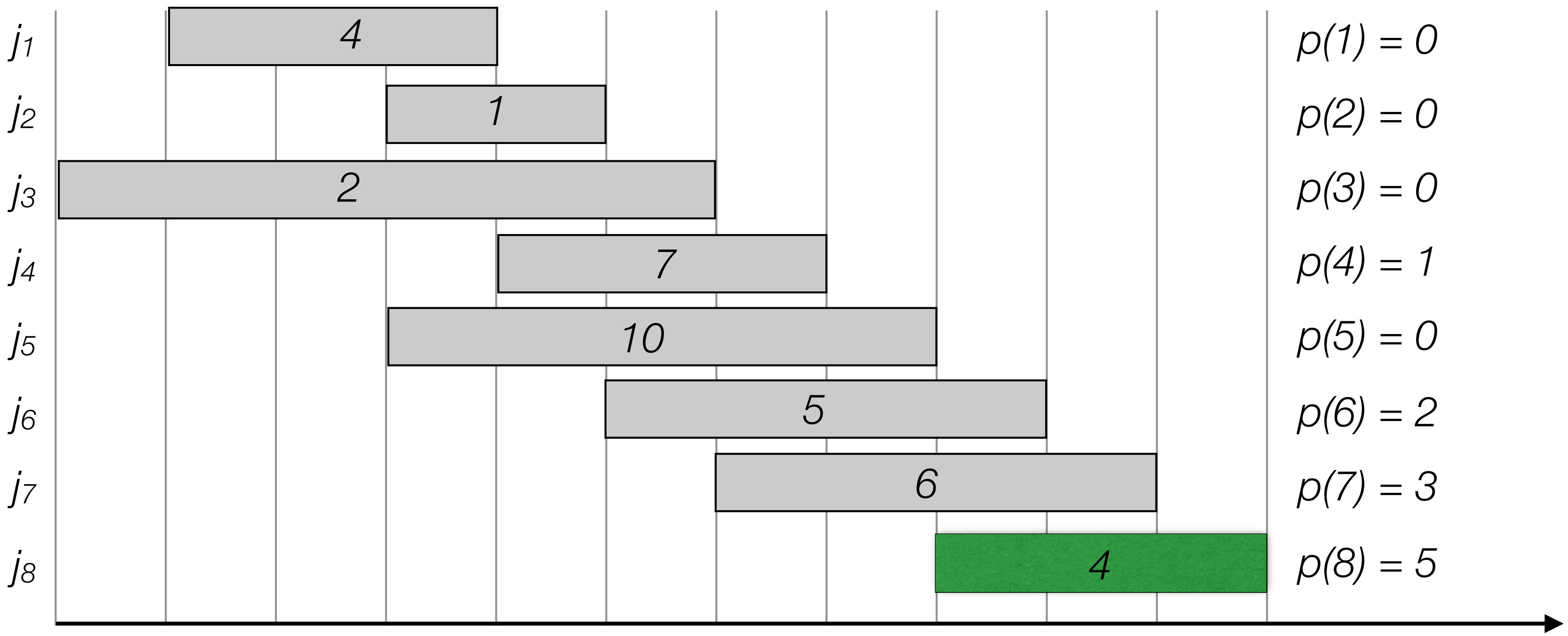
Weighted interval scheduling

- Label/sort 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.
- Optimal solution OPT:



Weighted interval scheduling

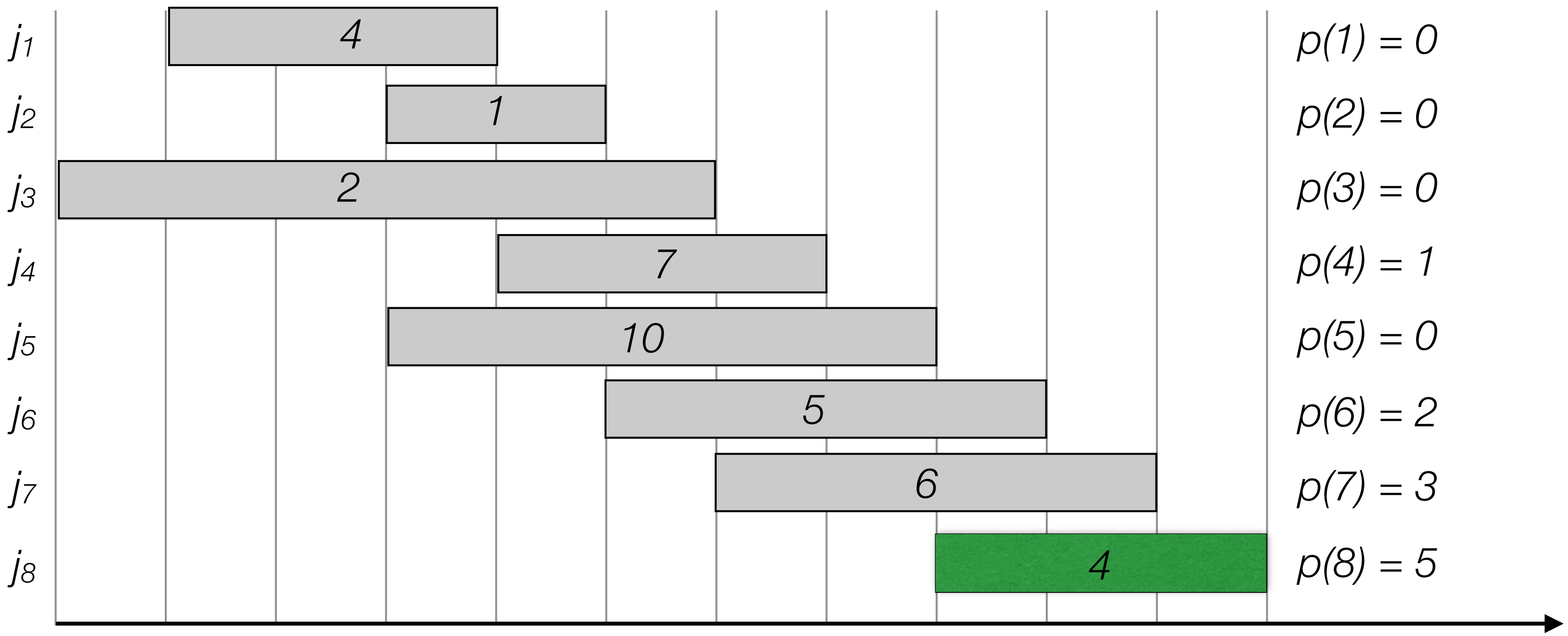
- Label/sort jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$
- $p(j)$ = largest index $i < j$ such that job i is compatible with j.
- Optimal solution OPT:
  - Case 1. OPT selects last job



Weighted interval scheduling

- Label/sort 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.
- Optimal solution OPT:
  - Case 1. OPT selects last job

$$OPT = v_n + \text{optimal solution to subproblem on } 1,\ldots,p(n)$$

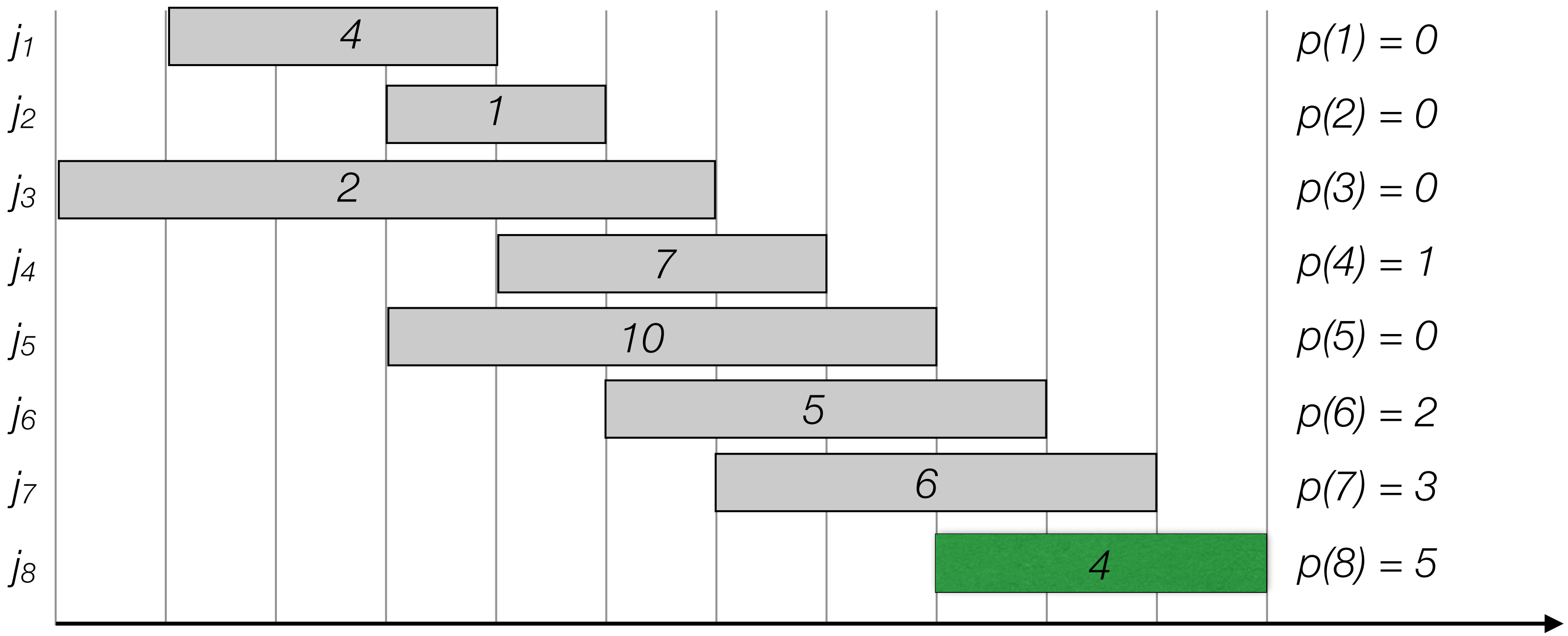


Weighted interval scheduling

- Label/sort 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.
- Optimal solution OPT:
  - Case 1. OPT selects last job

    $$OPT = v_n + \text{optimal solution to subproblem on } 1,\ldots,p(n)$$

  - Case 2. OPT does not select last job

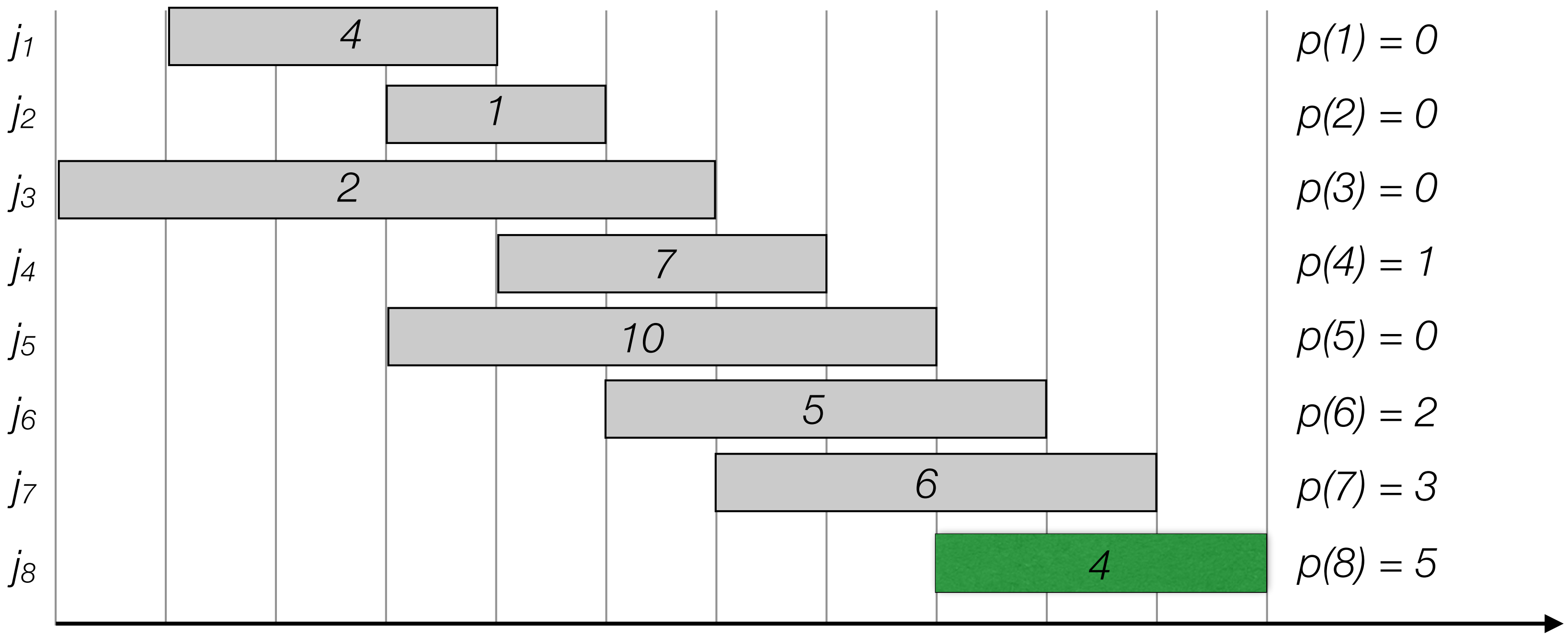


Weighted interval scheduling

- Label/sort 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.
- Optimal solution OPT:
  - Case 1. OPT selects last job

    $$OPT = v_n + \text{optimal solution to subproblem on } 1,\ldots,p(n)$$
  - Case 2. OPT does not select last job

    $$OPT = \text{optimal solution to subproblem on } 1,\ldots,n$$



Weighted interval scheduling

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

  - Case 1. OPT(*j*) selects job j

    OPT(j) = v_j + optimal solution to subproblem on 1,…,p(j)

  - Case 2. OPT(*j*) does not job j

    OPT = optimal solution to subproblem 1,…j-1

Weighted interval scheduling

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

  - Case 1. OPT(*j*) selects job j

    OPT(j) = v_j + optimal solution to subproblem on 1,…,p(j)

  - Case 2. OPT(*j*) does not job j

    OPT = optimal solution to subproblem 1,…j-1

- Recursion:

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

Weighted interval scheduling: brute force

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

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```

Weighted interval scheduling: brute force

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

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```

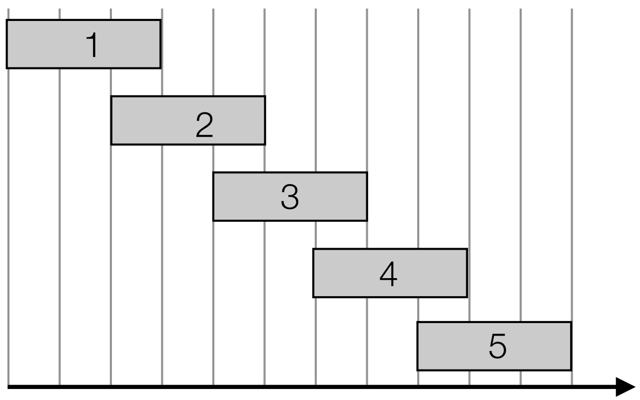
Weighted interval scheduling: brute force

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

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```



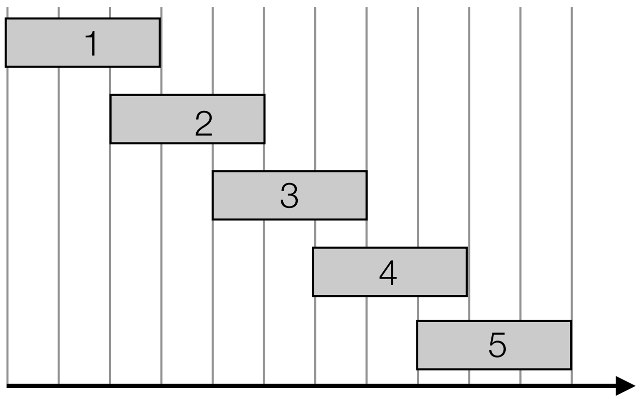
Weighted interval scheduling: brute force

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

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```



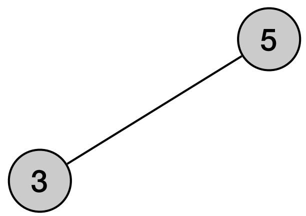
Weighted interval scheduling: brute force

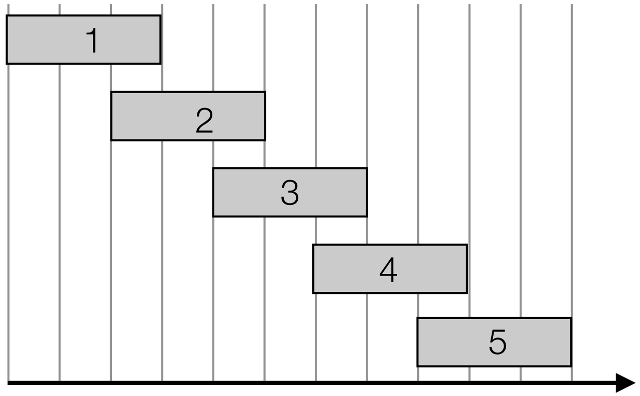
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```





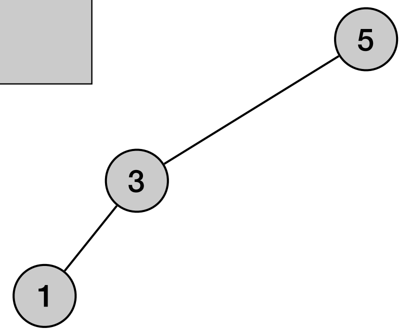
Weighted interval scheduling: brute force

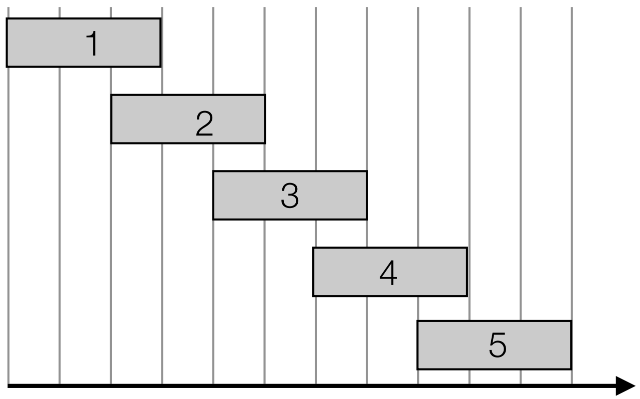
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```





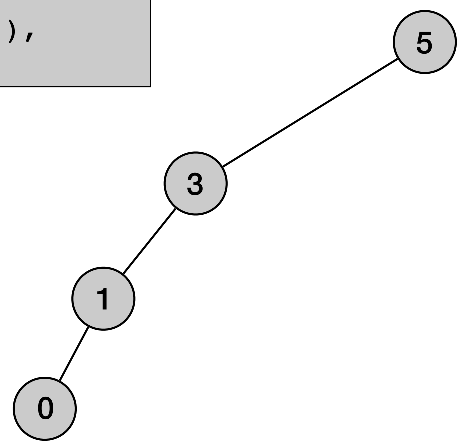
Weighted interval scheduling: brute force

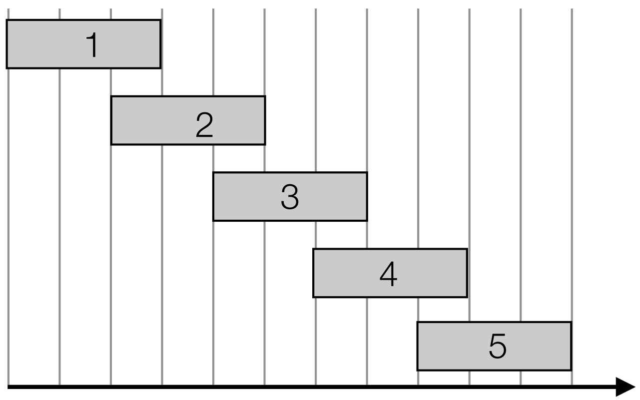
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```





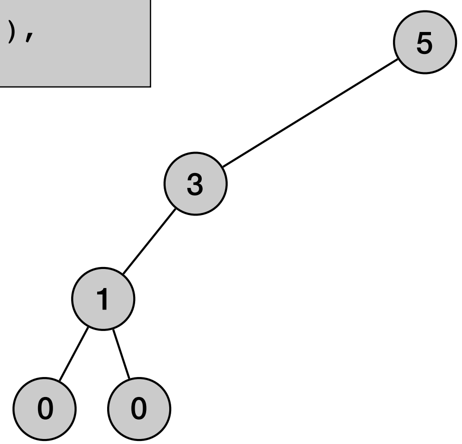
Weighted interval scheduling: brute force

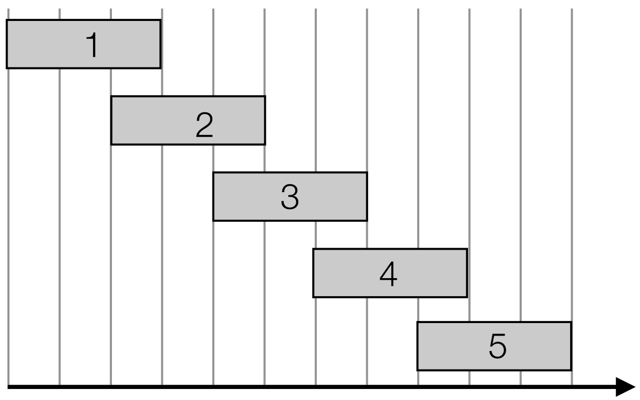
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```





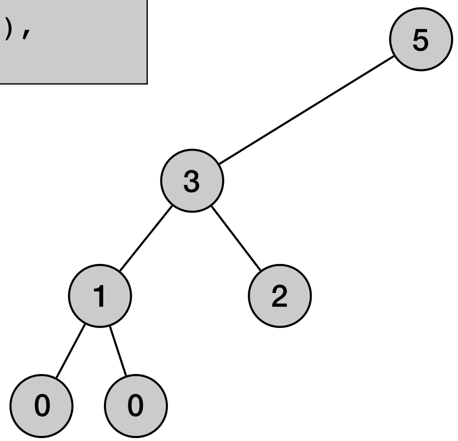
Weighted interval scheduling: brute force

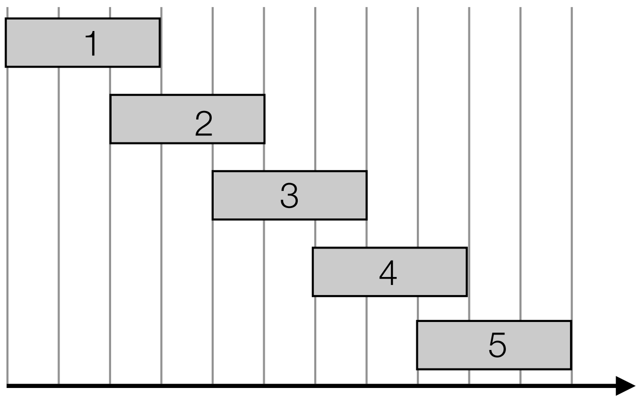
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```





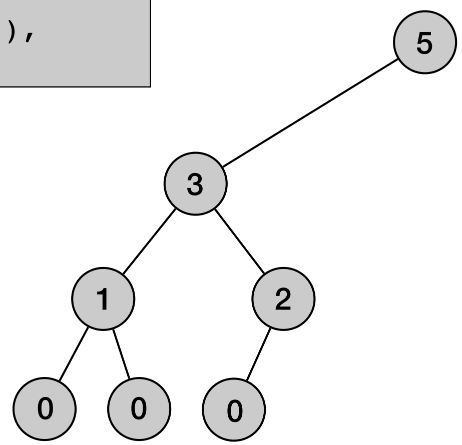
Weighted interval scheduling: brute force

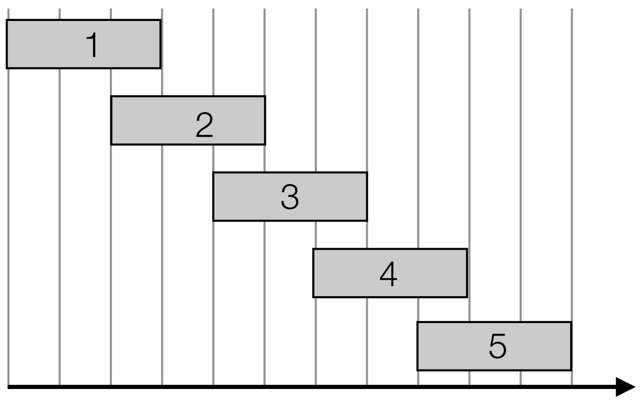
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
        Compute-Brute-Force-Opt(j-1))
```





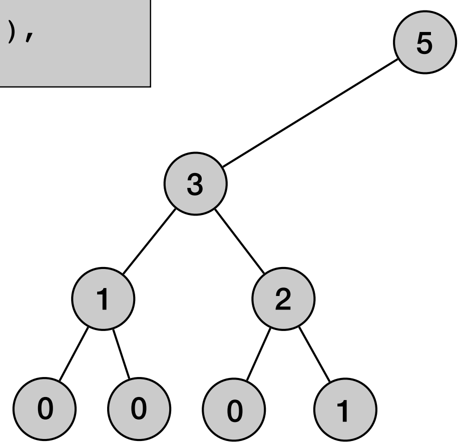
Weighted interval scheduling: brute force

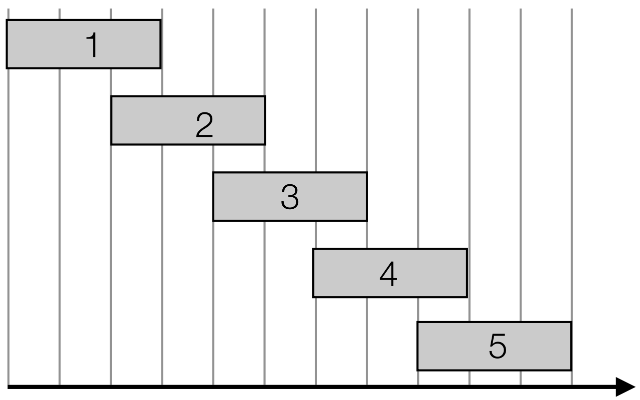
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```





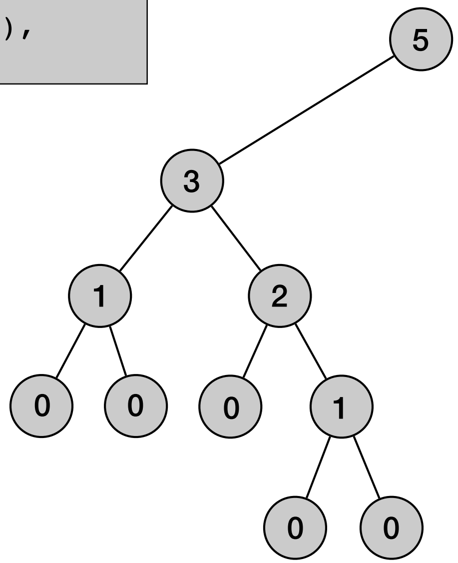
Weighted interval scheduling: brute force

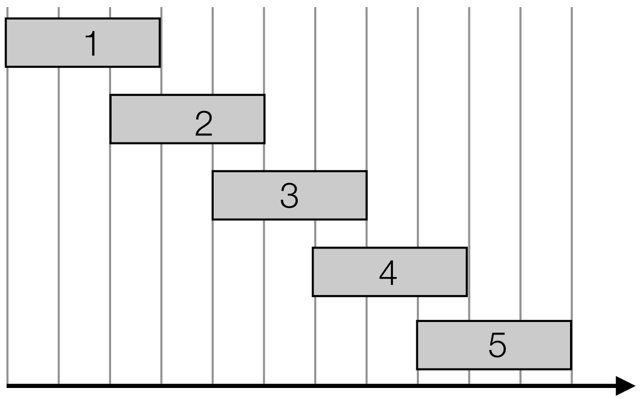
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```





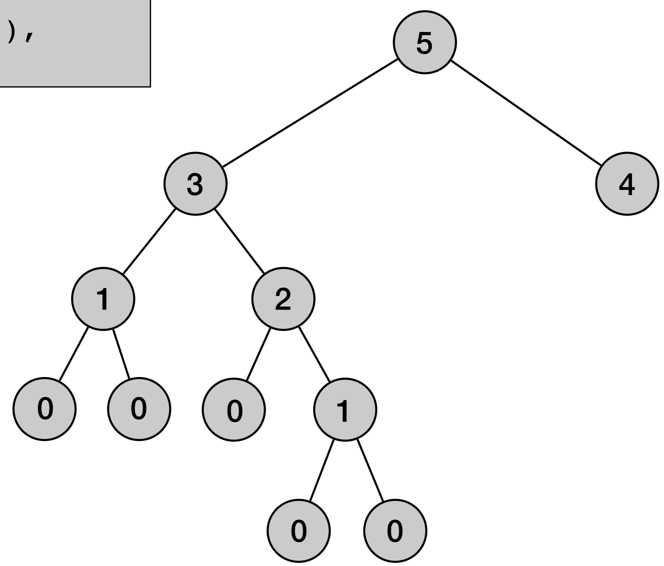
Weighted interval scheduling: brute force

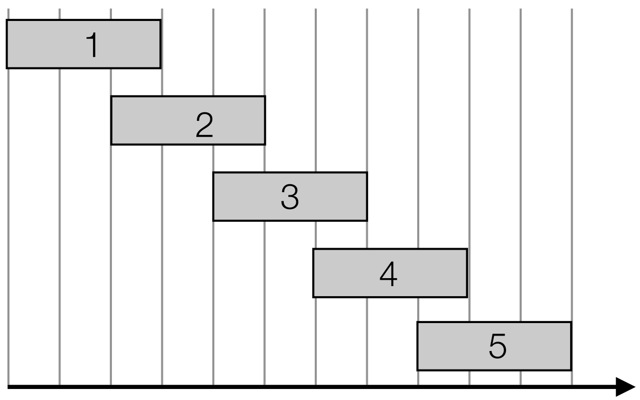
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```





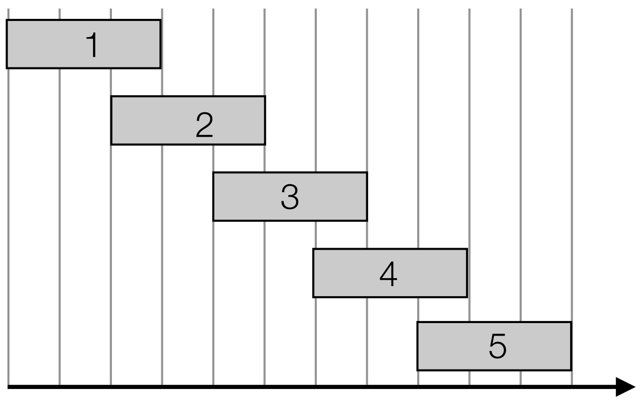
Weighted interval scheduling: brute force

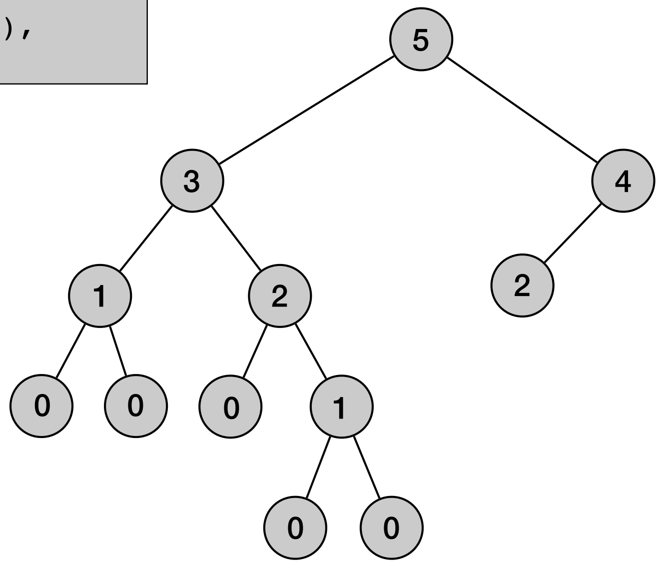
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```





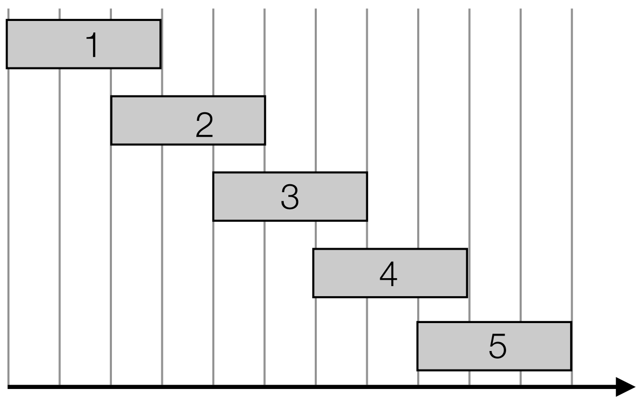
Weighted interval scheduling: brute force

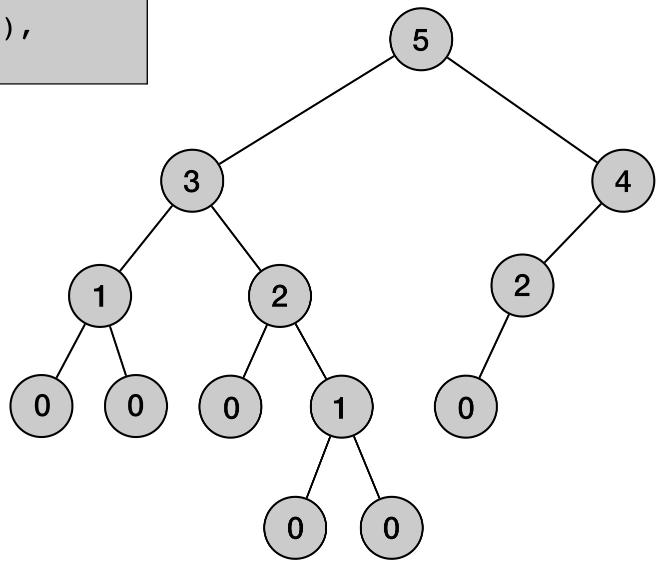
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```





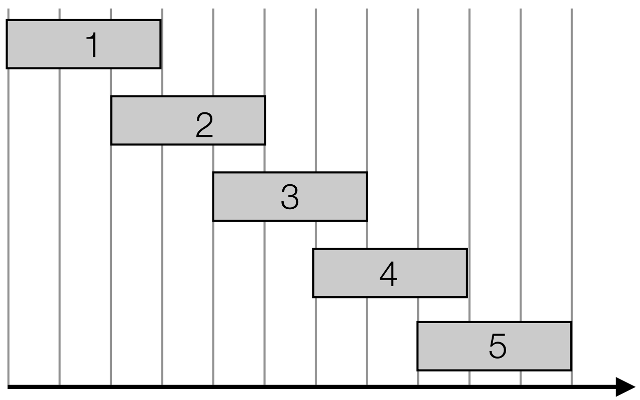
Weighted interval scheduling: brute force

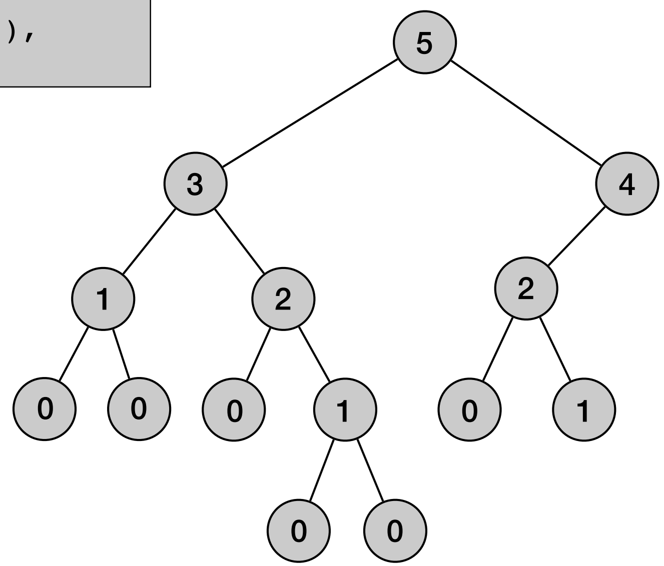
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```





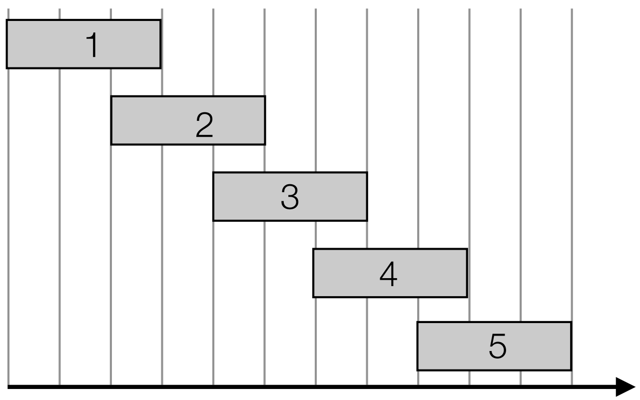
Weighted interval scheduling: brute force

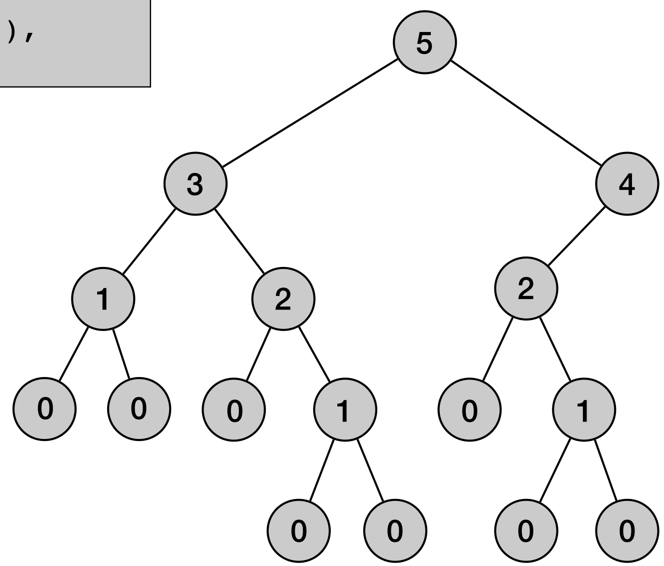
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```





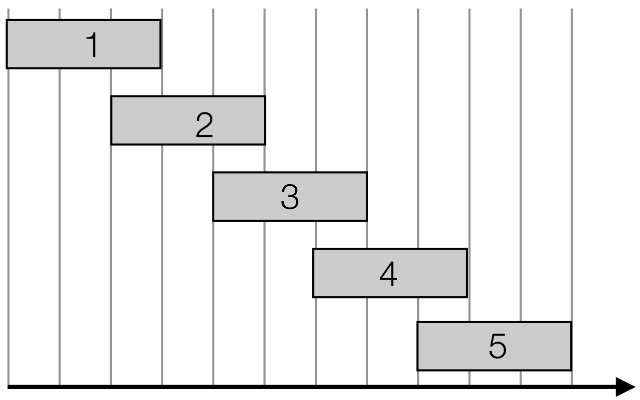
Weighted interval scheduling: brute force

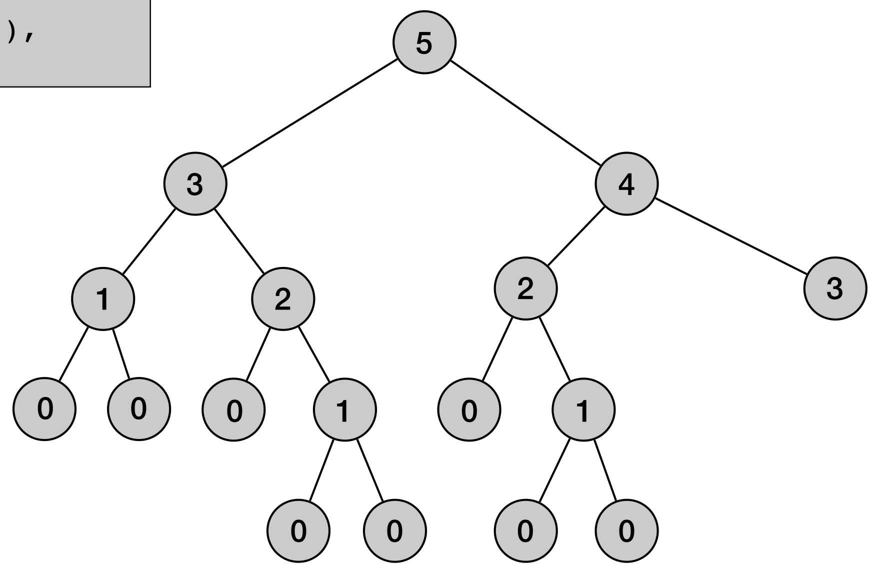
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```





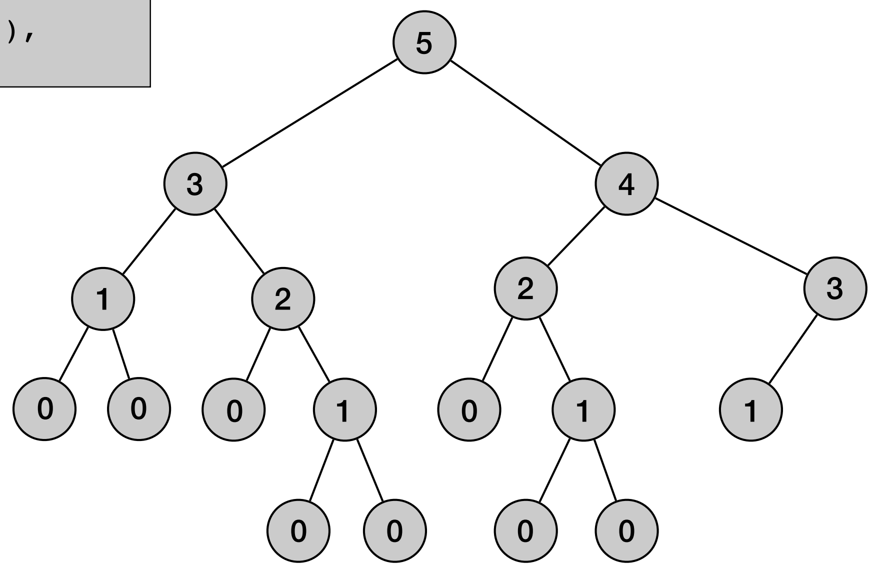
Weighted interval scheduling: brute force

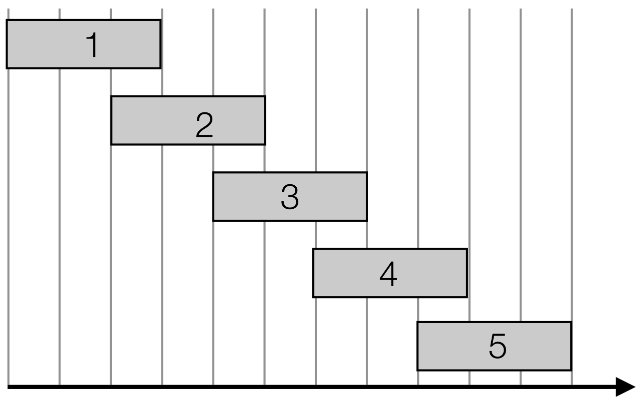
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
        Compute-Brute-Force-Opt(j-1))
```





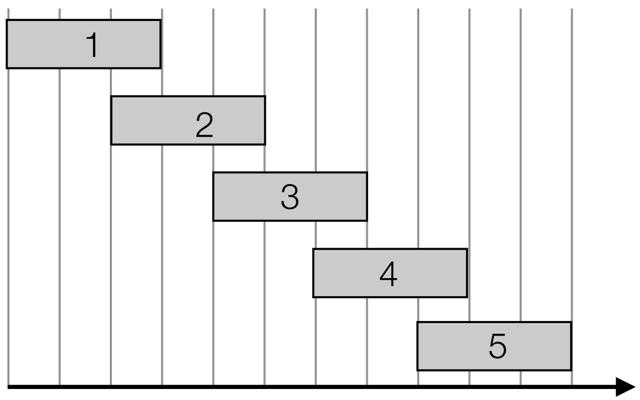
Weighted interval scheduling: brute force

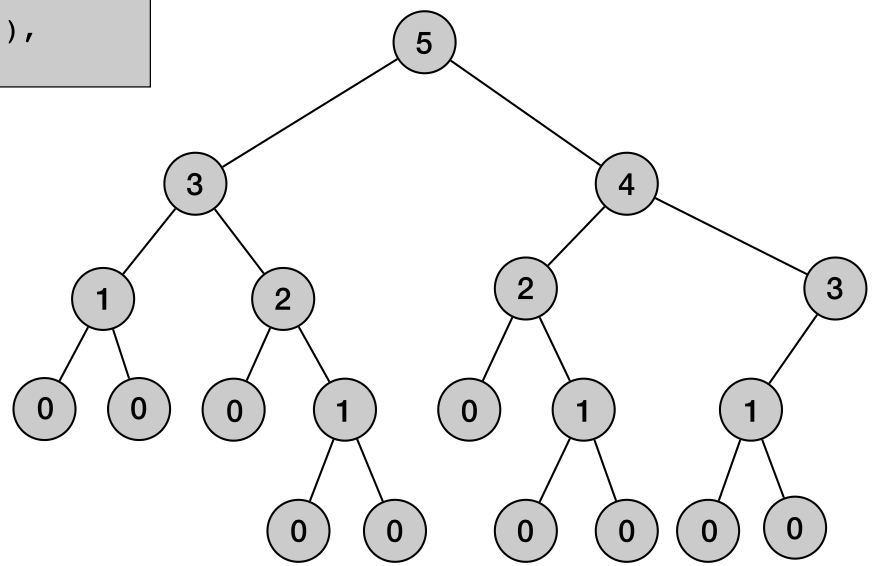
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```





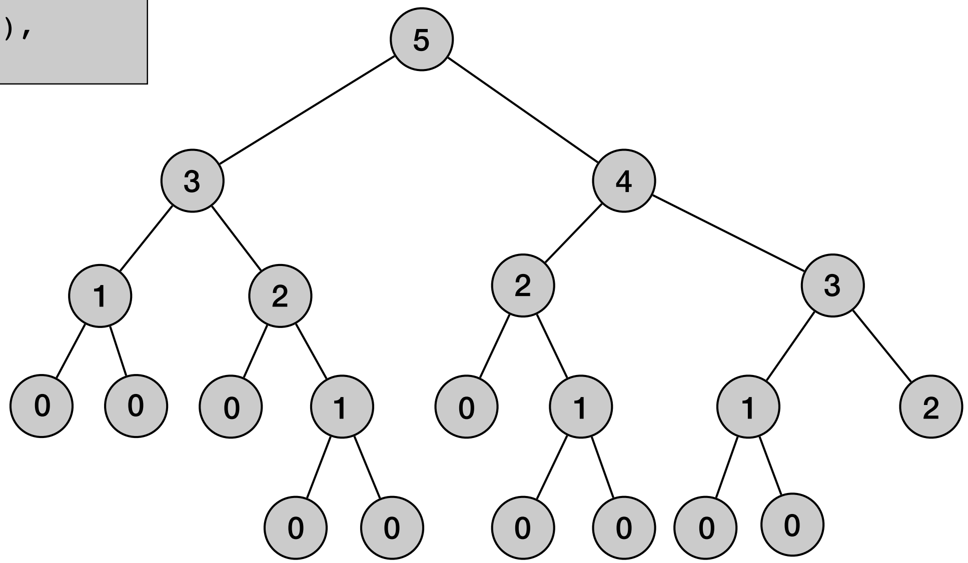
Weighted interval scheduling: brute force

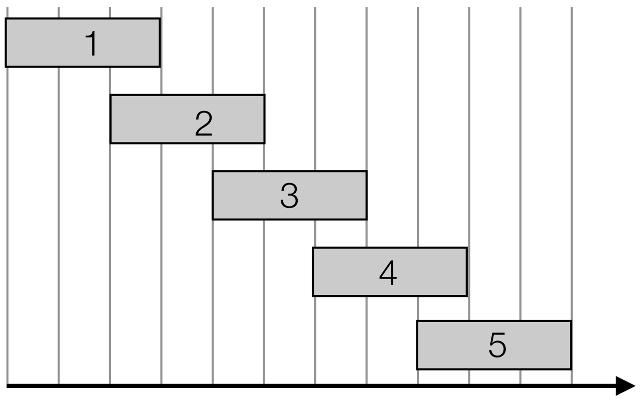
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
        Compute-Brute-Force-Opt(j-1))
```





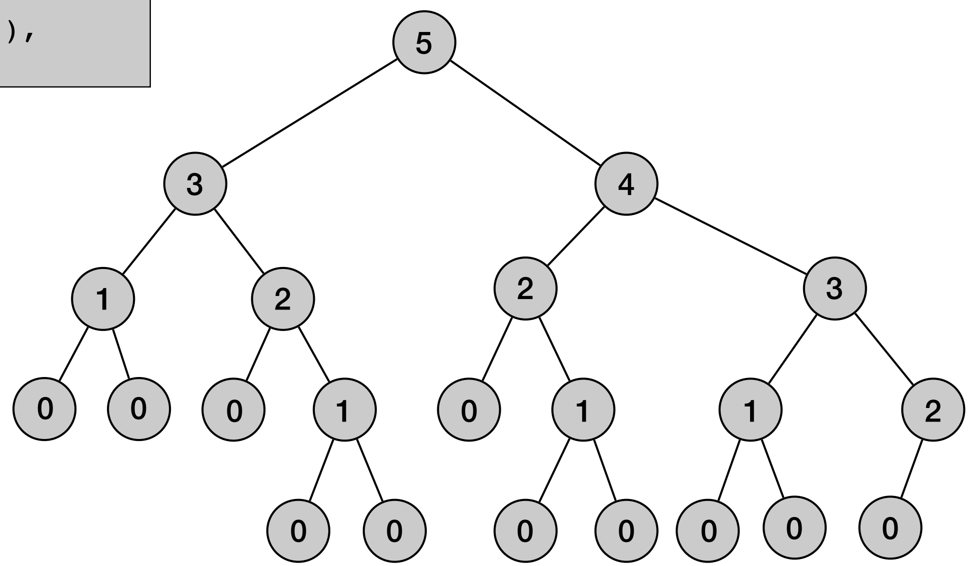
Weighted interval scheduling: brute force

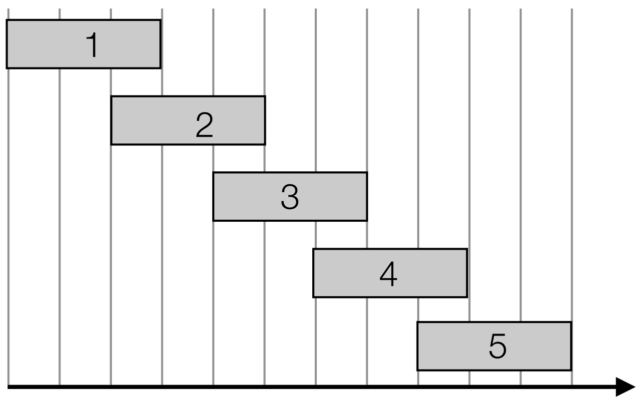
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```





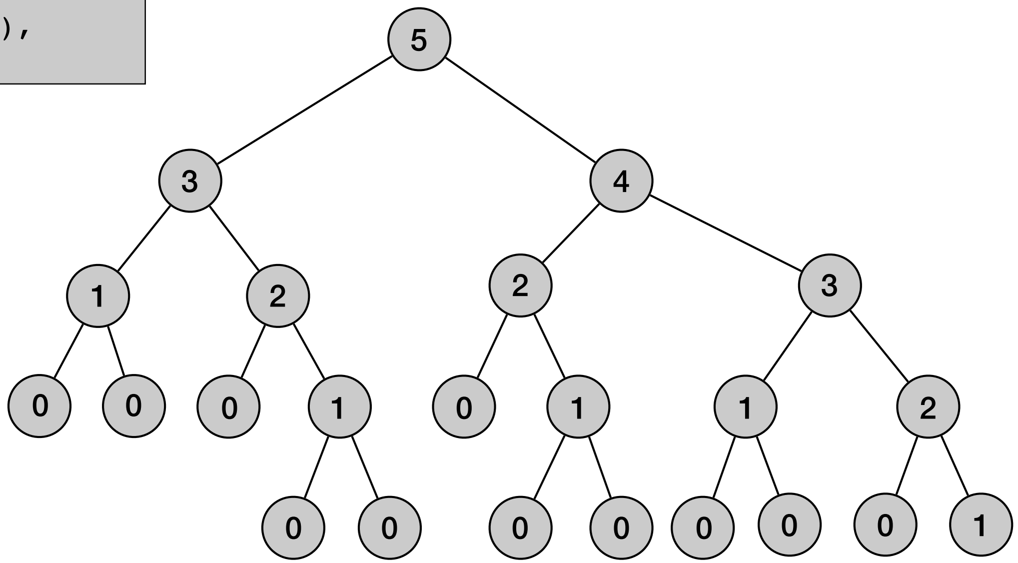
Weighted interval scheduling: brute force

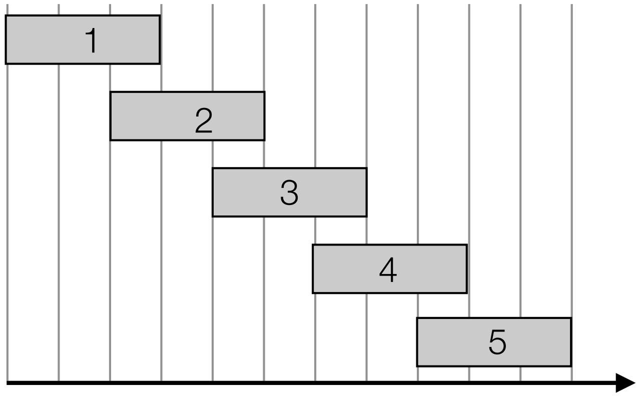
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```





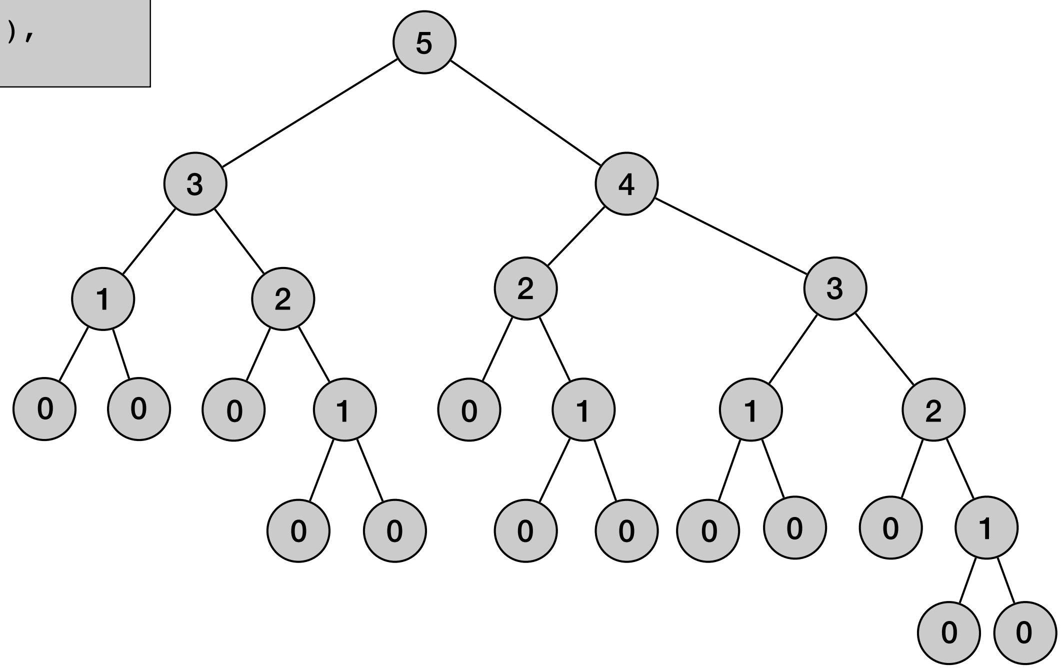
Weighted interval scheduling: brute force

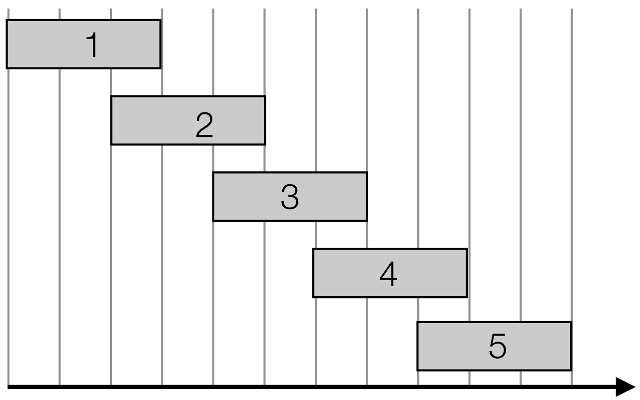
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```





Weighted interval scheduling: brute force

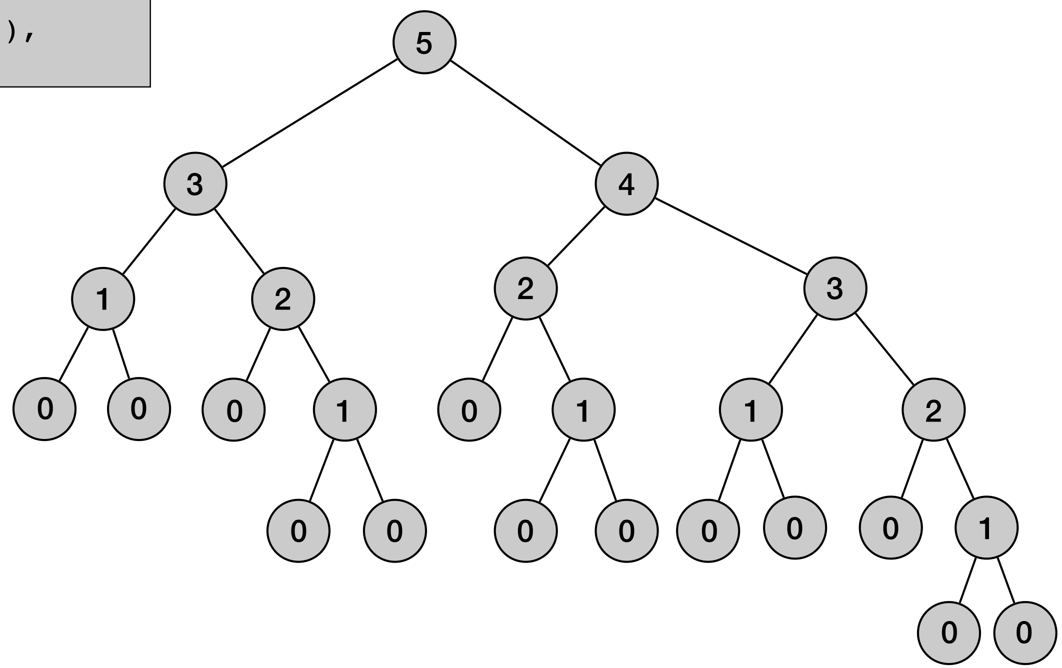
$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \text{otherwise} \end{cases}$$

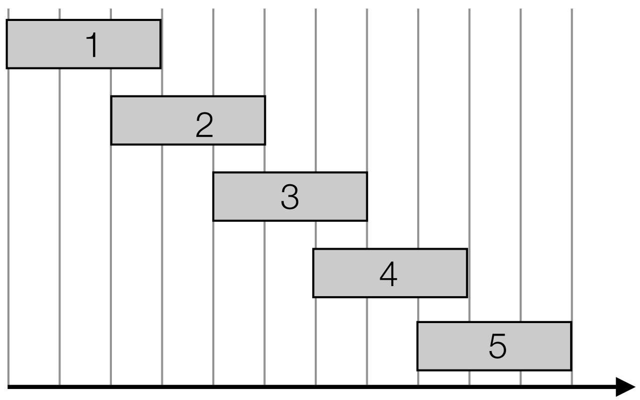
```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```

time Θ(2^n)





Weighted interval scheduling: brute force

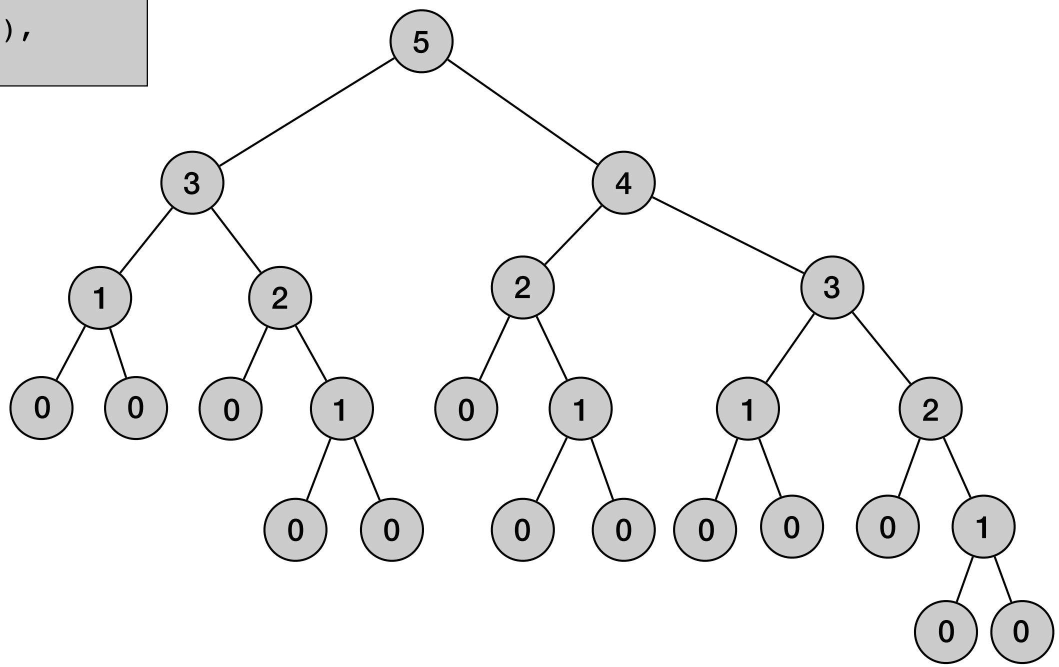
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

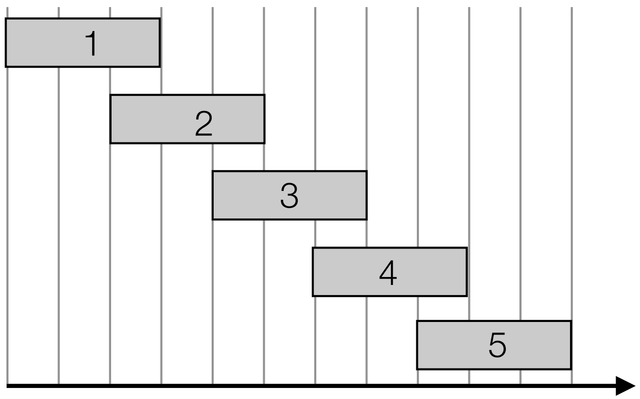
```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```

time Θ(2^n)





Weighted interval scheduling: brute force

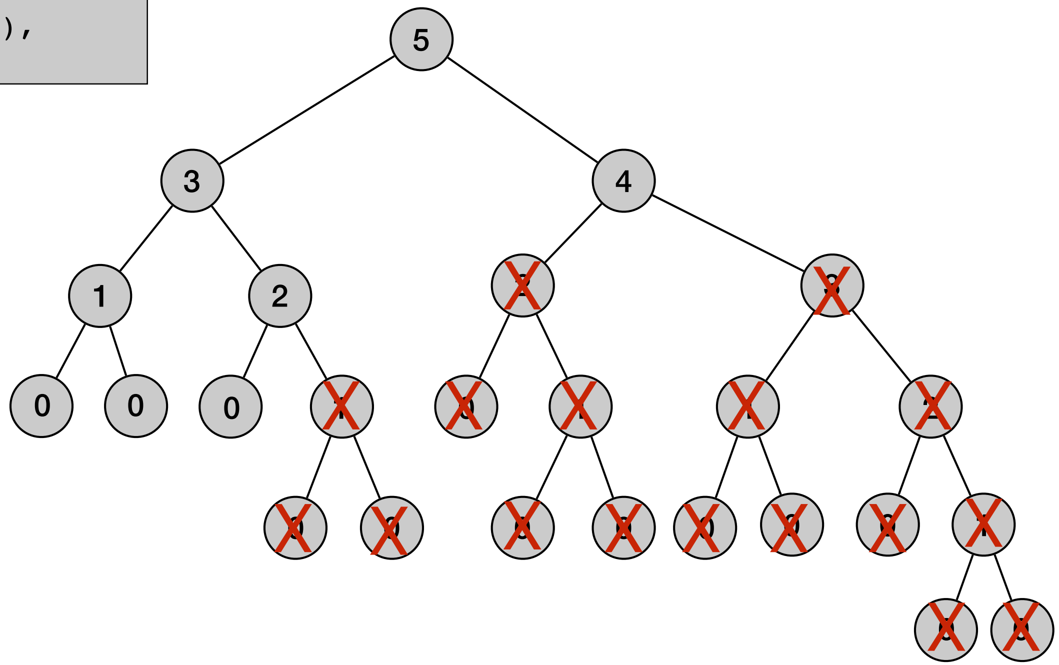
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

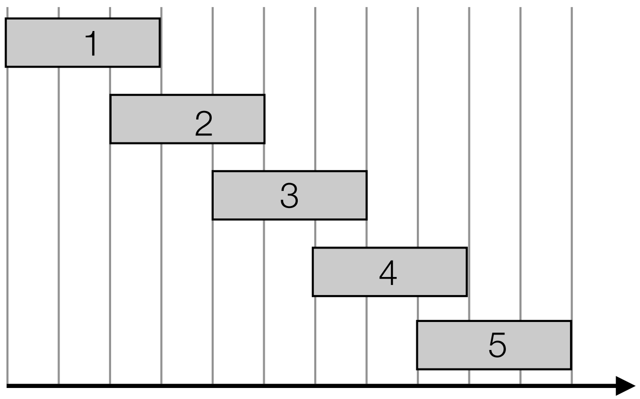
```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```

time Θ(2^n)





Weighted interval scheduling: brute force

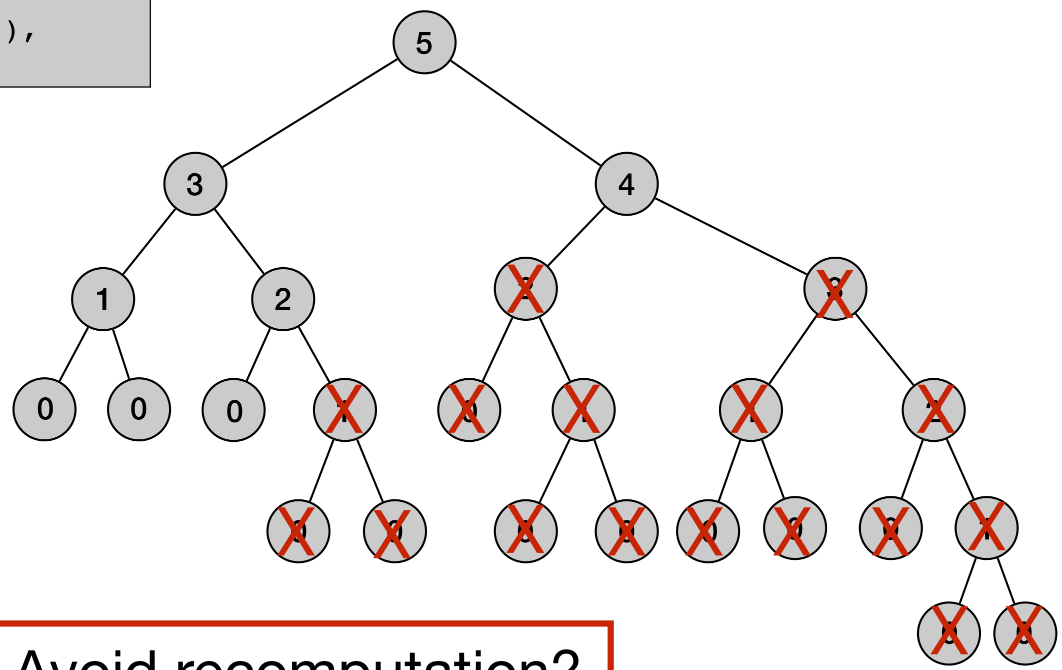
$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \texttt{otherwise} \end{cases}$$

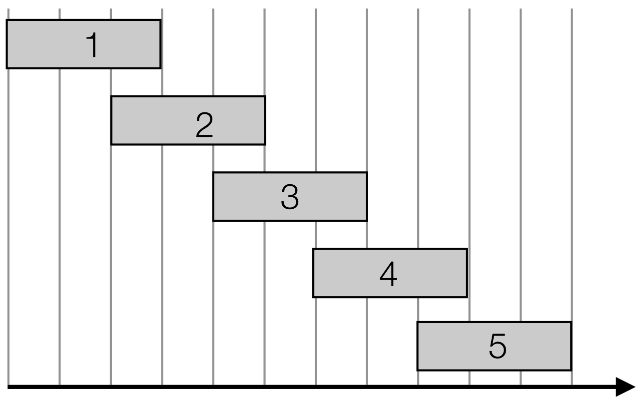
```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

Compute-Brute-Force-Opt(j)
if j = 0
  return 0
else
  return max(v[j] + Compute-Brute-Force-Opt(p[j]),
         Compute-Brute-Force-Opt(j-1))
```

time Θ(2^n)





Avoid recomputation?

Weighted interval scheduling: memoization

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

for j=1 to n
  M[j] = null
M[0] = 0.

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

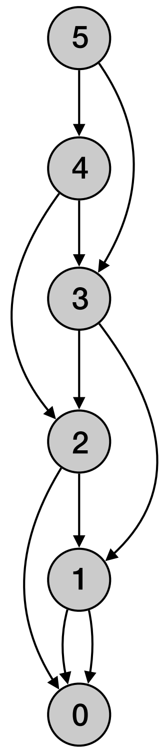
Weighted interval scheduling: memoization

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

for j=1 to n
  M[j] = null
M[0] = 0.

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



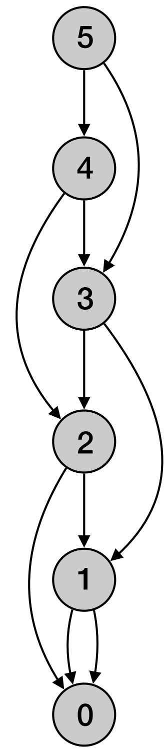
Weighted interval scheduling: memoization

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

for j=1 to n
  M[j] = null
M[0] = 0.

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



- Running time O(n log n):

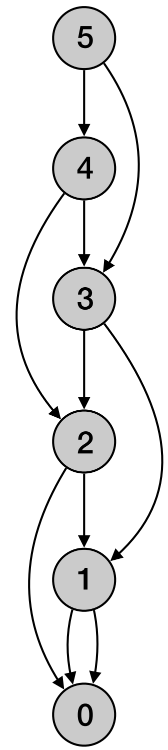
Weighted interval scheduling: memoization

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

for j=1 to n
  M[j] = null
M[0] = 0.

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



- Running time O(n log n):
  - Sorting takes O(n log n) time.

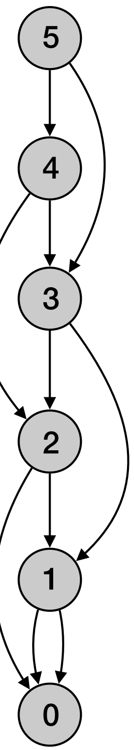
Weighted interval scheduling: memoization

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

for j=1 to n
  M[j] = null
M[0] = 0.

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



- Running time O(n log n):
  - Sorting takes O(n log n) time.
  - Computing p(n): O(n log n) by using sort by start time

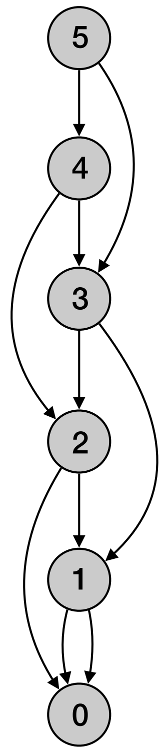
Weighted interval scheduling: memoization

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

for j=1 to n
  M[j] = null
M[0] = 0.

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



- Running time O(n log n):
  - Sorting takes O(n log n) time.
  - Computing p(n): O(n log n) by using sort by start time
  - Each subproblem solved once.

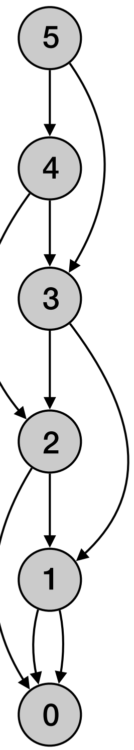
Weighted interval scheduling: memoization

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

for j=1 to n
  M[j] = null
M[0] = 0.

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



- Running time O(n log n):
  - Sorting takes O(n log n) time.
  - Computing p(n): O(n log n) by using sort by start time
  - Each subproblem solved once.
  - Time to solve a subproblem constant.

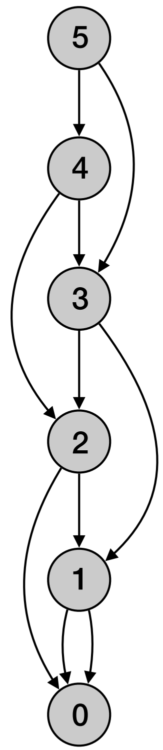
Weighted interval scheduling: memoization

```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

for j=1 to n
  M[j] = null
M[0] = 0.

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



- Running time O(n log n):
  - Sorting takes O(n log n) time.
  - Computing p(n): O(n log n) by using sort by start time
  - Each subproblem solved once.
  - Time to solve a subproblem constant.
- Space O(n)

Weighted interval scheduling: memoization

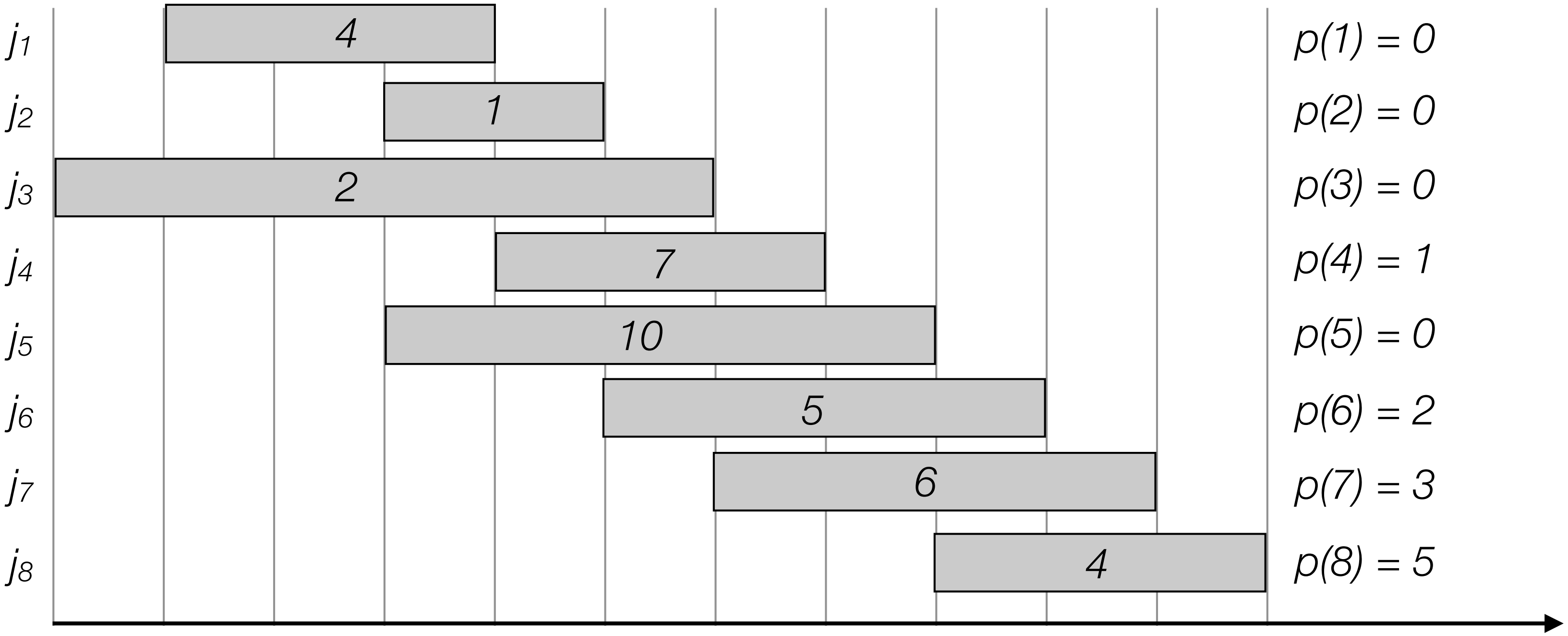
```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time 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.

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

i	M[i]
0	
1	
2	
3	
4	
5	
6	
7	
8	



Weighted interval scheduling: memoization

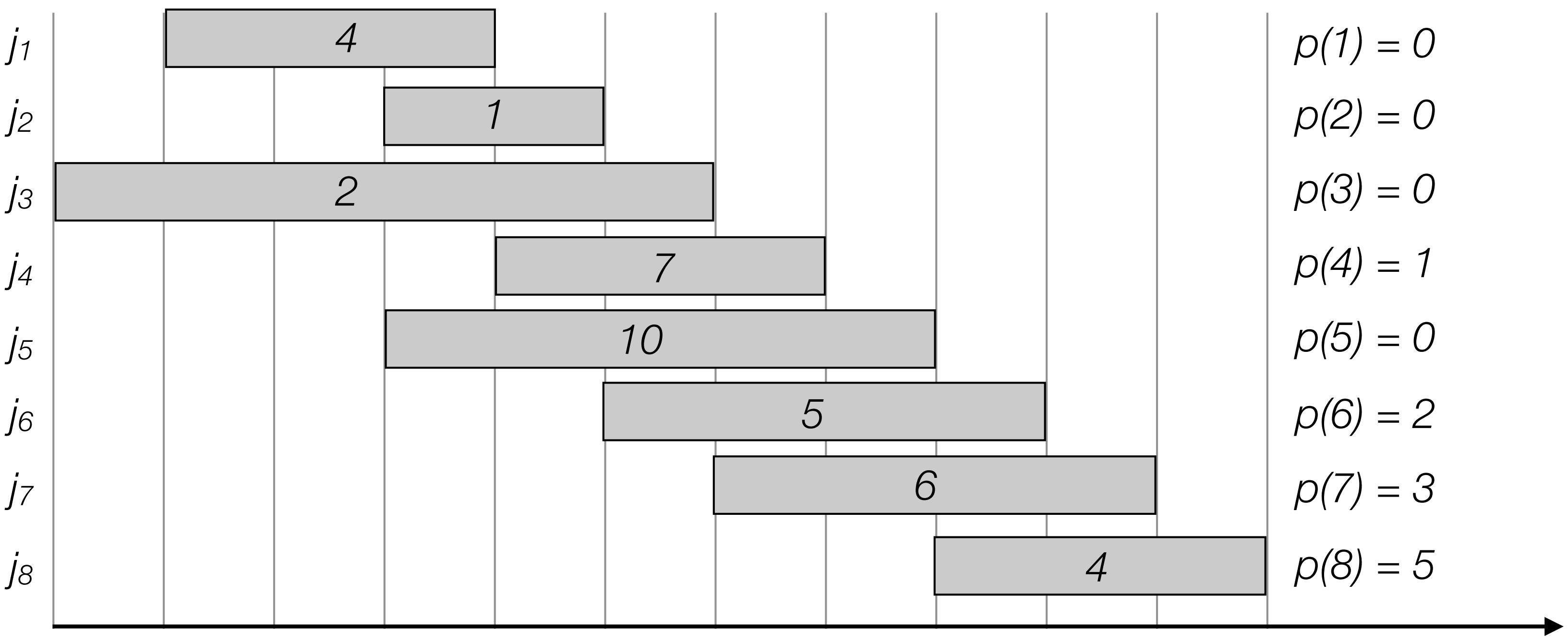
```
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time 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.

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

i	M[i]
0	0
1	4
2	4
3	4
4	11
5	11
6	11
7	11
8	15

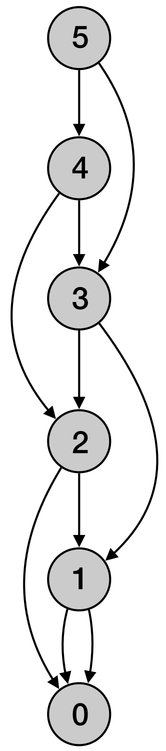


Weighted interval scheduling: bottom-up

```
Compute-Bottom-Up–Opt(n, s[1..n], f[1..n], v[1..n])

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

M[0] = 0.
for j=1 to n
  M[j] = max(v[j] + M(p[j]), M(j-1))
return M[n]
```

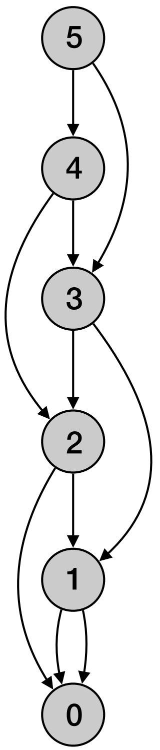


Weighted interval scheduling: bottom-up

```
Compute-Bottom-Up—Opt(n, s[1..n], f[1..n], v[1..n])

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

M[0] = 0.
for j=1 to n
  M[j] = max(v[j] + M(p[j]), M(j-1))
return M[n]
```



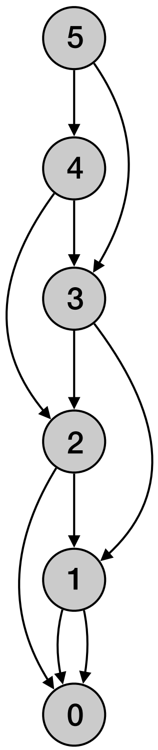
- Running time O(n log n):

Weighted interval scheduling: bottom-up

```
Compute-Bottom-Up—Opt(n, s[1..n], f[1..n], v[1..n])

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

M[0] = 0.
for j=1 to n
  M[j] = max(v[j] + M(p[j]), M(j-1))
return M[n]
```



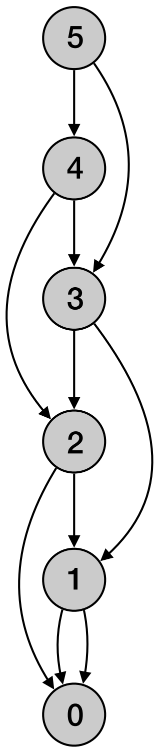
- Running time O(n log n):
  - Sorting takes O(n log n) time.

Weighted interval scheduling: bottom-up

```
Compute-Bottom-Up—Opt(n, s[1..n], f[1..n], v[1..n])

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

M[0] = 0.
for j=1 to n
  M[j] = max(v[j] + M(p[j]), M(j-1))
return M[n]
```



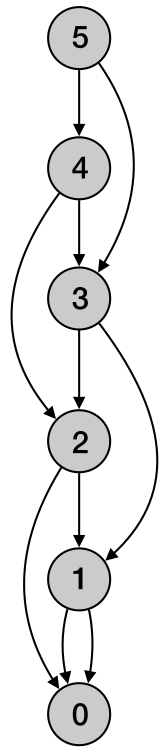
- Running time O(n log n):
  - Sorting takes O(n log n) time.
  - Computing p(n): O(n log n) by using sort by start time

Weighted interval scheduling: bottom-up

```
Compute-Bottom-Up—Opt(n, s[1..n], f[1..n], v[1..n])

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

M[0] = 0.
for j=1 to n
  M[j] = max(v[j] + M(p[j]), M(j-1))
return M[n]
```



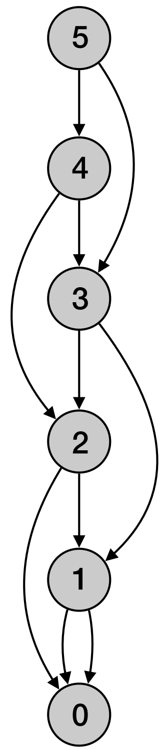
- Running time O(n log n):
  - Sorting takes O(n log n) time.
  - Computing p(n): O(n log n) by using sort by start time
  - For loop: O(n) time

Weighted interval scheduling: bottom-up

```
Compute-Bottom-Up–Opt(n, s[1..n], f[1..n], v[1..n])

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

M[0] = 0.
for j=1 to n
  M[j] = max(v[j] + M(p[j]), M(j-1))
return M[n]
```



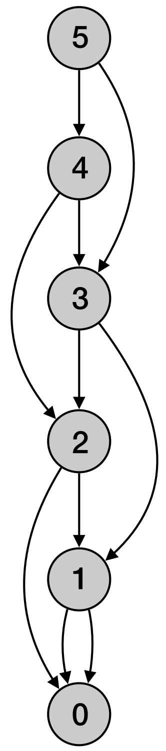
- Running time O(n log n):
  - Sorting takes O(n log n) time.
  - Computing p(n): O(n log n) by using sort by start time
  - For loop: O(n) time
    - Each iteration takes constant time.

Weighted interval scheduling: bottom-up

```
Compute-Bottom-Up—Opt(n, s[1..n], f[1..n], v[1..n])

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

M[0] = 0.
for j=1 to n
  M[j] = max(v[j] + M(p[j]), M(j-1))
return M[n]
```



- Running time O(n log n):
  - Sorting takes O(n log n) time.
  - Computing p(n): O(n log n) by using sort by start time
  - For loop: O(n) time
    - Each iteration takes constant time.
- Space O(n)

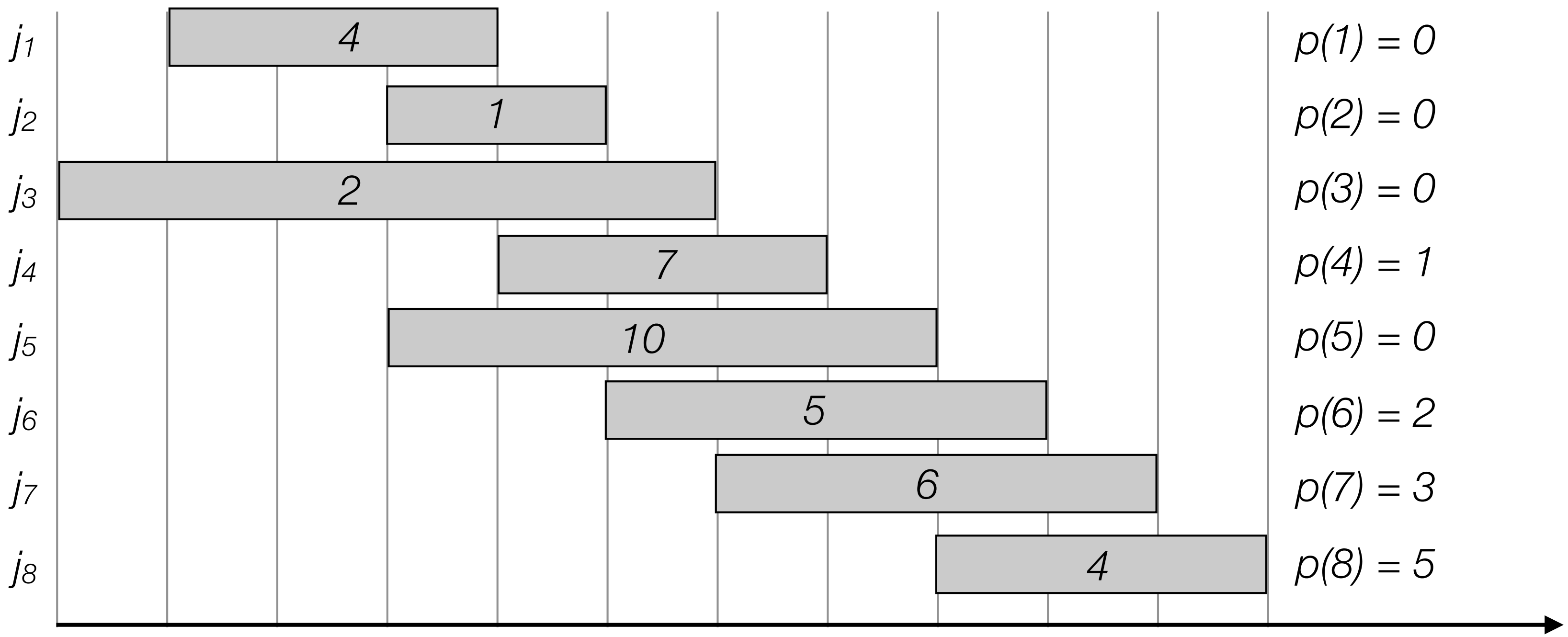
Weighted interval scheduling: bottom-up

```
Compute-Bottom-Up-Opt(n, s[1..n], f[1..n], v[1..n])

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

M[0] = 0.
for j=1 to n
   M[j] = max(v[j] + M(p[j]), M(j-1))
return M[n]
```

i	M[i]
0	
1	
2	
3	
4	
5	
6	
7	
8	



$p(1) = 0$

$p(2) = 0$

$p(3) = 0$

$p(4) = 1$

$p(5) = 0$

$p(6) = 2$

$p(7) = 3$

$p(8) = 5$

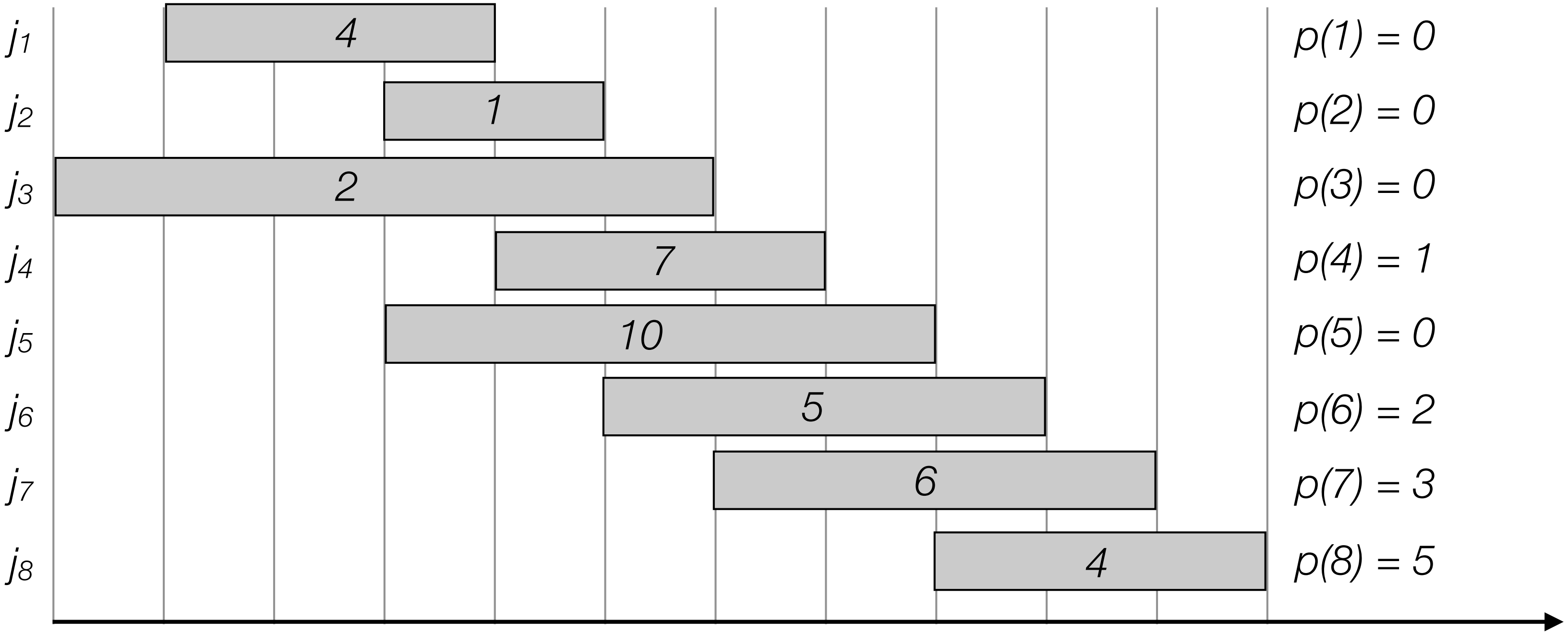
Weighted interval scheduling: bottom-up

```
Compute-Bottom-Up—Opt(n, s[1..n], f[1..n], v[1..n])

Sort jobs by finish time so that f[1] ≤ f[2]≤ … ≤ f[n]
Compute p[1], p[2], …, p[n]

M[0] = 0.
for j=1 to n
   M[j] = max(v[j] + M(p[j]), M(j-1))
return M[n]
```

i	M[i]
0	0
1	4
2	4
3	4
4	11
5	11
6	11
7	11
8	15



Weighted interval scheduling: finding solution

- DP algorithm returns value. How do we find the solution itself?
- Make a second pass:

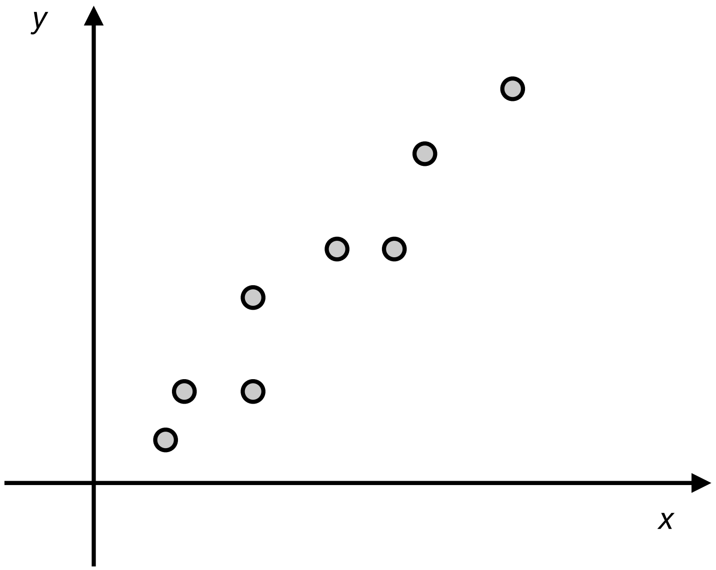
```
Find-Solution(j)
if j=0
  Return emptyset
else if v[j] + M[p[j]] > M[j-1]
  return {j} ∪ Find-Solution(p[j])
else
  return Find-Solution(j-1)
```

- Analysis: #recursive calls ≤ n => O(n) time.

Segmented Least Squares

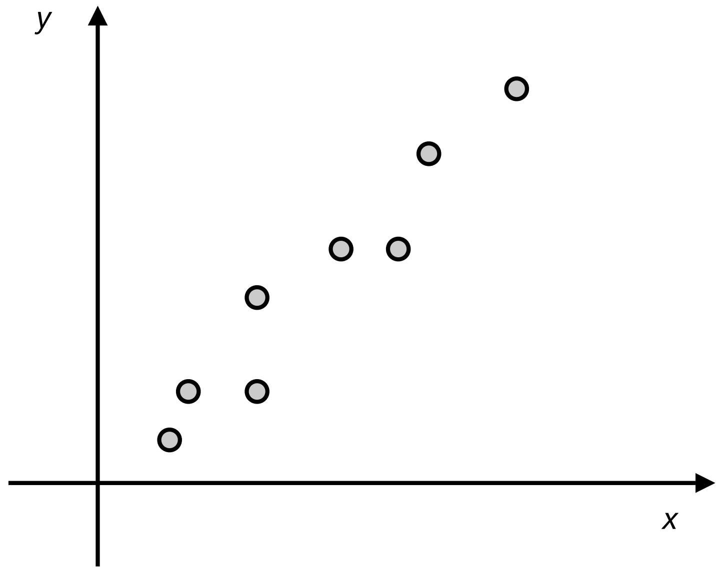
Least squares

- Least squares.



Least squares

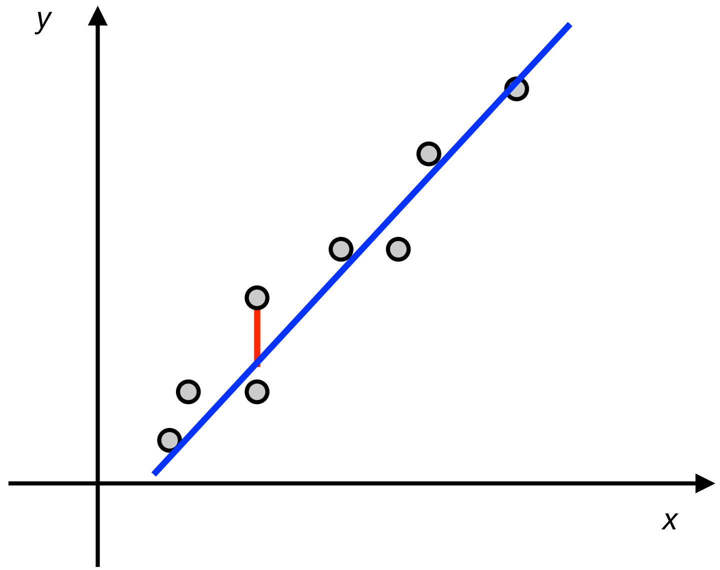
- Least squares.
  - Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.



Least squares

- Least squares.
  - Given n points in the plane: (x_1,y_1), (x_2,y_2), …, (x_n,y_n).
  - Find a line $y = ax + b$ that minimizes the sum of the squared error:

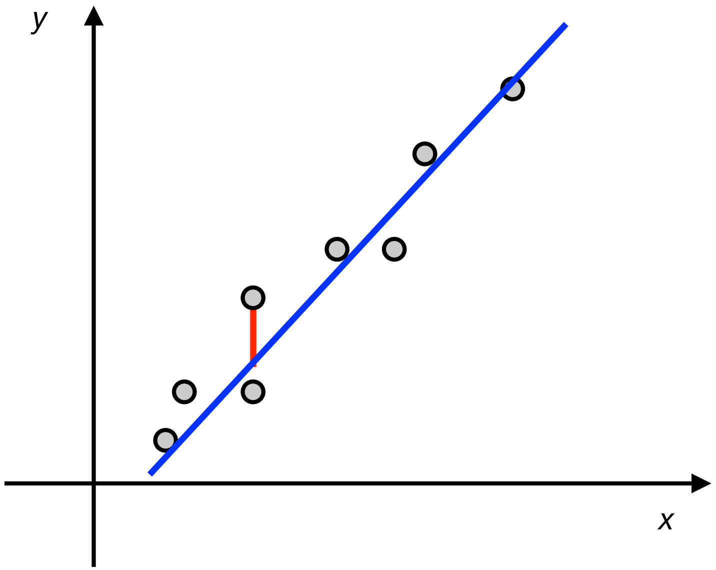
$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$



Least squares

- Least squares.
  - Given n points in the plane: (x_1,y_1), (x_2,y_2), …, (x_n,y_n).
  - Find a line $y = ax + b$ that minimizes the sum of the squared error:

$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

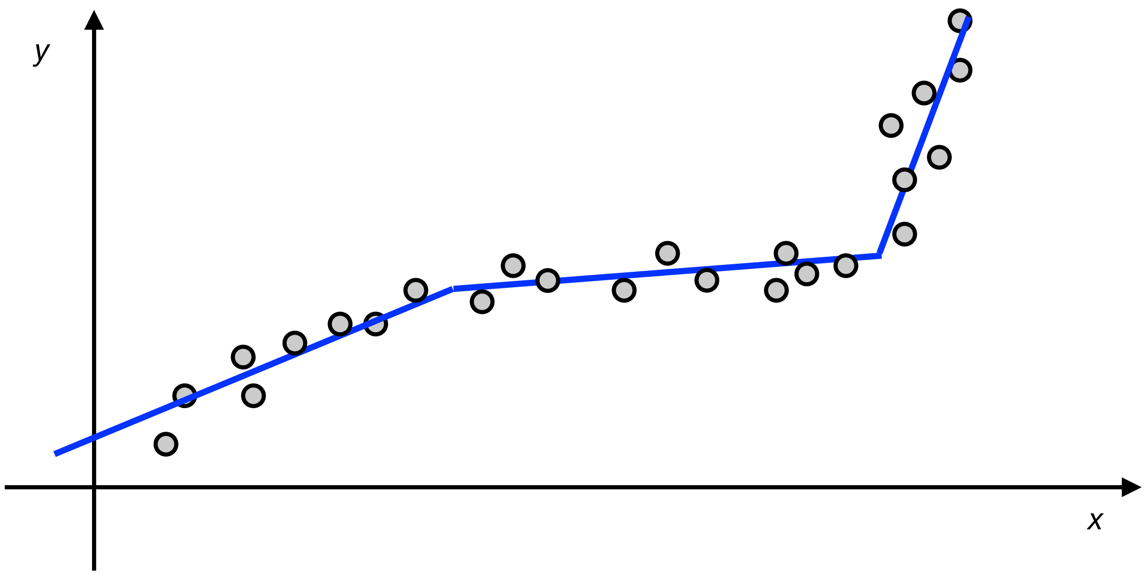


- Solution. Calculus => minimum error is achieved when

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \qquad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

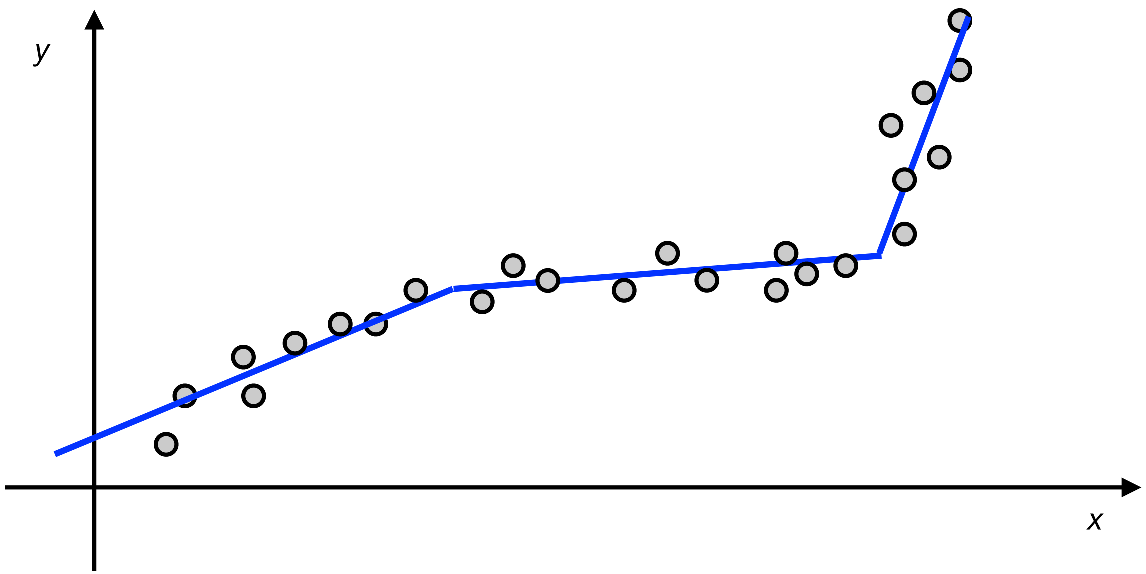
Segmented least squares

- Segmented least squares
  - Points lie roughly on a *sequence* of line segments.
  - Given n points in the plane (x_1, y_1), (x_2, y_2), …, (x_n, y_n).
  - Find a sequence of lines that minimizes $f(x)$.
- What is a good choice for $f(x)$ that balance accuracy and number of lines?

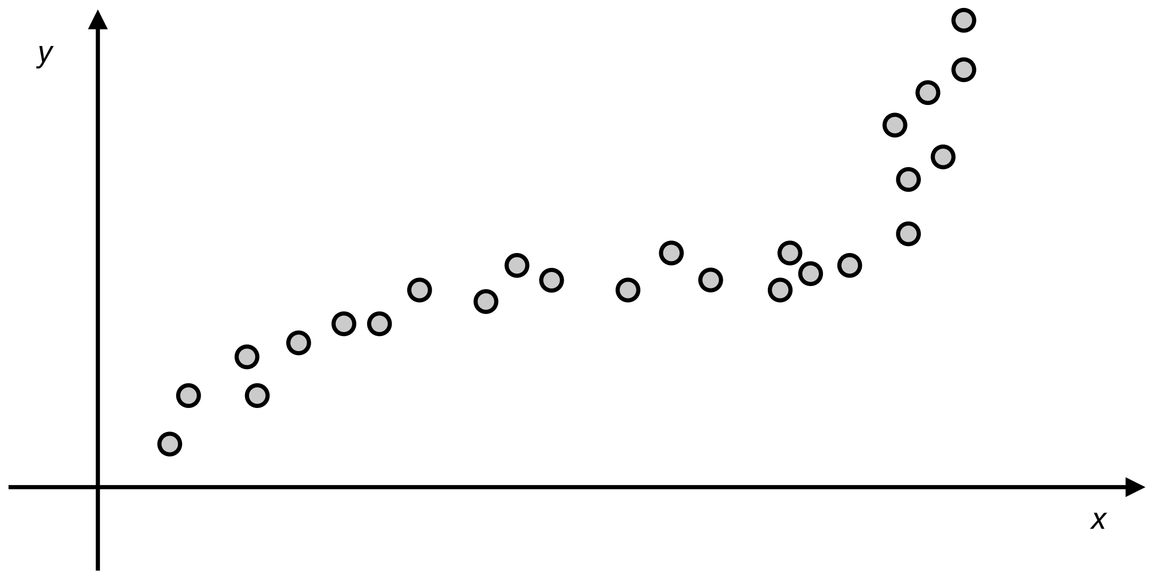


Segmented least squares

- Segmented least squares. Given n points in the plane (x_1, y_1), (x_2, y_2), …, (x_n, y_n) and a constant $c > 0$ find a sequence of lines that minimizes $f(x) = E + cL$:
    - E = sum of sums of the squared errors in each segment.
    - L = number of lines

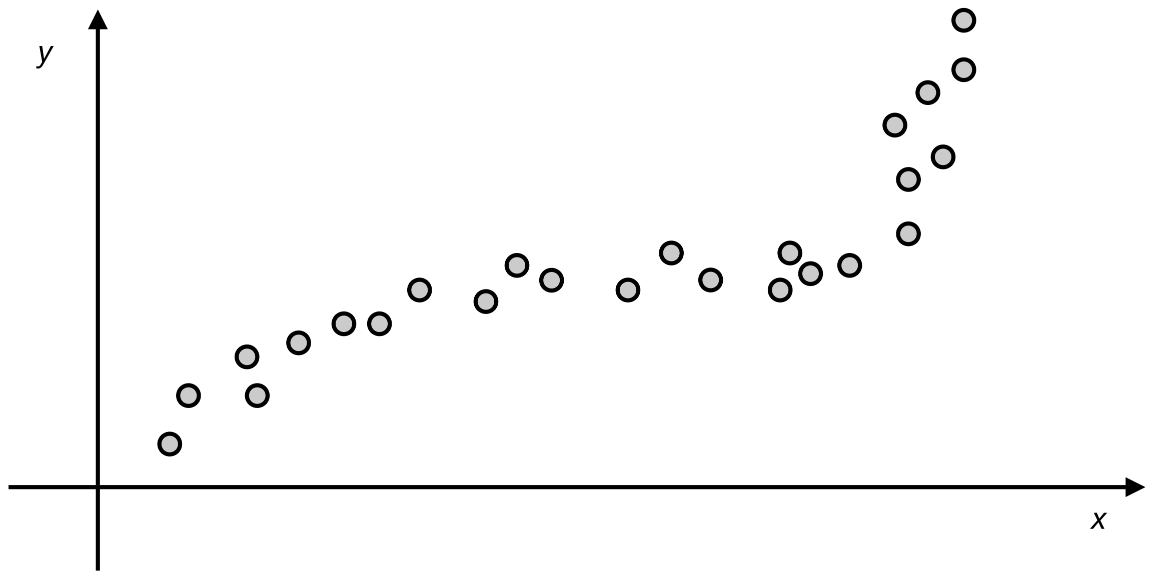


Dynamic programming: multiway choice



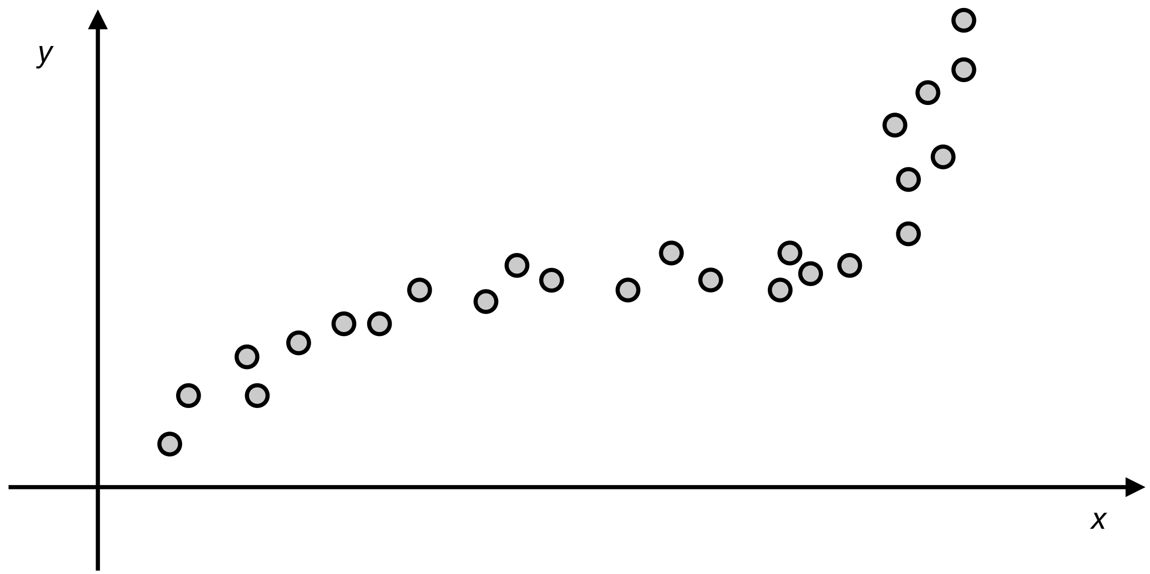
Dynamic programming: multiway choice

- OPT(j) = minimum cost for points p_1, p_2,…, p_j.



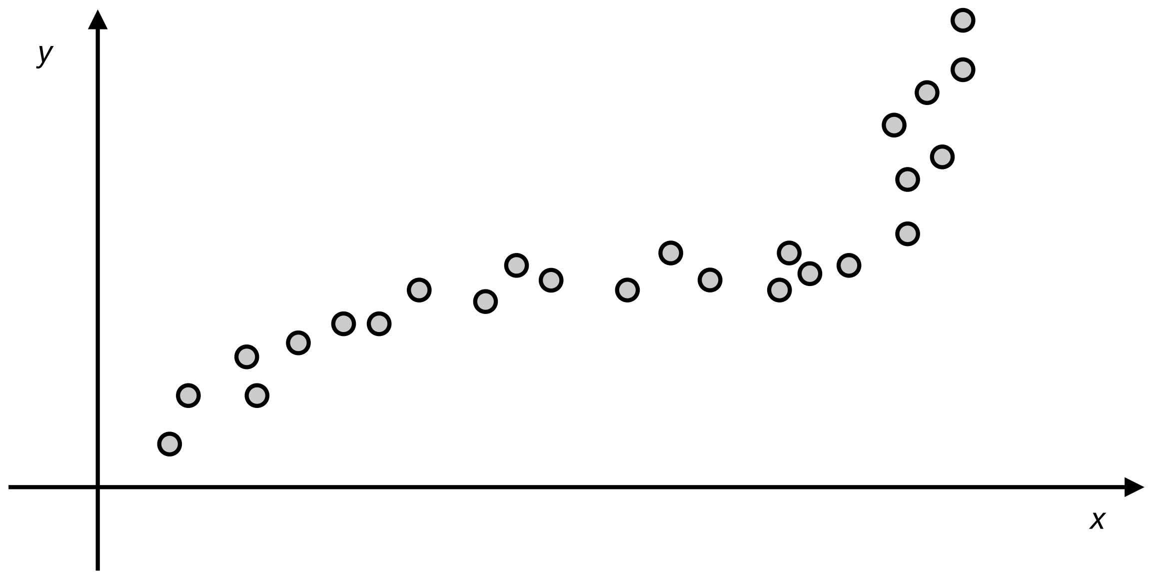
Dynamic programming: multiway choice

- OPT(j) = minimum cost for points p_1, p_2,…, p_j.
- $e(i,j)$ = minimum sum of squares for points p_i, p_{i+1},…, p_j.



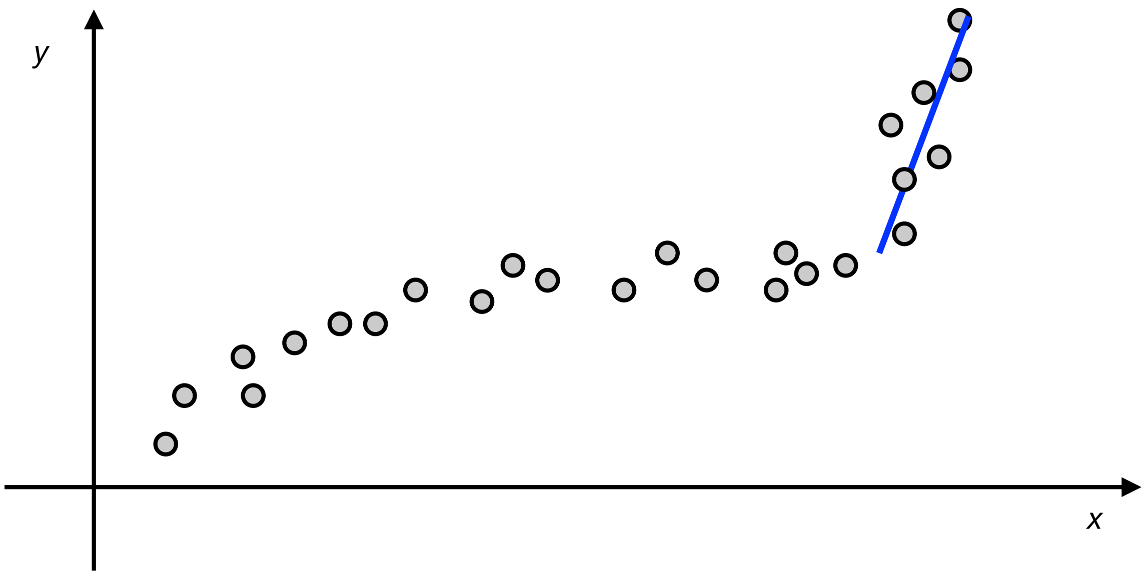
Dynamic programming: multiway choice

- OPT(j) = minimum cost for points $p_1, p_2, \ldots, p_j$.
- $e(i,j)$ = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.
- To compute OPT(j):



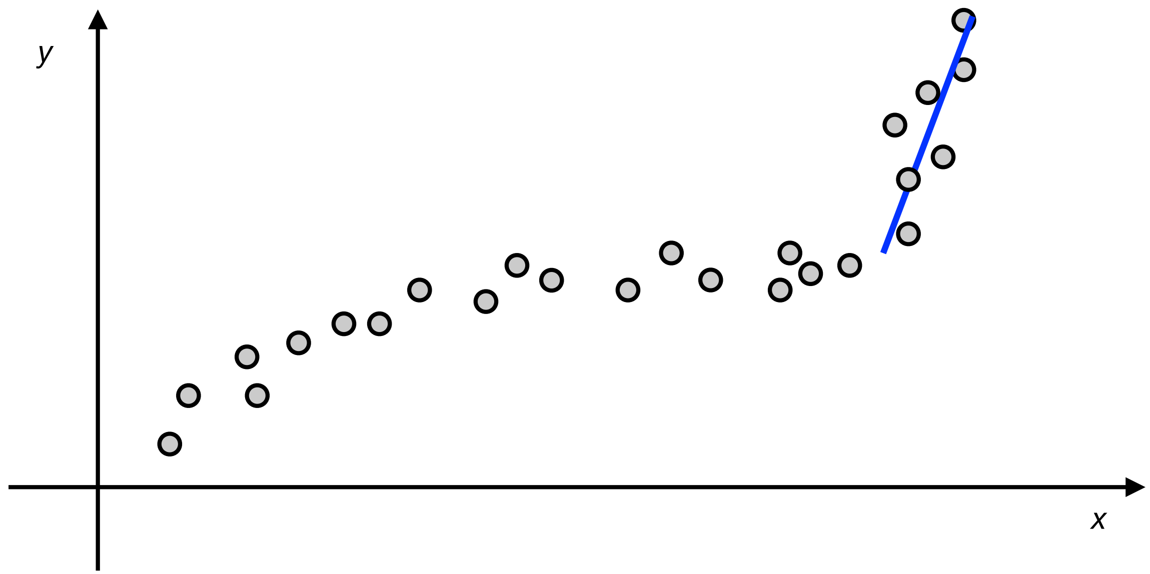
Dynamic programming: multiway choice

- OPT(j) = minimum cost for points $p_1, p_2, \ldots, p_j$.
- $e(i,j)$ = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.
- To compute OPT(j):



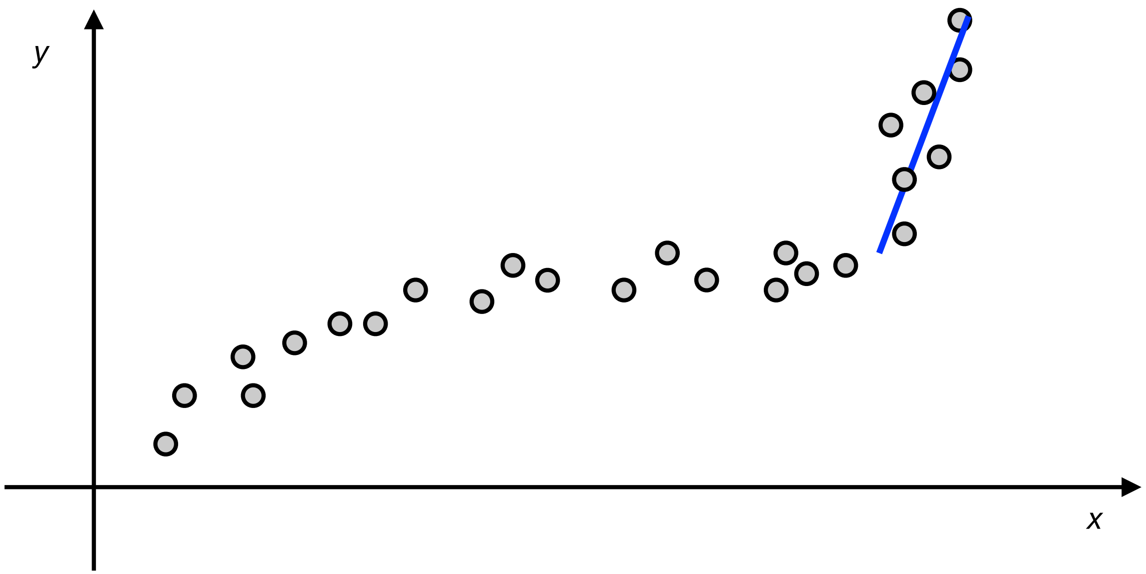
Dynamic programming: multiway choice

- OPT(j) = minimum cost for points p_1, p_2,…, p_j.
- $e(i,j)$ = minimum sum of squares for points p_i, p_{i+1},…, p_j.
- To compute OPT(j):
  - Last segment uses points p_i, p_{i+1},…, p_j for some i.



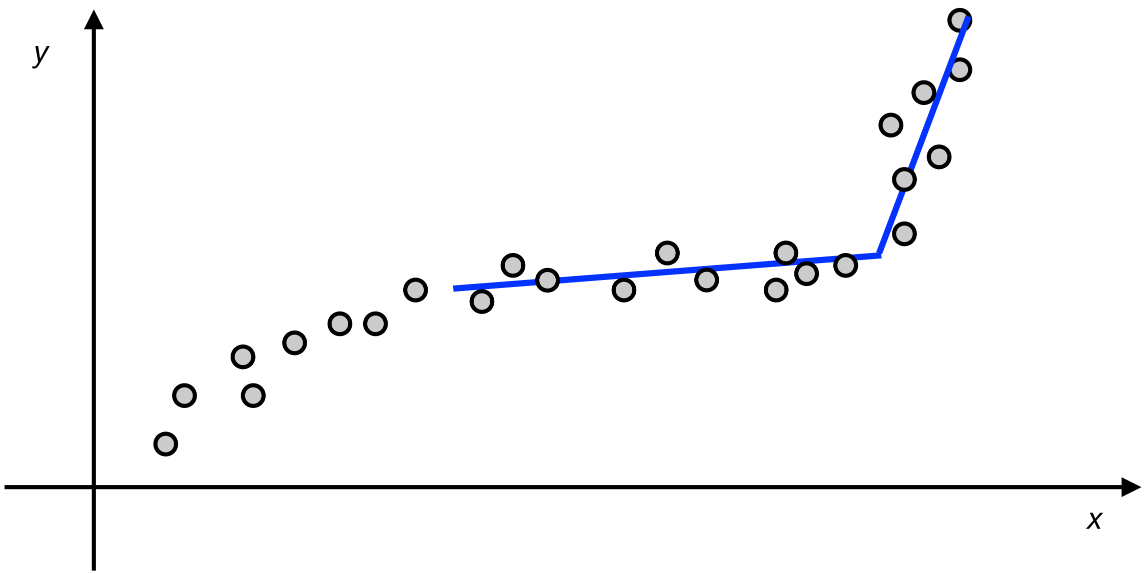
Dynamic programming: multiway choice

- OPT(j) = minimum cost for points $p_1, p_2, \ldots, p_j$.
- $e(i,j)$ = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.
- To compute OPT(j):
  - Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some i.
  - Cost = $e(i,j) + c + OPT(i-1)$.



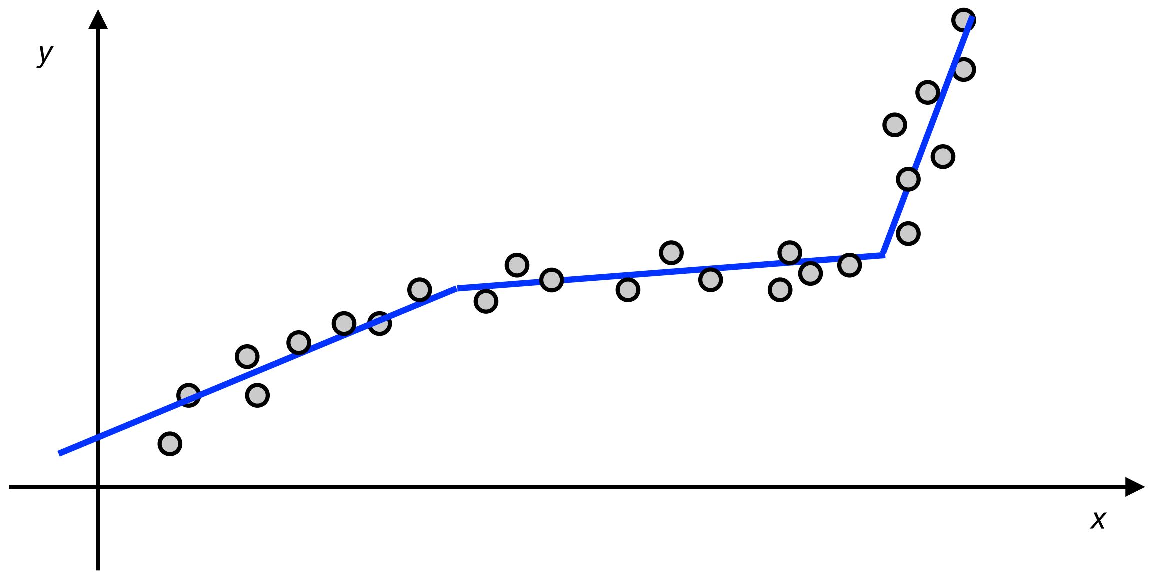
Dynamic programming: multiway choice

- OPT(j) = minimum cost for points $p_1, p_2, \ldots, p_j$.
- $e(i,j)$ = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.
- To compute OPT(j):
  - Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some i.
  - Cost = $e(i,j) + c + OPT(i-1)$.



Dynamic programming: multiway choice

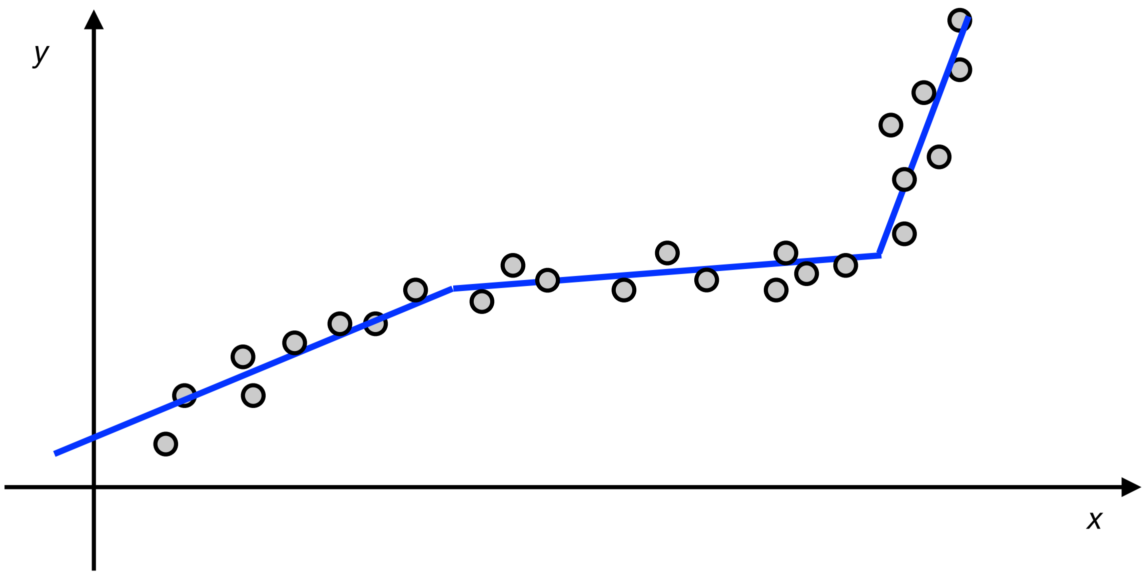
- OPT(j) = minimum cost for points $p_1, p_2, \ldots, p_j$.
- $e(i,j)$ = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.
- To compute OPT(j):
  - Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some i.
  - Cost = $e(i,j) + c + OPT(i-1)$.



Dynamic programming: multiway choice

- OPT(j) = minimum cost for points $p_1, p_2, \ldots, p_j$.
- $e(i,j)$ = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.
- To compute OPT(j):
  - Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some i.
  - Cost = $e(i,j) + c + OPT(i-1)$.

$$OPT(j) = \begin{cases} 0 & \texttt{if } j = 0 \\ \min_{1 \le i \le j}\{e(i,j) + c + OPT(i-1)\} & \texttt{otherwise} \end{cases}$$



Segmented least squares algorithm

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \le i \le j}\{e(i,j) + c + OPT(i-1)\} & \text{otherwise} \end{cases}$$

```
Segmented-least-squares(n, p1, p2, …,pn,c)

for j=1 to n
  for i=1 to j
    Compute the least squares e(i,j) for the segment
    pi, pi+1, …,pj.

M[0] = 0.
for j=1 to n
  M[j] = ∞
  for i=1 to j
    M[j] = min(M[j],e(i,j) + c + M[i-1])

Return M[n]
```

Segmented least squares algorithm

- Time.
  - $O(n^3)$ for computing $e(i,j)$ for $O(n^2)$ pairs ($O(n)$ per pair).
  - $O(n^2)$ for computing M.
  - Total $O(n^3)$
- Space
  - $O(n^2)$.

```
Segmented-least-squares(n, p1, p2, …,pn,c)

for j=1 to n
   for i=1 to j
      Compute the least squares e(i,j) for the segment
      pi, pi+1, …,pj.

M[0] = 0.
for j=1 to n
   M[j] = ∞
   for i=1 to j
      M[j] = min(M[j],e(i,j) + c + M[i-1])

Return M[n]
```