COMP251: Dynamic programming (1)

Jérôme Waldispühl

School of Computer Science

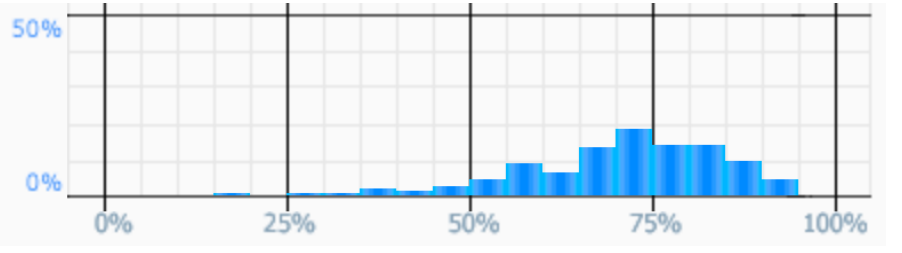
McGill University

Based on (Cormen *et al.*, 2002) & (Kleinberg & Tardos, 2005)

Announces

Midterm:

- Mean 70%, Median 72%, Best 94%.



- Available for review (with solution) during my office hours.

Office hours:

- Today: Moved to 3pm to 4pm.

Assignment 3:

- Deadline postponed to March 22.
- Assignment 4 will be released on Monday.

Algorithms paradigms

- **Greedy:**
  - Build up a solution incrementally.
  - Iteratively decompose and reduce the size of the problem.
  - Top-down approach.

- **Dynamic programming**:
  - Solve all possible sub-problems.
  - Assemble them to build up solutions to larger problems.
  - Bottom-up approach.

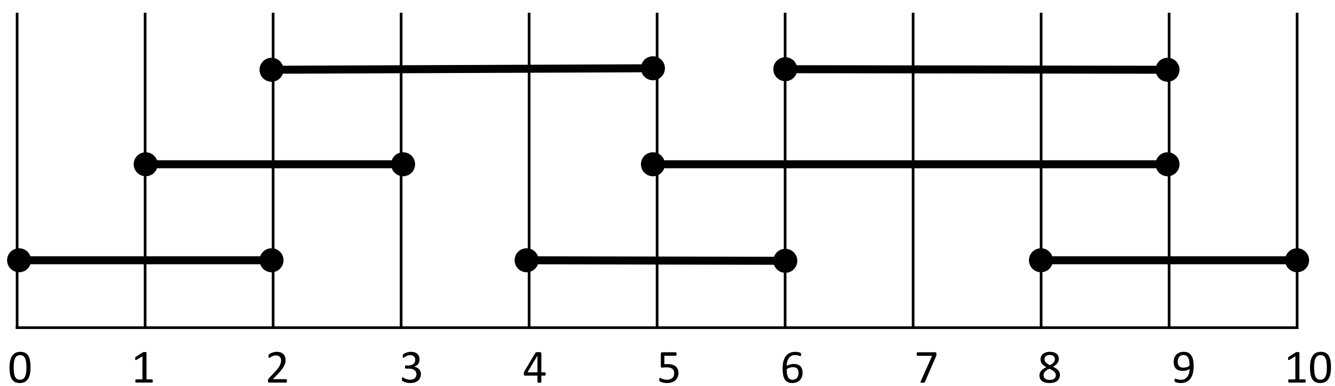
INTRODUCTION

Activity-selection Problem

- <u>Input:</u> Set S of n activities, a_1, a_2, …, a_n.
  - s_i = start time of activity i.
  - f_i = finish time of activity i.
- <u>Output:</u> Subset A of maximum number of compatible activities.
  - 2 activities are compatible, if their intervals do not overlap.

Example:

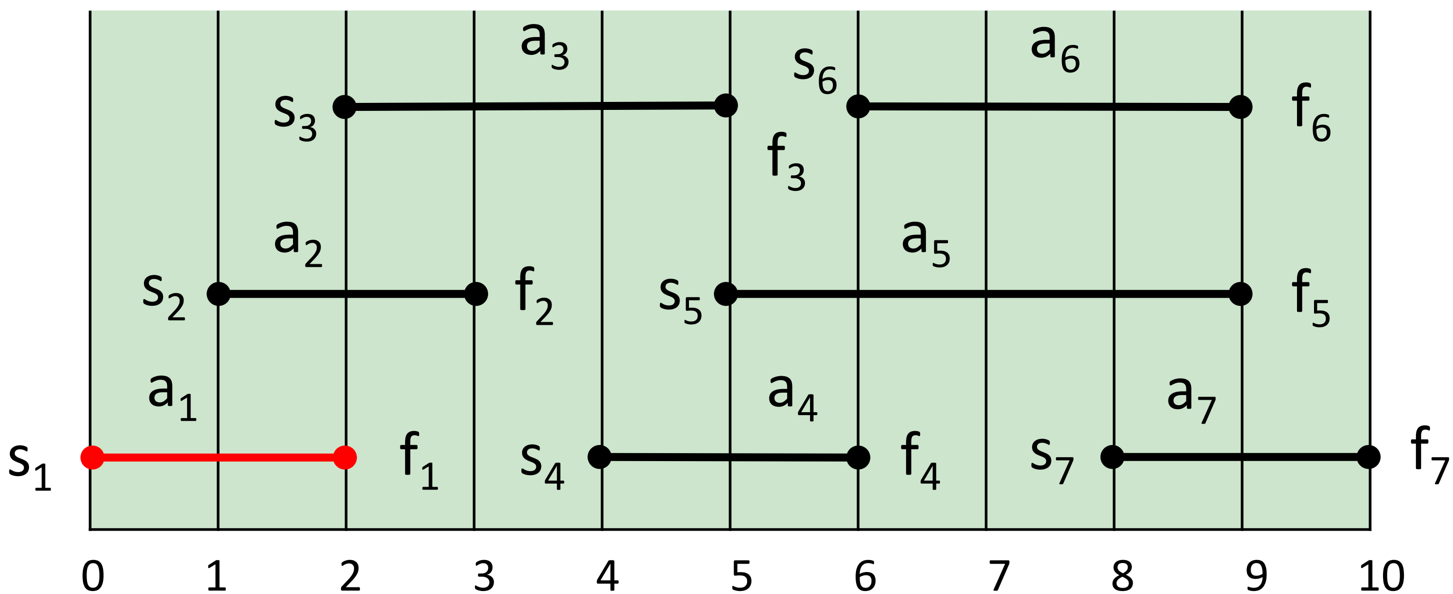
Activities in each line are compatible.



Activity-selection Problem

i	1	2	3	4	5	6	7
s_i	0	1	2	4	5	6	8
f_i	2	3	5	6	9	9	10

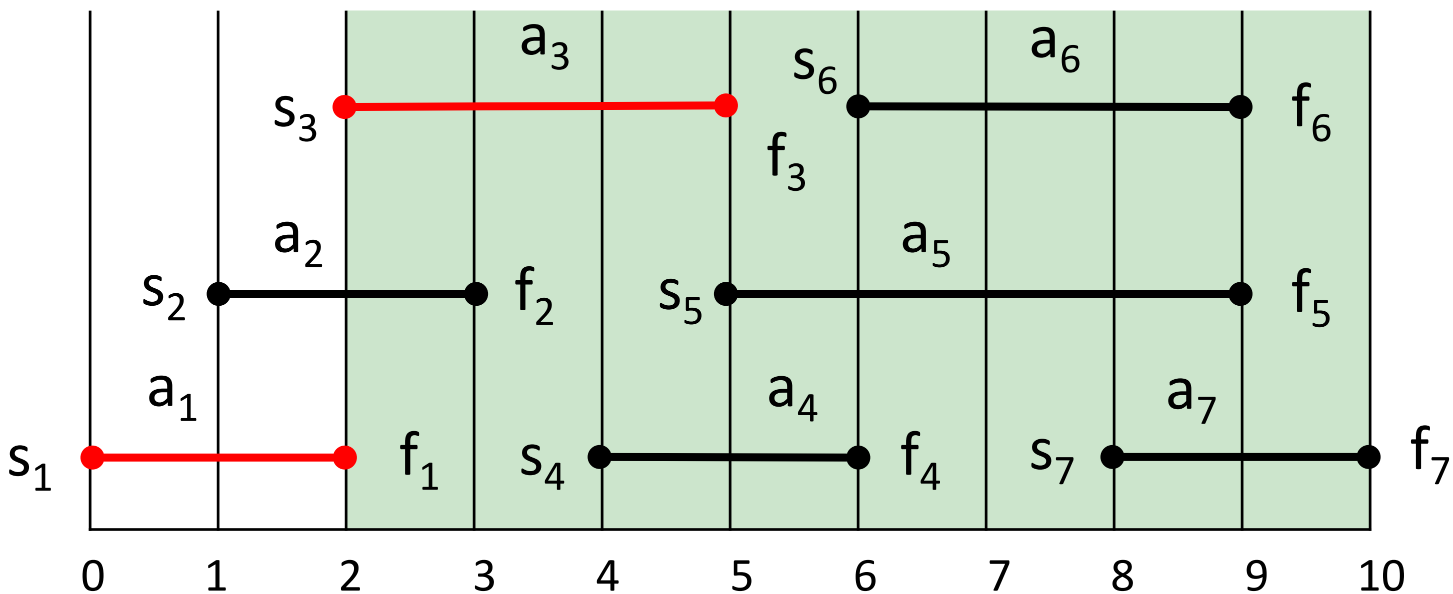
Activities sorted by finishing time.



Activity-selection Problem

i	1	2	3	4	5	6	7
s_i	0	1	2	4	5	6	8
f_i	2	3	5	6	9	9	10

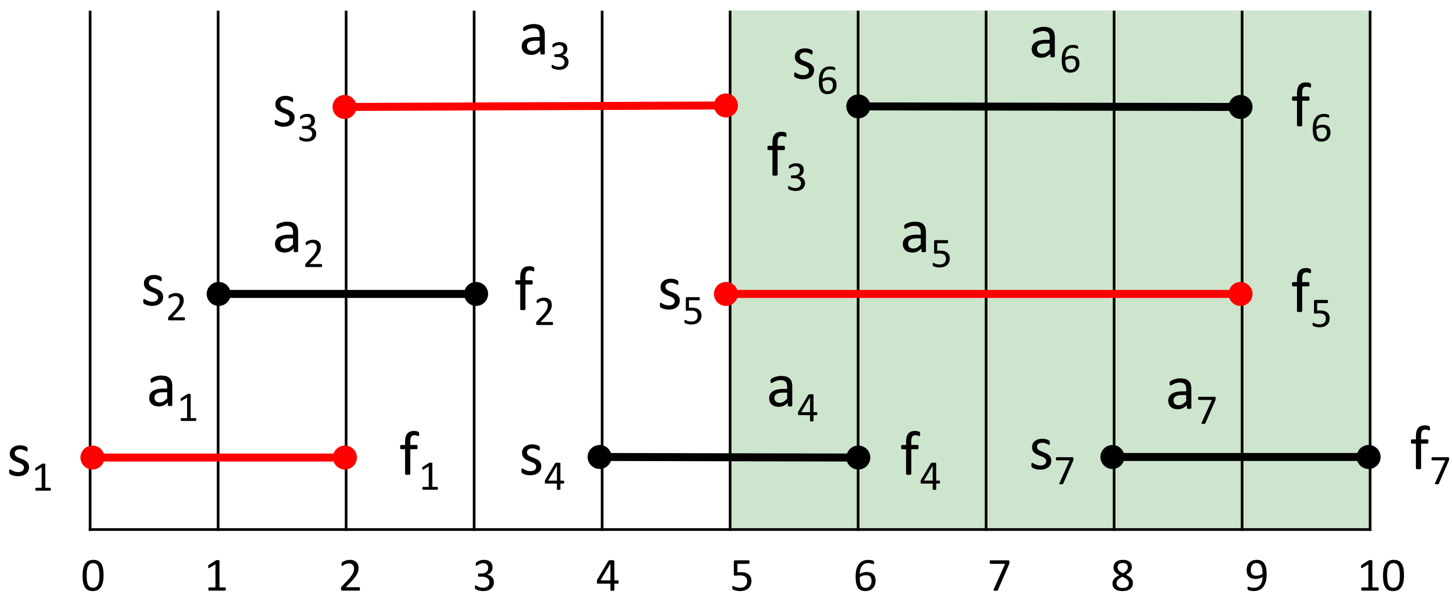
Activities sorted by finishing time.



Activity-selection Problem

i	1	2	3	4	5	6	7
s_i	0	1	2	4	5	6	8
f_i	2	3	5	6	9	9	10

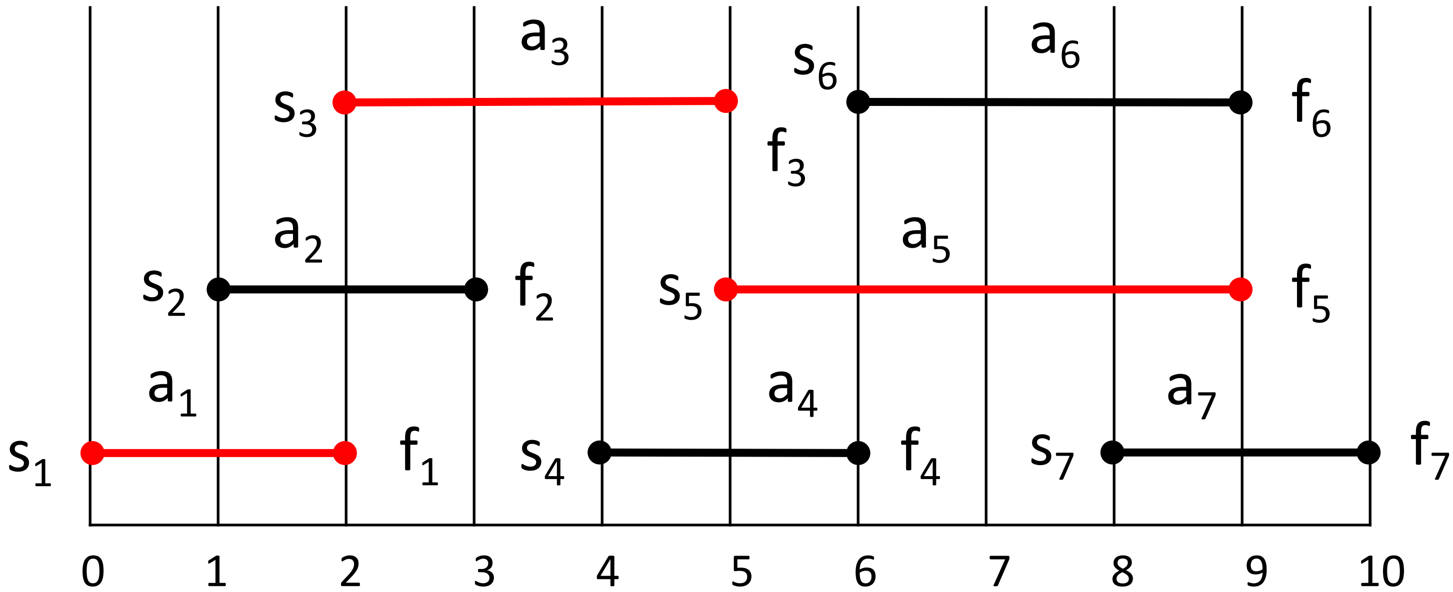
Activities sorted by finishing time.



Activity-selection Problem

i	1	2	3	4	5	6	7
s_i	0	1	2	4	5	6	8
f_i	2	3	5	6	9	9	10

Activities sorted by finishing time.



Optimal sub-structure

- Let S_{ij} = subset of activities in S that start after a_i finishes and finish before a_j starts.

$$S_{ij} = \left\{ a_k \in S : \forall i, j \quad f_i \le s_k < f_k \le s_j \right\}$$

- A_{ij} = optimal solution to S_{ij}
- $\mathbf{A_{ij} = A_{ik} \cup \{ a_k \} \cup A_{kj}}$

Greedy choice

	Before theorem
# subproblems in optimal solution	2
# choices to consider	j-i-1

$A_{ij} = A_{ik} \cup \{ a_k \} \cup A_{kj}$

We can solve the problem S_{ij} top-down:

- Consider all $a_m \in S_{ij}$
- Solve S_{im} and S_{mj}
- Pick the best m such that $A_{im} = A_{im} \cup \{ a_k \} \cup A_{im}$

Greedy choice

Theorem:

Let $S_{ij} \neq \varnothing$, and let a_m be the activity in S_{ij} with the earliest finish time: $f_m = \min\{ f_k : a_k \in S_{ij}\}$. Then:

1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.

2. $S_{im} = \varnothing$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem.

Greedy choice

	Before theorem	After theorem
# subproblems in optimal solution	2	1
# choices to consider	j-i-1	1

$\underbrace{\hspace{6em}}_{A_{ij} = A_{ik} \cup \{ a_k \} \cup A_{kj}}$ $\underbrace{\hspace{6em}}_{A_{ij} = \{ a_m \} \cup A_{mj}}$

We can now solve the problem S_{ij} top-down:

- Choose $a_m \in S_{ij}$ with the earliest finish time (greedy choice).
- Solve S_{mj}.

Challenges

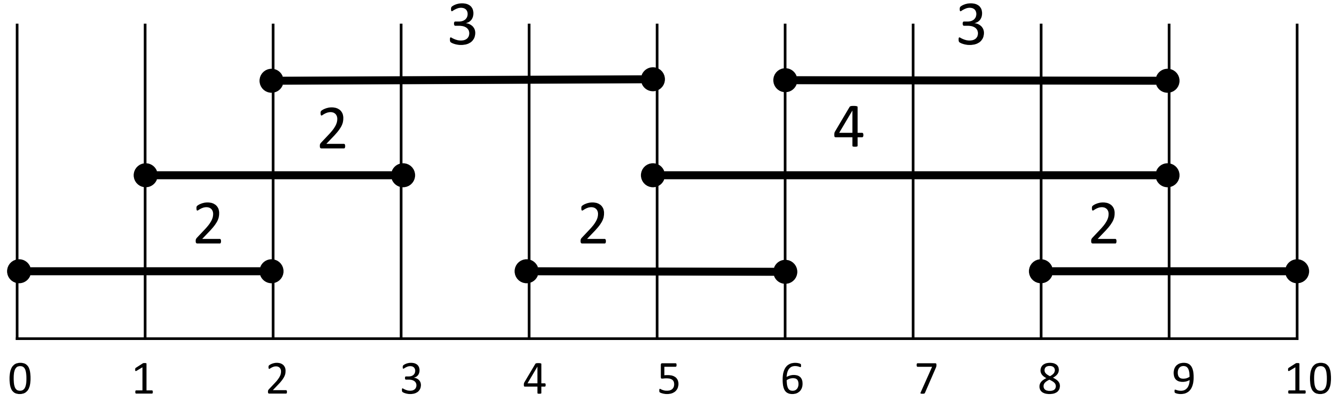
- Greedy choice is not always available.
- How to solve problem that have optimal substructures?

WEIGHTED INTERVAL SCHEDULING

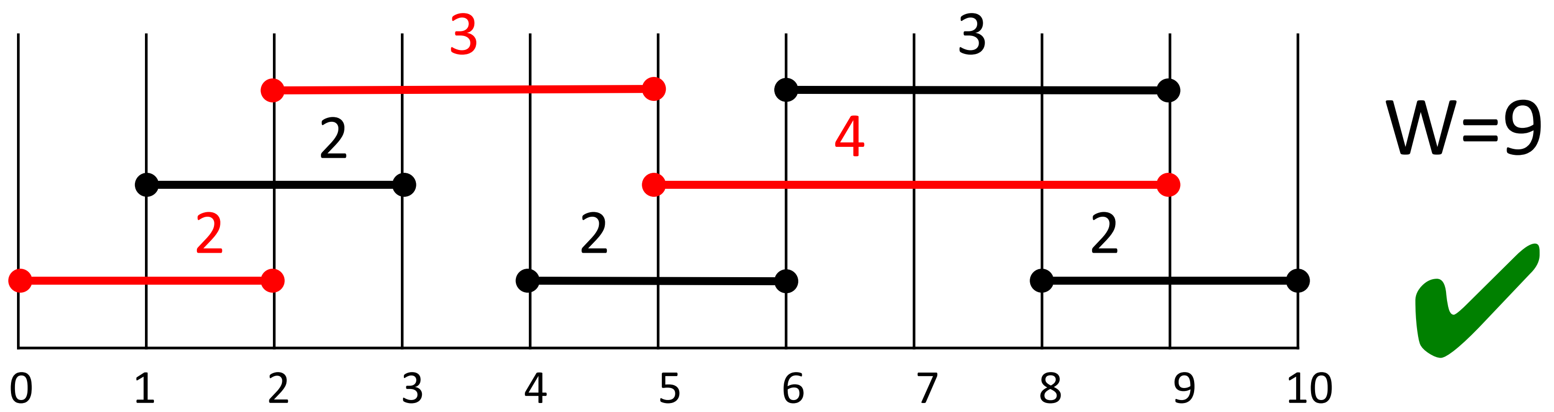
Weighted interval scheduling

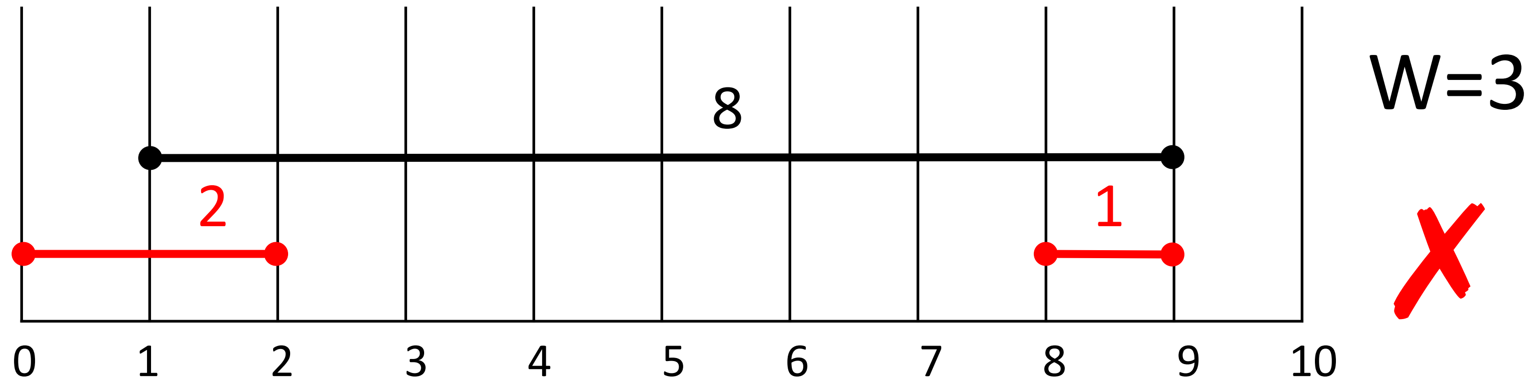
- **Input:** Set S of n activities, $a_1, a_2, \ldots, a_n$.
  - s_i = start time of activity i.
  - f_i = finish time of activity i.
  - w_i= weight of activity i
- **Output:** find maximum weight subset of mutually compatible activities.
  - 2 activities are compatible, if their intervals do not overlap.

Example:



Application of the greedy algorithm





Discussion

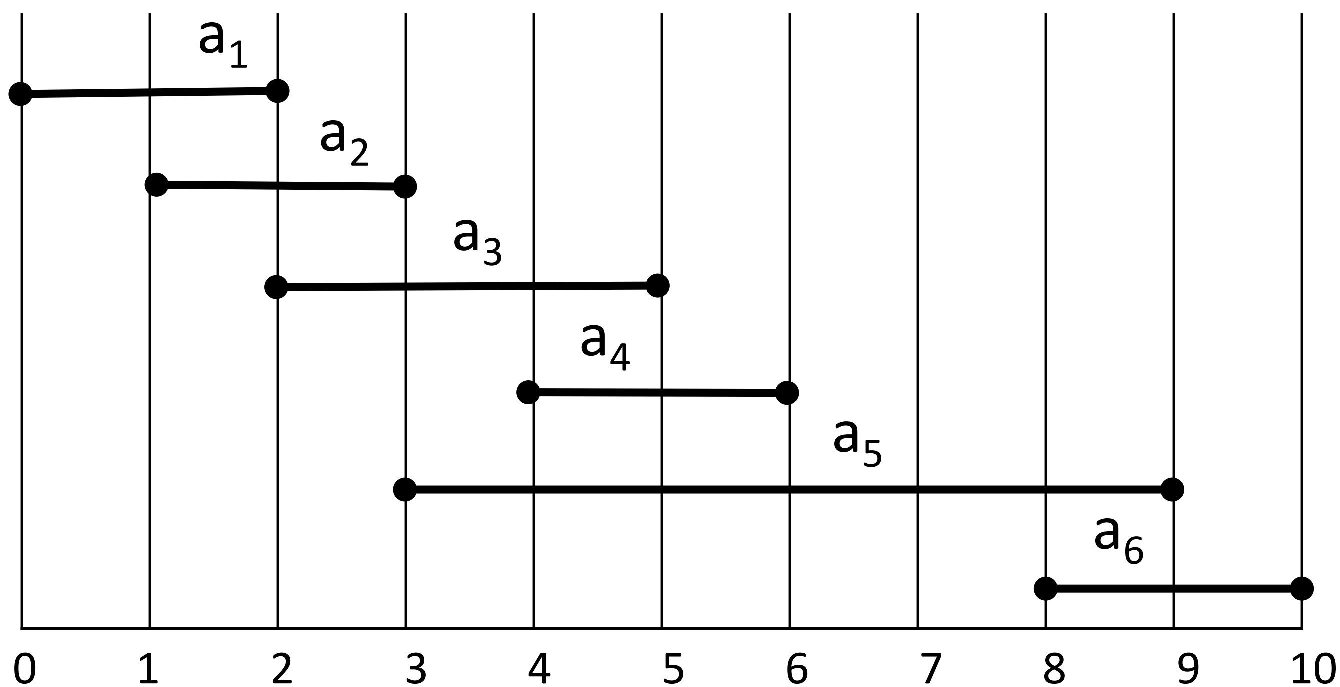
- **Optimal substructure:** ✓
  - A_{ij} = optimal solution to S_{ij}
  - $A_{ij} = A_{ik} \cup \{ a_k \} \cup A_{kj}$
- **Greedy Choice:** ✗
  - Select the activity with earliest finish time.

Data structure

Notation: All activities are sorted by finishing time $f_1 \leq f_2 \leq \ldots \leq f_n$

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

Examples: p(6)=4, p(5)=2, p(4)=2, p(2)=0.



Binary Choice

Notation: OPT(j) = value of the optimal solution to the problem
= max total weight of compatible activities 1 … j

Case 1: OPT selects activity j

- Add weight w_j
- Cannot use incompatible activities
- Must include optimal solution on remaining compatible activities { 1, 2, … , p(j) }.

Optimal substructure property

Case 2: OPT does not select activity j

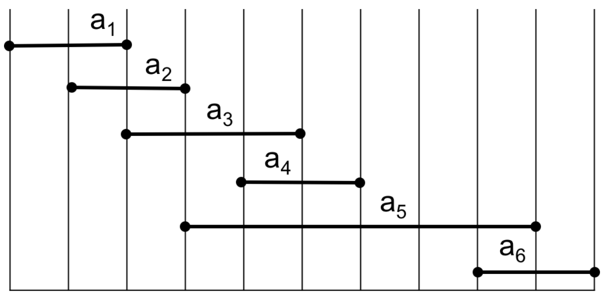
Must include optimal solution on others activities { 1, 2, …, j-1 }.

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

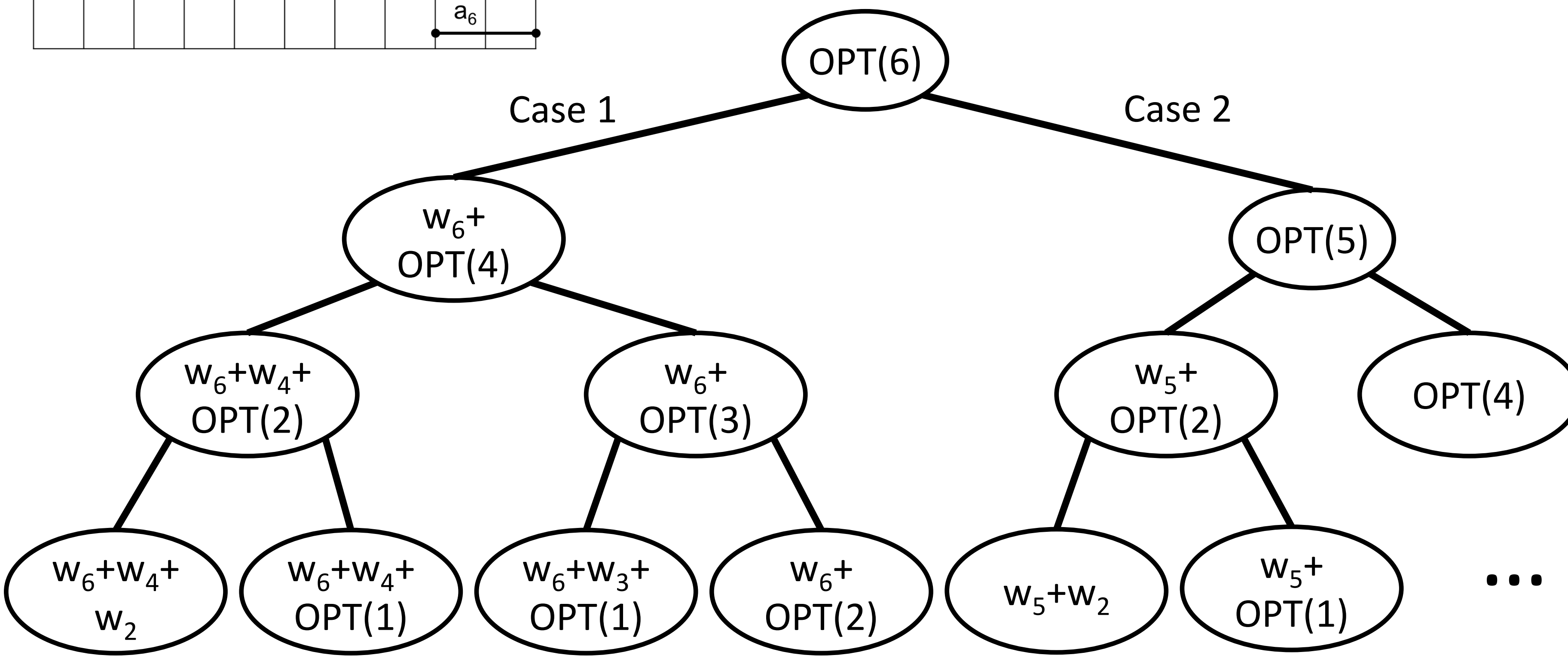
Recursive call

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

Brute Force Approach



Observation: OPT(j) is calculated multiple times…



Memoization

Memoization: Cache results of each subproblem; lookup as needed.

```
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.

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

Running time

Claim. Memoized version of algorithm takes $O(n \log n)$ time.

- Sort by finish time: $O(n \log n)$.
- Computing $p(\cdot)$: $O(n \log n)$ via sorting by start time.

- M-COMPUTE-OPT(j): each invocation takes $O(1)$ time and either
  - (i) returns an existing value `M[j]`
  - (ii) fills in one new entry `M[j]` and makes two recursive calls

- Progress measure $\Phi = \#$ nonempty entries of `M[]`.
  - initially $\Phi = 0$, throughout $\Phi \leq n$.
  - (ii) increases Φ by $1 \Rightarrow$ at most $2n$ recursive calls.

- Overall running time of M-COMPUTE-OPT(n) is $O(n)$. ▪

Remark. $O(n)$ if jobs are presorted by start and finish times.

DYNAMIC PROGRAMMING

Bottom-up

Observation: When we compute M[j], we only need values M[k] for k<j.

```
BOTTOM-UP (n;s1,...,sn;f1,...,fn;v1,...,vn)

Sort jobs by finish time so that f1≤f2≤...≤fn.
Compute p(1), p(2), ..., p(n).
M[0]←0
for j = 1 TO n
   M[j] ← max { vj + M[p(j)], M[j–1] }
```

Main Idea of Dynamic Programming: Solve the sub-problems in an order that makes sure when you need an answer, it's already been computed.

Finding a solution

Dyn. Prog. algorithm computes optimal value.

Q: How to find solution itself?

A: Bactrack!

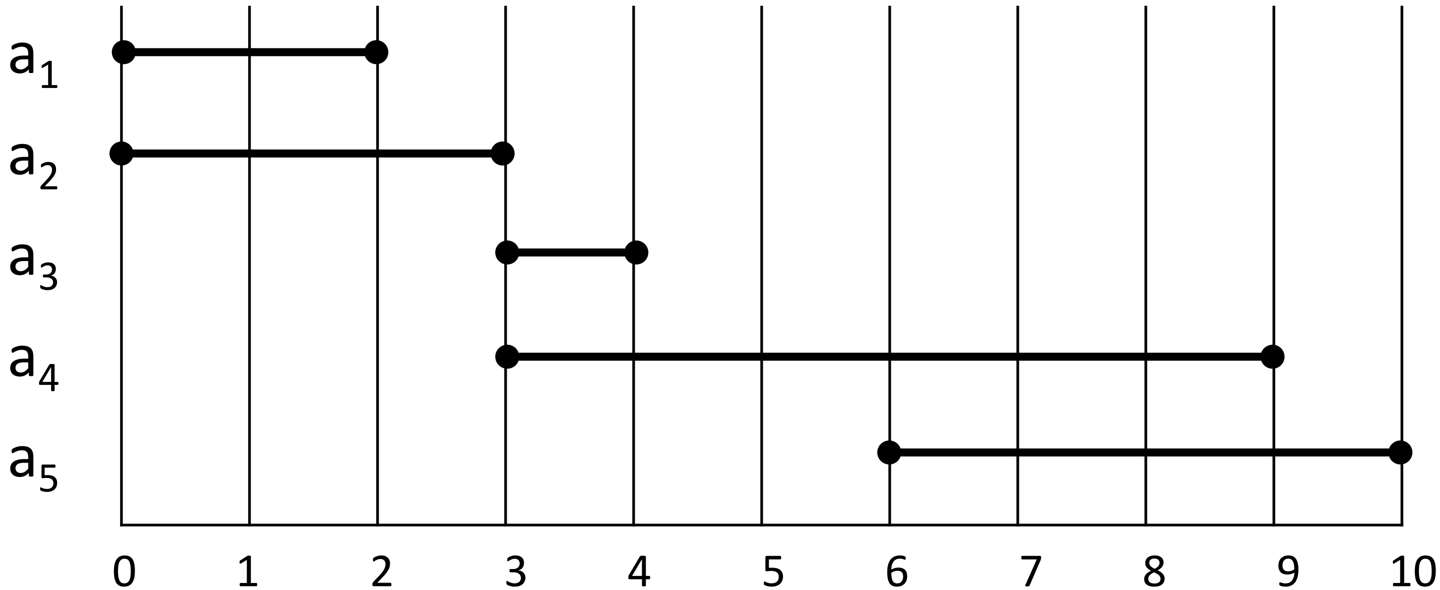
```
Find-Solution(j)
if j = 0
   return ∅.
else if (v[j] + M[p[j]] > M[j–1])
   return { j } ∪ Find-Solution(p[j])
else
   return Find-Solution(j–1).
```

Analysis. # of recursive calls $\leq n \Rightarrow O(n)$.

Example: Computing solution

activity	1	2	3	4	5
predecessor	0	0	2	2	3
Best weight M	-	-	-	-	-
V_j+M[p(j)]	-	-	-	-	-
M[j-1]	-	-	-	-	-

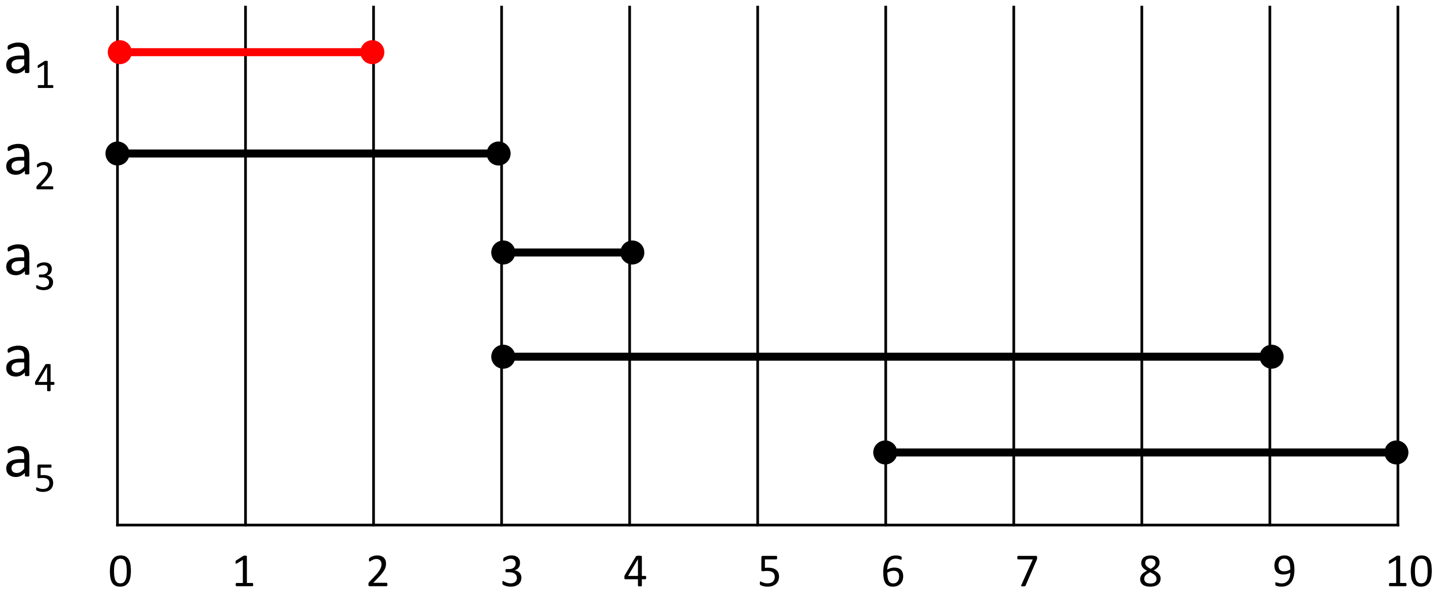
(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



Example: Computing solution

activity	1	2	3	4	5
predecessor	0	0	2	2	3
Best weight M	2	-	-	-	-
V_j+M[p(j)]	**2**	-	-	-	-
M[j-1]	0	-	-	-	-

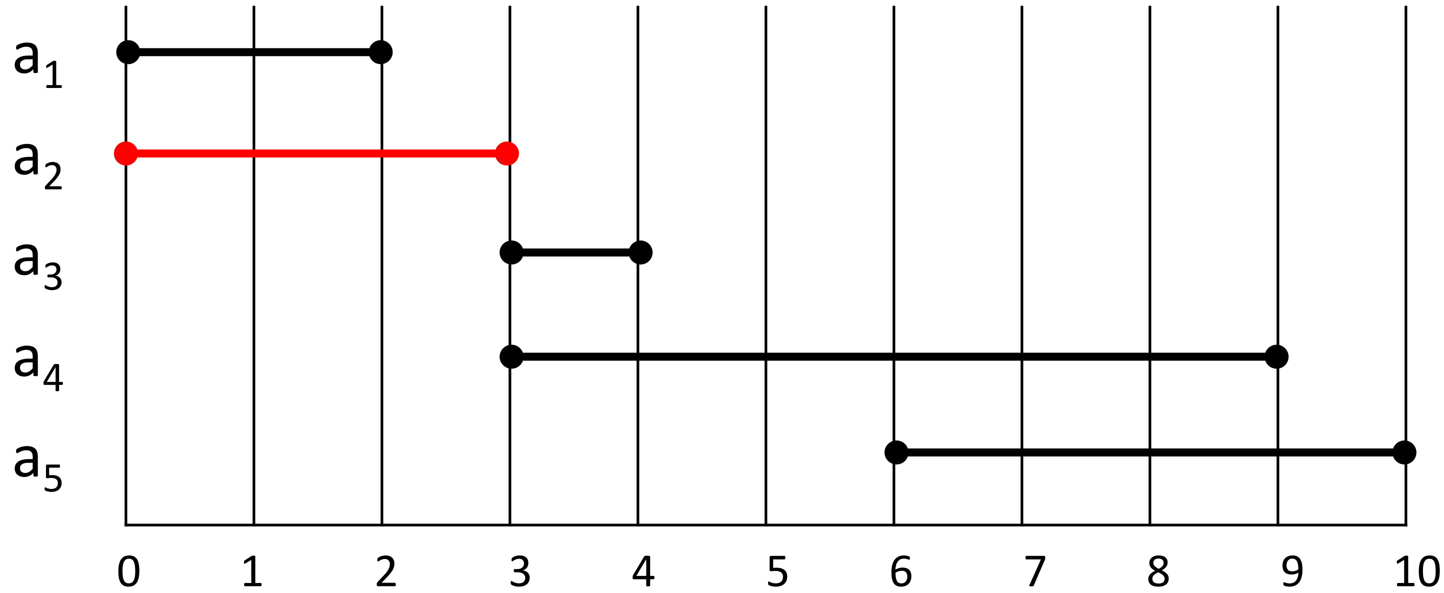
(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



Example: Computing solution

activity	1	2	3	4	5
predecessor	0	0	2	2	3
Best weight M	2	3	-	-	-
V_j+M[p(j)]	2	**3**	-	-	-
M[j-1]	0	2	-	-	-

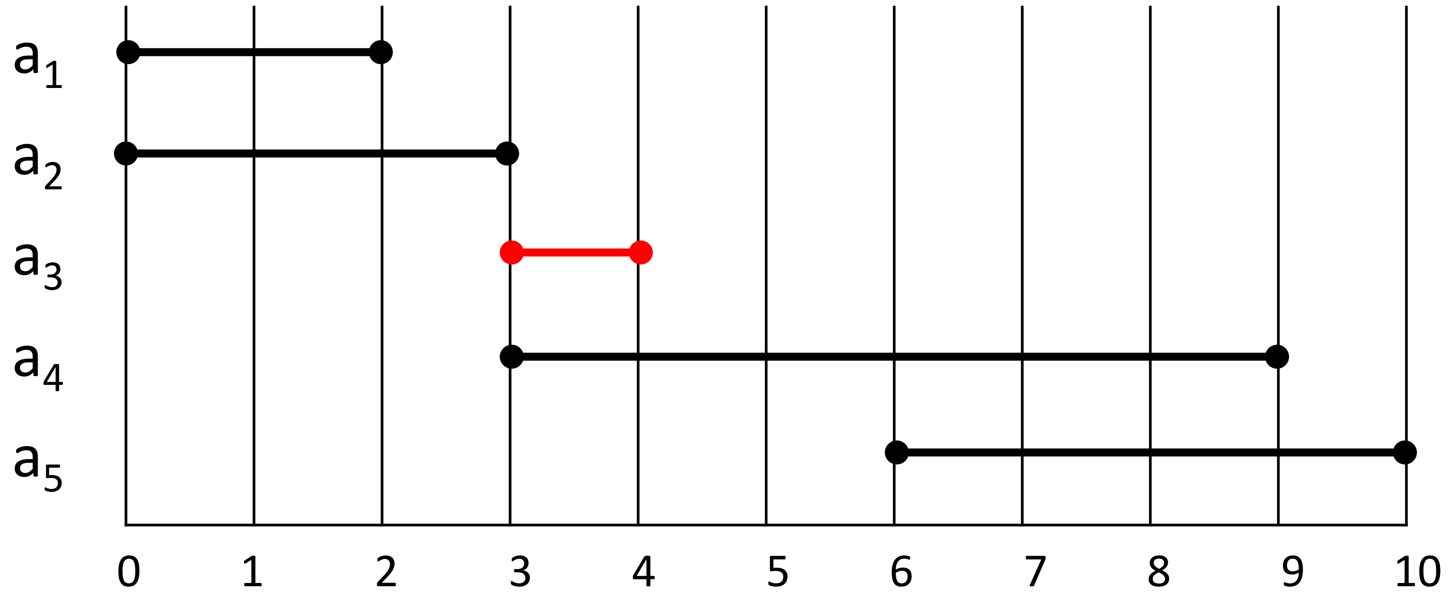
(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



Example: Computing solution

activity	1	2	3	4	5
predecessor	0	0	2	2	3
Best weight M	2	3	4	-	-
V_j+M[p(j)]	2	3	**4**	-	-
M[j-1]	0	2	3	-	-

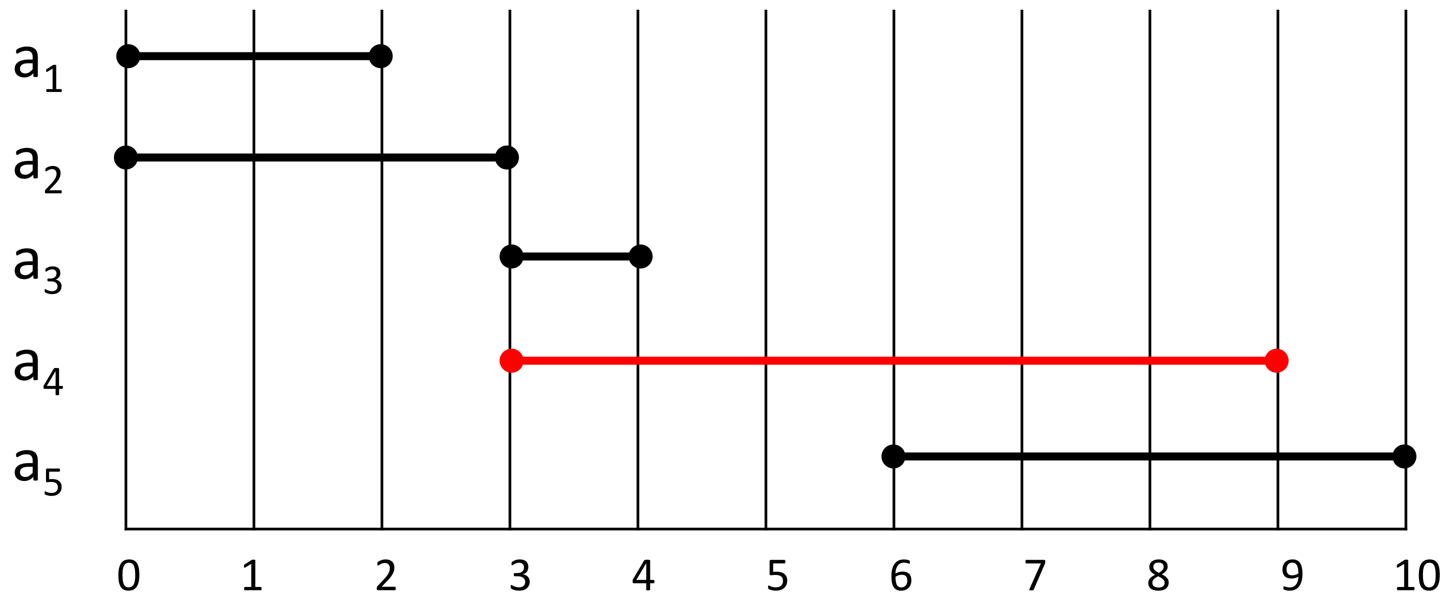
(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



Example: Computing solution

activity	1	2	3	4	5
predecessor	0	0	2	2	3
Best weight M	2	3	4	9	-
V_j+M[p(j)]	2	3	4	**9**	-
M[j-1]	0	2	3	4	-

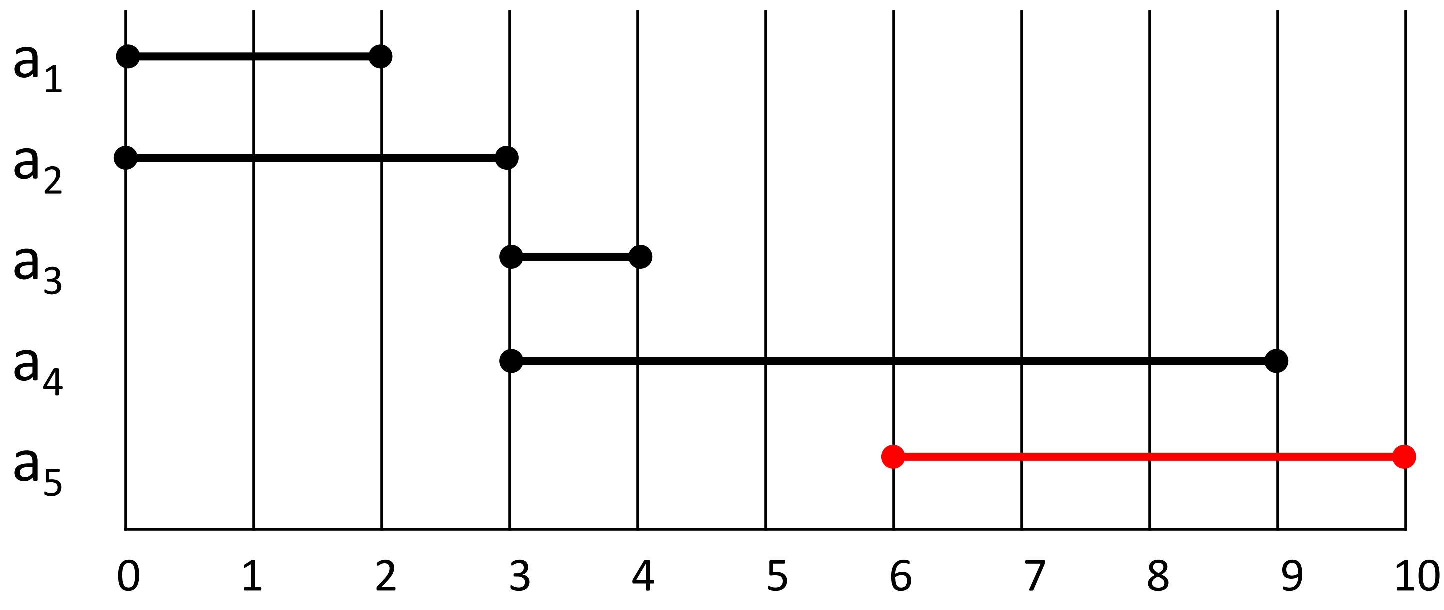
(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



Example: Computing solution

activity	1	2	3	4	5
predecessor	0	0	2	2	3
Best weight M	2	3	4	9	9
$V_j+M[p(j)]$	2	3	4	9	8
M[j-1]	0	2	3	4	**9**

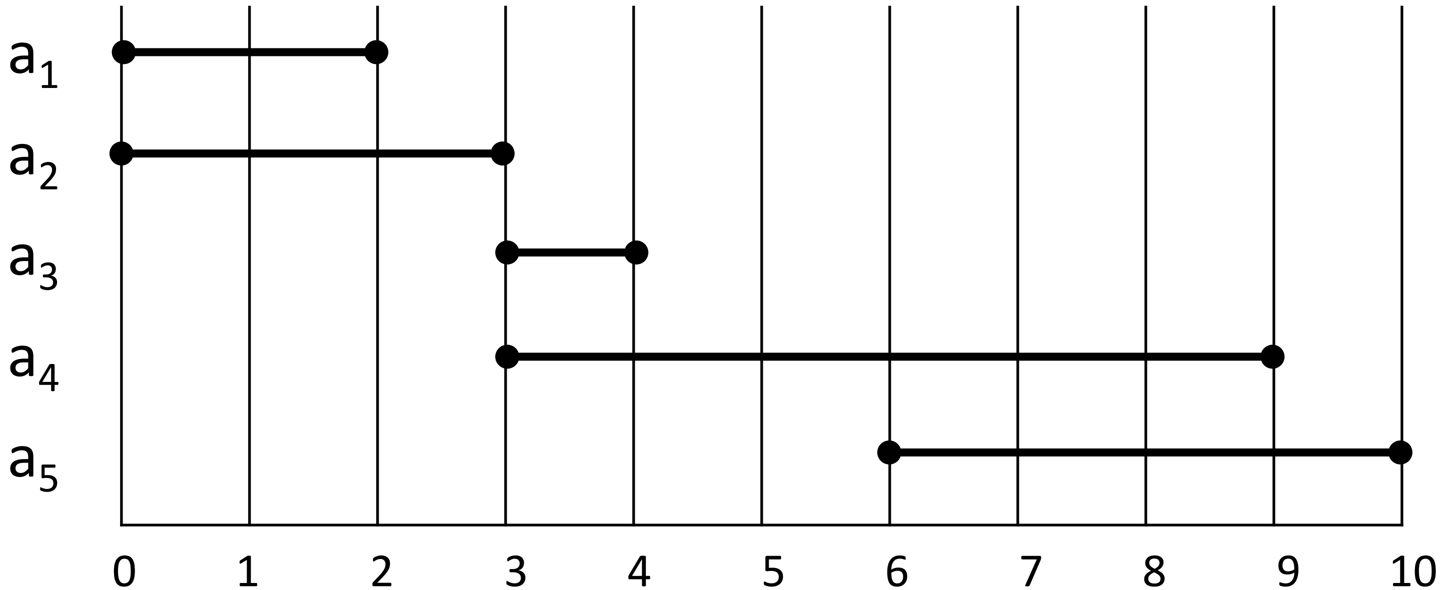
(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



Example: Reconstruction

activity	1	2	3	4	5
predecessor	0	0	2	2	3
Best weight M	2	3	4	9	9
V_j+M[p(j)]	2	3	4	9	8
M[j-1]	0	2	3	4	9

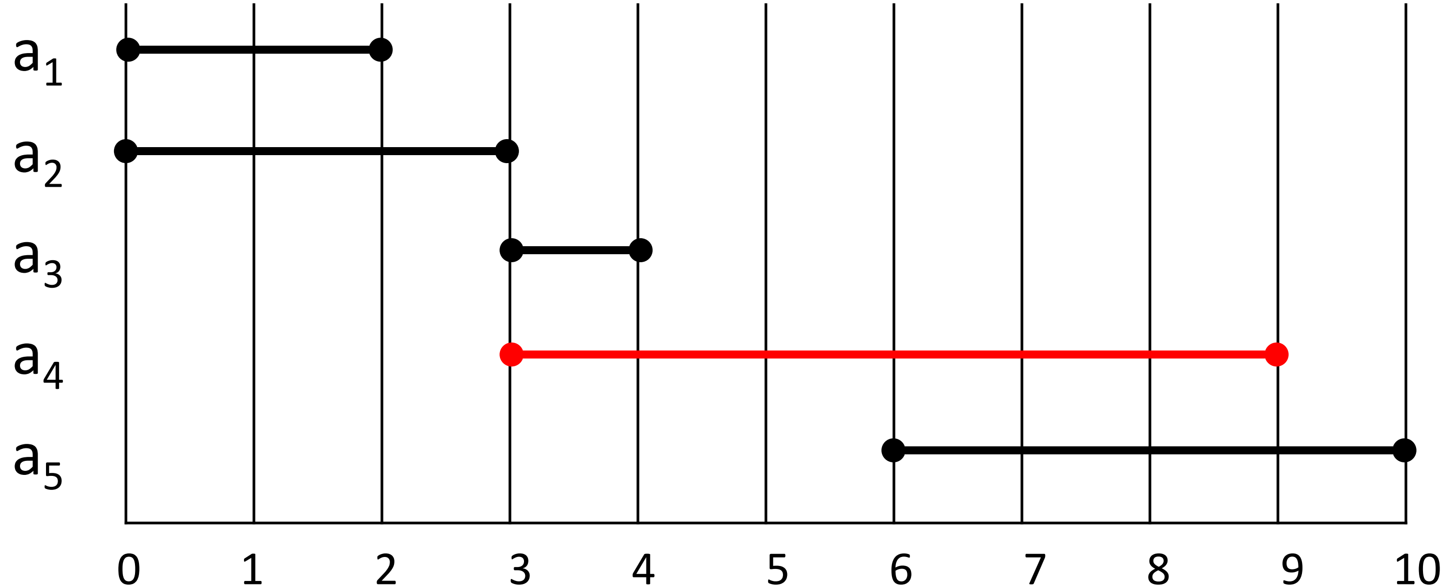
(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



Example: Reconstruction

activity	1	2	3	4	5
predecessor	0	0	2	2	3
Best weight M	2	3	4	9	9
V_j+M[p(j)]	2	3	4	9	8
M[j-1]	0	2	3	4	9

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



Example: Reconstruction

activity	1	2	3	4	5
predecessor	0	0	2	2	3
Best weight M	2	3	4	9	9
V_j+M[p(j)]	2	3	4	9	8
M[j-1]	0	2	3	4	9

(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.