

COMP251: Dynamic programming (1)

Jérôme Waldispühl & Roman Sarrazin-Gendron

School of Computer Science

McGill University

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

Algorithm 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.



Although both techniques seems disconnected, we will highlight similarities.

An example?

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 = ?$$

20!

$$1+1 = ?$$

21

Principle: Use answers previously computed for a smaller instance

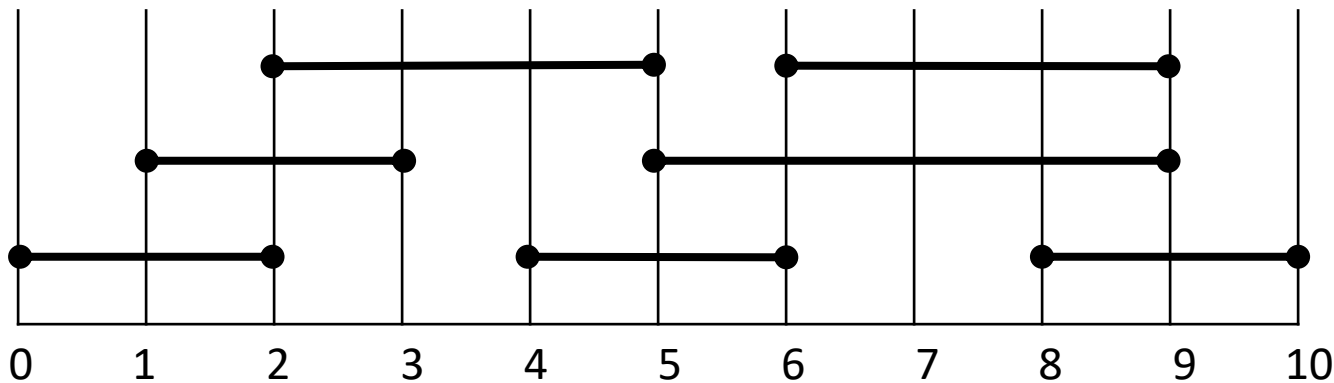
INTRODUCTION

Activity-selection Problem

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

Example:

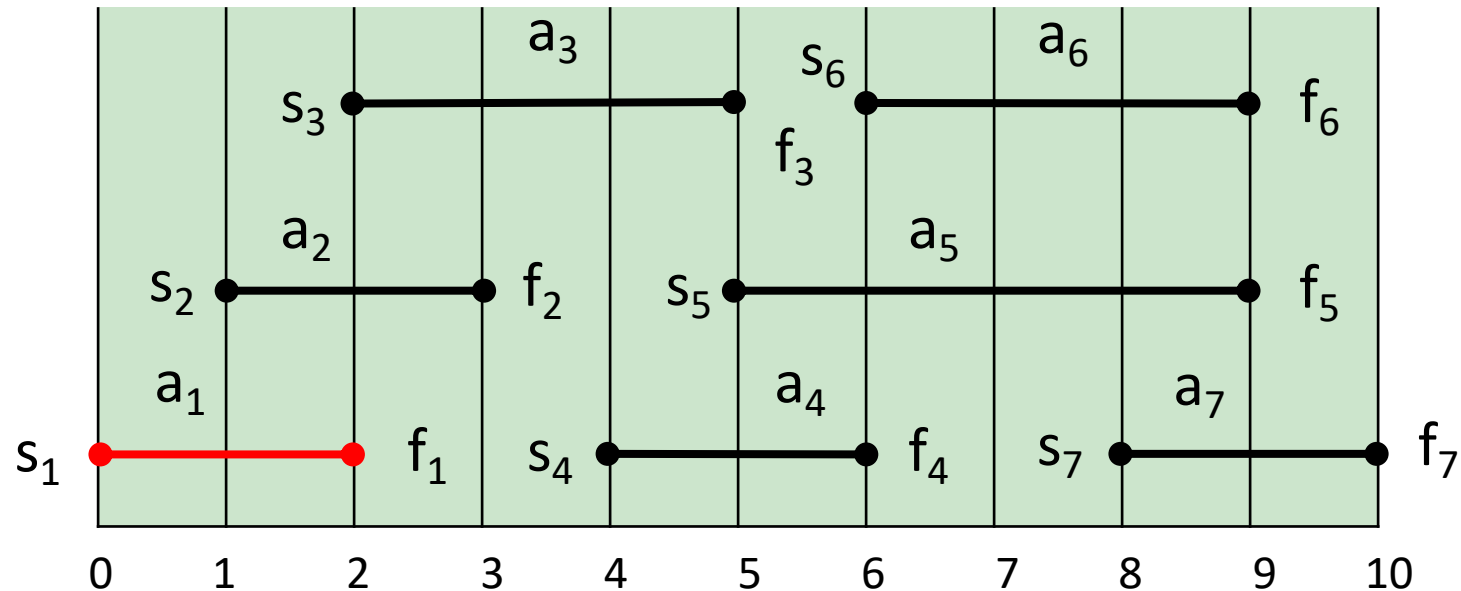
Activities in each line
are compatible.



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

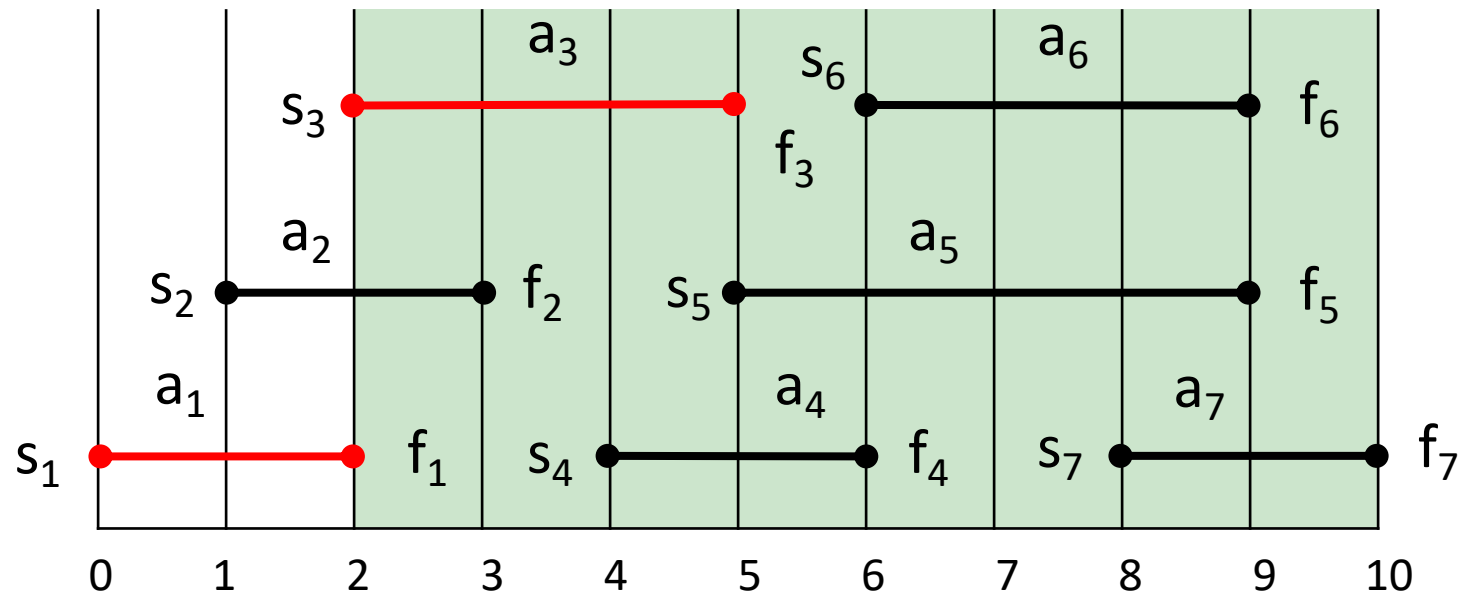
Activities sorted by finishing time.



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

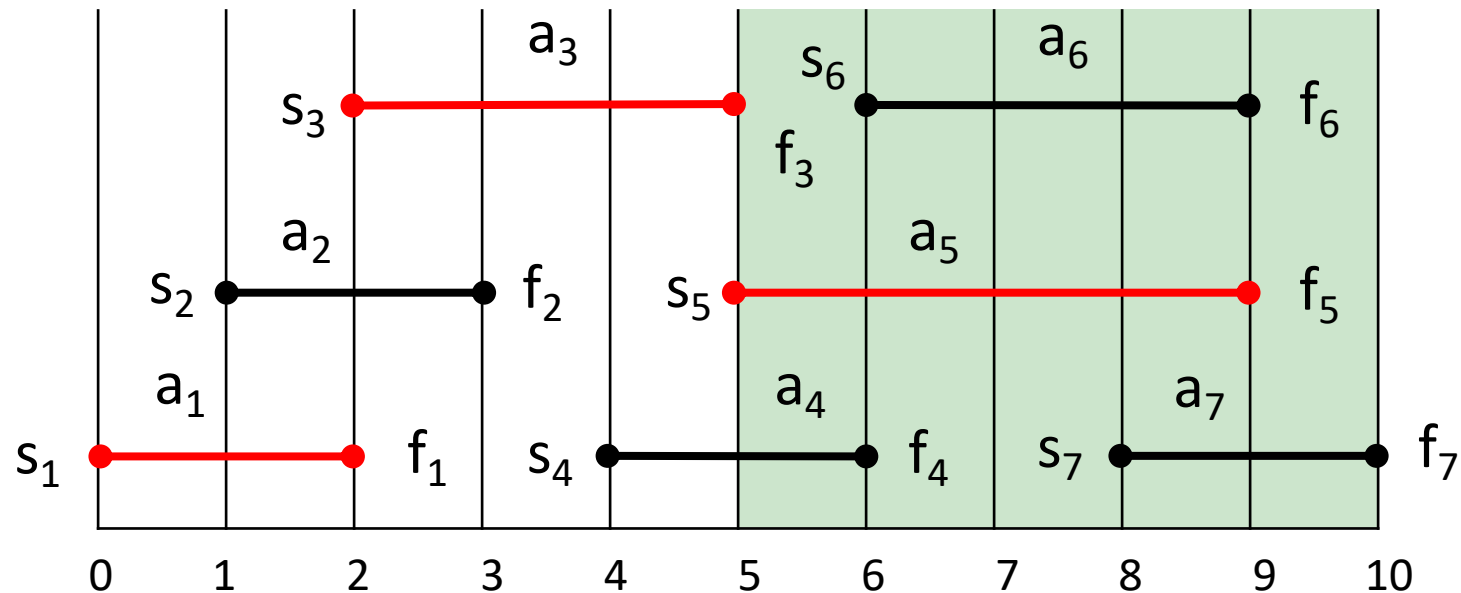
Activities sorted by finishing time.



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

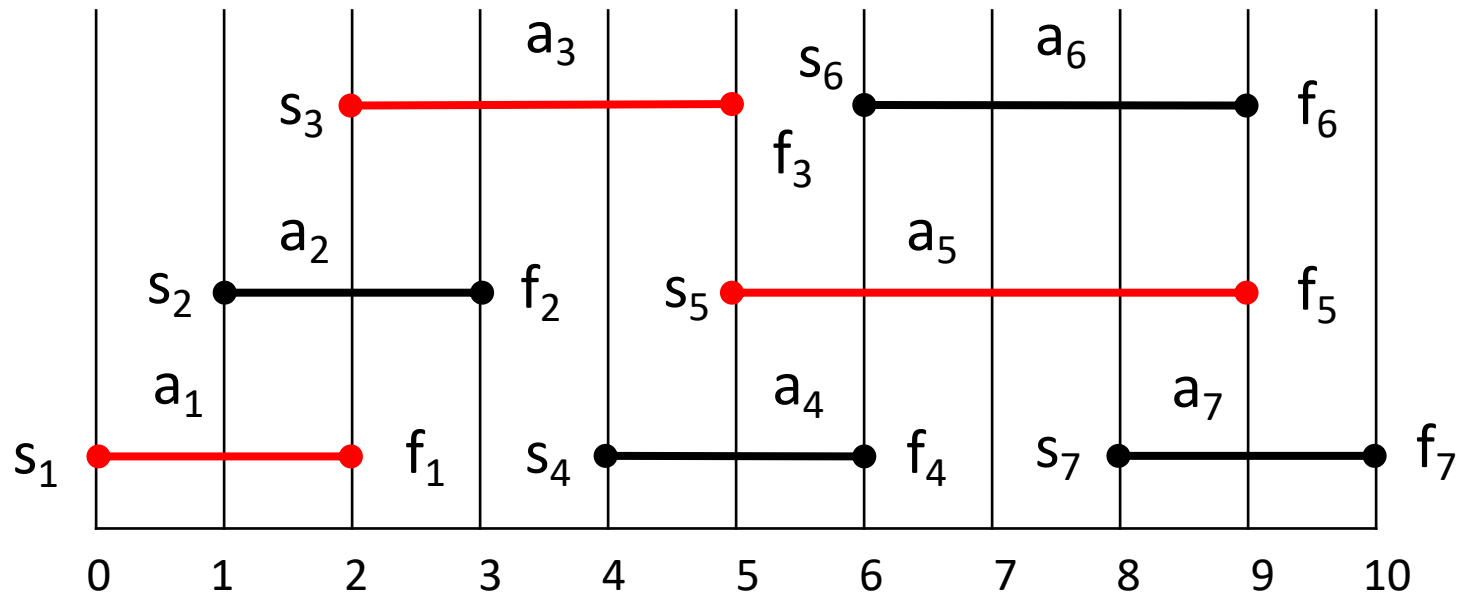
Activities sorted by finishing time.



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.

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

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

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

Greedy choice

Theorem:

Let $S_{ij} \neq \emptyset$, 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. $S_{im} = \emptyset$, 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
	$A_{ij} = A_{ik} \cup \{ a_k \} \cup A_{kj}$	$A_{ij} = \{ a_m \} \cup A_{mj}$

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} .

Objective

- A greedy algorithm can compute an optimal solution if we identify:
 - a greedy choice
 - an optimal substructure property
- A greedy choice is not always available.
- What can we do if we have an optimal substructures property but not a greedy choice?
- How can we use the optimal substructures property to design an efficient algorithm?

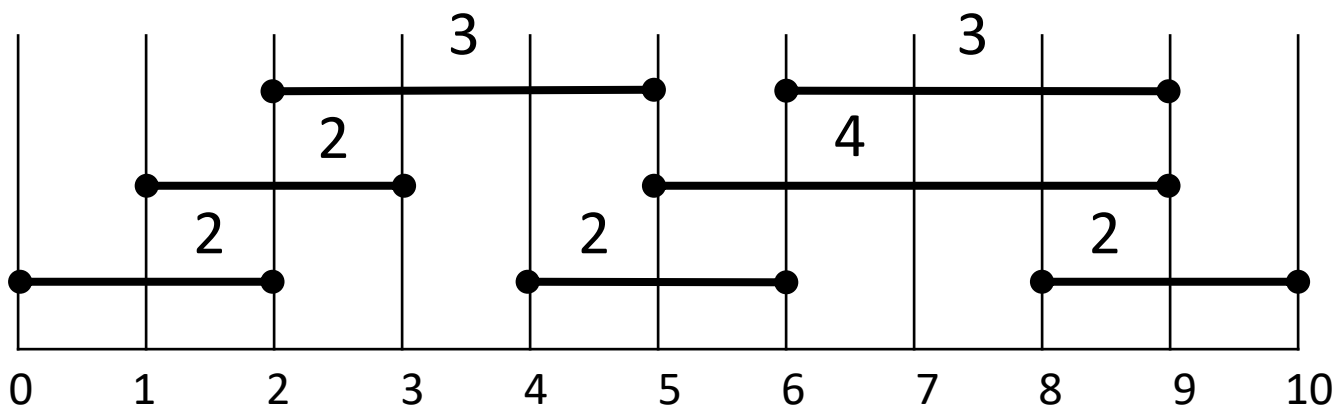
We will illustrate this approach on a variant of the interval scheduling problem. Next week, we will review more examples.

WEIGHTED INTERVAL SCHEDULING

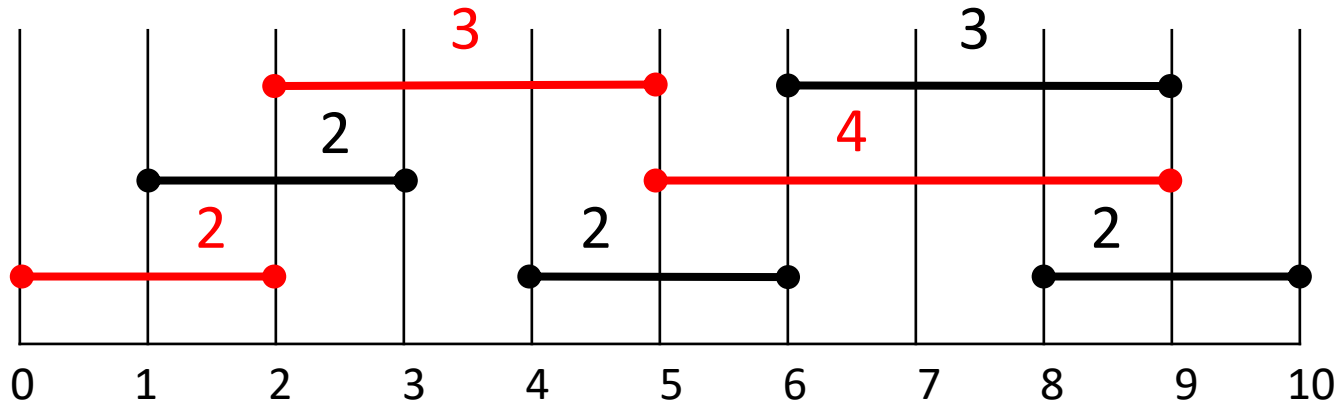
Weighted interval scheduling

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

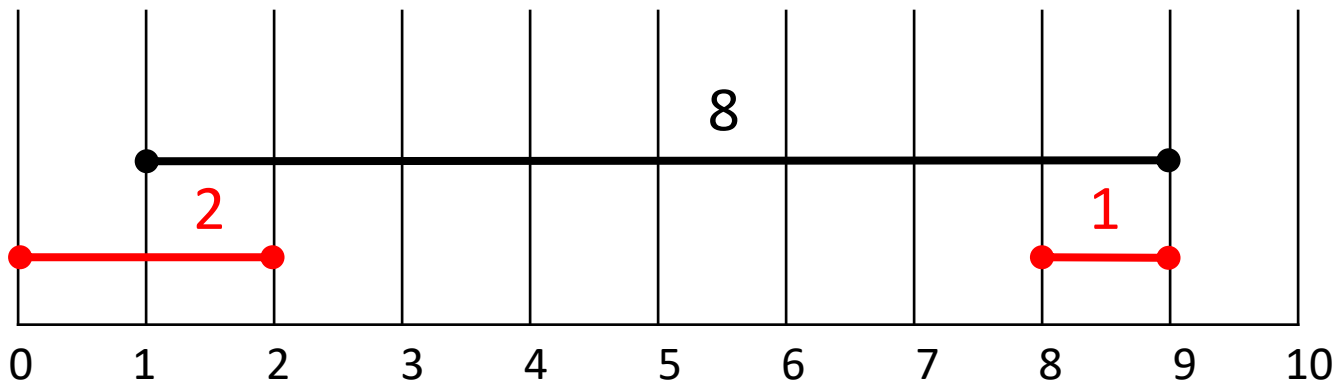
Example:



Application of the greedy algorithm



$W=9$



$W=3$



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 finishing time.

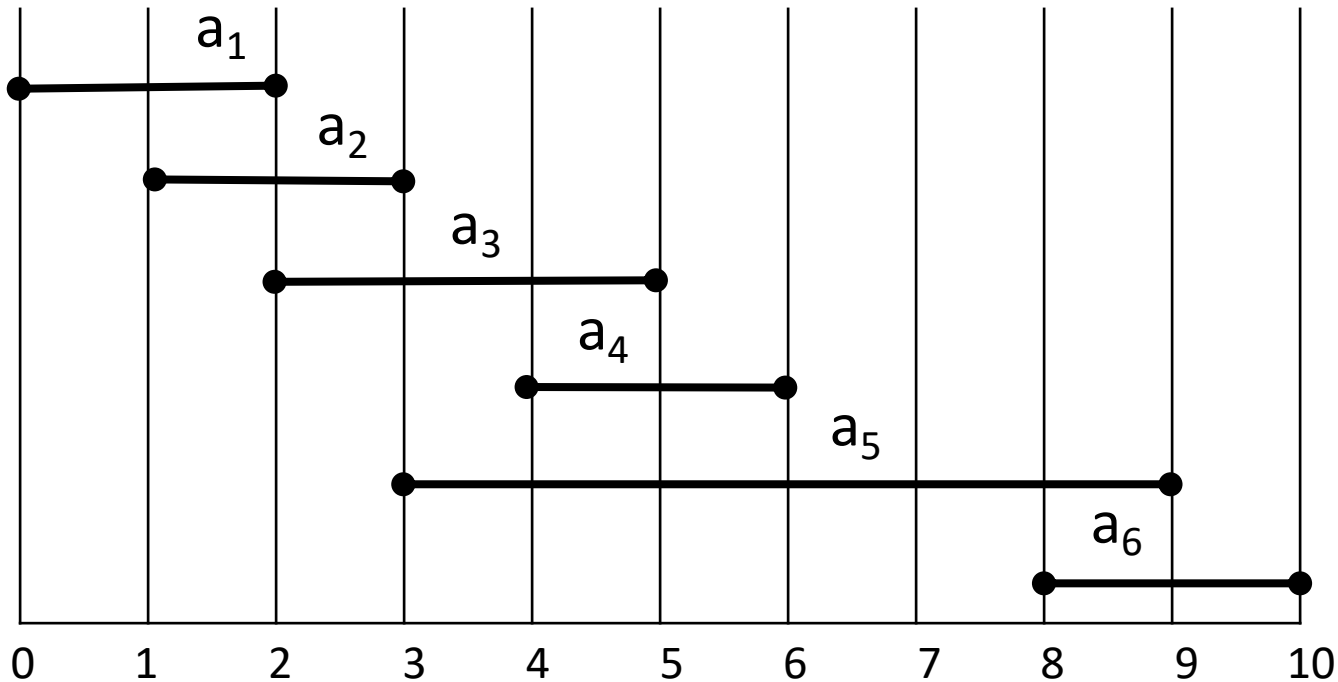
Without the greedy choice property, we need to consider all possible decompositions of A_{ij} to find the optimal one.

Data structure (1)

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

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

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



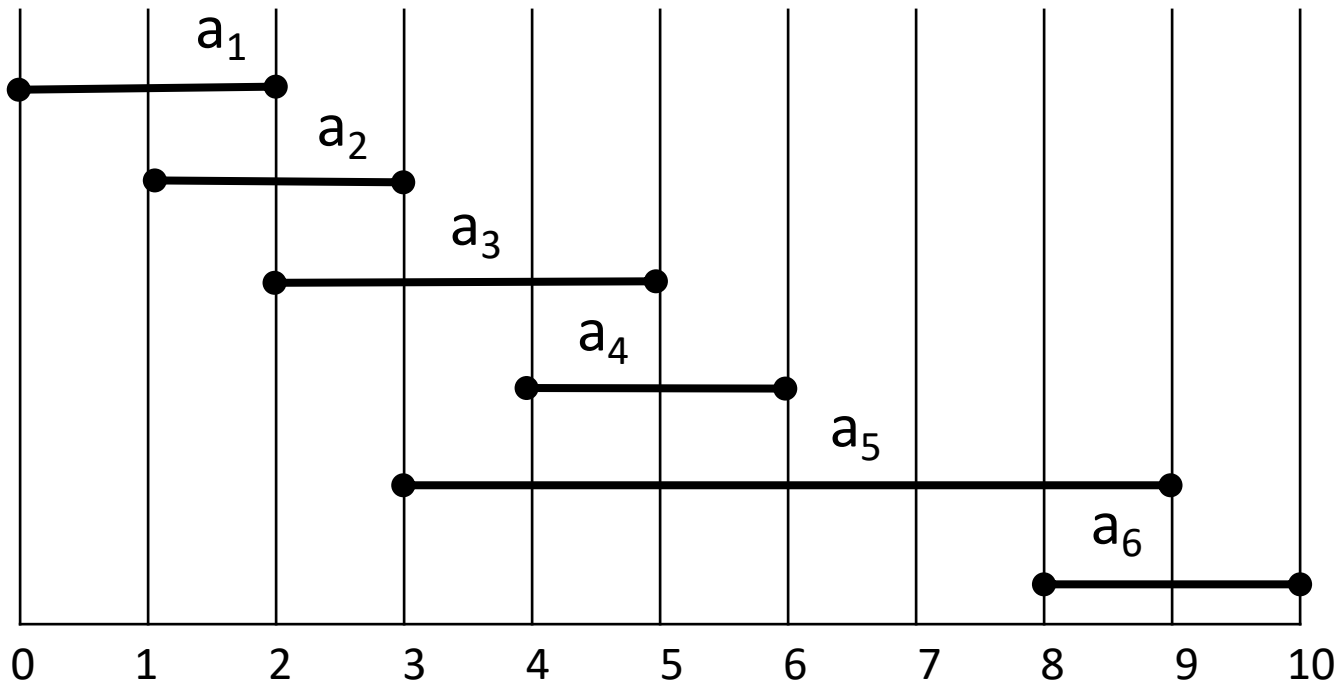
OPT() stores the value we want to optimize

Data Structure (2)

Note: OPT(n) is the solution to our problem, where n is the number of activities.

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

Examples: OPT(6) = 8, OPT(3)=5, OPT(1)=2



Binary Choice

Objective: We want to recursively compute OPT.

Question: Is activity j used in the optimal solution $OPT(j)$?

Case 1: *OPT uses activity j*

- The weight w_j is used to compute $OPT(j)$
- But we cannot use activities incompatible with j
- The activities compatible with activity j are $\{ 1, 2, \dots, p(j) \}$.

Case 2: *OPT does not use activity j*

The optimal solution is made with other activities $\{ 1, 2, \dots, j-1 \}$.

Optimal substructure property



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

Recursive Algorithm

Input: $n, s[1..n], f[1..n], w[1..n]$

Number of activities, starting and finishing times, weights

Preprocessing:

- Sort activities by finishing time $f[1] \leq \dots \leq f[n]$
- Compute $p[1], p[2], \dots, p[n]$

Main:

Compute-Opt(j)

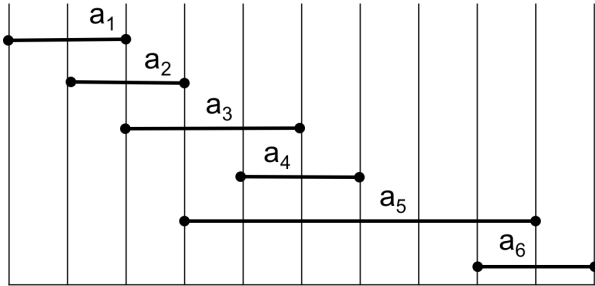
if $j = 0$

 return 0

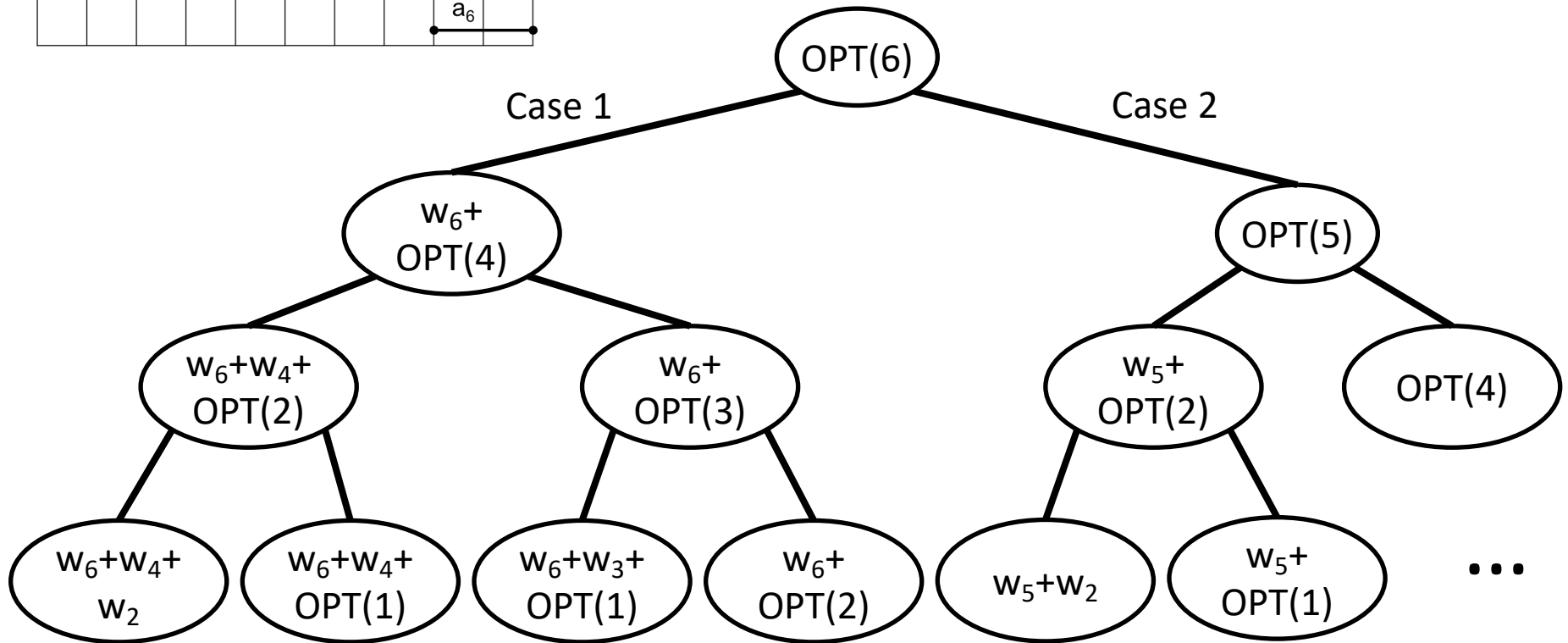
else

 return $\max(w[j] + \text{Compute-Opt}(p[j]), \text{Compute-Opt}(j-1))$

Brute Force Approach



Observation: $OPT(j)$ is calculated multiple times...



Memoization

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

Input: $n, s[1..n], f[1..n], w[1..n]$

Sort jobs by finish time so that $f[1] \leq f[2] \leq \dots \leq f[n]$.

Compute $p[1], p[2], \dots, p[n]$.

for $j = 1$ to n

$OPT[j] \leftarrow \text{empty}$.

$OPT[0] \leftarrow 0$. Initialization of OPT .

We store the values of $OPT(j)$ in a table, so that we can re-use them instead of computing them again.

Compute-Opt(j)

if $OPT[j]$ is empty

$OPT[j] \leftarrow \max(w[j] + \text{Compute-Opt}(p[j]), \text{Compute-Opt}(j-1))$

return $OPT[j]$.

Running time

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

- Sort by finishing time: $O(n \log n)$
- Computing $p()$: $O(n \log n)$ via sorting by starting time
- Compute-Opt(j): each invocation takes $O(1)$ time, and either:
 - i. Returns an existing value OPT(j)
 - ii. Fills in one new entry OPT(j) and makes two recursive calls
- Progress measure $\phi = \#$ non-empty entries of OPT
 - i. Initially $\phi = 0$, throughout $\phi \leq n$
 - ii. Increases ϕ by 1 \Rightarrow at most $2n$ recursive calls
- Overall running time of Compute-Opt(n) is $O(n)$

Note: $O(n)$ if the activities are presorted

DYNAMIC PROGRAMMING

Bottom-up

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

BOTTOM-UP $(n; s_1, \dots, s_n; f_1, \dots, f_n; w_1, \dots, w_n) :$

Sort jobs by finish time so that $f_1 \leq f_2 \leq \dots \leq f_n$

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

$OPT[0] \leftarrow 0$

for $j = 1$ to n

$OPT[j] \leftarrow \max \{ w_j + OPT[p(j)], OPT[j-1] \}$

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.

For now, you can see it as a variant of the memoization algorithm that incrementally compute the $OPT(k)$

Finding a solution

Dyn. Prog. algorithm computes optimal value.

Q: How to find solution itself?

A: Backtrack!

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

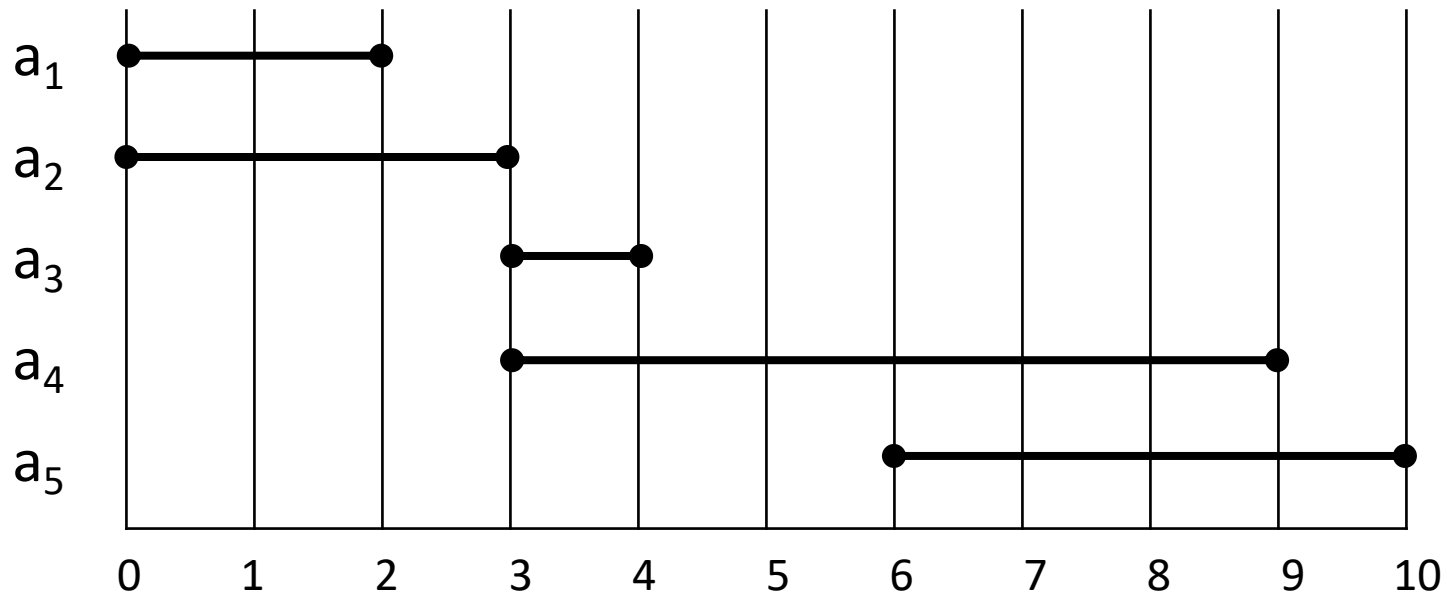
Analysis. # of recursive calls $\leq n \Rightarrow O(n)$.

Example: Computing solution

This is $p()$

activity	1	2	3	4	5
predecessor	0	0	2	2	3
OPT[j]	-	-	-	-	-
$w_j + \text{OPT}[p(j)]$	-	-	-	-	-
OPT[j-1]	-	-	-	-	-

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

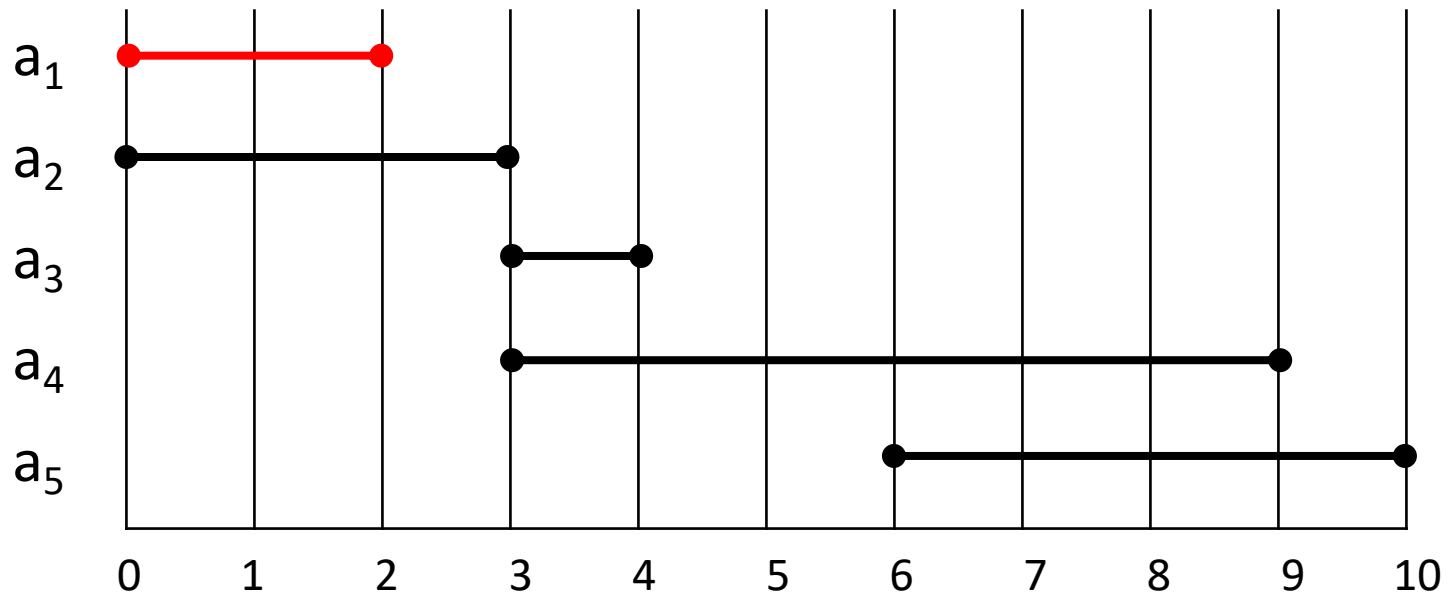


Example: Computing solution

$M[0]=0$

activity	1	2	3	4	5
predecessor	0	0	2	2	3
OPT[j]	2	-	-	-	-
$w_j + \text{OPT}[p(j)]$	2	-	-	-	-
OPT[j-1]	0	-	-	-	-

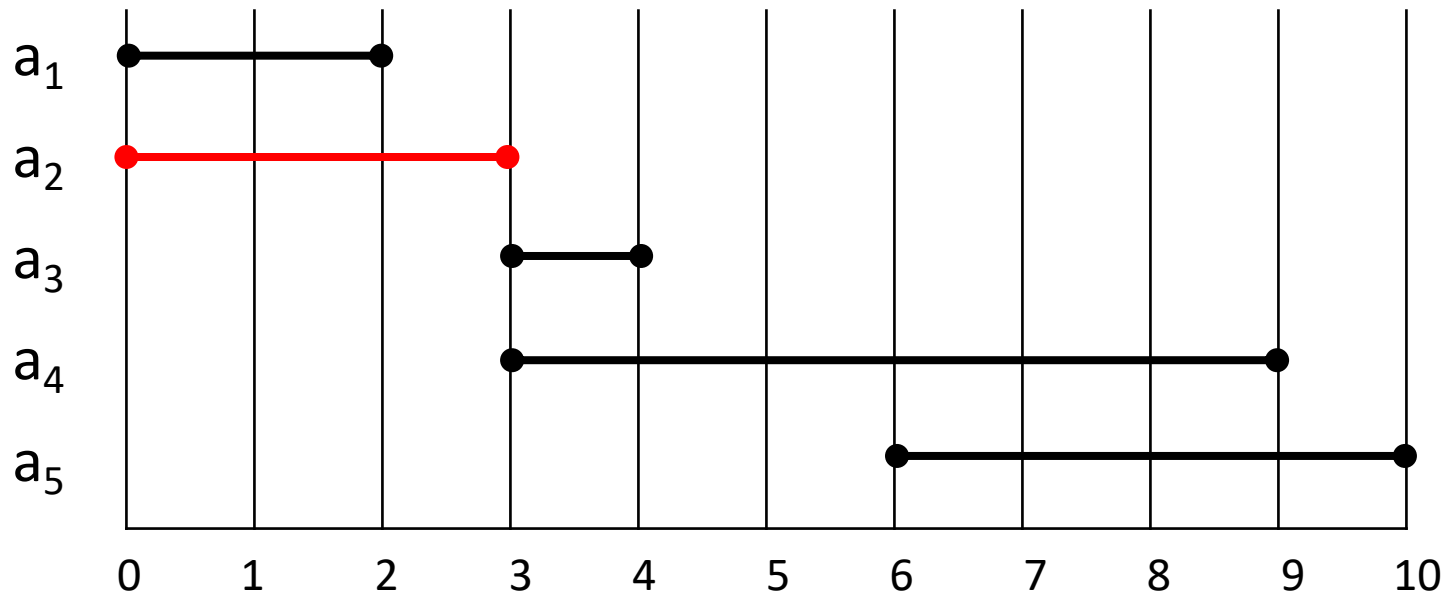
(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



Example: Computing solution

activity	1	2	3	4	5
predecessor	0	0	2	2	3
OPT[j]	2	3	-	-	-
$w_j + \text{OPT}[p(j)]$	2	3	-	-	-
OPT[j-1]	0	2	-	-	-

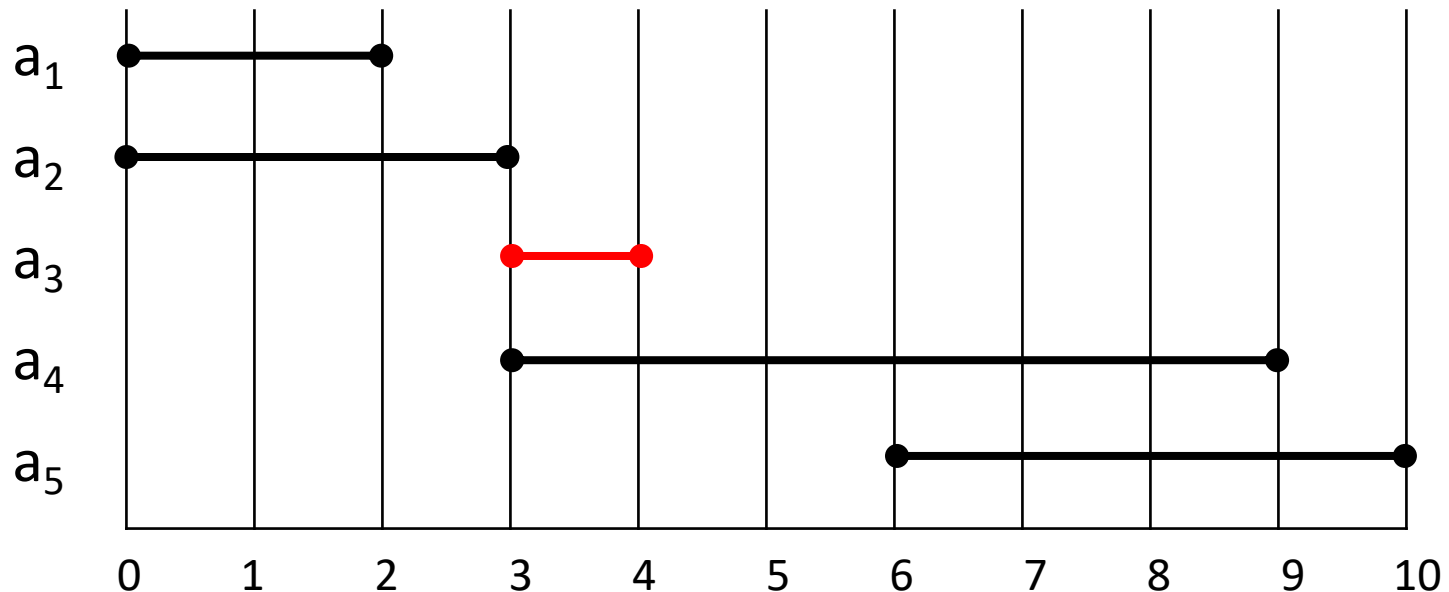
(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



Example: Computing solution

activity	1	2	3	4	5
predecessor	0	0	2	2	3
OPT[j]	2	3	4	-	-
$w_j + \text{OPT}[p(j)]$	2	3	4	-	-
OPT[j-1]	0	2	3	-	-

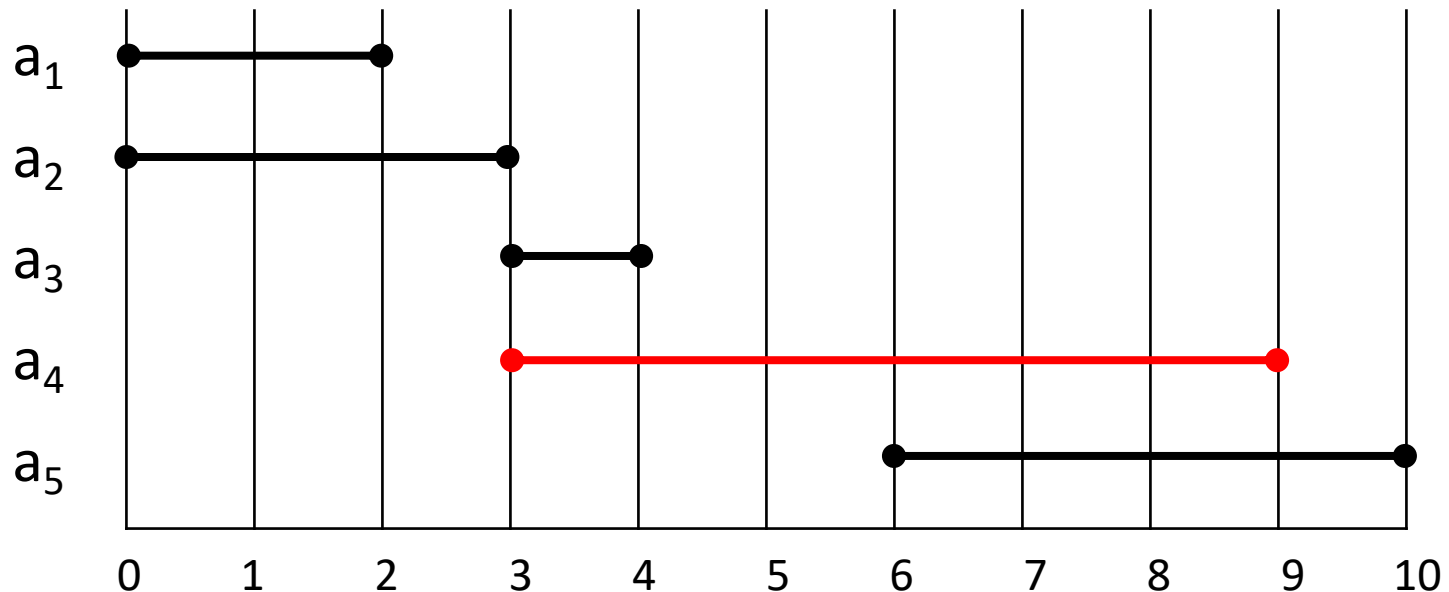
(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



Example: Computing solution

activity	1	2	3	4	5
predecessor	0	0	2	2	3
OPT[j]	2	3	4	9	-
$w_j + \text{OPT}[p(j)]$	2	3	4	9	-
OPT[j-1]	0	2	3	4	-

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

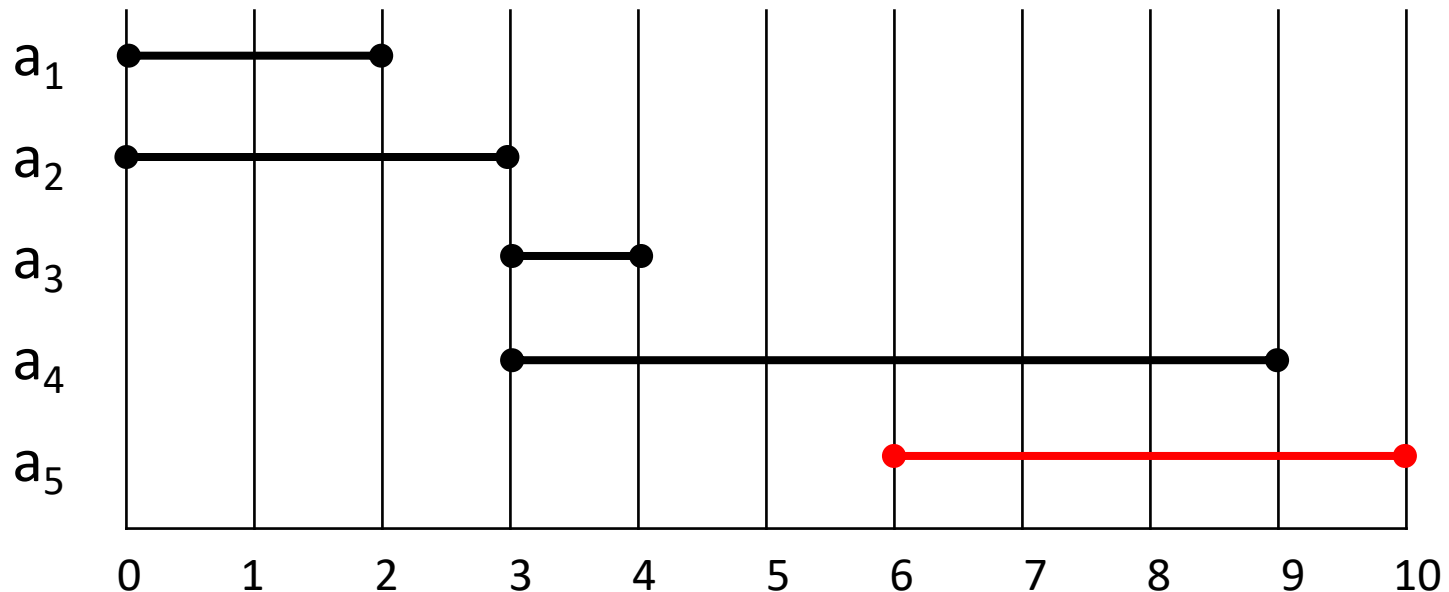


Example: Computing solution

activity	1	2	3	4	5
predecessor	0	0	2	2	3
OPT[j]	2	3	4	9	9
$w_j + \text{OPT}[p(j)]$	2	3	4	9	8
OPT[j-1]	0	2	3	4	9

Your solution

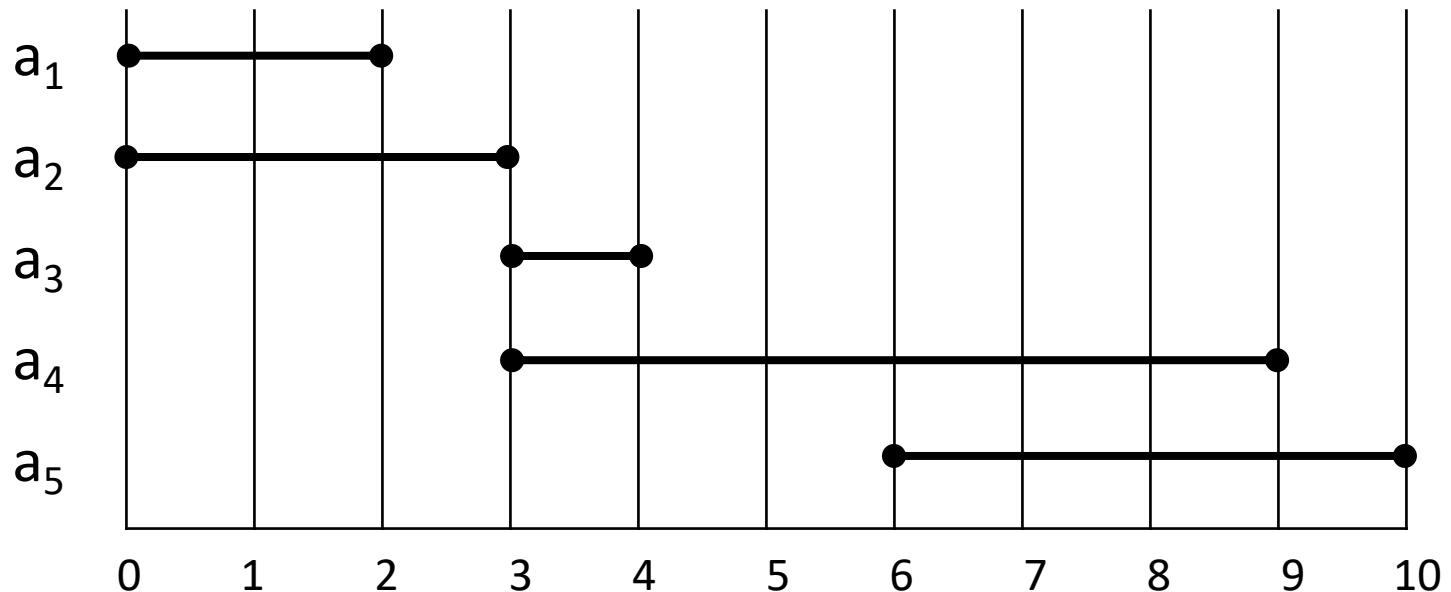
(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



Example: Reconstruction

activity	1	2	3	4	5
predecessor	0	0	2	2	3
OPT[j]	2	3	4	9	9
$w_j + \text{OPT}[p(j)]$	2	3	4	9	8
OPT[j-1]	0	2	3	4	9

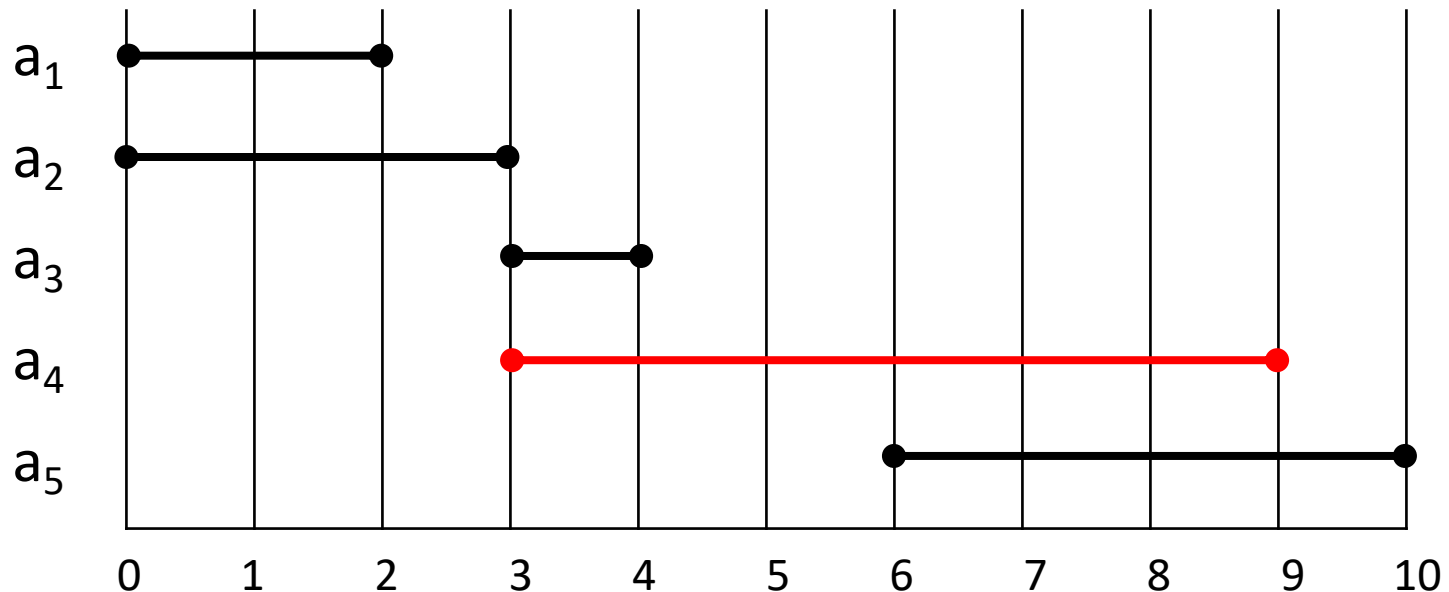
(1) Activities sorted by finishing time. (2) Weight equal to the length of activity.



Example: Reconstruction

activity	1	2	3	4	5
predecessor	0	0	2	2	3
OPT[j]	2	3	4	9	9
$w_j + \text{OPT}[p(j)]$	2	3	4	9	8
OPT[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
OPT[j]	2	3	4	9	9
$w_j + \text{OPT}[p(j)]$	2	3	4	9	8
OPT[j-1]	0	2	3	4	9

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

