# Fall 2024 - Analysis and Design of Algorithms

# Lecture 7: Greedy Algorithms

**Ahmed Kosba**

Department of Computer and Systems Engineering
Faculty of Engineering
Alexandria University

# Greedy Algorithms

- Greedy Strategy:
  - At any step, when we have multiple options to choose from, choose the best option at the moment, i.e., the option that offers the highest *immediate* benefit.
  - This certainly does not lead to optimal solutions to all problems.
    - We have seen several examples in the dynamic programming lectures where the greedy strategy fails.
- In this lecture, we will see several examples where the greedy strategy works.

# Outline

- Activity selection
- Fractional knapsack
- Huffman codes

# Activity Selection [CLRS 16.1]

Given a set of $n$ activities, $S = \{a_1, a_2, \ldots, a_n\}$, where each activity $a_i$ has a start time $s_i$ and a finish time $f_i$, find a maximum-size subset of mutually compatible activities.

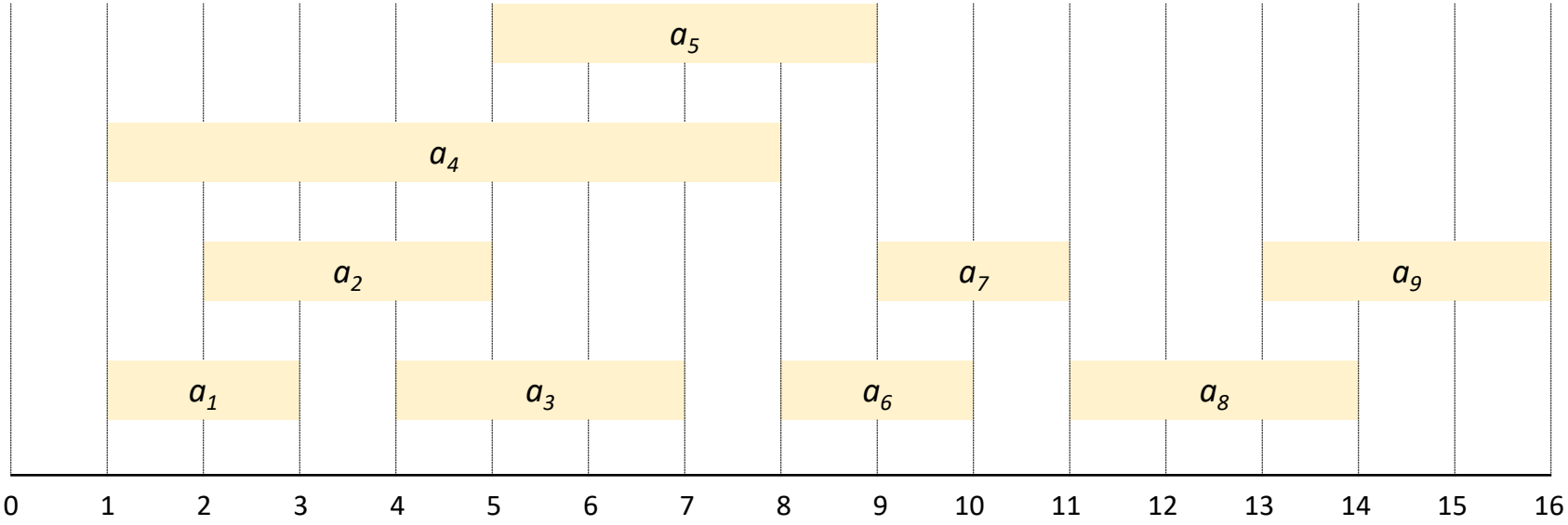
- Each activity $a_i$ takes place during the half-open interval $[s_i, f_i)$.
- Two activities $a_i$ and $a_j$ are compatible iff

  $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.
- Assume that the activities are already sorted by their finish times.

# Activity Selection

- Example [CLRS notes]:

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| $s_i$ | 1 | 2 | 4 | 1 | 5 | 8 | 9 | 11 | 13 |
| $f_i$ | 3 | 5 | 7 | 8 | 9 | 10 | 11 | 14 | 16 |

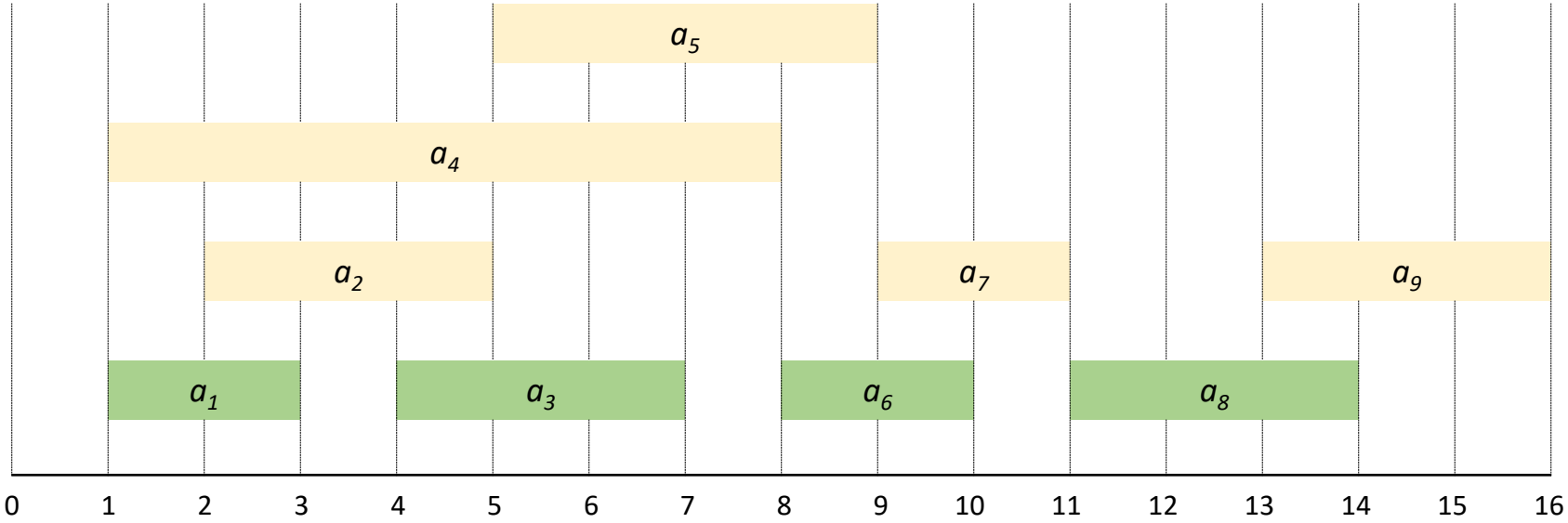


# Activity Selection

- Example [CLRS notes]:

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| $s_i$ | 1 | 2 | 4 | 1 | 5 | 8 | 9 | 11 | 13 |
| $f_i$ | 3 | 5 | 7 | 8 | 9 | 10 | 11 | 14 | 16 |

Solution 1

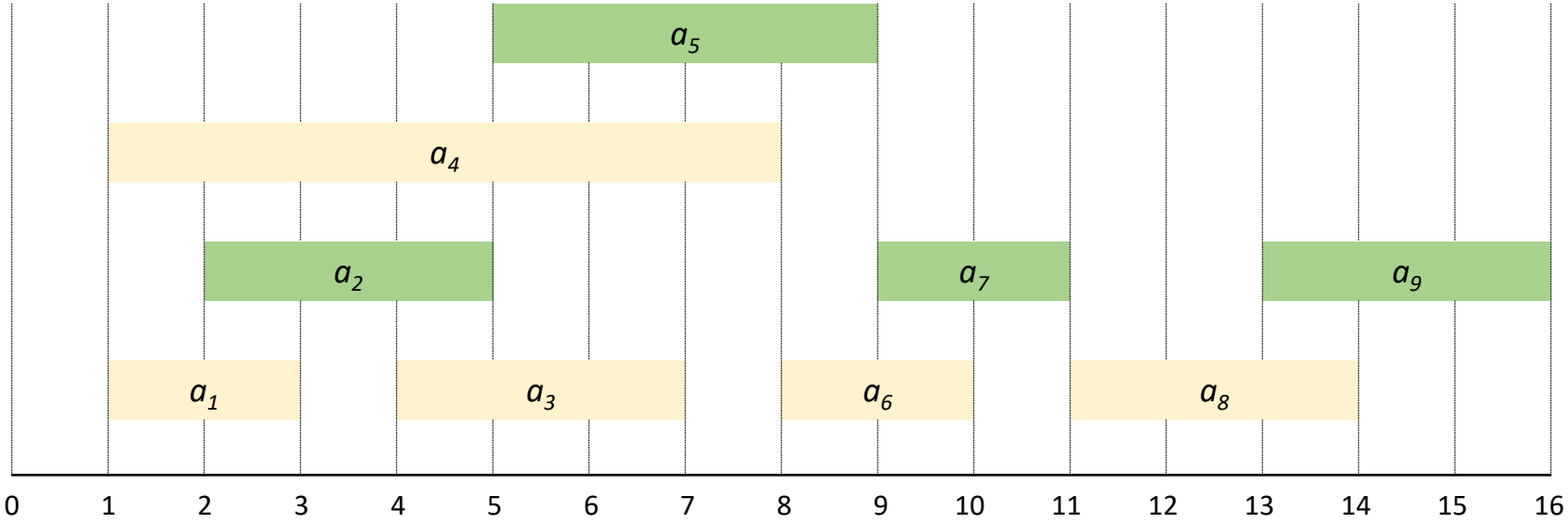


# Activity Selection

- Example [CLRS notes]:

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| $s_i$ | 1 | 2 | 4 | 1 | 5 | 8 | 9 | 11 | 13 |
| $f_i$ | 3 | 5 | 7 | 8 | 9 | 10 | 11 | 14 | 16 |

Solution 2



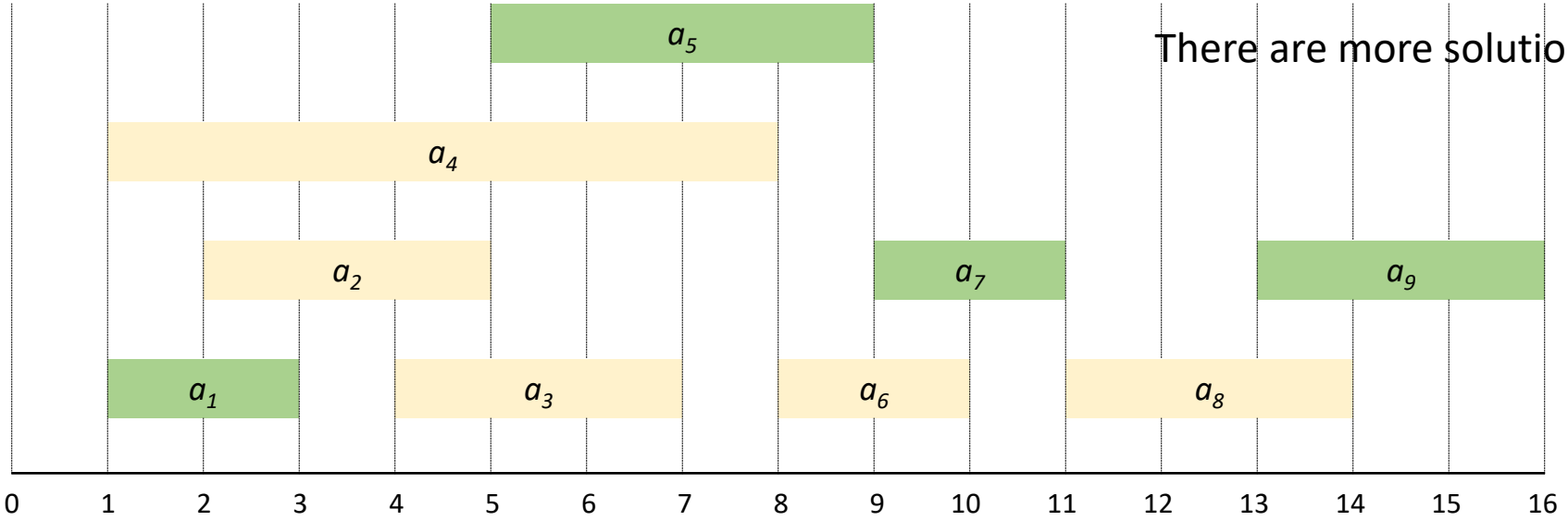
# Activity Selection

- Example [CLRS notes]:

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| $s_i$ | 1 | 2 | 4 | 1 | 5 | 8 | 9 | 11 | 13 |
| $f_i$ | 3 | 5 | 7 | 8 | 9 | 10 | 11 | 14 | 16 |

Solution 3

There are more solutions.

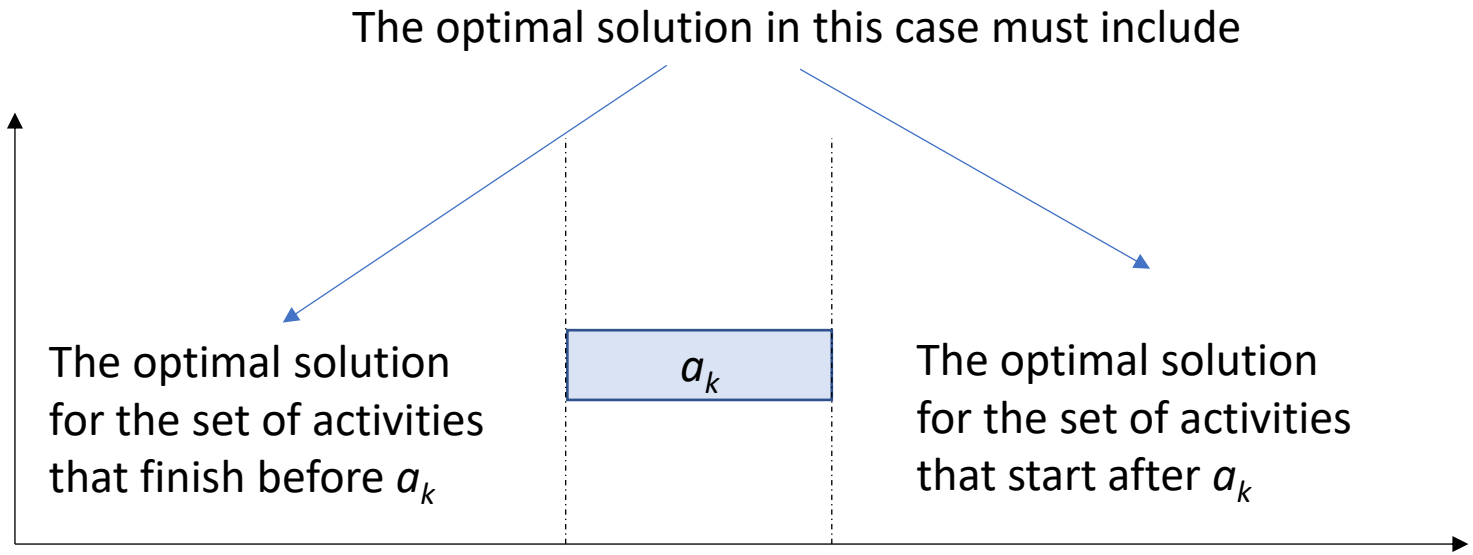


# Activity Selection

- To solve the previous problem, we can use dynamic programming.
- However, we will discover a simpler *greedy* algorithm.
- We will start by the DP solution as a review, then we will discuss the greedy one.
  - Note: The DP solution presented next is **not** the most efficient DP solution to the problem.
  - The goal is to illustrate the difference between DP and greedy algorithms.
  - We use the formalization used in CLRS.

# Activity Selection

- Examining the structure of the problem
  - Suppose some activity $a_k$ is part of the optimal solution for the set S, i.e., $a_k$ belongs to the maximum subset of mutually compatible activities.




As in the examples covered previously, we don't know which $a_k$ belong to the optimal solution, so we have to consider all options when writing the recursive definition.

# Activity Selection

- The problem has optimal substructure.
  - The optimal solution of the original problem includes optimal solutions to the subproblems.

- Following the DP paradigm of the previous lecture
  - Next step: find a recursive definition

# Activity Selection

Some notation:

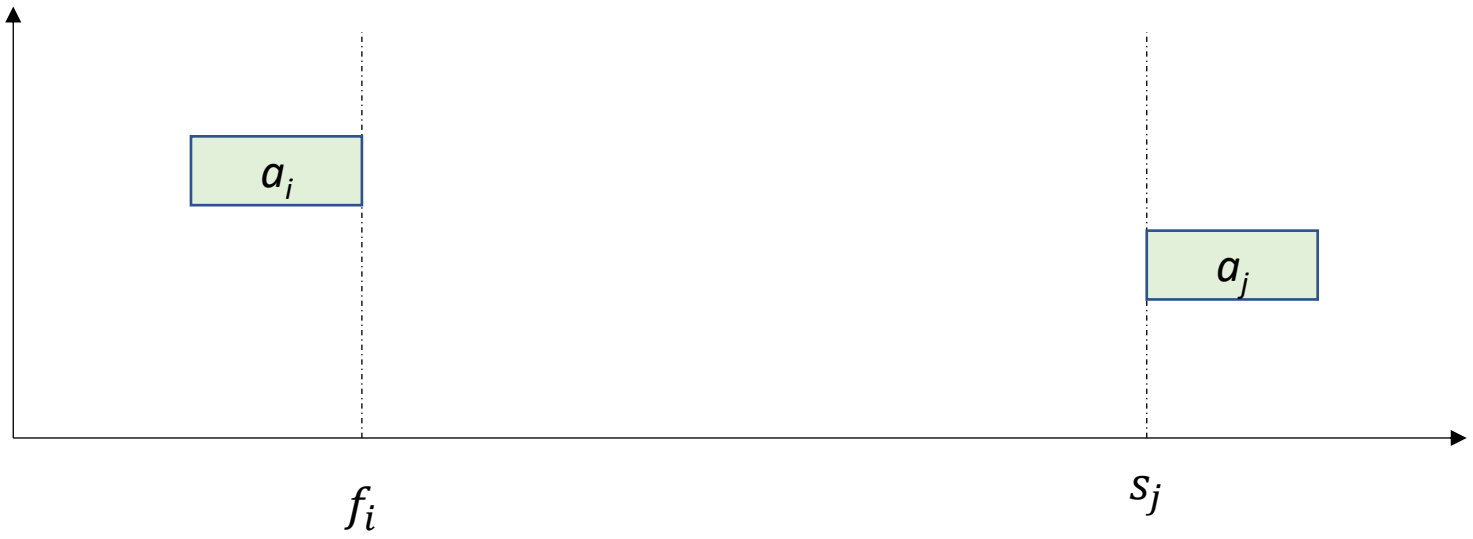
- $S_{ij}$ is the set of activities that start after activity $a_i$ finishes and that finish before activity $a_j$ starts.

$$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$$

- $A_{ij}$ is the maximum-size subset of mutually compatible activities in $S_{ij}$.

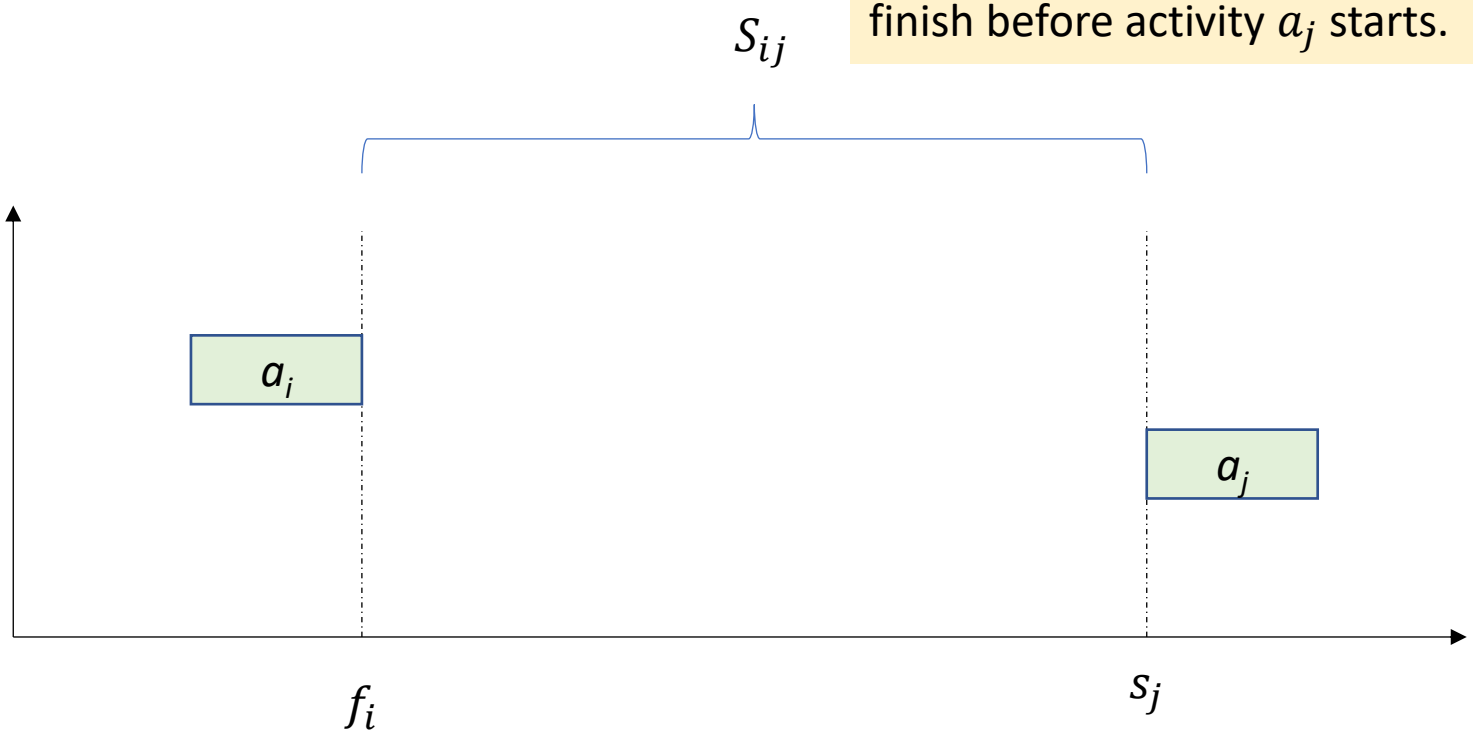
- $|A_{ij}|$ is the size of the set $A_{ij}$

# Activity Selection



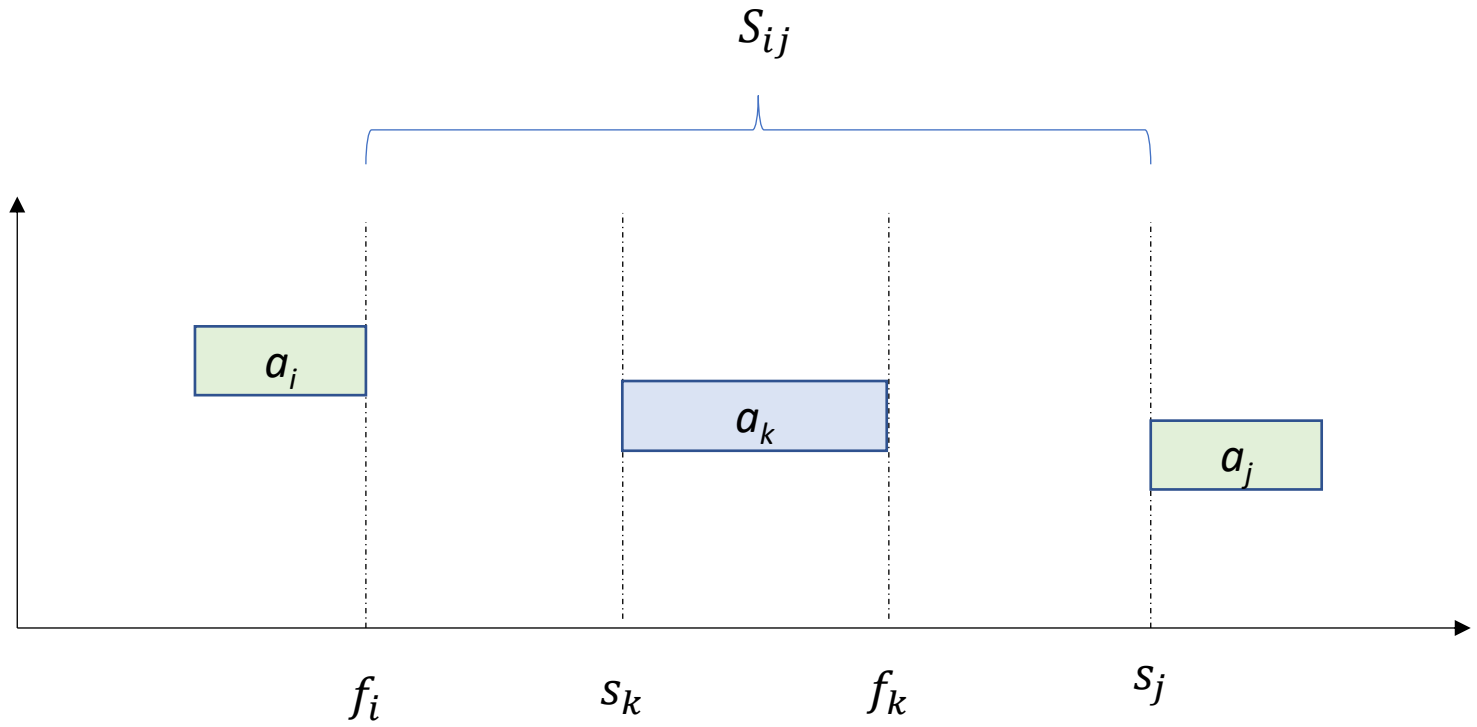
# Activity Selection

$S_{ij}$ is the set of activities that start after activity $a_i$ finishes and that finish before activity $a_j$ starts.



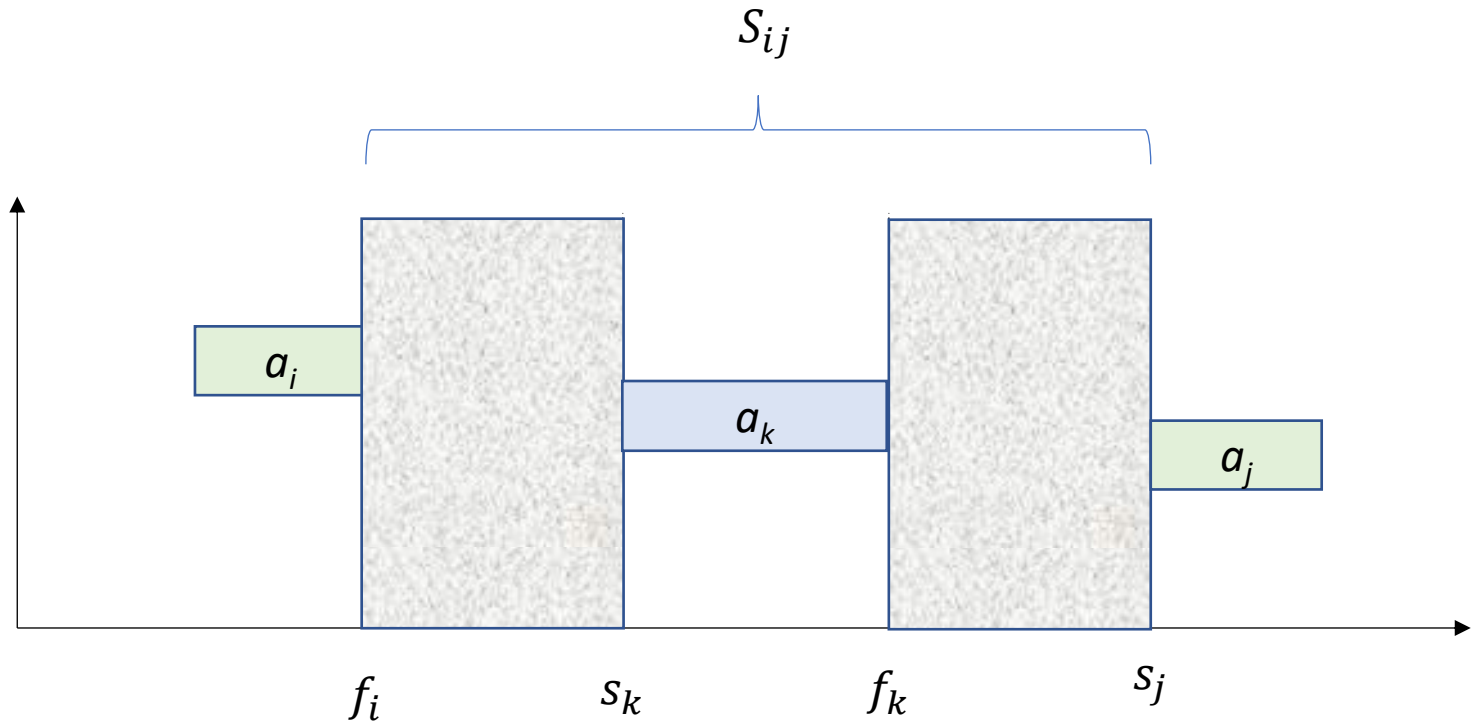

All activities that start and finish in this interval belong to $S_{ij}$ .

# Activity Selection



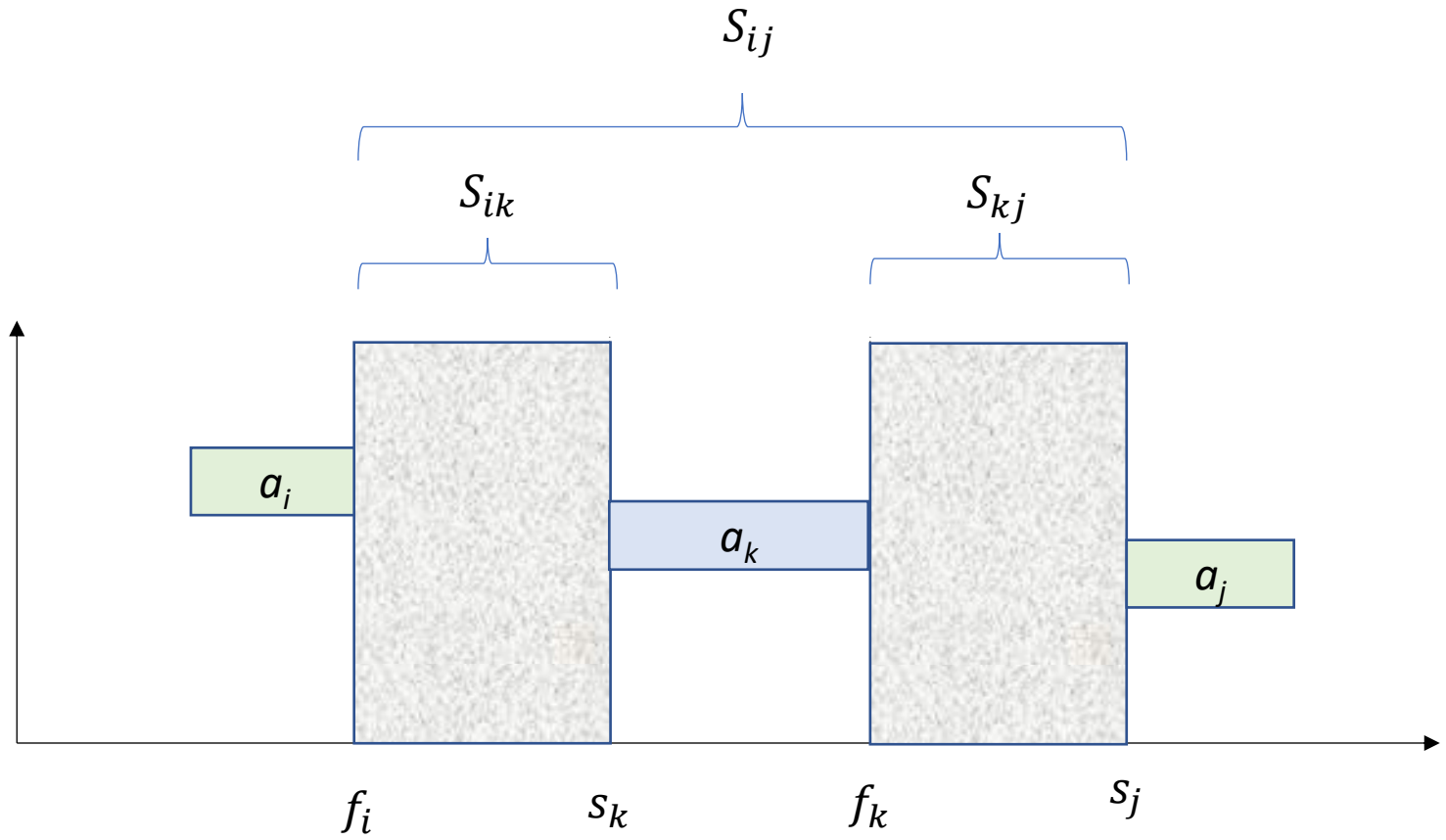

Suppose the optimal solution for $S_{ij}$ includes activity $a_k$.

# Activity Selection




Suppose the optimal solution for $S_{ij}$ includes activity $a_k$, then it must include the optimal solutions for the shaded intervals as well.

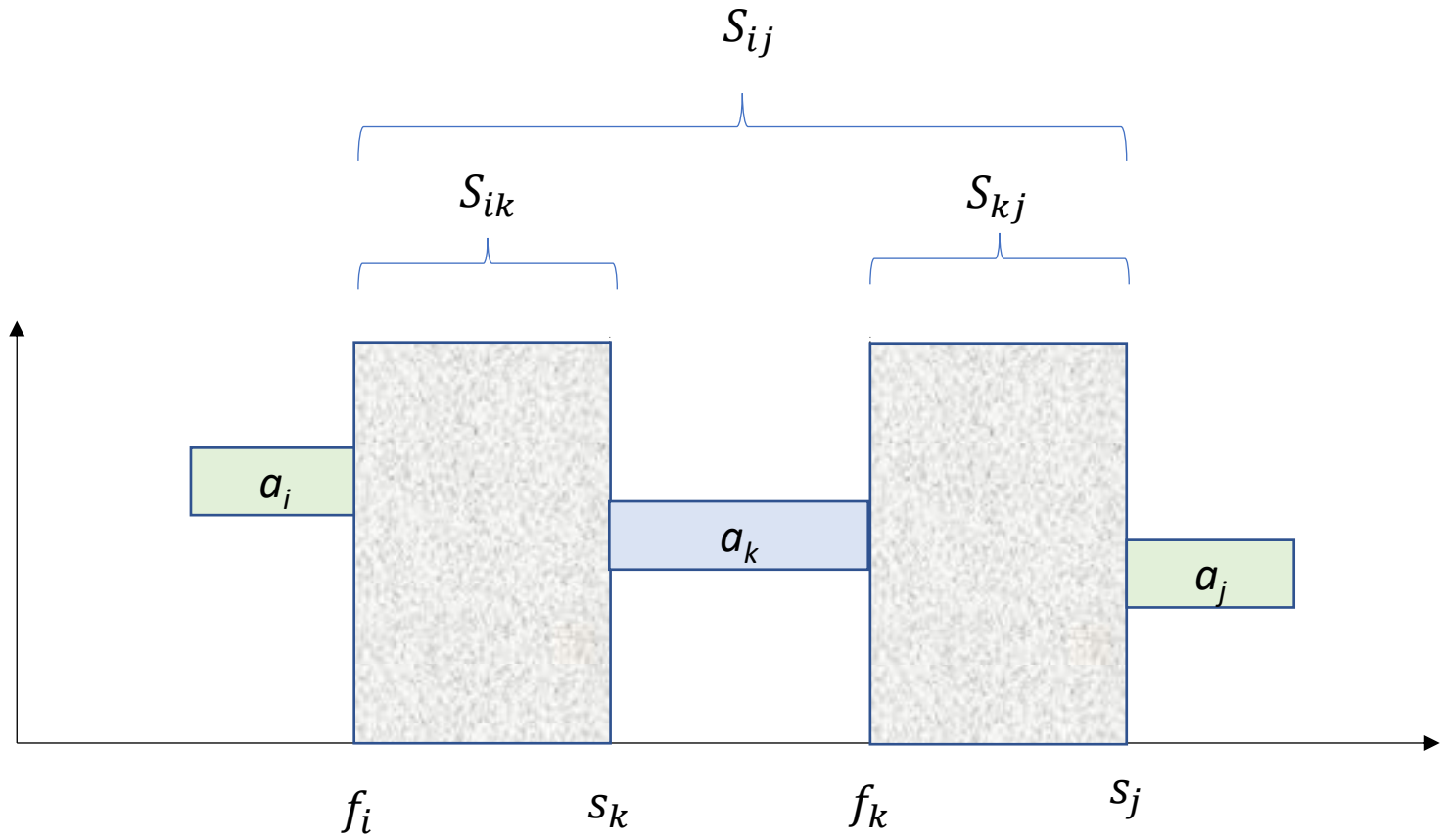
# Activity Selection




Suppose the optimal solution for $S_{ij}$ includes activity $a_k$, then it must include the optimal solutions for the shaded intervals as well.

# Activity Selection

$$|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$$




Suppose the optimal solution for $S_{ij}$ includes activity $a_k$, then it must include the optimal solutions for the shaded intervals as well.

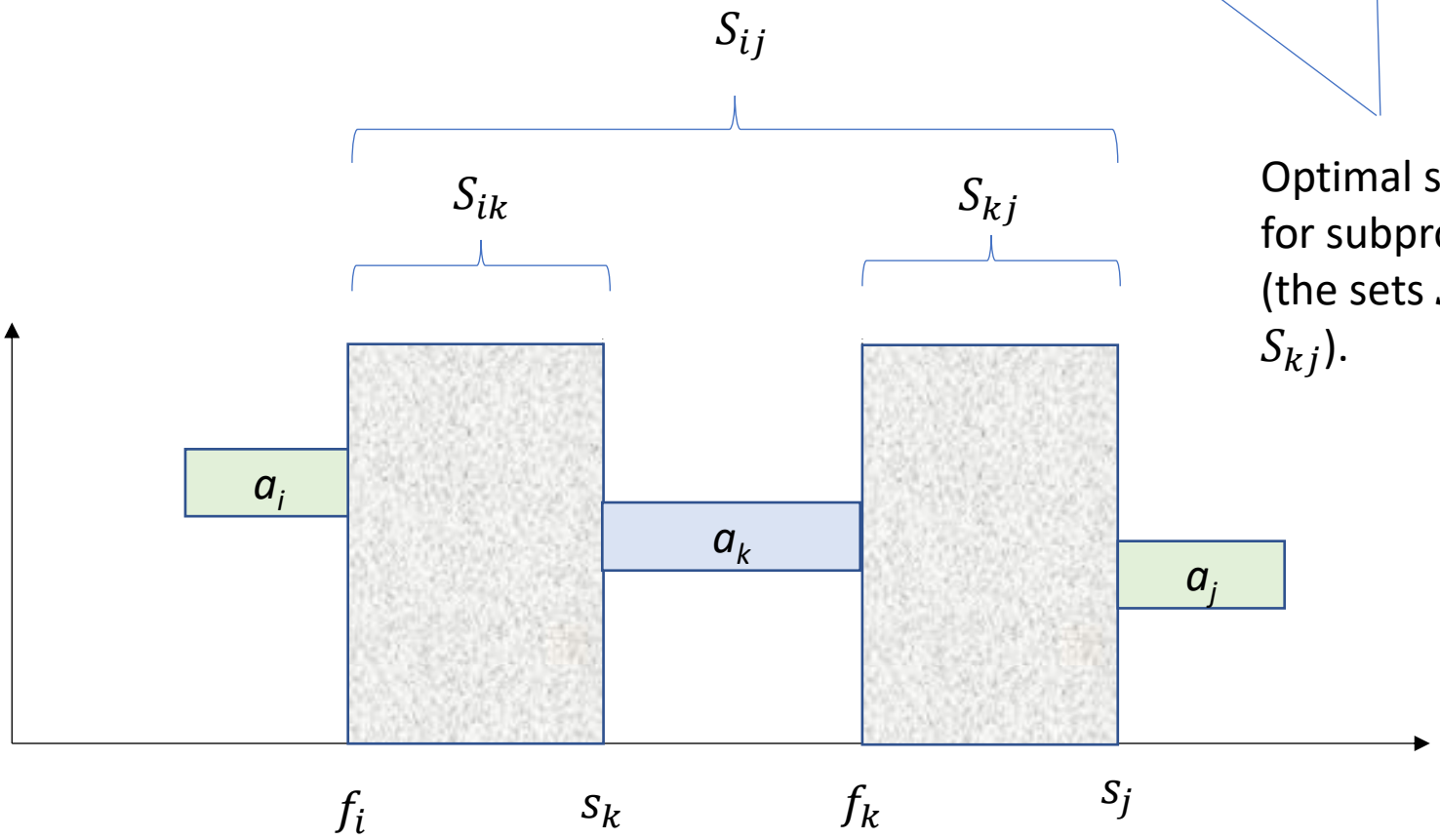
# Activity Selection

Optimal solution for the set $S_{ij}$

*Counting* $a_k$

$$|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$$

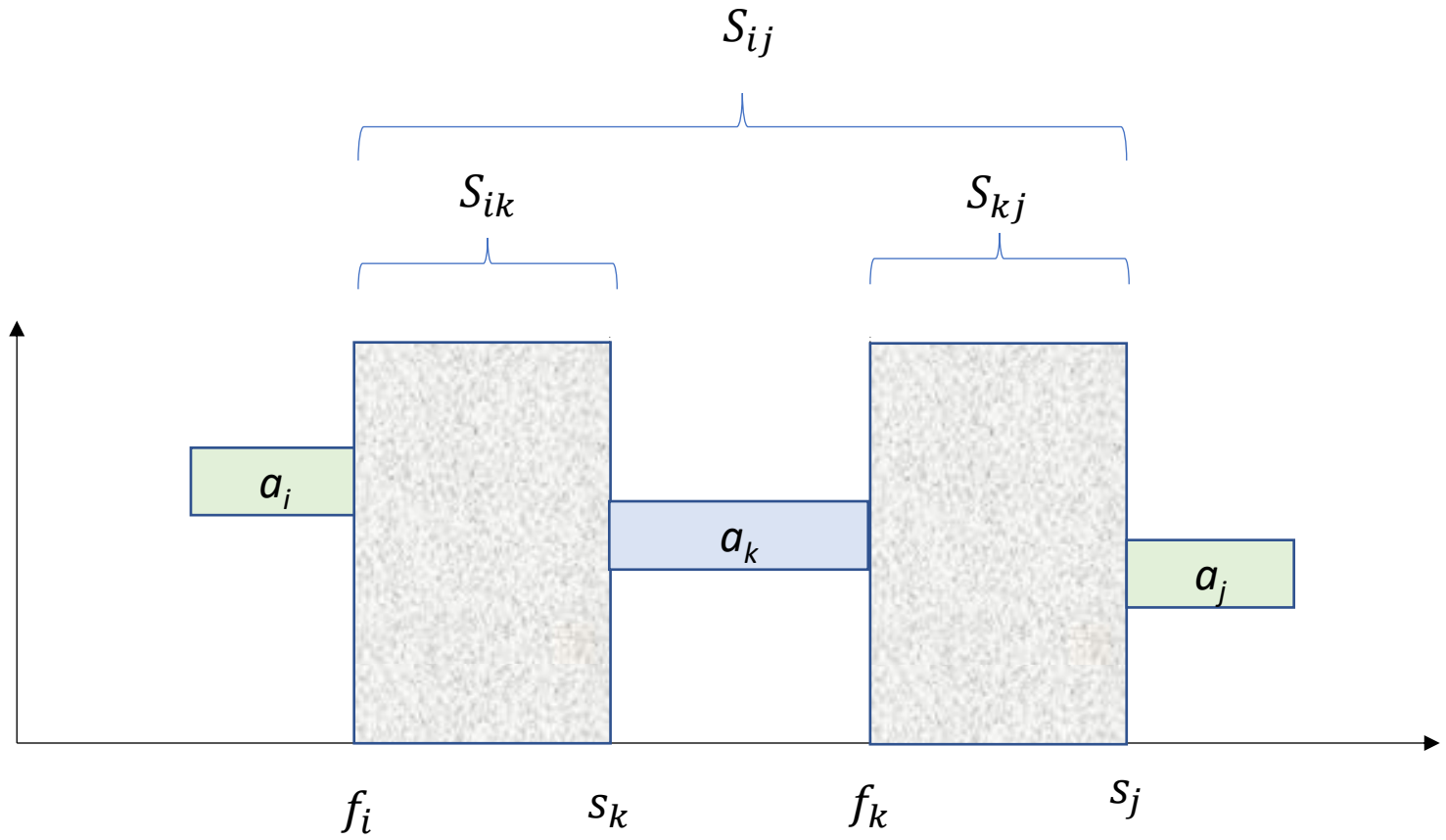
Optimal solution for subproblems (the sets $S_{ik}$ and $S_{kj}$).




Suppose the optimal solution for $S_{ij}$ includes activity $a_k$, then it must include the optimal solutions for the shaded intervals as well.

# Activity Selection

Check your understanding
Is $S_{ij} = S_{ik} \cup S_{kj} \cup \{a_k\}$ ?

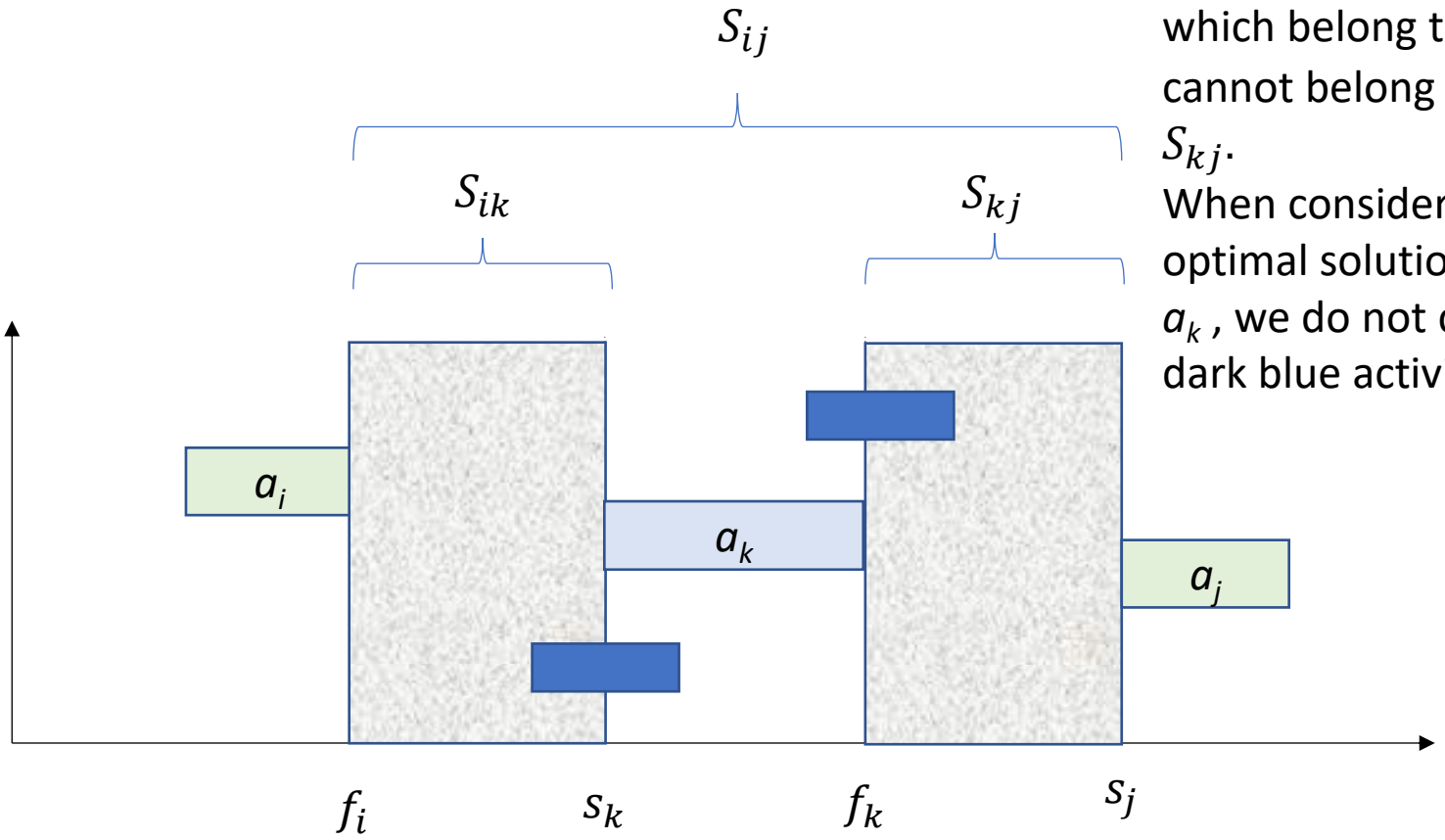



Suppose the optimal solution for $S_{ij}$ includes activity $a_k$, then it must include the optimal solutions for the shaded intervals as well.

# Activity Selection

Exercise:
Is $S_{ij} = S_{ik} \cup S_{kj} \cup \{a_k\}$ ?

Not necessarily. There could be activities like the dark blue ones below which belong to $S_{ij}$, but cannot belong to $S_{ik} \cup S_{kj}$.
When considering an optimal solution that has $a_k$ , we do not consider the dark blue activities.




Suppose the optimal solution for $S_{ij}$ includes activity $a_k$, then it must include the optimal solutions for the shaded intervals as well.

# Activity Selection – DP Solution

- Recursive definition:

  As before, we don't know which k would lead to the optimal solution, so we loop over all $a_k$ in $S_{ij}$ and select what leads to the maximum.

Let $c[i,j]$ be the size of the optimal solution for $S_{ij}$

$$c[i,j] = 0 \quad if \quad S_{ij} = \emptyset$$

$$c[i,j] = \max_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} \quad if \quad S_{ij} \neq \emptyset$$

  - As in the DP lecture, we could implement this by either a bottom-up approach or top-down approach with memoization.
  - However, is this the best we can do?

# Activity Selection – DP Solution

- Notes
  - The previous DP formalization can be simplified.
  - Furthermore, there is a more efficient DP solution than the previous one.

  - Exercise: Can you find a simplified recursive definition that won't require solving two subproblems?

# Activity Selection
# A simpler solution

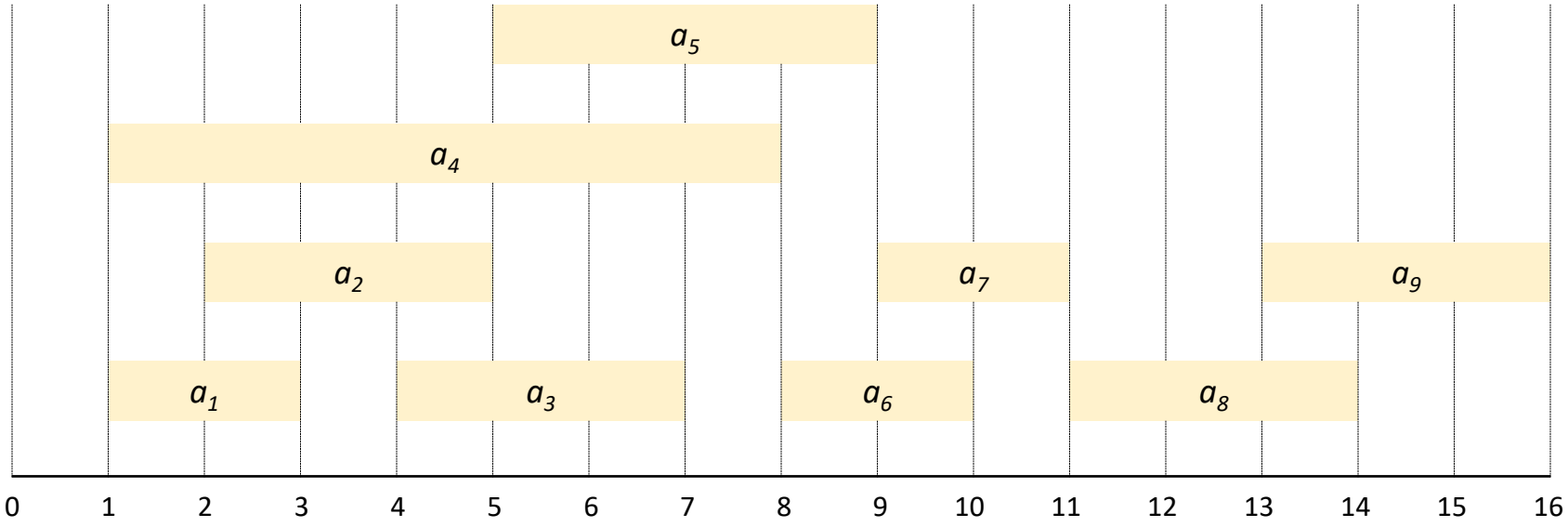
- Greedy strategy:
  - Instead of solving all the subproblems for each possible $a_k$, choose the activity $a_k$ in a greedy way (before solving any of the subproblems!)
  - In this context, a possible greedy choice is to select an activity that would leave more space for the other activities.
    - The greedy choice we will use is based on the earliest finish time.
    - We will show other ways that do not work.
  - Note that the choice is made without considering the future choices, i.e., before solving any of the next subproblems.

# Activity Selection – Greedy Solution

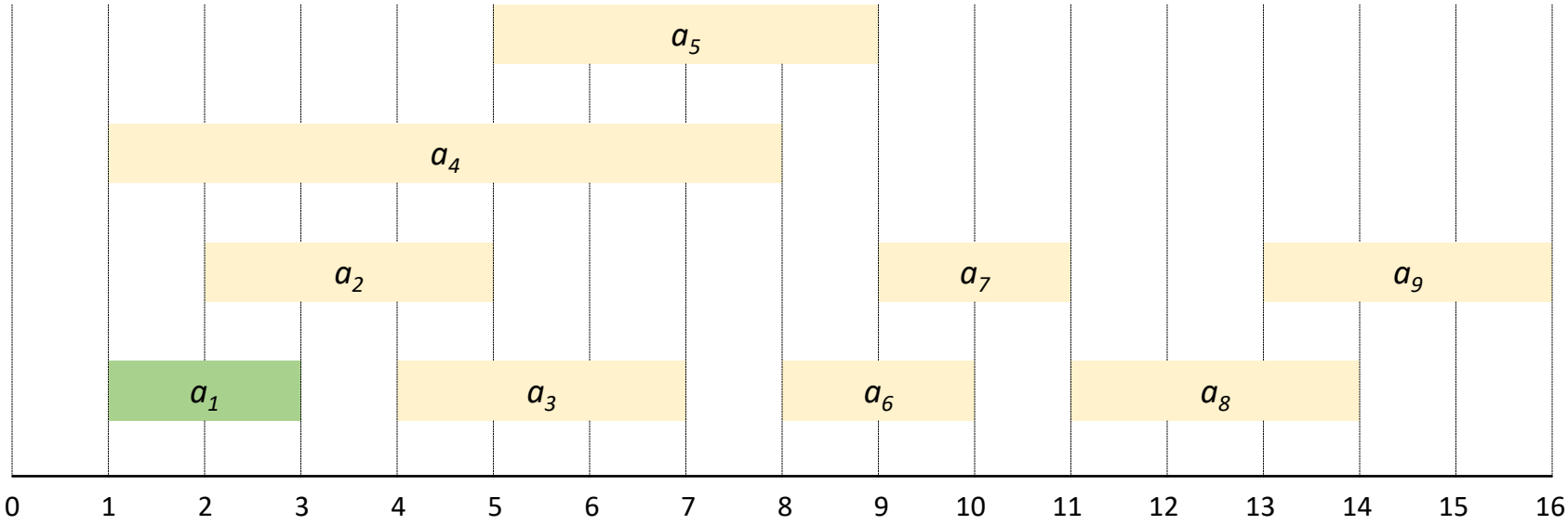
- Greedy strategy:
  - **Choose the first activity to finish.**
  - As the activities are sorted by the finish time, this means that we select the first activity in the interval we are considering.
  - When the first activity is selected for the optimal solution, note that only one subproblem remains.

# Activity Selection – Greedy Solution

Back to our example:

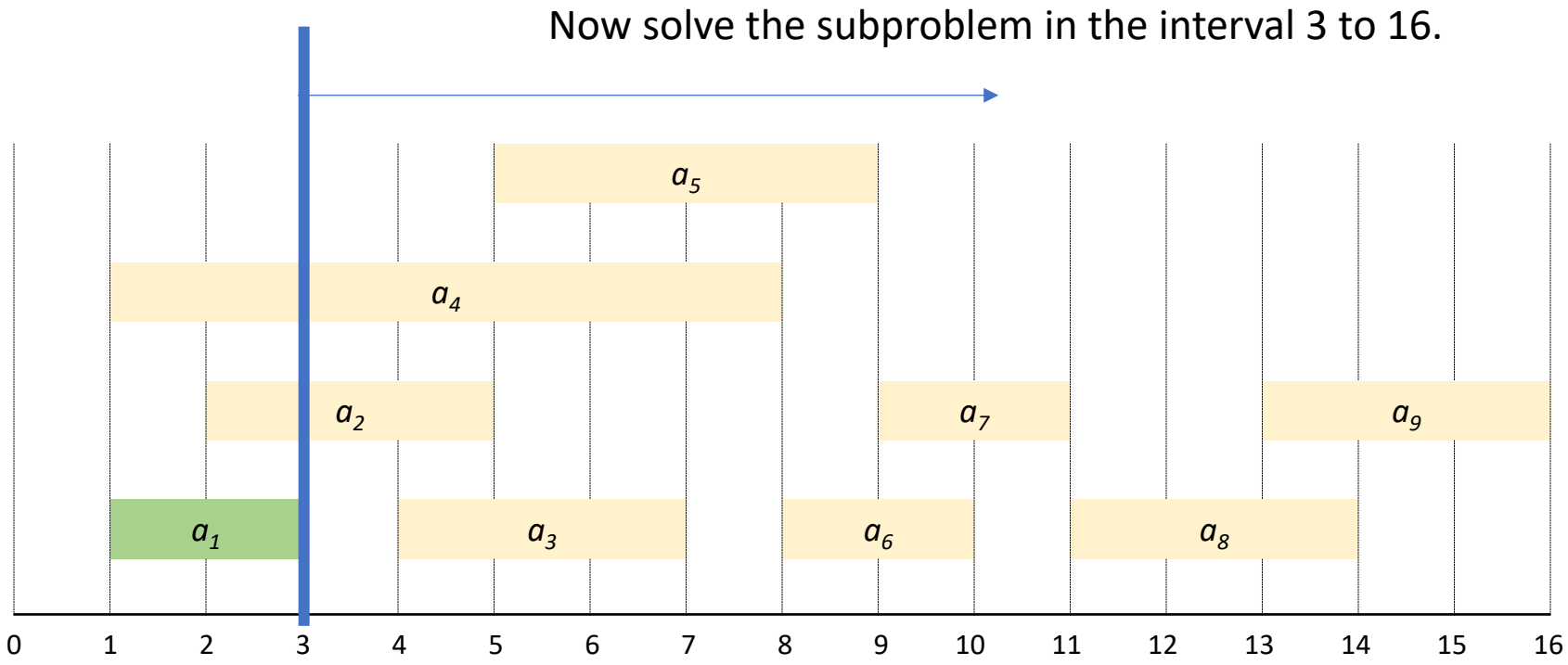


# Activity Selection – Greedy Solution

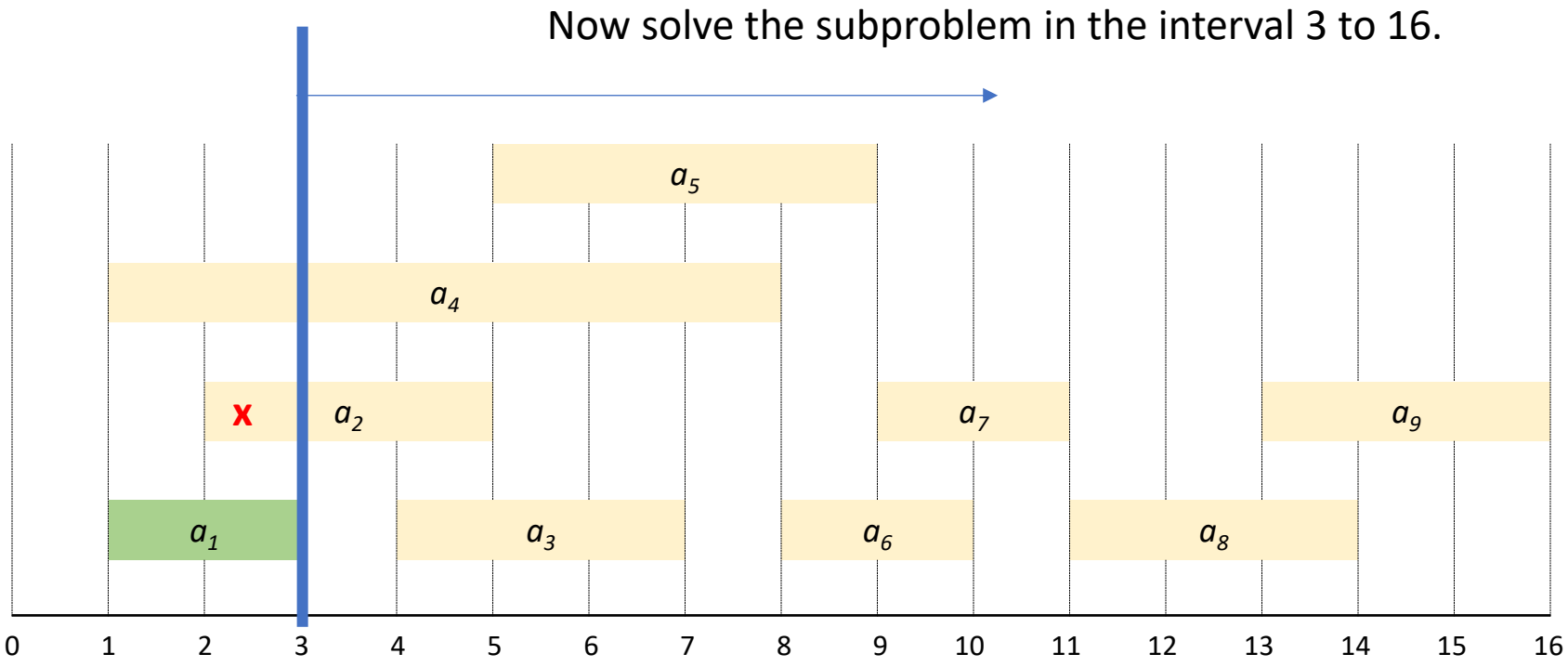



1- Select $a_1$ as it has the earliest finish time.

# Activity Selection – Greedy Solution

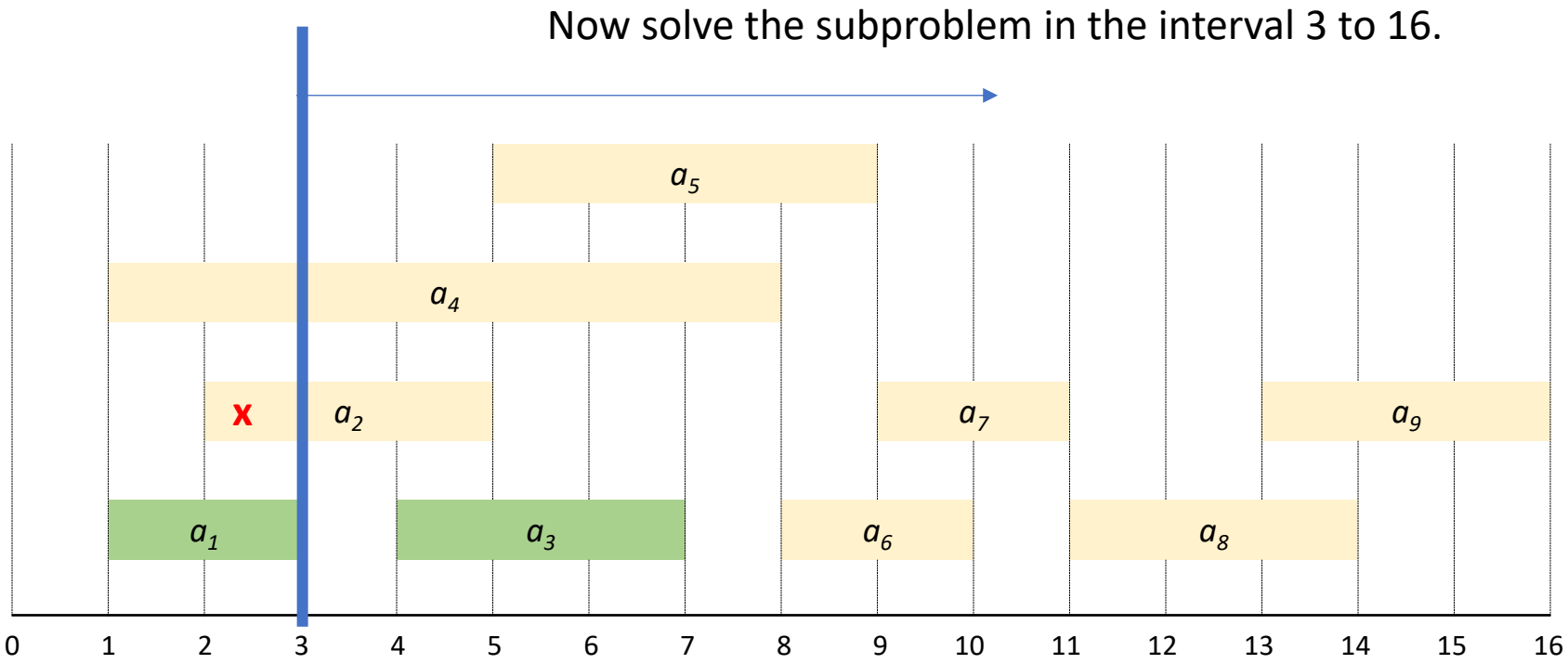


# Activity Selection – Greedy Solution



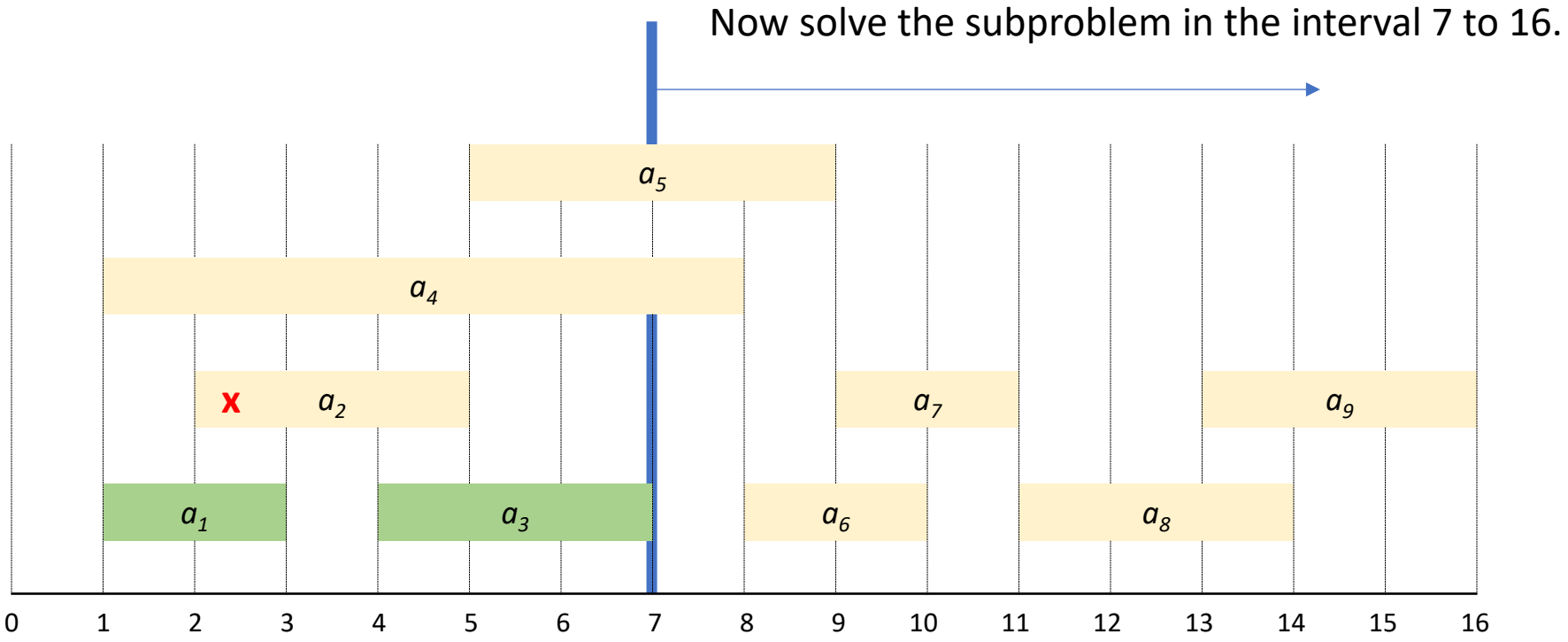

2- The next activity with earliest finish time is $a_2$, but we will skip it, as its start time < 3.

# Activity Selection – Greedy Solution

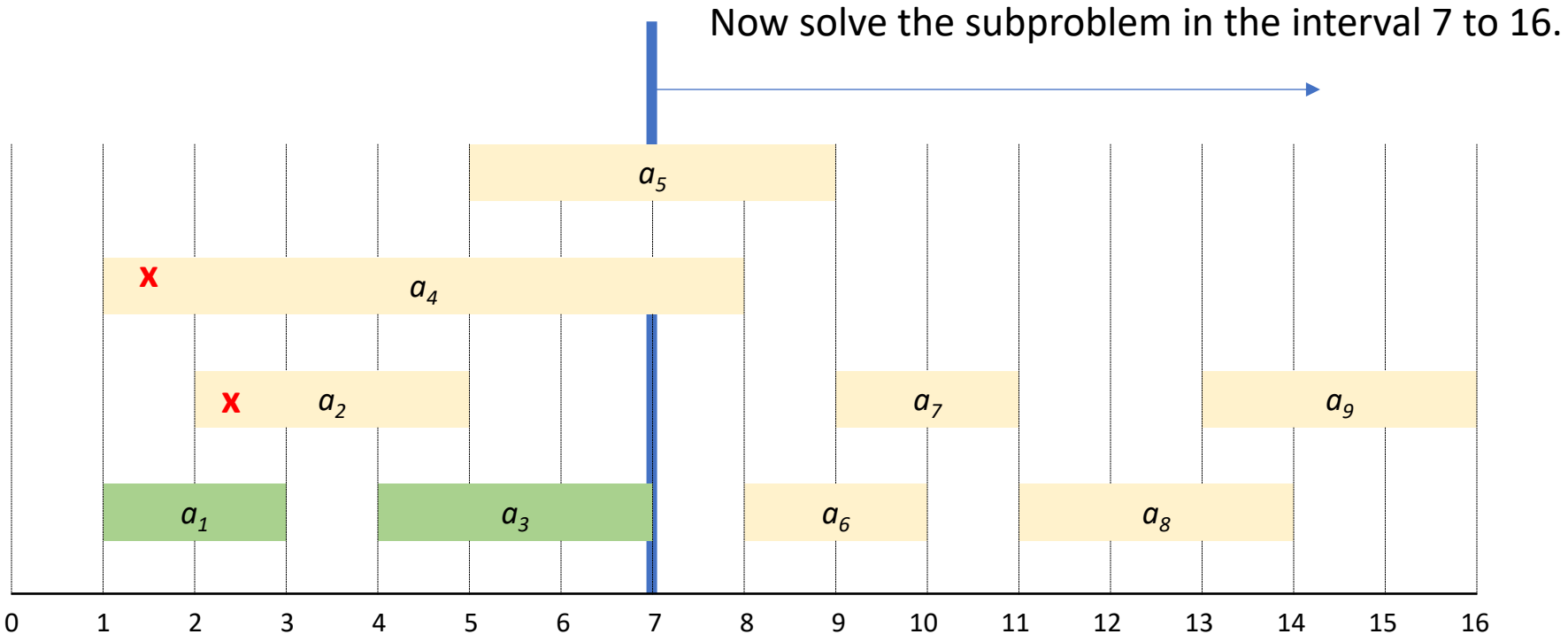



3- Select $a_3$ as it has the next earliest finish time, and it start time >= 3

# Activity Selection – Greedy Solution

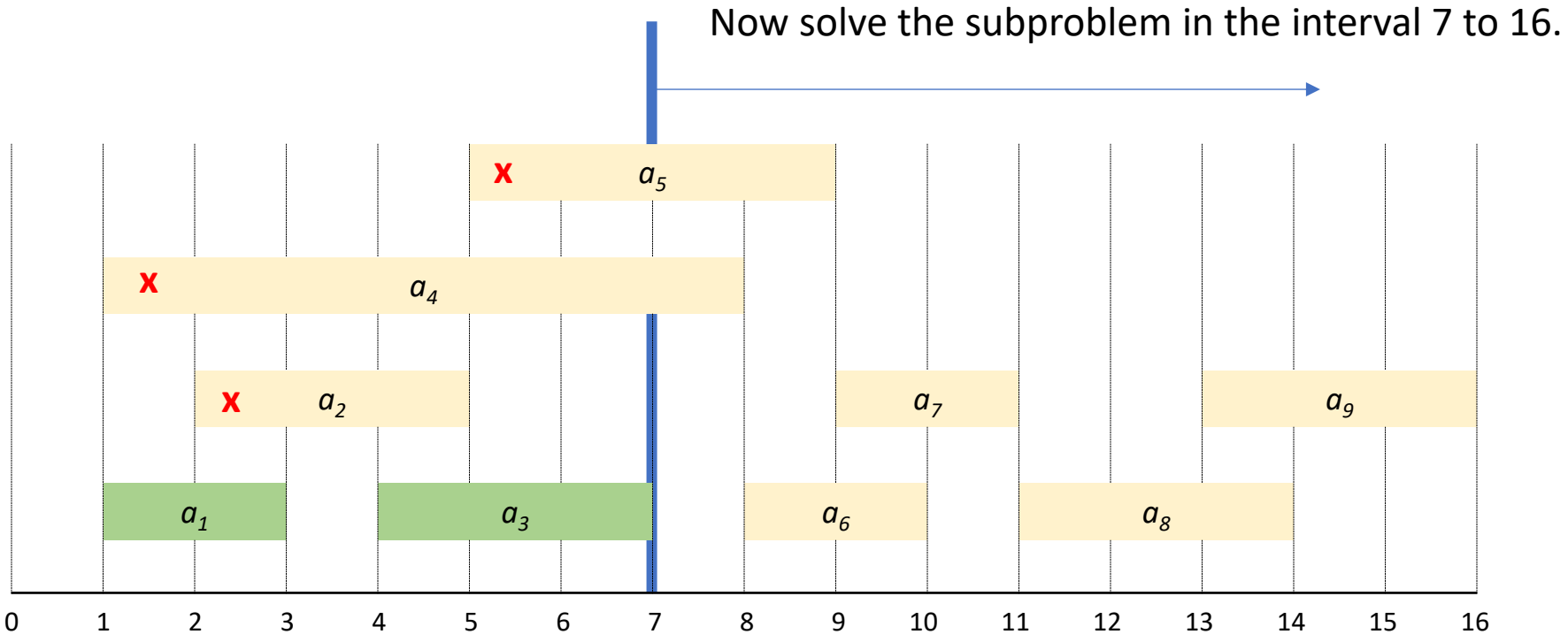


# Activity Selection – Greedy Solution



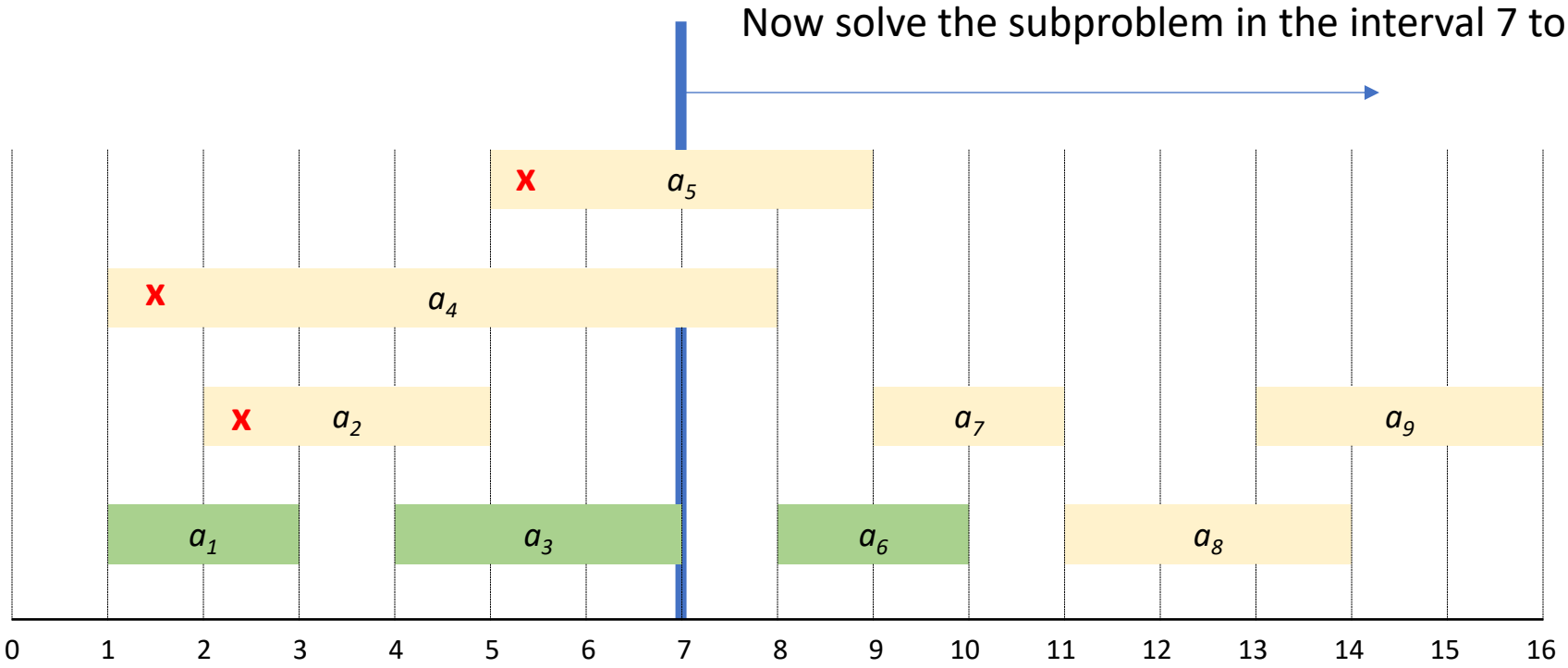

4- The next activity with earliest finish time is $a_4$, but we will skip it, as its start time < 7.

# Activity Selection – Greedy Solution




5- The next activity with earliest finish time is $a_5$, but we will skip it, as its start time < 7.

# Activity Selection – Greedy Solution




6- Select $a_6$ as it has the next earliest finish time, and it start time >= 7

And so on.

# Other options for the greedy choice?

- In the previous example, we used "the earliest finish" criteria.
  - We will prove shortly that it will lead to an optimal solution.

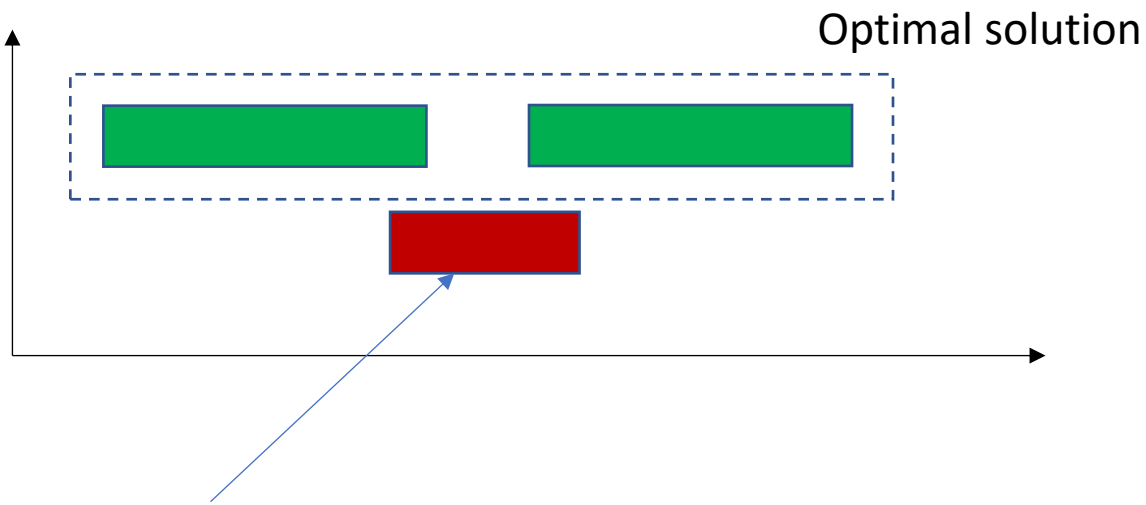
- This does not mean that any greedy criteria will work. For example, think about the following criteria:
  - Choose the shortest activity first.
  - Choose the activity which has the minimum number of conflicts.

Both of them won't work.

# Other Greedy Criteria
## Choosing shortest activity first?

- Choosing the shortest activity first will not necessarily lead to an optimal solution.

- Consider this counter example:

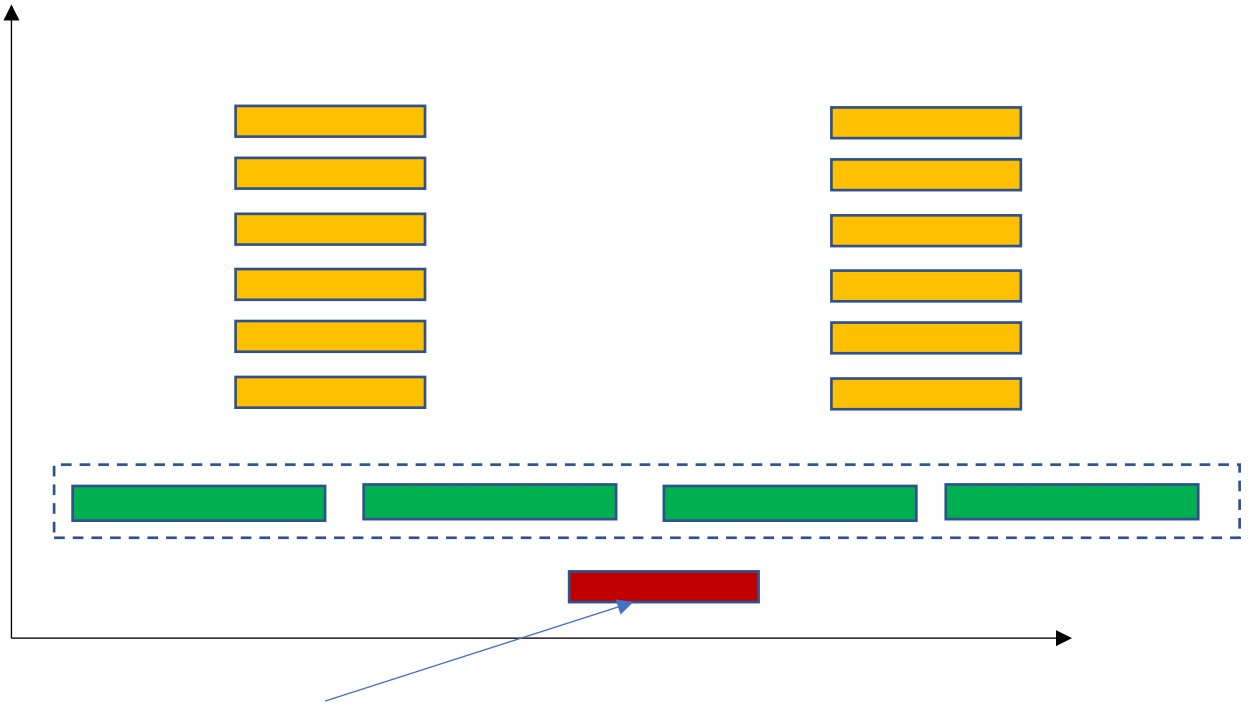



Choosing the activity with the shortest duration will not lead to an optimal solution.

# Other Greedy Criteria

## Choosing activity with minimum number of conflicts?

- This will not necessarily lead to an optimal solution as well. Counter example:




Choosing the activity with minimum number of conflicts will not lead to optimal solution.

# Greedy Criteria

- This implies that not any criteria that just makes sense would work.

- We need to prove that the strategy will lead to an optimal solution.

- In greedy algorithms, we prove two properties:
  - The greedy-choice property
    - The greedy choice will lead to an optimal solution.
  - The optimal substructure property
    - Combining the greedy choice with the optimal solution of the subproblem will lead to an optimal solution of the problem.

# Activity Selection – Greedy Solution

- As we now consider one subproblem only, the notation can be simplified.
- Let $S_k$ be the set of activities that start after activity $a_k$ finishes.

$$S_k = \{a_i \in S : s_i \geq f_k\}$$

- Optimal substructure property:
  - If $a_1$ is part of the optimal solution, then the optimal solution must contain the optimal solution of $S_1$. (This can be proven by a simple contradiction)
- Next, we will prove that the greedy choice will lead to an optimal solution.

# Activity Selection – Greedy Solution Proof of Optimality [CLRS]

**Theorem:**

If $S_k$ is nonempty and $a_m$ has the earliest finish time in $S_k$, then $a_m$ is included in some optimal solution for $S_k$.

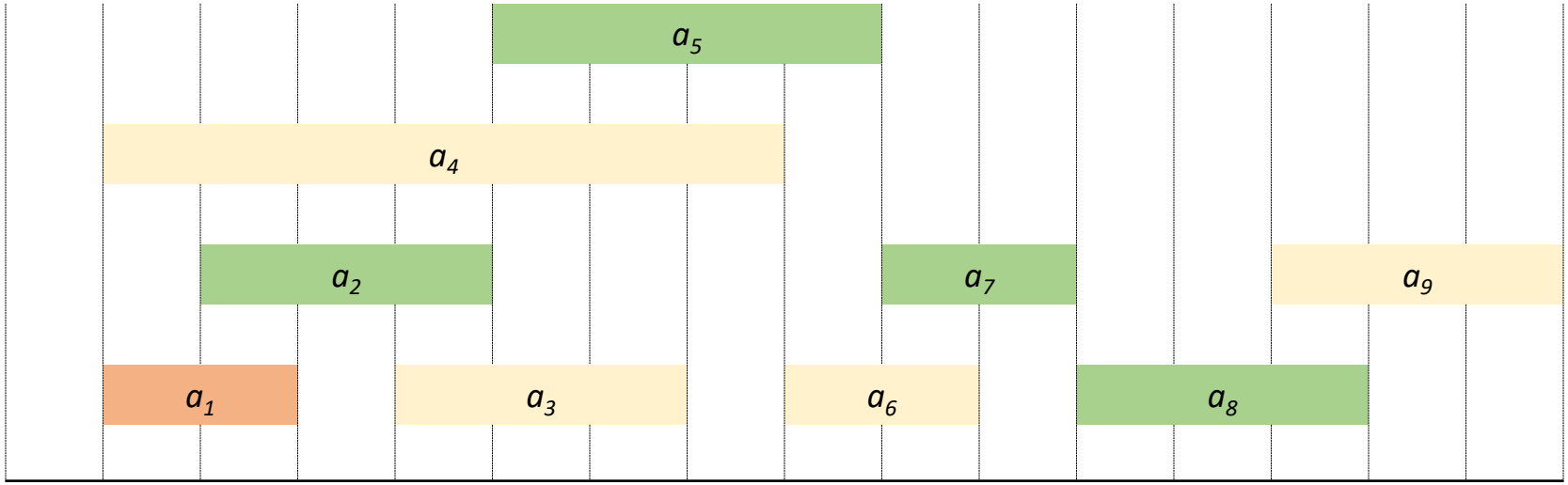
Proof:

- Let $A_k$ be an optimal solution to $S_k$, i.e., $A_k$ is a maximum-size subset of mutually compatible activities in $S_k$.
- Assume $a_j$ is the activity with the earliest finish time in $A_k$.
- If $a_j = a_m$ , done.
- If $a_j \neq a_m$
  - Let $A'_k = A_k - \{a_j\} \cup \{a_m\}$ // include $a_m$ instead of $a_j$
  - The activities in $A'_k$ must be non-overlapping, as the activities of the optimal solution $A_k$ must be non-overlapping, and $f_m \leq f_j$.
  - Therefore $|A'_k| = |A_k|$ = the size of the maximum-size subset of mutually compatible activities in $S_k$, i.e, $a_m$ is part of a maximum-size subset.

# Activity Selection – Greedy Solution Proof of Optimality (Intuition)

Proof illustration via an example:

Note: you cannot use examples to prove a claim. This is for illustration.

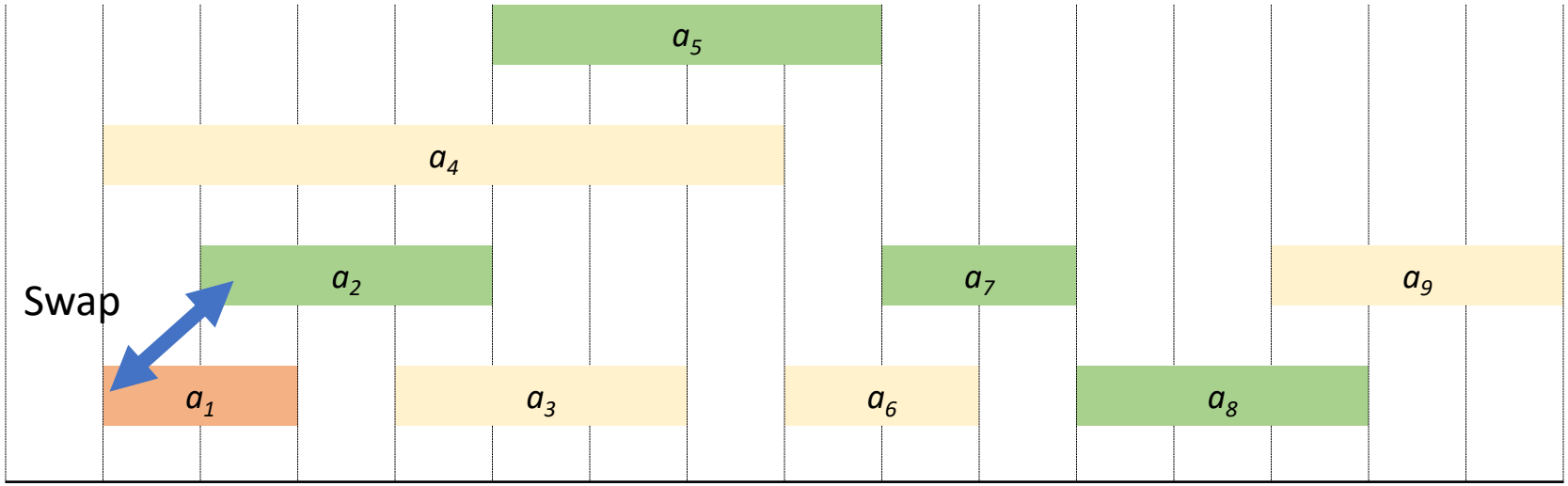



S = {$a_1$, …, $a_9$}

Given an optimal solution A = {$a_2$, $a_5$, $a_7$, $a_8$}, show that we can construct an optimal solution using the activity with the earliest time $a_1$

# Activity Selection – Greedy Solution Proof of Optimality (Intuition)

Proof illustration via an example:



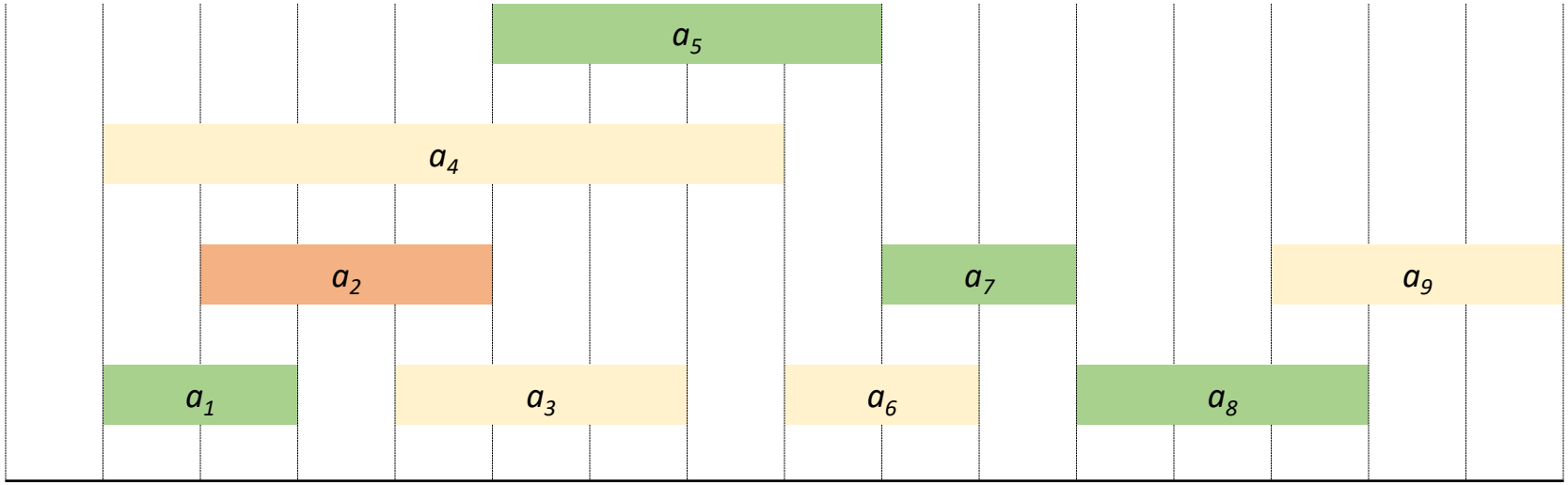

S = {$a_1$, …, $a_9$}

Given an optimal solution A = {$a_2$, $a_5$, $a_7$, $a_8$}, show that we can construct an optimal solution using the activity with the earliest time $a_1$

As $a_2$ has the earliest time in A, let $A'_1 = A_1 - \{a_2\} \cup \{a_1\} = \{a_1, a_5, a_7, a_8\}$

# Activity Selection – Greedy Solution Proof of Optimality (Intuition)

Proof illustration via an example:




$A'_1 = \{a_1, a_5, a_7, a_8\}$ is an optimal solution as well.

Note that because $a_1$ finishes earliest, it was possible to swap it with $a_2$ without making overlaps with any other activity. **This is generalized in the proof.**

# Activity Selection – Greedy Solution Implementation

- Assuming the activities are sorted by the finish times already, the running time of the greedy solution will be $\theta(n)$.

- If the activities are not sorted, the cost will be $O(n \lg n)$.

# Activity Selection – Greedy Solution Implementation

- Iterative implementation [CLRS]

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5      if s[m] ≥ f[k]
6          A = A ∪ {a_m}
7          k = m
8  return A
```

Add the first activity $a_1$ to A
Recall that the activities are sorted by finish times.

Find the first activity that starts after f[k]

# Activity Selection – Greedy Solution Implementation

- Recursive implementation [CLRS]

  Assume having a dummy activity $a_0$ with $f_0 = 0$

  First call: Recursive-Activity-Selector(s, f, 0, n)

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]      // find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

Greedy choice

One subproblem to solve

# The Greedy Strategy – Summary [CLRS]

- Express the optimization problem as a problem in which we can make a choice, then solve one subproblem.
- Show that there is always an optimal solution that includes the greedy choice.
- Show that combining the optimal solution of the subproblem with the greedy choice leads to an optimal solution.

# The Greedy Strategy

- Greedy-choice property:
  - A globally optimal solution is reached by making locally optimal (greedy) choices (without considering solutions to subproblems).
  - In dynamic programming, the situation was different. We make a choice after finding the solutions to subproblems.
    - This is why the solutions of DP can be built in a bottom-up manner.
  - The greedy approach usually works in a top-down manner, as the subproblem is solved after making a choice.

# The Greedy Strategy

- Optimal substructure
  - Recall: A problem has optimal substructure if the optimal solution incorporates optimal solutions to subproblems.
  - In the context of greedy algorithms, we show that combining
    - the greedy choice
    - the optimal solution to the subproblem that we have to solve after making the greedy choice

  will lead to an optimal solution to the problem.

# Knapsack Problem

- To see the difference between greedy algorithms and dynamic programming, we will revisit the knapsack problem covered previously.

- We discussed dynamic programming solutions to 0-1 knapsack and unbounded knapsack last time.

- In this lecture, we will see a variant of this problem that can be solved by a greedy algorithm.

# Knapsack Problem

Given a knapsack (bag) that can hold a weight of at most W, and n items to pick from.

Each item has weight $w_i$ kg and is worth $v_i$ dollars.

How to choose items to put in the knapsack, such that the total value of the items in the knapsack is maximized?

Different versions of this problem:

- Knapsack with repetition (Unbounded Knapsack)
  - There is no limit on the quantity of each item. An item can appear 0, 1 or more times.
- Knapsack without repetition (0-1 Knapsack)
  - Each item can appear at most once.
- Fractional Knapsack
  - We can take a fraction of an item.

# Knapsack Example

Example [CLRS]:

W = 50

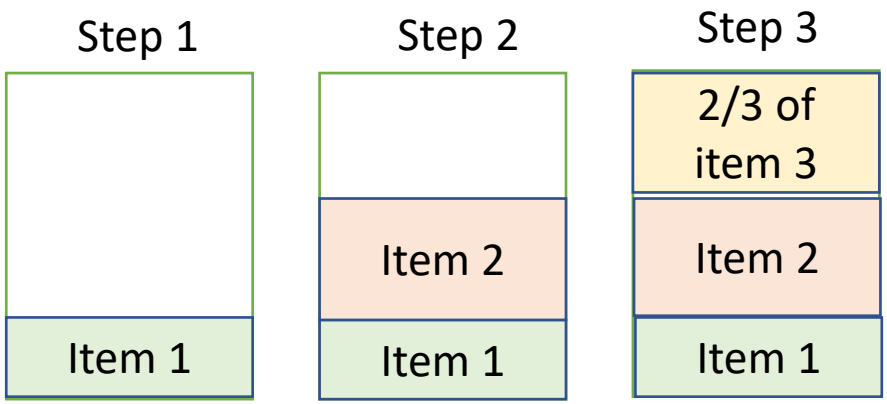
| i | $w_i$ | $v_i$ |
|---|---|---|
| 1 | 10 | 60 |
| 2 | 20 | 100 |
| 3 | 30 | 120 |

- For 0-1 knapsack, the optimal solution will be items 2 and 3.
  - The max total value will be 220.
- What if we are allowed to take fractions of items?
  - The optimal solution will be items 1 and 2, and $\frac{2}{3}$ of item 3.
  - The max total value will be 240

# Fractional Knapsack

- Can be solved using a greedy strategy.
- Compute the value per kg of each item and sort the items accordingly.

| i | $w_i$ | $v_i$ | $v_i / w_i$ |
|---|---|---|---|
| 1 | 10 | 60 | 6 |
| 2 | 20 | 100 | 5 |
| 3 | 30 | 120 | 4 |

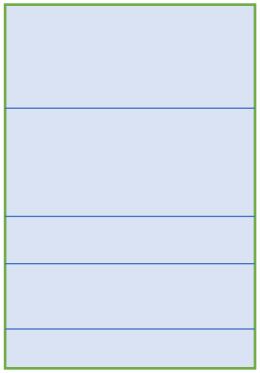



- Take as much as possible from the item with the highest value per kg, until its supply ends, or the knapsack is full.
- If there is room in the knapsack, move to the next item with the 2nd highest value per kg, and repeat.

# Fractional Knapsack – Correctness Proof Sketch [informal]

- Prove that the greedy choice can lead to an optimal solution, then prove the optimal substructure.
- We will discuss the greedy choice first.
- Assume we have an optimal solution, in which we did not include as much as possible from item 1 (the item that has the highest $v_i / w_i$)

Assume this is the optimal solution with total value V



Knapsack

Item 1 has the highest value per kg, and some of it has not been included in the knapsack

Item 1

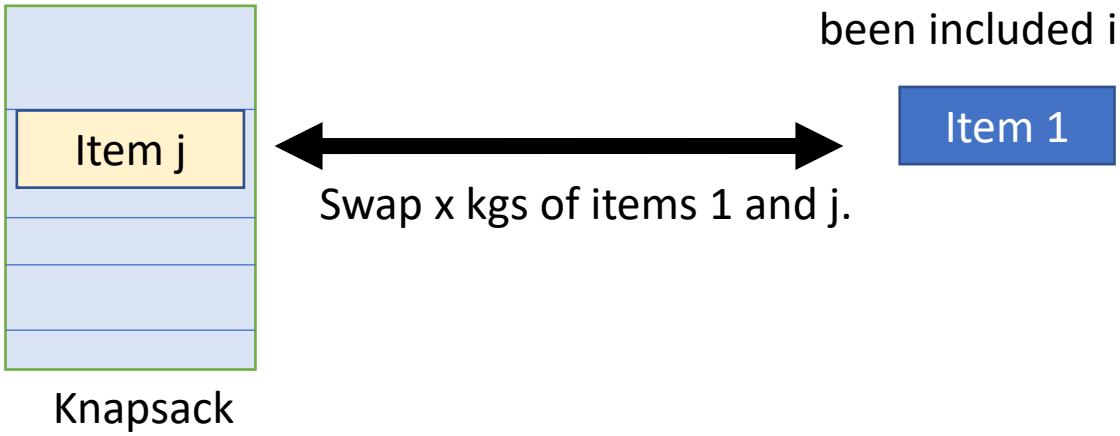
Consider the case, where items have distinct $v_i/w_i$

# Fractional Knapsack – Correctness Proof Sketch [informal]

- If we replace x kgs of some item j in the optimal solution with x kgs of item 1 (j ≠ 1)

Assume this is the optimal solution with total value V

Item 1 has the highest value per kg, and some of it has not been included in the knapsack




New value of knapsack V' = V – x * $\frac{v_j}{w_j}$ + x * $\frac{v_1}{w_1}$

As $\frac{v_1}{w_1} > \frac{v_j}{w_j}$, then V' > V

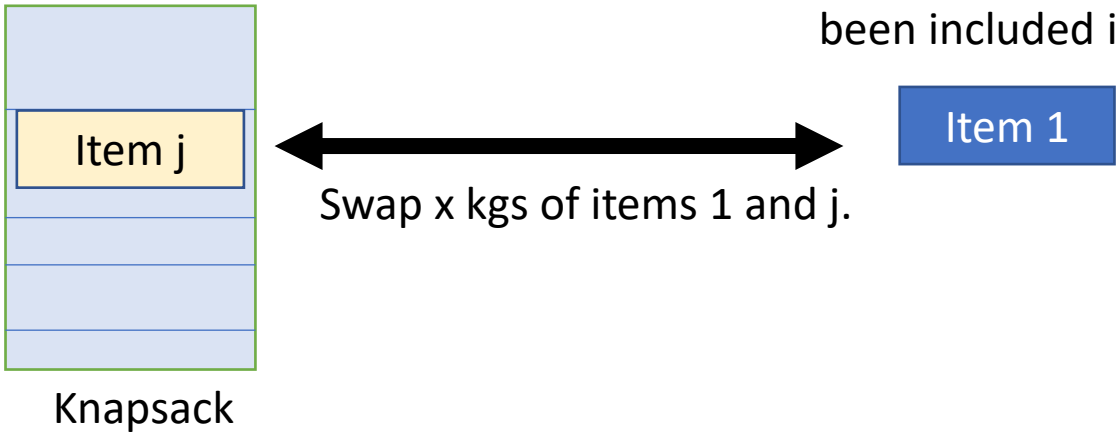
This is a contradiction. All of item 1 must be in the optimal solution.

# Fractional Knapsack – Correctness Proof Sketch [informal]

- If we replace x kgs of some item j in the optimal solution with x kgs of item 1 (j ≠ 1)

Assume this is the optimal solution with total value V

Item 1 has the highest value per kg, and some of it has not been included in the knapsack




If we don't assume distinct $v_i$ / $w_i$, then we can show that V' ≥ V.
Including item 1 will never make the solution worse.

New value of knapsack V' = V – x * $\frac{v_j}{w_j}$ + x * $\frac{v_1}{w_1}$ . As $\frac{v_1}{w_1} \geq \frac{v_j}{w_j}$ , then V' ≥ V

# Fractional Knapsack – Correctness Proof Sketch [informal]

Optimal substructure [CLRS]:

- If the optimal solution for weight W contains (some of) item i, then if we remove x kgs of item i, what remains in the knapsack is the optimal solution for weight W – x using the other n – 1 items and $w_i$ – x kgs from item i.

- The optimal substructure can be proven by contradiction.
  - If the optimal solution for weight W includes x kgs of item i and some non-optimal solution for weight W – x
  - Then, we can simply show that the solution of the problem for weight W cannot be optimal, because if we made the subproblem solution better, then we can use it to make the total value of the knapsack for weight W higher.

# Fractional Knapsack – Correctness Proof Sketch [informal]

- Therefore, we can reach the optimal solution by combining the greedy choice and the optimal solution to the subproblem that we have to solve after making the greedy choice.

# Greedy Strategy for 0-1 Knapsack?

- Note that the greedy approach won't work for the 0-1 knapsack.

| **i** | **$w_i$** | **$v_i$** | **$v_i / w_i$** |
|---|---|---|---|
| 1 | 10 | 60 | 6 |
| 2 | 20 | 100 | 5 |
| 3 | 30 | 120 | 4 |

- The greedy strategy based on $\mathbf{v_i / w_i}$ for the 0-1 knapsack in the above example will lead to items 1 and 2 only, which is not optimal.

# Huffman Codes

# Data Compression

- Needed for many applications in practice
- Two categories:
  - **Lossless data compression**
    - Allows reconstructing the original data completely from the compressed version without any loss of information
    - Used when changes in the uncompressed data are not tolerable, e.g., text, programs, etc.
  - **Lossy data compression**
    - Allows reconstructing an approximation of the original data.
    - Used to compress audio, video and images.

# Huffman Codes

- A technique for lossless data compression
- According to CLRS, it can achieve savings between 20% and 90%, depending on the characteristics of the input data.
- Uses a greedy method to find an optimal way for representing characters.

# Designing a Binary Code

- How can characters be represented in binary?
  - Fixed-length codes
    - Each character is represented by a unique binary string (codeword) of a fixed length.
    - Example: ASCII, Unicode
  - Variable-length codes
    - The codewords representing the characters vary in their length.
    - Can be utilized in compression, by assigning short codewords to the characters that appear frequently.

| Character | Fixed-length code | Variable-length code |
|---|---|---|
| A | 00 | 0 |
| B | 01 | 111 |
| C | 10 | 110 |
| D | 11 | 10 |

# Designing a Binary Code

- How to decode when using variable-length codes?
  - While encoding is straightforward, the decoding might not result into a single result if the code is not designed carefully.
  - Example

    Assume the codewords representing {A, B, C, D} are {1, 10, 110, 111}, how to decode the string 1110?

    Both AAB and AC are possible (ambiguity).

To avoid this, we use **prefix codes,** in which no codeword is a prefix of any other codeword.

These are known also as prefix-free codes.

# Prefix Codes

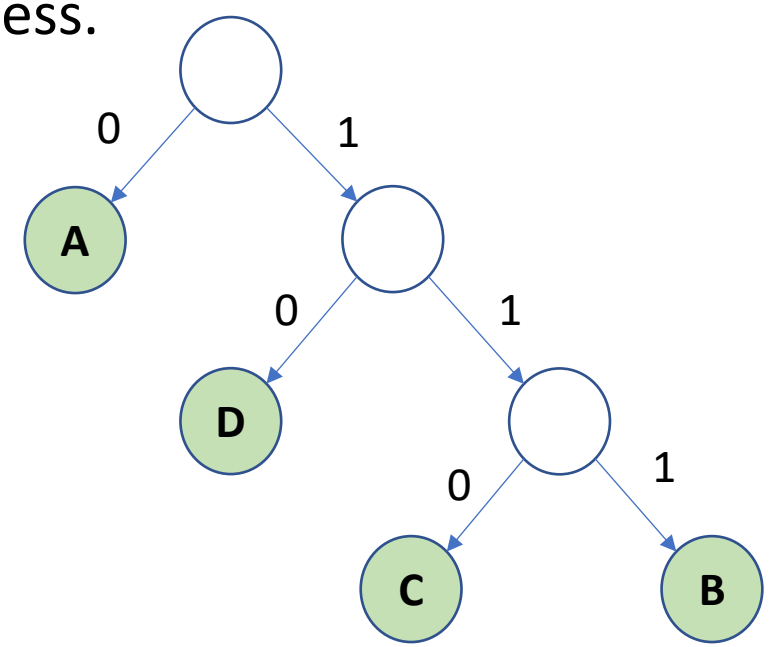
- No codeword can be a prefix of any other codeword.
- Example codewords: {0, 10, 110, 111}
  - Given a string 001010111110, it's straightforward to decode it.

0**0**1010111110
0**0**1010111110
00**1**010111110
00**1**010111110
0010**1**0111110
0010**1**0111110
001010**111**110
001010**111**110
001010111**1**10
001010111**1**10

# Prefix Codes

- Can be represented by a binary tree.
- Each path from the root to the leaves generates a codeword.
- Helps during the decoding process.

| Character | Codeword |
|---|---|
| A | 0 |
| B | 111 |
| C | 110 |
| D | 10 |




Example: 001010111110

# Prefix Codes

Given a prefix code tree T, the number of bits needed to encode a file can be calculated as:

$$B(T) = \sum_{c \in C} d_T(c) * c.freq$$

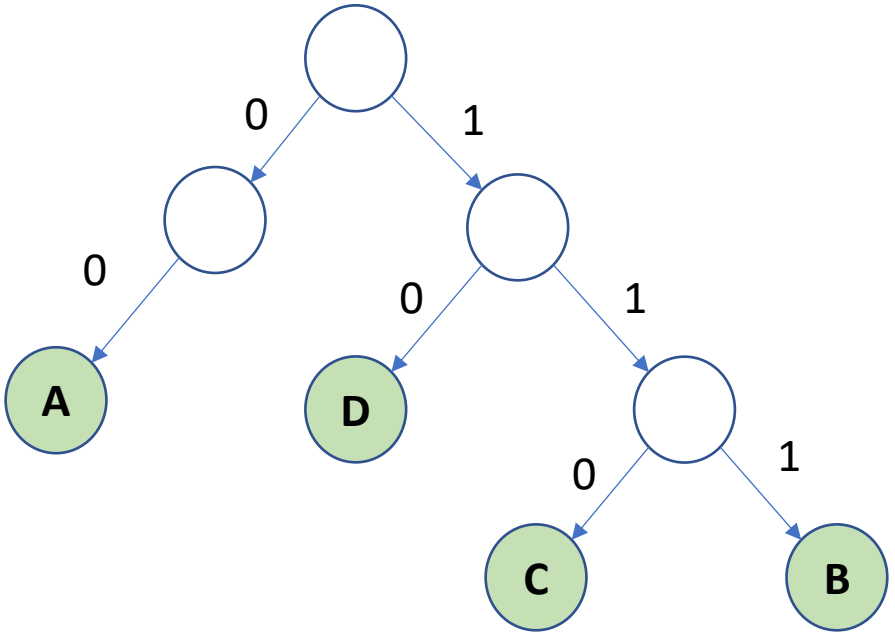
Length of codeword representing c

Frequency of character c

**Goal:** Find a prefix code that would minimize the above for a given file.

# Trees of Optimal Prefix Codes

- Binary trees corresponding to optimal prefix codes must be **full** binary trees.
- The number of leaves should be equal to the alphabet size |C| and the number of internal nodes should be |C| - 1.




For example, this cannot correspond to an optimal code. Why?

- These properties can be proven formally.
- These are necessary conditions but not sufficient for optimality. We will use Huffman's algorithm to get an optimal solution.

# Huffman Codes

- Huffman codes: Optimal prefix codes
- To illustrate the algorithm, we will trace an example first.
- The following example is from CLRS

| | **a** | **b** | **c** | **d** | **e** | **f** |
|---|---|---|---|---|---|---|
| **Freq.** | 45 | 13 | 12 | 16 | 9 | 5 |

# Huffman Codes

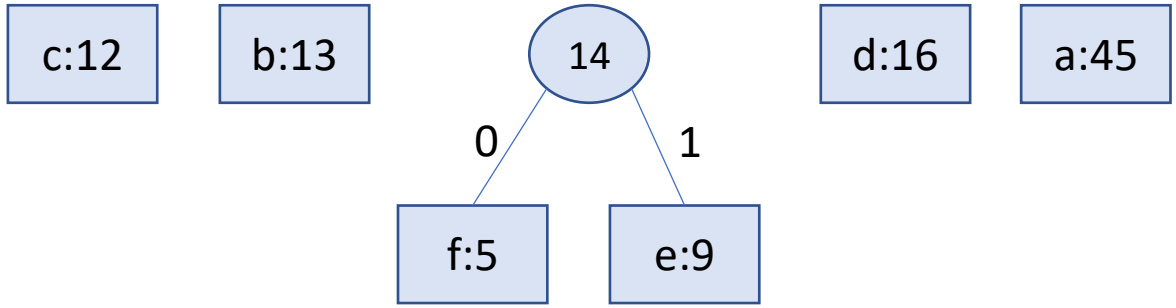
Example from CLRS

| | **a** | **b** | **c** | **d** | **e** | **f** |
|---|---|---|---|---|---|---|
| **Freq.** | 45 | 13 | 12 | 16 | 9 | 5 |

# Huffman Codes

Example from CLRS

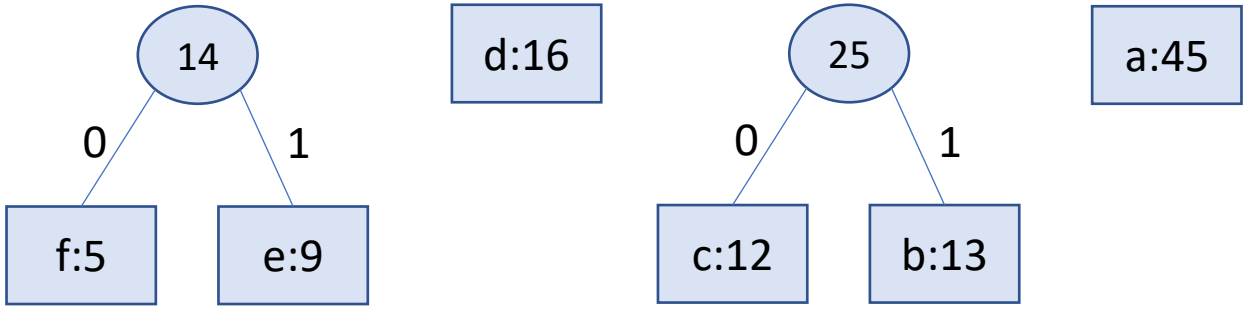
| | **a** | **b** | **c** | **d** | **e** | **f** |
|---|---|---|---|---|---|---|
| **Freq.** | 45 | 13 | 12 | 16 | 9 | 5 |



# Huffman Codes

Example from CLRS

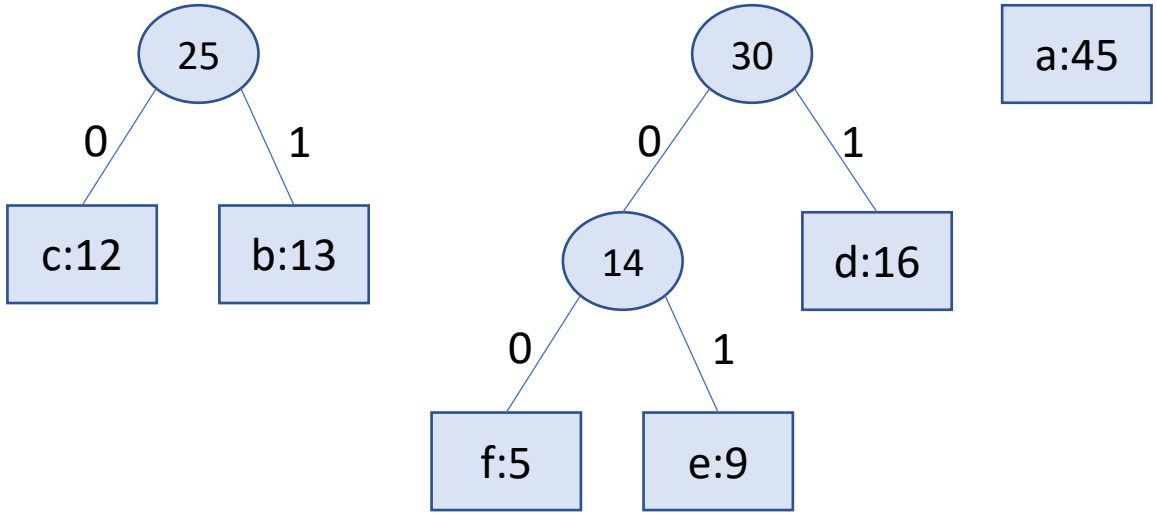
| | **a** | **b** | **c** | **d** | **e** | **f** |
|---|---|---|---|---|---|---|
| **Freq.** | 45 | 13 | 12 | 16 | 9 | 5 |



# Huffman Codes

Example from CLRS

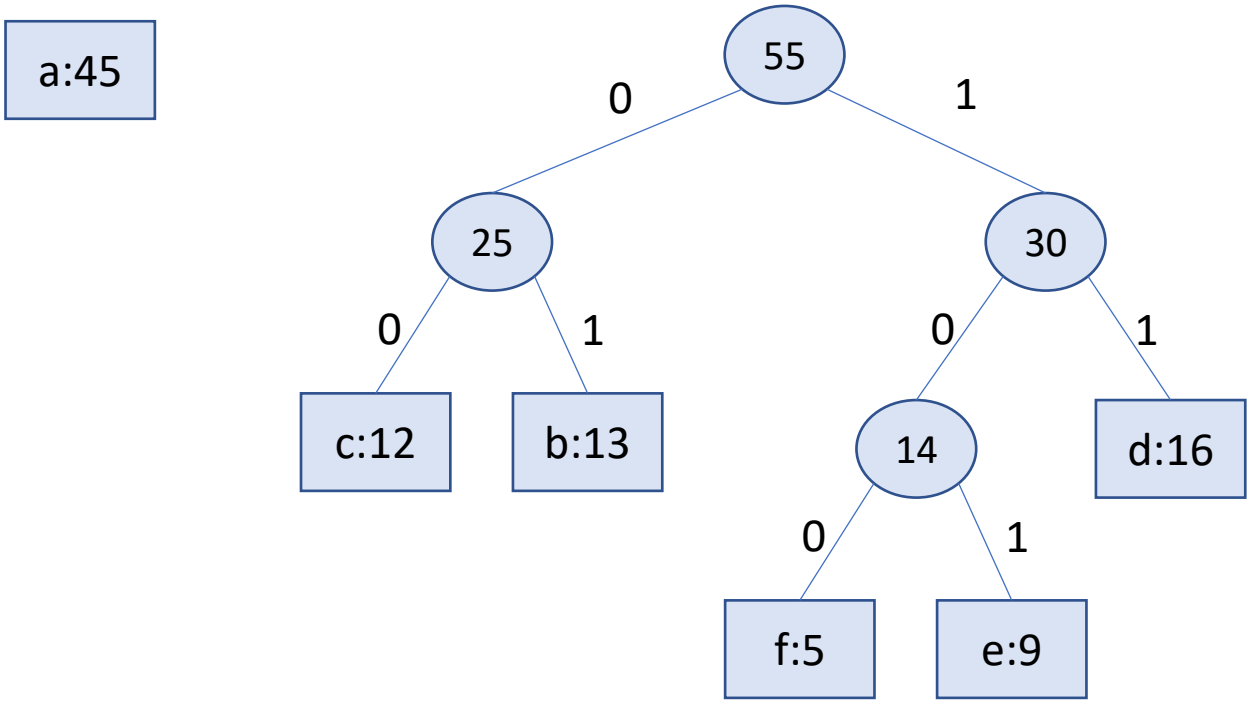
| | **a** | **b** | **c** | **d** | **e** | **f** |
|---|---|---|---|---|---|---|
| **Freq.** | 45 | 13 | 12 | 16 | 9 | 5 |



# Huffman Codes

Example from CLRS

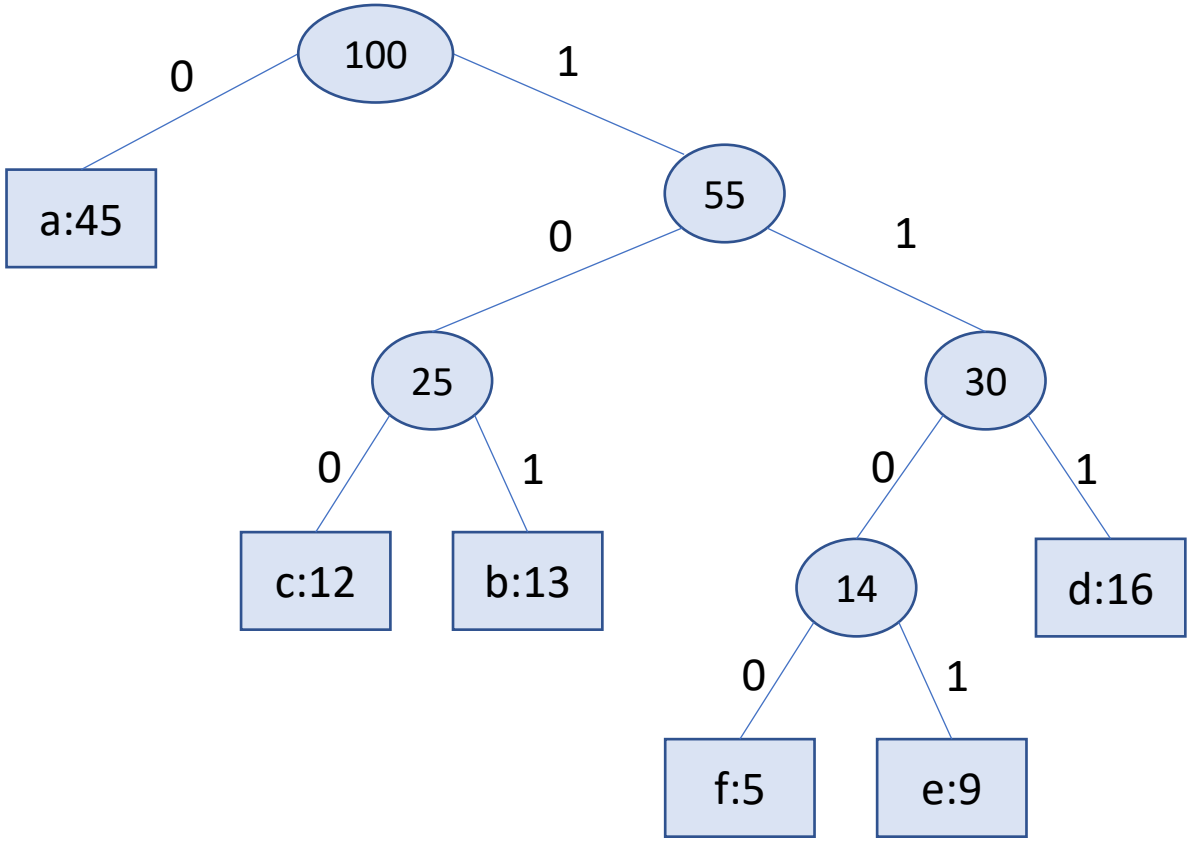
| | **a** | **b** | **c** | **d** | **e** | **f** |
|---|---|---|---|---|---|---|
| **Freq.** | 45 | 13 | 12 | 16 | 9 | 5 |



# Huffman Codes

Example from CLRS

| | **a** | **b** | **c** | **d** | **e** | **f** |
|---|---|---|---|---|---|---|
| **Freq.** | 45 | 13 | 12 | 16 | 9 | 5 |



# Huffman Codes

- How to implement the previous algorithm efficiently?
  - Need a data structure that supports minimum extraction and insertion.
  - Use a priority queue.
    - For simplicity, we will assume a binary min-heap is used for implementing the priority queue.
    - Advanced data structures could be used to achieve a better cost. See discussion in CLRS. This is extracurricular.

# Huffman Codes

- Pseudocode [CLRS]:

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4      allocate a new node z
5      z.left = x = EXTRACT-MIN(Q)
6      z.right = y = EXTRACT-MIN(Q)
7      z.freq = x.freq + y.freq
8      INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

Size of alphabet

All characters are added to the queue.
Note: each character has an attribute freq

Get the two nodes with the lowest frequencies, and combine them.

Running time: O(n lg n)

# Huffman Code – Correctness Proof

- Need to prove that the problem exhibits both the greedy choice and optimal substructure properties.