CS 401

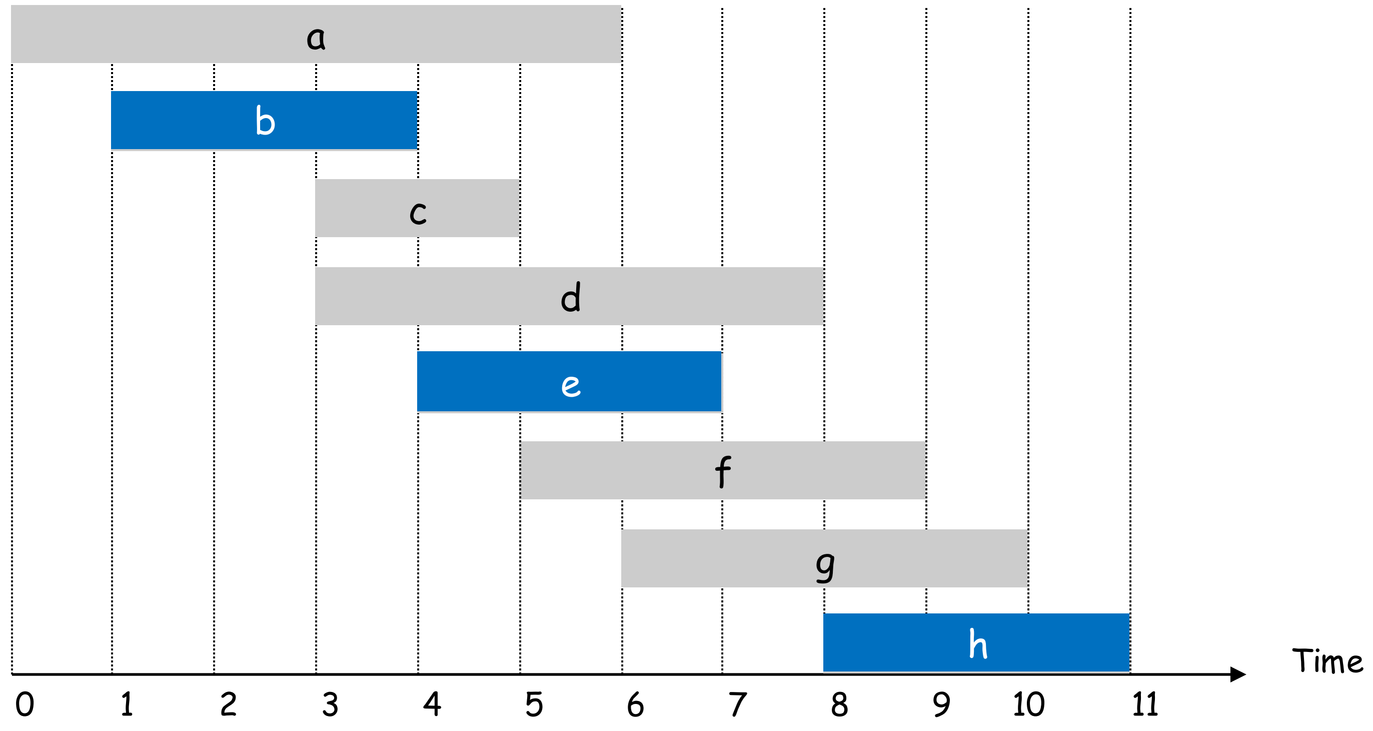
Dynamic Programming

Xiaorui Sun

Weighted Interval Scheduling

Weighted Interval Scheduling

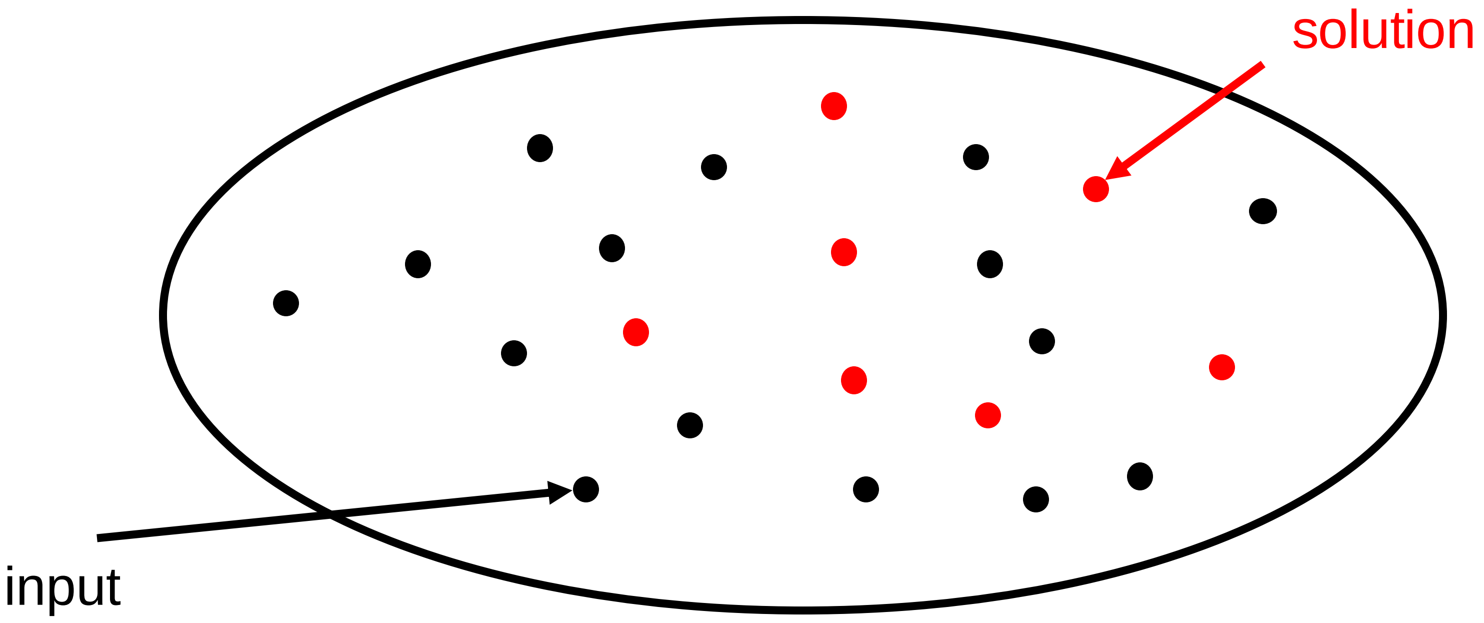
- Job j starts at $s(j)$ and finishes at $f(j)$ and has weight w_j
  - Two jobs compatible if they don't overlap.
  - Goal: find maximum weight subset of mutually compatible jobs.



Dynamic Programming

Principle:

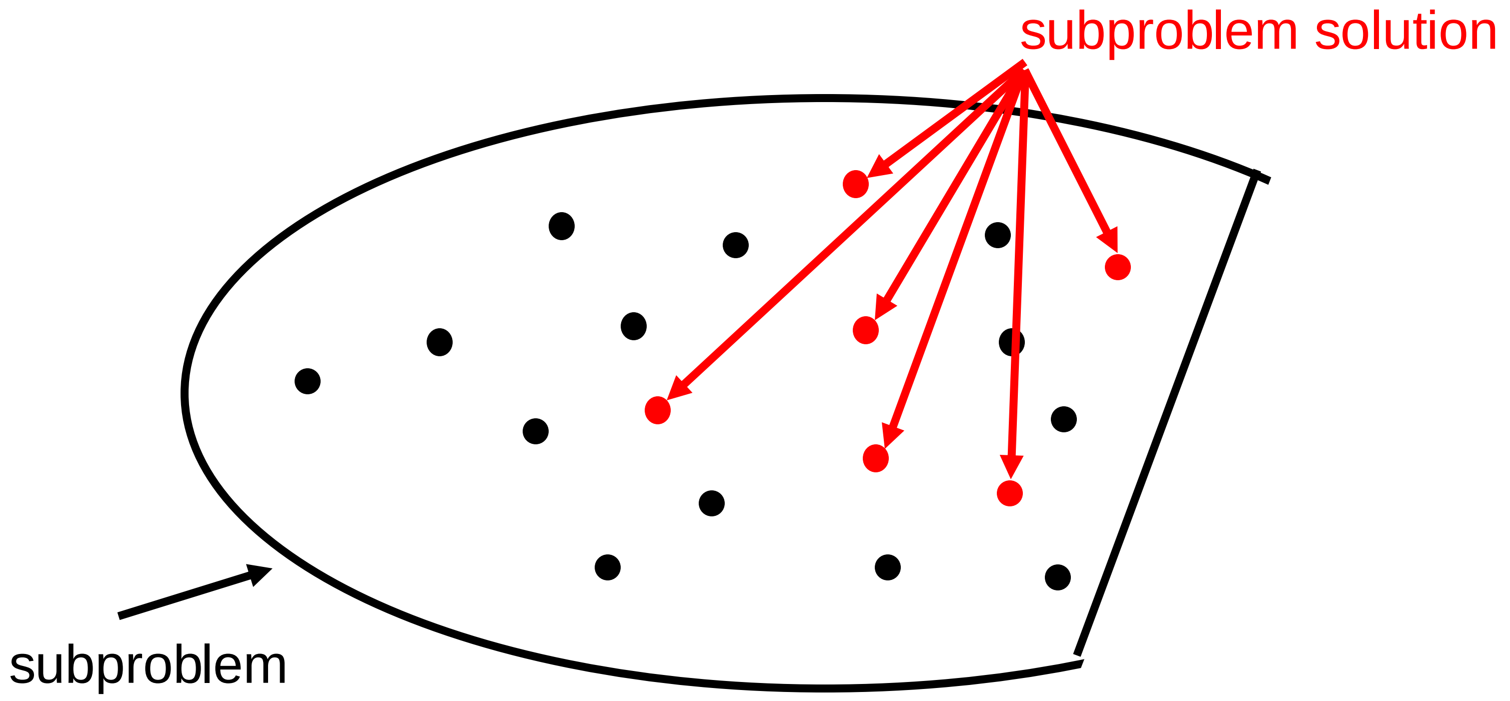
- Optimal substructure: Remove certain part of the optimal solution (for the entire problem) is an optimal solution of a subproblem



Dynamic Programming

Principle:

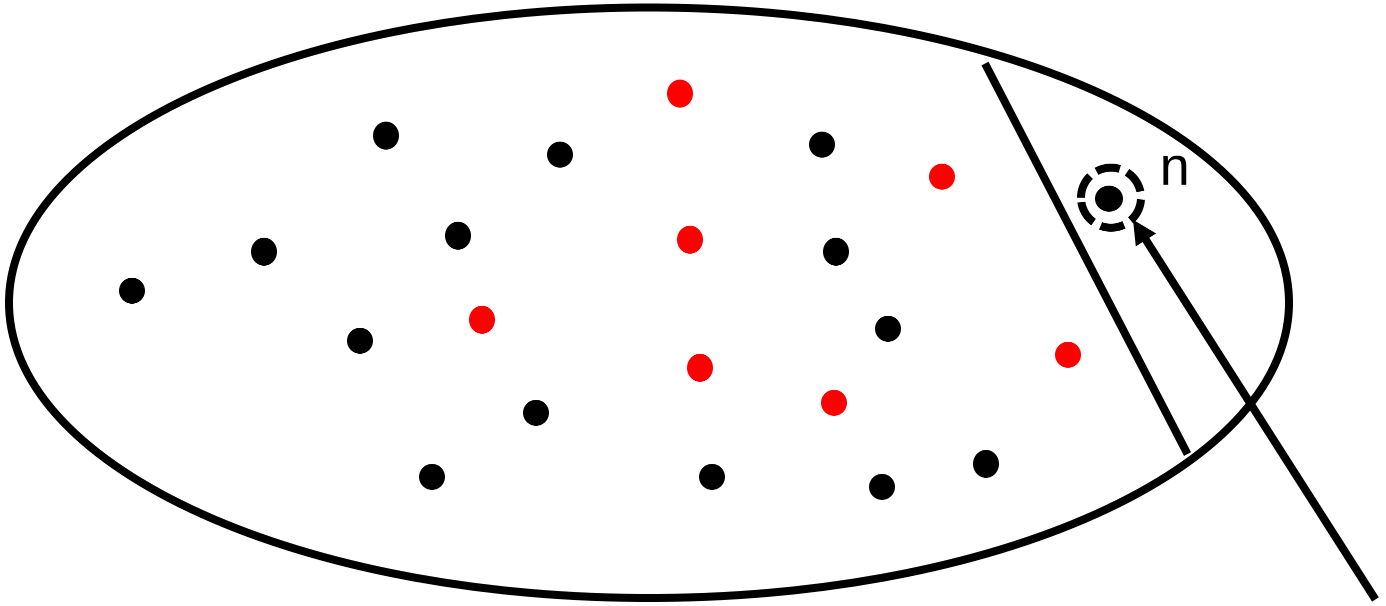
- Optimal substructure: Remove certain part of the optimal solution (for the entire problem) is an optimal solution of a subproblem



Dynamic Programming

Principle:

- Optimal substructure: Remove certain part of the optimal solution (for the entire problem) is an optimal solution of a subproblem
- Case analysis for optimal solution (e.g. weighted interval scheduling)

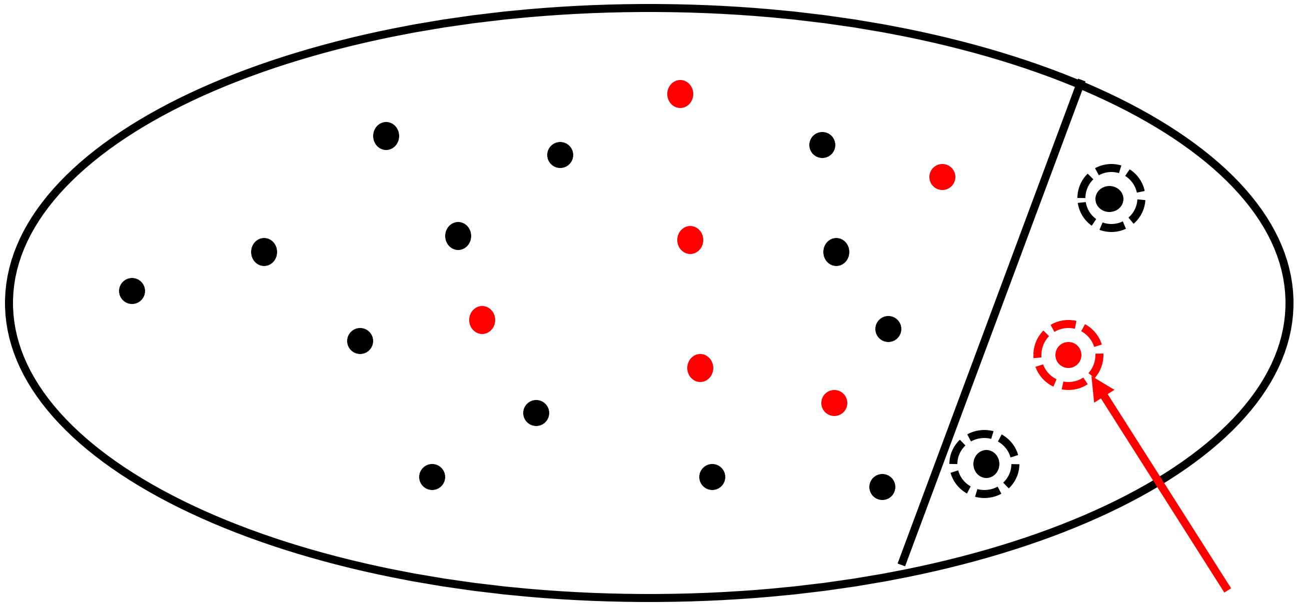


case 1: if n is not in the optimal solution

Dynamic Programming

Principle:

- Optimal substructure: Remove certain part of the optimal solution (for the entire problem) is an optimal solution of a subproblem
- Case analysis for optimal solution (e.g. weighted interval scheduling)



case 2: if n is in the optimal solution

Weighted Job Scheduling by Induction

Suppose $1, \dots, n$

Optimal Substructure: Optimal solution of a problem can be obtained from optimal solutions of smaller sub-problems

IH: Suppose we
jobs of size $< n$.

IS: Goal: For any n jobs we can compute OPT.

Case 1: Job n is not in OPT.

-- Then, just return OPT of $1, \dots, n-1$.

Take best of the two

Case 2: Job n is in OPT.

-- Then, delete all jobs not compatible with n and recurse.

Key question: Too many subproblems need to compute.

Sorting to reduce Subproblems

Sorting Idea: Label jobs by finishing time $f(1) \leq \cdots \leq f(n)$
IS: For jobs $1, \ldots, n$ we want to compute OPT

Case 1: Suppose OPT has job n.
- So, all jobs i that are not compatible with n are not OPT
- Let $p(n)$ = largest index $i < n$ such that job i is compatible with n.
- Then, we just need to find OPT of $1, \ldots, p(n)$

Take best of the two

Case 2: OPT does not select job n.
- Then, OPT is just the OPT of $1, \ldots, n-1$

Weighted Job Scheduling by Induction

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

Def $OPT(j)$ denote the weight of OPT solution of $1, \dots, j$

The most important part of a correct DP; It fixes IH

To solve $OPT(j)$:

Case 1: $OPT(j)$ has job j.

- So, all jobs i that are not compatible with j are not $OPT(j)$.
- Let $p(j) =$ largest index $i < j$ such that job i is compatible with j.
- So $OPT(j) = OPT\big(p(j)\big) + w_j$.

Case 2: $OPT(j)$ does not select job j.

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

Dynamic programming equation

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

Algorithm with Memoization

Memorization. Compute and Store the solution of each sub-problem in a cache the first time that you face it. lookup as needed.

```
Input: $n$, $s(1),\dots,s(n)$ and $f(1),\dots,f(n)$ and $w_1,\dots,w_n$.

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

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

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

$OPT(j)$ {
   if (M[j] is empty)
      M[j] = $max\,(w_j + OPT\big(p($
   return M[j]
}
```

Dynamic programming: break complex problem down into simpler sub-problems in a recursive manner (can be viewed as a generalization of divide and conquer)

In practice, you may get stack overflow if $n \gg 10^6$ (depends on the language).

Bottom up Dynamic Programming

You can also avoid recursion

- recursion may be easier conceptually when you use induction

```
Input: $n$, $s(1), \dots, s(n)$ and $f(1), \dots, f(n)$ and $w_1, \dots, w_n$.

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

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

M[0] = 0
for j = 1 to n
    M[j] = $max\,(w_j + M[p(j)], M[j -$


Output M[n]
```

$\leftarrow$ Binary search

O(n log n)

O(n log n)

Dynamic programming: break complex problem down into a sequence of decision steps over time (can be viewed as a generalization of greedy)

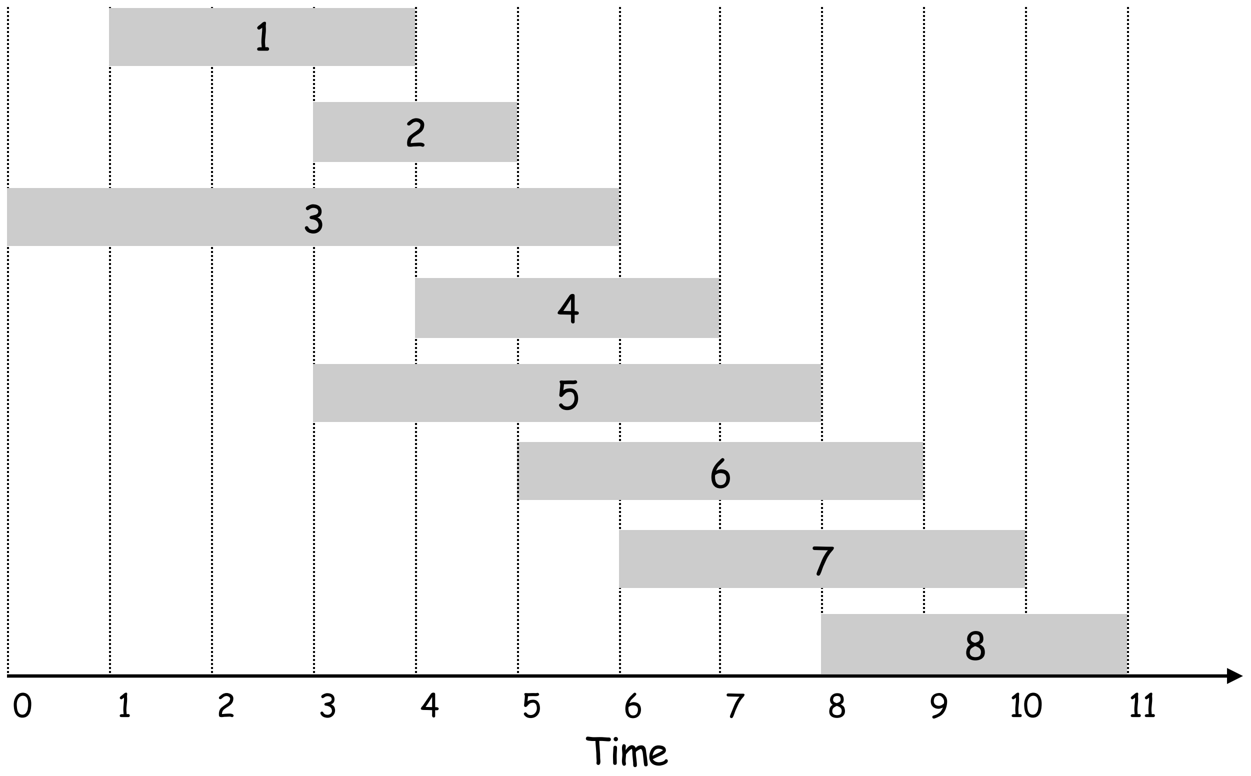
Claim: $M[j]$ is value of $OPT(j)$

Example

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

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

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



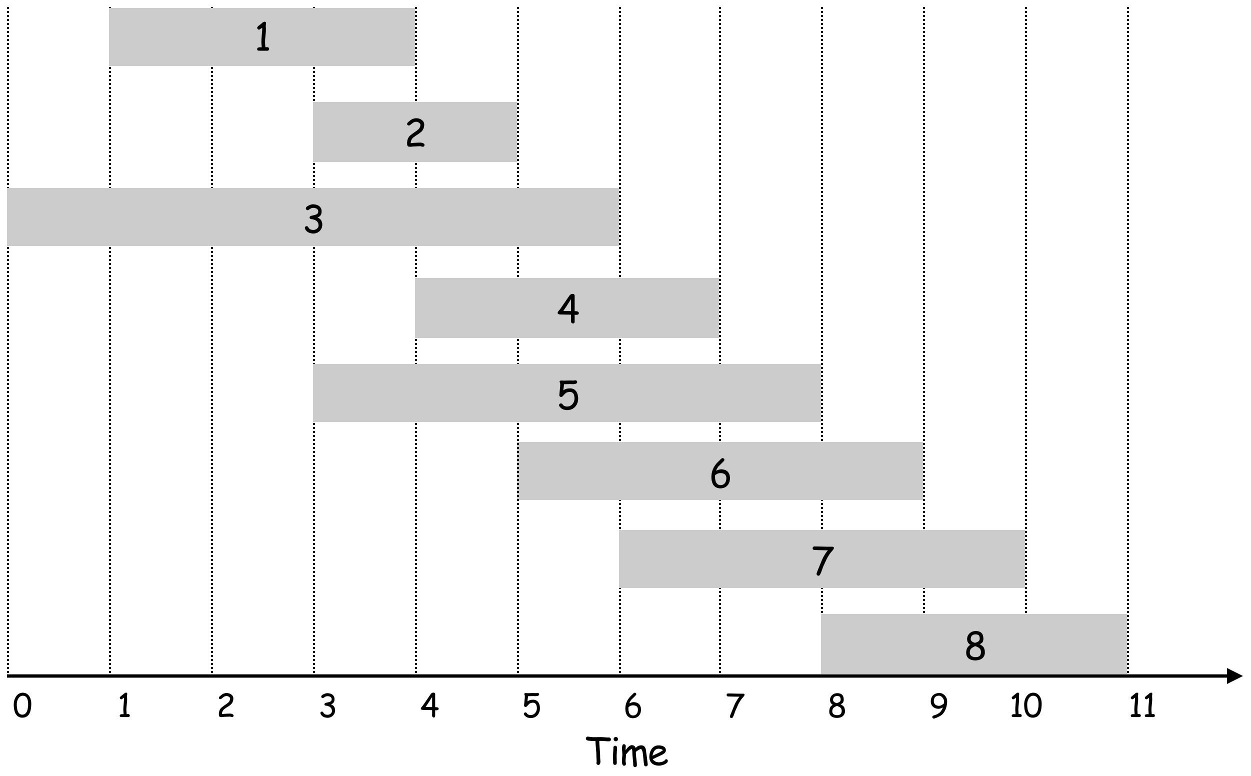
j	w_j	p(j)	OPT(j)
0			0
1	3	0	
2	4	0	
3	1	0	
4	3	1	
5	4	0	
6	3	2	
7	2	3	
8	4	5	

Example

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

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

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



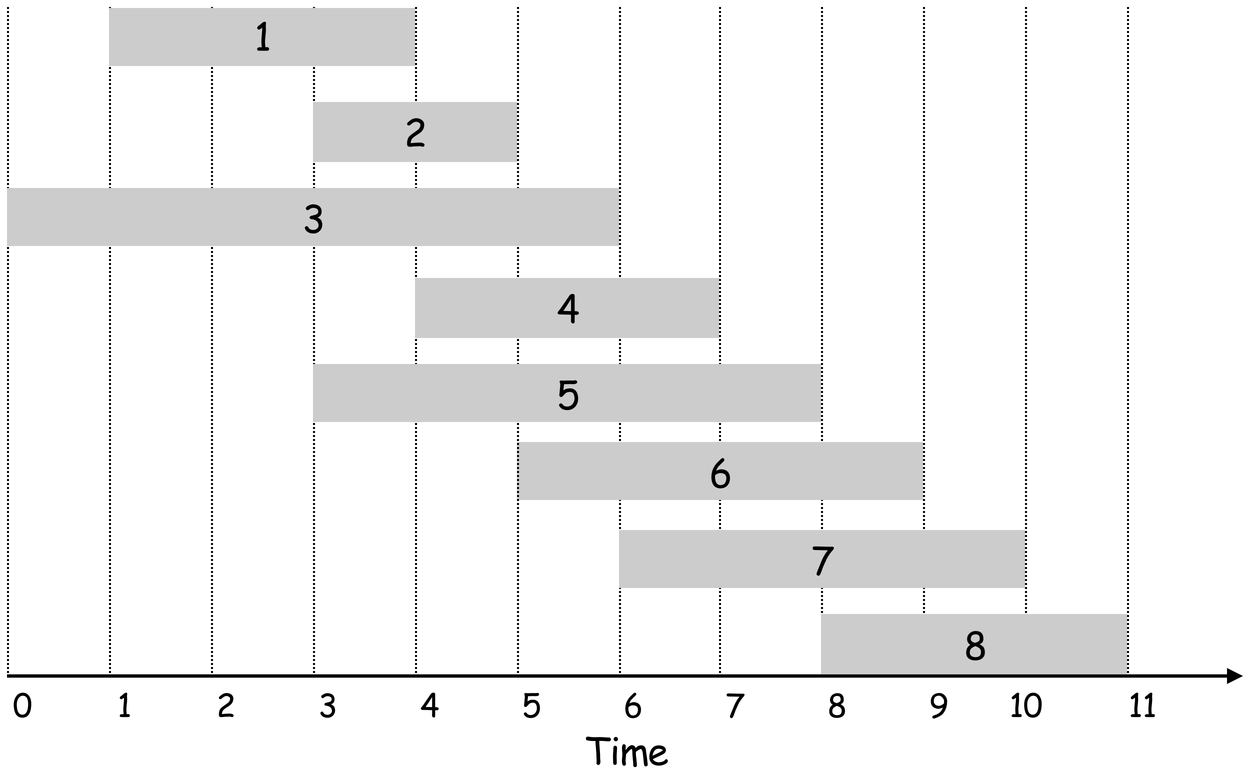
j	w_j	p(j)	OPT(j)
0			0
1	3	0	3
2	4	0	
3	1	0	
4	3	1	
5	4	0	
6	3	2	
7	2	3	
8	4	5	

Example

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

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

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



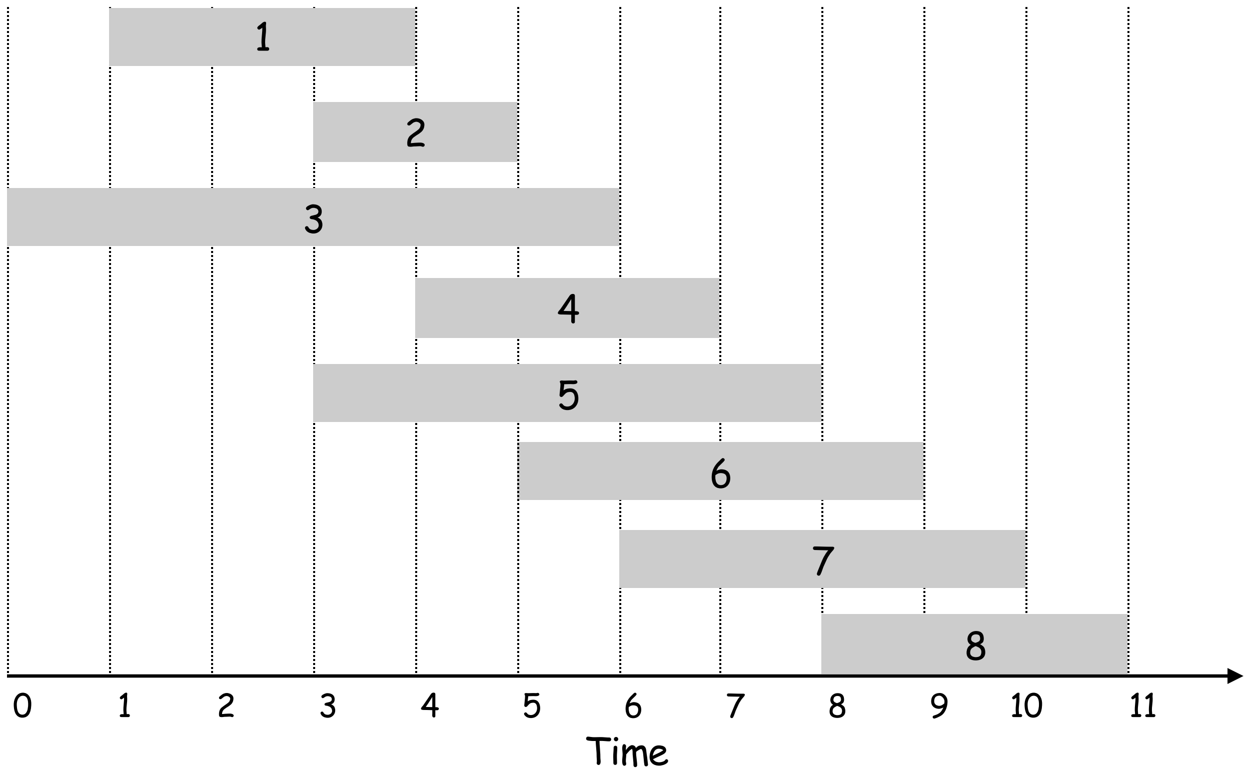
j	w_j	p(j)	OPT(j)
0			0
1	3	0	3
2	4	0	**4**
3	1	0	
4	3	1	
5	4	0	
6	3	2	
7	2	3	
8	4	5	

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

Example

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

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



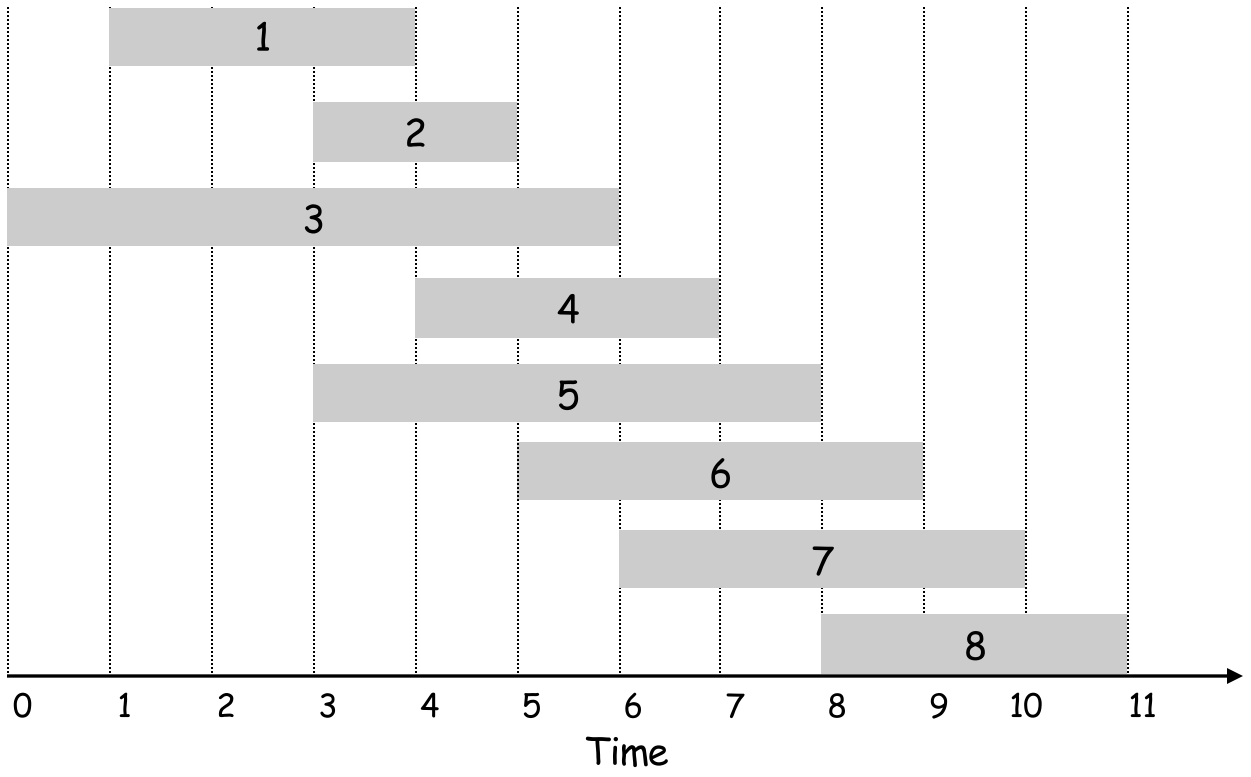
j	w_j	p(j)	OPT(j)
0			0
1	3	0	3
2	4	0	4
3	1	0	**4**
4	3	1	
5	4	0	
6	3	2	
7	2	3	
8	4	5	

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

Example

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

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



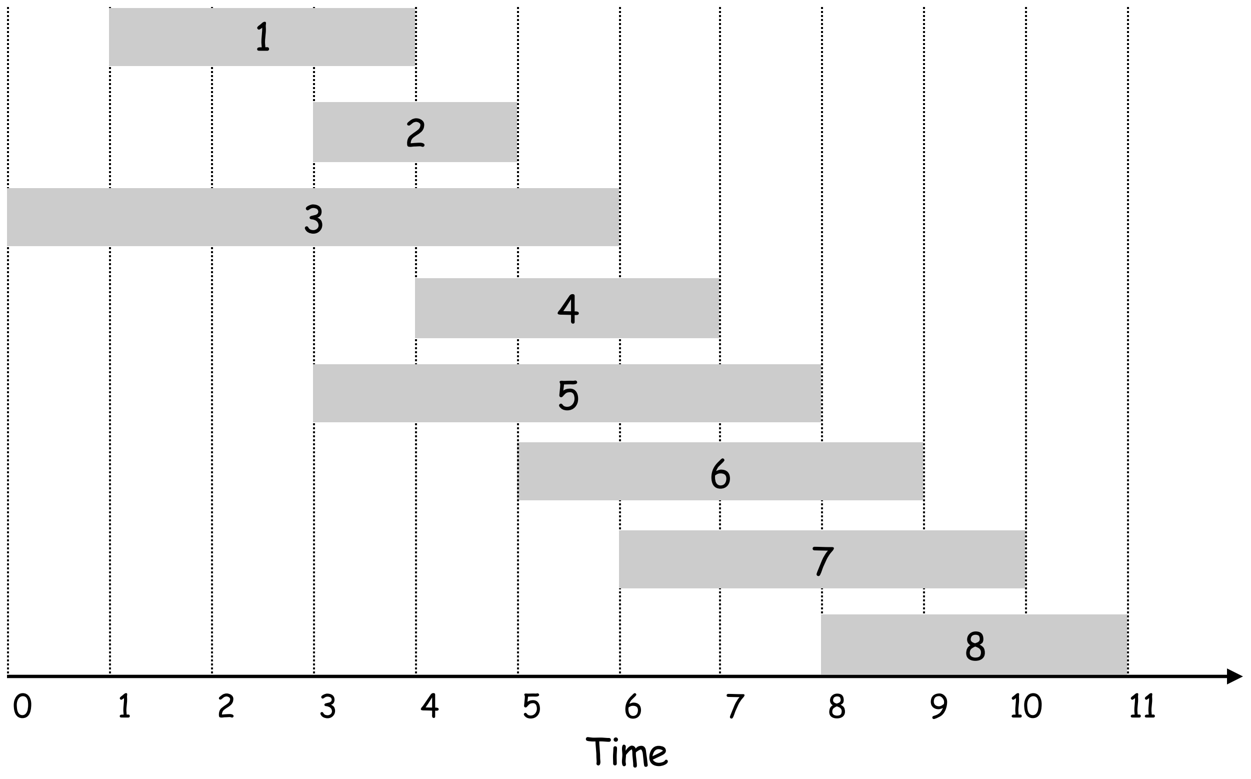
j	w_j	p(j)	OPT(j)
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	
6	3	2	
7	2	3	
8	4	5	

Example

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

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

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



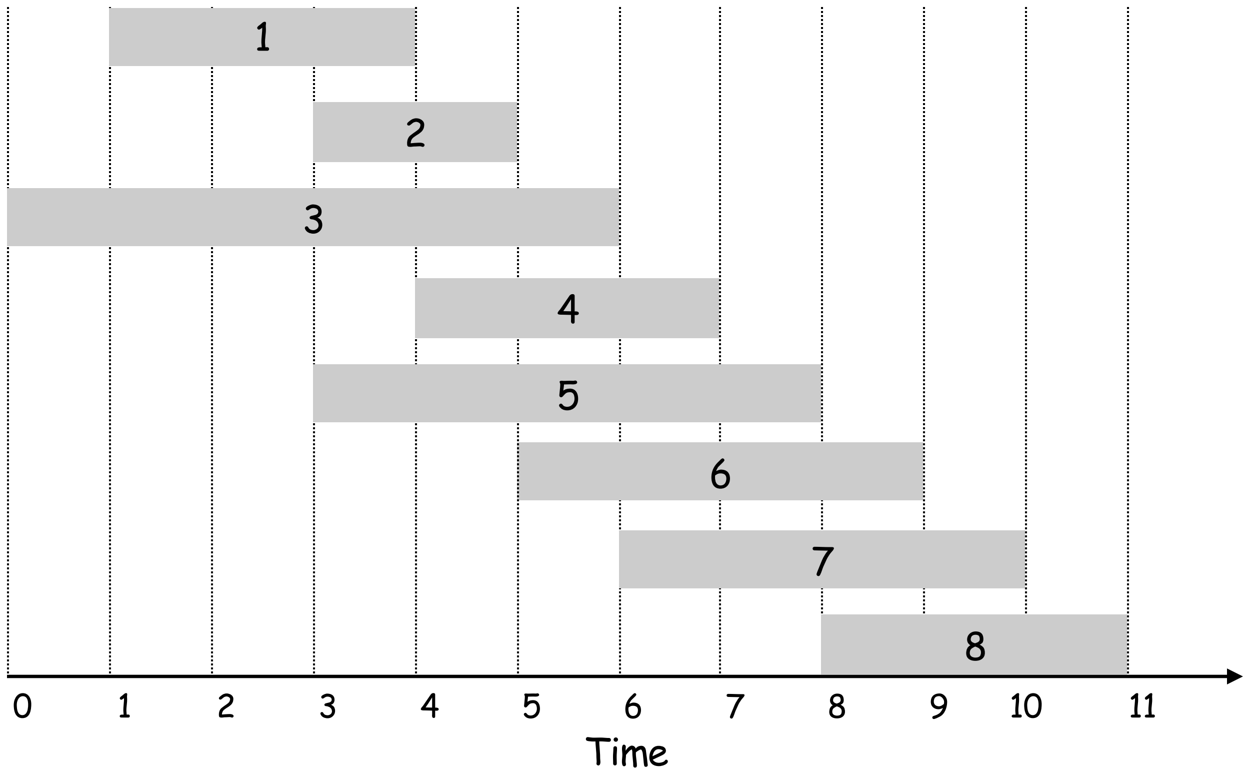
j	w_j	p(j)	OPT(j)
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	6
6	3	2	
7	2	3	
8	4	5	

Example

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

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

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



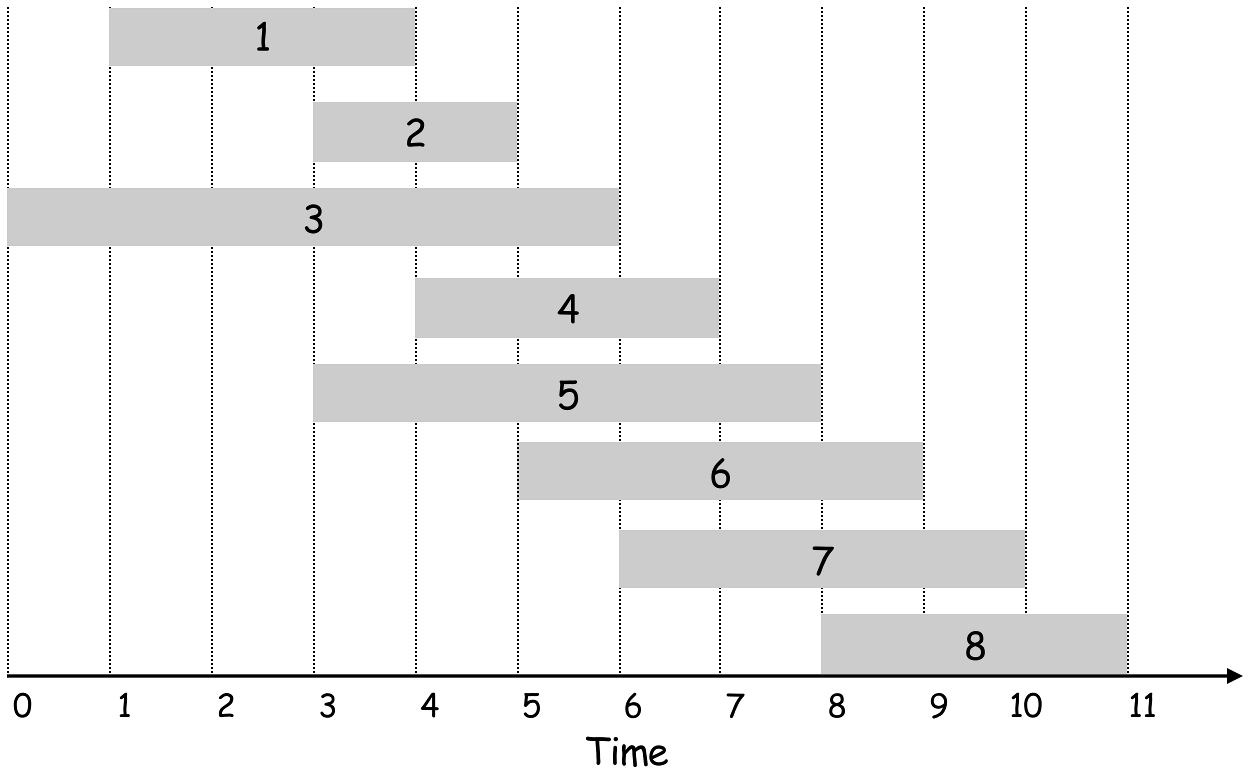
j	w_j	p(j)	OPT(j)
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	6
6	3	2	7
7	2	3	
8	4	5	

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

Example

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

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



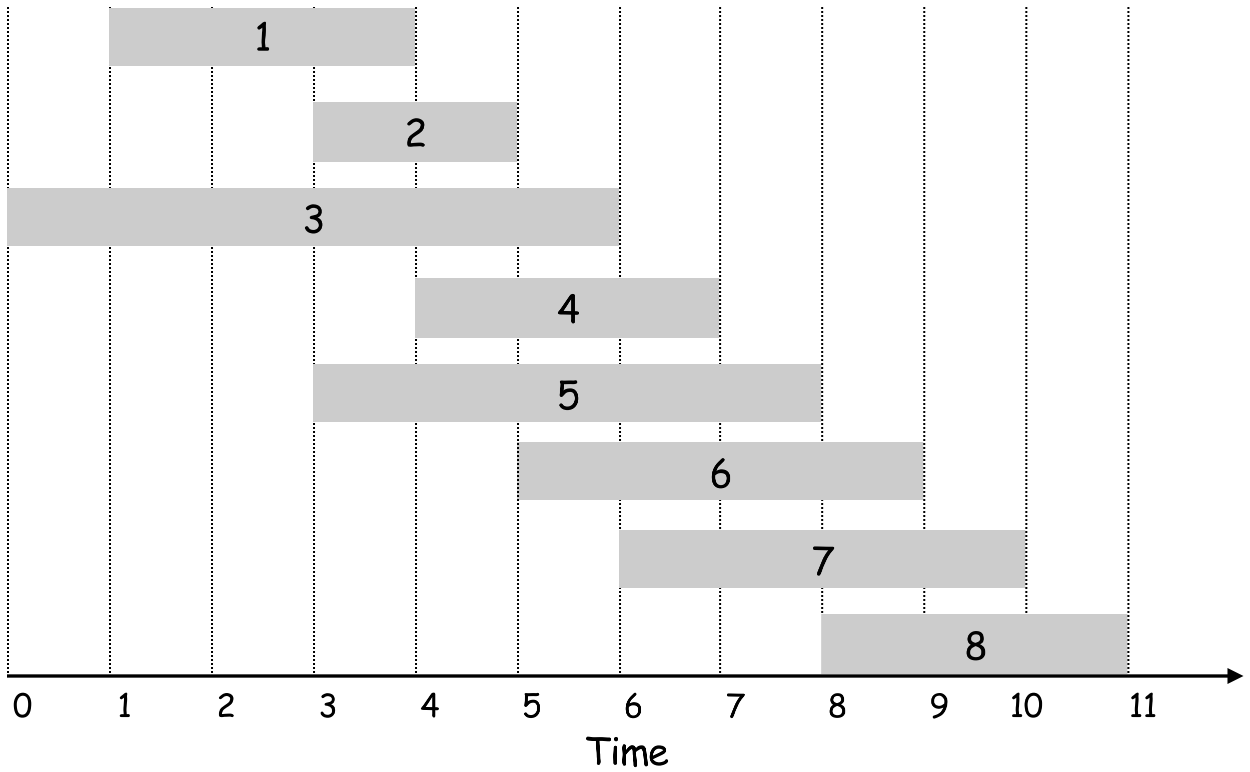
j	w_j	p(j)	OPT(j)
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	6
6	3	2	7
7	2	3	7
8	4	5	

Example

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

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

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



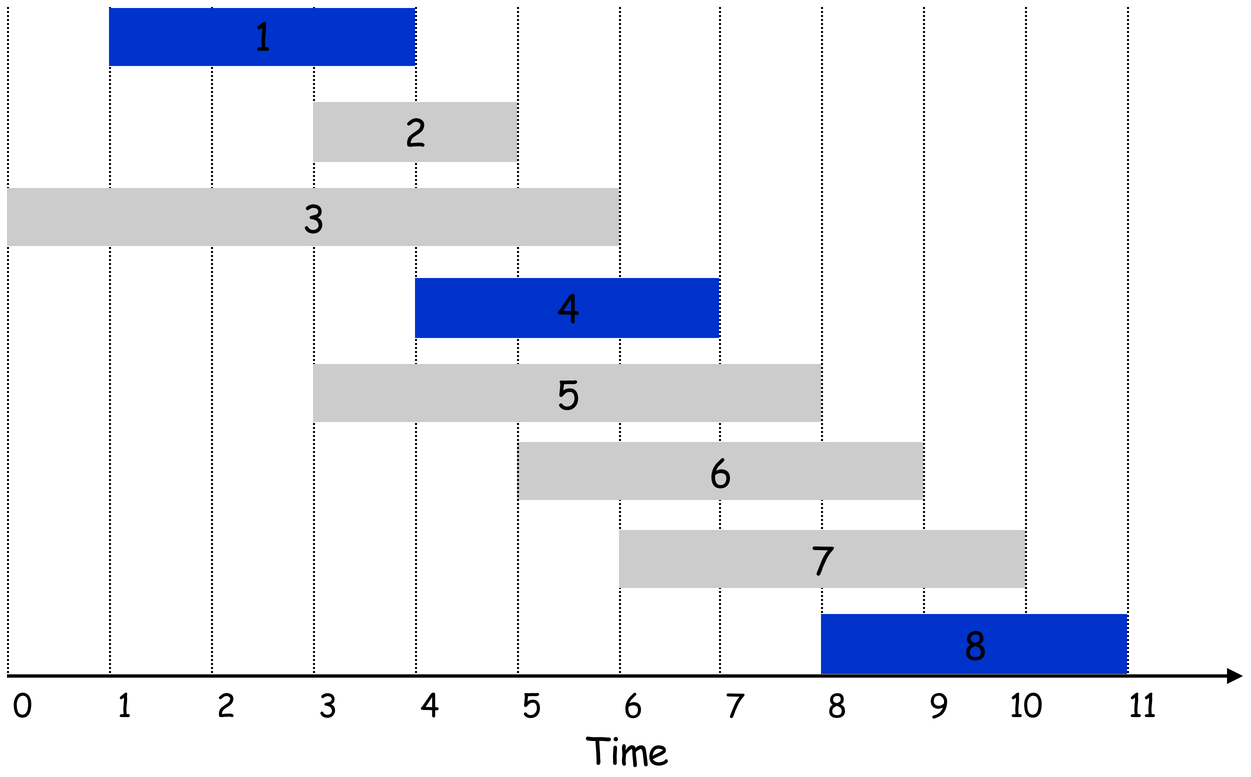
j	w_j	p(j)	OPT(j)
0			0
1	3	0	3
2	4	0	4
3	1	0	4
4	3	1	6
5	4	0	6
6	3	2	7
7	2	3	7
8	4	5	10

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

Example

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

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



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

Dynamic Programming

Principle:

- Optimal substructure: Remove certain part of the optimal solution (for the entire problem) is an optimal solution of a subproblem
- Typically, only a polynomial number of subproblems

Technique:

- Parameterization: Describe subproblems by parameters so that the optimal solution can be represented as a recurrence relation
- Memorization: Remember the solution of subproblems

Examples:

- Binary choice: weighted interval scheduling.
- Multiway choice: segmented least squares.
- Multidimensional dynamic programming: knapsack

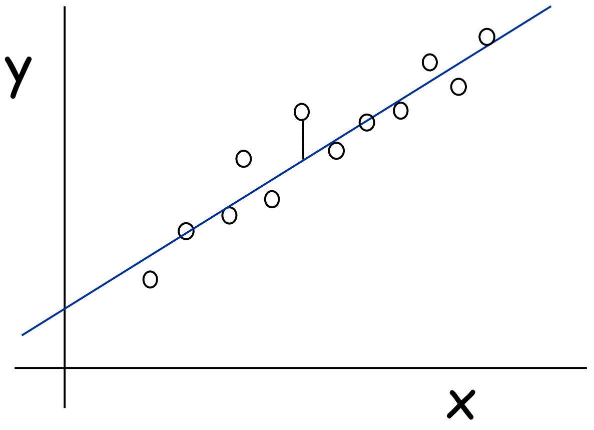
Segmented Least Squares

Segmented Least Squares

Least squares.

- Foundational problem in statistic and numerical analysis.
- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.
- Find a line $y = ax + b$ that minimizes the sum of the squared error:

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



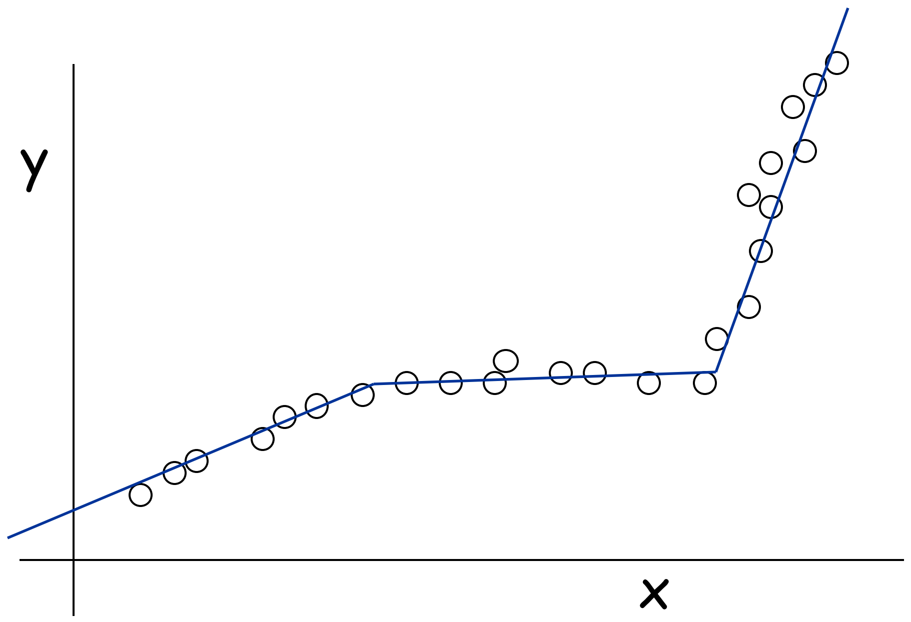
Solution. Calculus $\Rightarrow$ min error is achieved when

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

Segmented Least Squares

Segmented least squares.

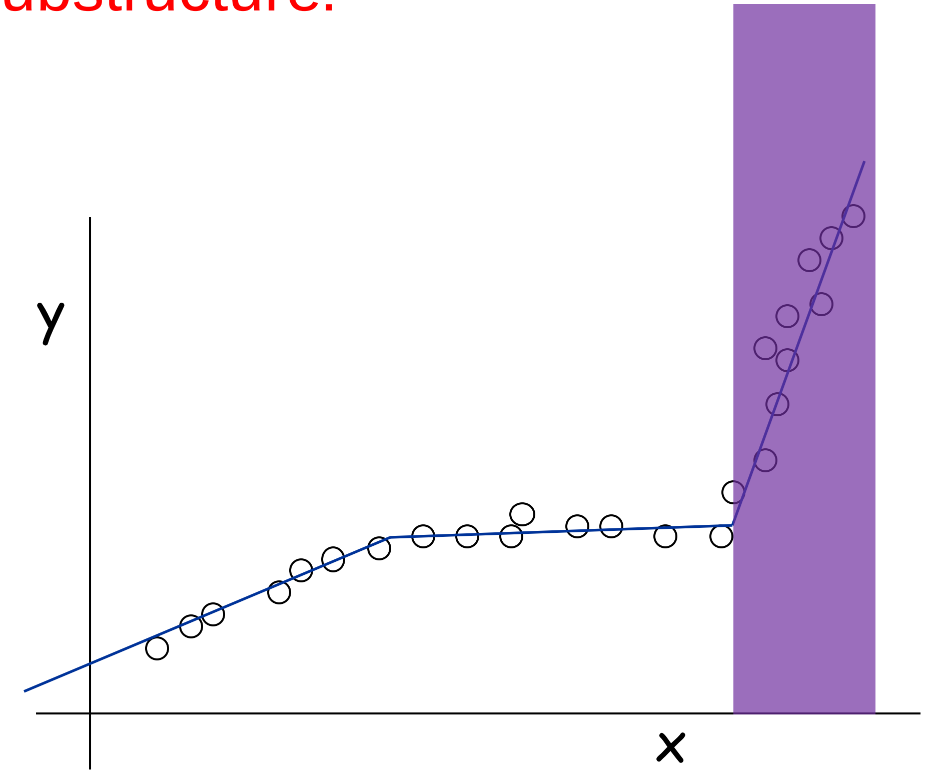
- Points lie roughly on a sequence of several line segments.
- Given n points in the plane (x_1, y_1), (x_2, y_2) , . . . , (x_n, y_n) with
- $x_1 < x_2 < ... < x_n$, find a sequence of lines that minimizes:
  - the sum of the sums of the squared errors E in each segment
  - the number of lines L
- Tradeoff function: E + c L, for some constant c > 0.



Dynamic programming

Suppose we know the last segment

- If all the points in last segment are removed, then the remaining segments must be the optimal solution for the the remaining points
- Optimal substructure!



Dynamic Programming: Multiway Choice

Notation.

- OPT(j) = minimum cost for points $p_1, \ldots, p_{i+1}, \ldots, p_j$.
- 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).

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

Segmented Least Squares: Algorithm

```
INPUT: n, $p_1,\dots,p_N$ , c

Segmented-Least-Squares() {
   M[0] = 0
   for j = 1 to n
      for i = 1 to j
         compute the least square error $e_{ij}$ for
         the segment $p_i,\dots, p_j$

   for j = 1 to n
      M[j] = $\min_{1 \le i \le j} (e_{ij} + c + M[i-1])$

   return M[n]
}
```

can be improved to $O(n^2)$ by pre-computing various statistics

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

Bottleneck = computing e(i, j) for $O(n^2)$ pairs, O(n) per pair using previous formula.