CS11313 - Spring 2023

Design & Analysis of Algorithms

Greedy Algorithms

Ibrahim Albluwi

Warmup: Change-Making Problem

Problem. Given a set of coin denominations (e.g. quarter, dime, nickel, etc.) and an amount of money, find the *minimum number of coins* that add up to this amount of money.

Example. What is the minimum number of coins needed for paying 99¢ using Jordanian currency?

Warmup: Change-Making Problem

Problem. Given a set of coin denominations (e.g. quarter, dime, nickel, etc.) and an amount of money, find the *minimum number of coins* that add up to this amount of money.

Example. What is the minimum number of coins needed for paying 99¢ using Jordanian currency?

1x50¢ + 1x25¢ + 2x10¢ + 4x1¢

Warmup: Change-Making Problem

Problem. Given a set of coin denominations (e.g. quarter, dime, nickel, etc.) and an amount of money, find the *minimum number of coins* that add up to this amount of money.

Example. What is the minimum number of coins needed for paying 99¢ using Jordanian currency?

`1x50¢ + 1x25¢ + 2x10¢ + 4x1¢`

Possible Solution. Sort the denominations in decreasing order. Pick from the largest denomination as much as possible and then move to the next.

Warmup: Change-Making Problem

Problem. Given a set of coin denominations (e.g. quarter, dime, nickel, etc.) and an amount of money, find the *minimum number of coins* that add up to this amount of money.

Example. What is the minimum number of coins needed for paying 99¢ using Jordanian currency?

`1x50¢ + 1x25¢ + 2x10¢ + 4x1¢`

Possible Solution. Sort the denominations in decreasing order. Pick from the largest denomination as much as possible and then move to the next.

Counter Example. Consider the denominations `{25¢, 20¢, 10¢, 1¢}`. What is the minimum number of coins needed to pay 40¢ ?

Warmup: Change-Making Problem

Problem. Given a set of coin denominations (e.g. quarter, dime, nickel, etc.) and an amount of money, find the *minimum number of coins* that add up to this amount of money.

Example. What is the minimum number of coins needed for paying 99¢ using Jordanian currency?

`1x50¢ + 1x25¢ + 2x10¢ + 4x1¢`

Possible Solution. Sort the denominations in decreasing order. Pick from the largest denomination as much as possible and then move to the next.

Counter Example. Consider the denominations `{25¢, 20¢, 10¢, 1¢}`. What is the minimum number of coins needed to pay 40¢ ?

Greedy: `25¢ + 10¢ + 1¢ + 1¢ + 1¢ + 1¢ + 1¢ = 40¢` (7 coins).

Optimal: `20¢ + 20¢ = 40¢` (2 coins)

Greedy Algorithms

Greedy Algorithm.

- Makes the best choice given the available information at the moment (without looking ahead or backtracking).
- Used for solving *optimization problems*.

Greedy Algorithms

Greedy Algorithm.

- Makes the best choice given the available information at the moment (without looking ahead or backtracking).
- Used for solving *optimization problems*.

Optimization problem. The problem of finding the *best* solution among all *feasible* solutions.

maximizes or minimizes an objective function

valid (*respects the constraints*)

Greedy Algorithms

Greedy Algorithm.

- Makes the best choice given the available information at the moment (without looking ahead or backtracking).
- Used for solving *optimization problems*.

Optimization problem. The problem of finding the *best* solution among all *feasible* solutions.

Example. Change Making.

Input: A set of coin denominations and an amount of money.
Objective Function: Choose the *minimum* number of coins.
Constraints: The *total coins value* must equal the given amount of money.

Greedy Algorithms

Greedy Algorithm.

- Makes the best choice given the available information at the moment (without looking ahead or backtracking).
- Used for solving *optimization problems*.

Optimization problem. The problem of finding the *best* solution among all *feasible* solutions.

Example. Change Making.

Input: A set of coin denominations and an amount of money.
Objective Function: Choose the *minimum* number of coins.
Constraints: The *total coins value* must equal the given amount of money.

Greedy Solution. Take as much as possible from the *largest* coin denomination. If taking more makes the total coins value larger than the given amount, start taking from the *next largest* coin denomination, etc.

Greedy Algorithms

Greedy Algorithm.

- Makes the best choice given the available information at the moment (without looking ahead or backtracking).
- Used for solving *optimization problems*.

Optimization problem. The problem of finding the *best* solution among all *feasible* solutions.

Example. Change Making.

Input: A set of coin denominations and an amount of money.
Objective Function: Choose the *minimum* number of coins.
Constraints: The *total coins value* must equal the given amount of money.

Greedy Solution. Take as much as possible from the *largest* coin denomination. If taking more makes the total coins value larger than the given amount, start taking from the *next largest* coin denomination, etc.

We need either a proof to show that the greedy solution always works or only one counterexample to show that it doesn't!

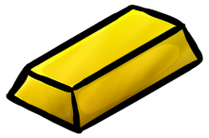
Fractional Knapsack

Input. N items, each of weight $w_i > 0$ and value $v_i > 0$. and a knapsack of weight capacity $W > 0$.

Goal. Find the *maximum* value the knapsack can hold.

Note. Taking a fraction of an item is allowed.

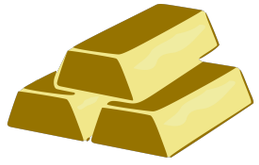
capacity = 13 kg



weight = 4 kg
value = $3000

weight = 12 kg
value = $8000

weight = 5 kg
value = $4000



weight = 6 kg
value = $6000

weight = 1 kg
value = $1000

Fractional Knapsack

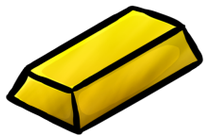
Input. N items, each of weight $w_i > 0$ and value $v_i > 0$. and a knapsack of weight capacity $W > 0$.

Goal. Find the *maximum* value the knapsack can hold.

Note. Taking a fraction of an item is allowed.

Greedy Strategy # 1.
Take the item with the highest value first.

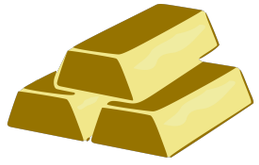
capacity = 13 kg



weight = 4 kg
value = $3000

weight = 12 kg
value = $8000

weight = 5 kg
value = $4000



weight = 6 kg
value = $6000

weight = 1 kg
value = $1000

Fractional Knapsack

Input. N items, each of weight $w_i > 0$ and value $v_i > 0$. and a knapsack of weight capacity $W > 0$.

Goal. Find the *maximum* value the knapsack can hold.

Note. Taking a fraction of an item is allowed.

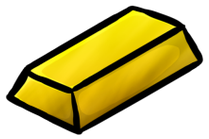
Greedy Strategy # 1.
Take the item with the highest value first.

12kg of = 12/12 * \$8000 = \$8000

1kg of = 1/6 * \$6000 = \$1000

= \$9000

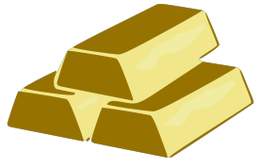
capacity = 13 kg



weight = 4 kg
value = \$3000

weight = 12 kg
value = \$8000

weight = 5 kg
value = \$4000



weight = 6 kg
value = \$6000

weight = 1 kg
value = \$1000

Fractional Knapsack

Input. N items, each of weight $w_i > 0$ and value $v_i > 0$. and a knapsack of weight capacity $W > 0$.

Goal. Find the *maximum* value the knapsack can hold.

Note. Taking a fraction of an item is allowed.

Greedy Strategy # 1.
Take the item with the highest value first.

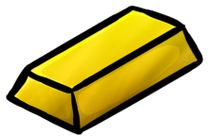
12kg of = 12/12 * \$8000 = \$8000

1kg of = 1/6 * \$6000 = \$1000

= \$9000

Can we do better?

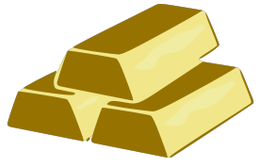
capacity = 13 kg



weight = 4 kg
value = \$3000

weight = 12 kg
value = \$8000

weight = 5 kg
value = \$4000



weight = 6 kg
value = \$6000

weight = 1 kg
value = \$1000

Fractional Knapsack

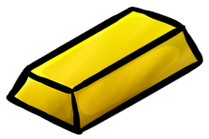
Input. N items, each of weight $w_i > 0$ and value $v_i > 0$. and a knapsack of weight capacity $W > 0$.

Goal. Find the *maximum* value the knapsack can hold.

Note. Taking a fraction of an item is allowed.

Greedy Strategy # 2.
Take the item with the highest value per Kg first.

capacity = 13 kg



weight = 4 kg
value = $3000

$750 per kg

weight = 12 kg
value = $8000

$666.7 per kg

weight = 5 kg
value = $4000

$800 per kg

weight = 6 kg
value = $6000

$1000 per kg

weight = 1 kg
value = $1000

$1000 per kg

Fractional Knapsack

Input. N items, each of weight $w_i > 0$ and value $v_i > 0$. and a knapsack of weight capacity $W > 0$.

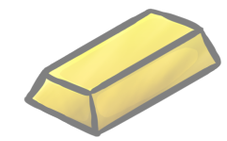
Goal. Find the *maximum* value the knapsack can hold.

Note. Taking a fraction of an item is allowed.

Greedy Strategy # 2.
Take the item with the highest value per Kg first.

6kg of = 6 * $1000 = $6000

capacity = 13 kg



weight = 4 kg
value = $3000

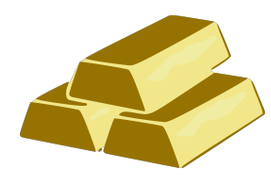
$750 per kg

weight = 12 kg
value = $8000

$666.7 per kg

weight = 5 kg
value = $4000

$800 per kg



weight = 6 kg
value = $6000

$1000 per kg

weight = 1 kg
value = $1000

$1000 per kg

Fractional Knapsack

Input. N items, each of weight $w_i > 0$ and value $v_i > 0$. and a knapsack of weight capacity $W > 0$.

Goal. Find the *maximum* value the knapsack can hold.

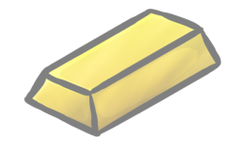
Note. Taking a fraction of an item is allowed.

Greedy Strategy # 2.
Take the item with the highest value per Kg first.

6kg of = 6 * $1000 = $6000

1kg of = 1 * $1000 = $1000

capacity = 13 kg



weight = 4 kg
value = $3000

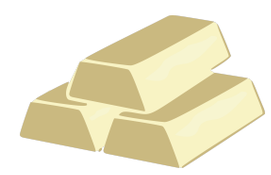
$750 per kg

weight = 12 kg
value = $8000

$666.7 per kg

weight = 5 kg
value = $4000

$800 per kg



weight = 6 kg
value = $6000

$1000 per kg

weight = 1 kg
value = $1000

$1000 per kg

Fractional Knapsack

Input. N items, each of weight $w_i > 0$ and value $v_i > 0$. and a knapsack of weight capacity $W > 0$.

Goal. Find the *maximum* value the knapsack can hold.

Note. Taking a fraction of an item is allowed.

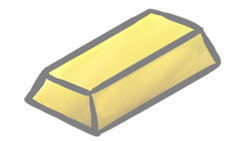
Greedy Strategy # 2.
Take the item with the highest value per Kg first.

6kg of = 6 * $1000 = $6000

1kg of = 1 * $1000 = $1000

5kg of = 5 * $800 = $4000

capacity = 13 kg



weight = 4 kg
value = $3000

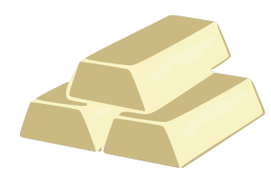
$750 per kg

weight = 12 kg
value = $8000

$666.7 per kg

weight = 5 kg
value = $4000

$800 per kg



weight = 6 kg
value = $6000

$1000 per kg

weight = 1 kg
value = $1000

$1000 per kg

Fractional Knapsack

Input. N items, each of weight $w_i > 0$ and value $v_i > 0$. and a knapsack of weight capacity $W > 0$.

Goal. Find the *maximum* value the knapsack can hold.

Note. Taking a fraction of an item is allowed.

Greedy Strategy # 2.
Take the item with the highest value per Kg first.

6kg of = 6 * $1000 = $6000

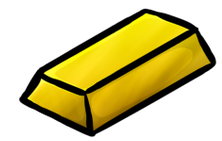
1kg of = 1 * $1000 = $1000

5kg of = 5 * $800 = $4000

1kg of = 1 * $750 = $750

= $11750

capacity = 13 kg



weight = 4 kg
value = $3000

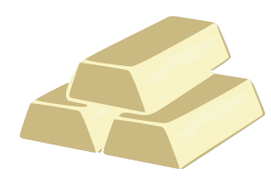
$750 per kg

weight = 12 kg
value = $8000

$666.7 per kg

weight = 5 kg
value = $4000

$800 per kg



weight = 6 kg
value = $6000

$1000 per kg

weight = 1 kg
value = $1000

$1000 per kg

Fractional Knapsack

Input. N items, each of weight $w_i > 0$ and value $v_i > 0$. and a knapsack of weight capacity $W > 0$.

Goal. Find the *maximum* value the knapsack can hold.

Note. Taking a fraction of an item is allowed.

Greedy Strategy # 2.
Take the item with the highest value per Kg first.

6kg of = 6 * $1000 = $6000

1kg of = 1 * $1000 = $1000

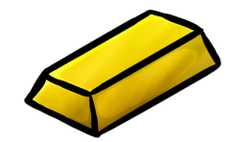
5kg of = 5 * $800 = $4000

1kg of = 1 * $750 = $750

= $11750

Can we do better?

capacity = 13 kg



weight = 4 kg
value = $3000

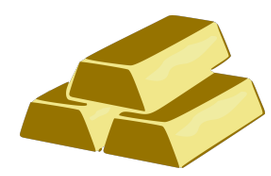
$750 per kg

weight = 12 kg
value = $8000

$666.7 per kg

weight = 5 kg
value = $4000

$800 per kg



weight = 6 kg
value = $6000

$1000 per kg

weight = 1 kg
value = $1000

$1000 per kg

Fractional Knapsack

Input. N items, each of weight $w_i > 0$ and value $v_i > 0$. and a knapsack of weight capacity $W > 0$.

Goal. Find the *maximum* value the knapsack can hold.

Note. Taking a fraction of an item is allowed.

Greedy Strategy # 2.
Take the item with the highest value per Kg first.

6kg of = 6 * $1000 = $6000

1kg of = 1 * $1000 = $1000

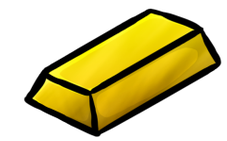
5kg of = 5 * $800 = $4000

1kg of = 1 * $750 = $750

= $11750

Can we do better? **IMPOSSIBLE!**

capacity = 13 kg



weight = 4 kg
value = $3000

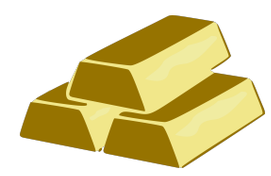
$750 per kg

weight = 12 kg
value = $8000

$666.7 per kg

weight = 5 kg
value = $4000

$800 per kg



weight = 6 kg
value = $6000

$1000 per kg

weight = 1 kg
value = $1000

$1000 per kg

Fractional Knapsack: Proof of Correctness

Consider a set of n items sorted by value per weight.

Item 1		Item 2		Item 3		Item 4		Item 5		· · · · · · ·		Item n
$\frac{v_1}{w_1}$	$>$	$\frac{v_2}{w_2}$	$>$	$\frac{v_3}{w_3}$	$>$	$\frac{v_4}{w_4}$	$>$	$\frac{v_5}{w_5}$	$>$		$>$	$\frac{v_n}{w_n}$

Fractional Knapsack: Proof of Correctness

Consider a set of n items sorted by value per weight.

Item 1		Item 2		Item 3		Item 4		Item 5		· · · · · · ·		Item n
$\frac{v_1}{w_1}$	$>$	$\frac{v_2}{w_2}$	$>$	$\frac{v_3}{w_3}$	$>$	$\frac{v_4}{w_4}$	$>$	$\frac{v_5}{w_5}$	$>$		$>$	$\frac{v_n}{w_n}$

Assume that there is an optimal solution for the fractional knapsack problem that *does not consume the item with the highest value per Kg.*

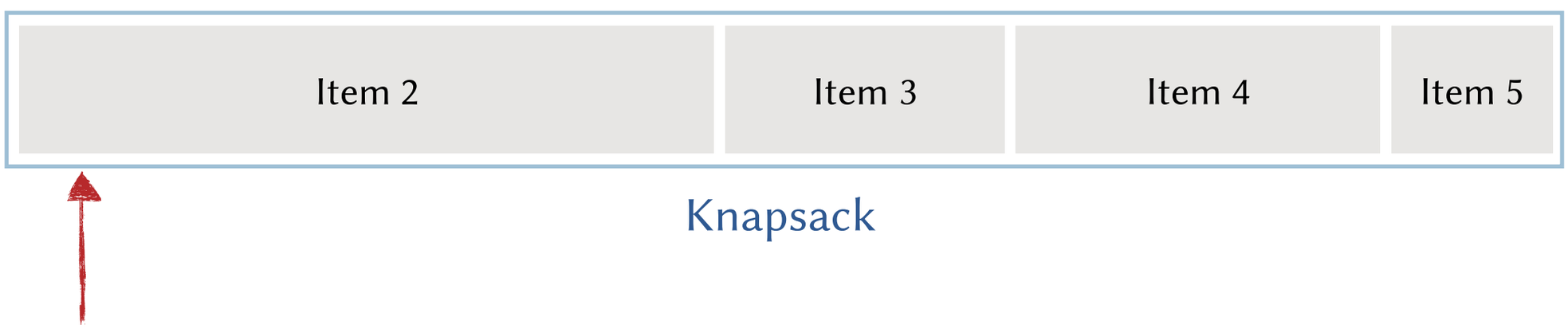
Knapsack

Fractional Knapsack: Proof of Correctness

Consider a set of n items sorted by value per weight.

Item 1		Item 2		Item 3		Item 4		Item 5		· · · · · · ·		Item n
$\frac{v_1}{w_1}$	$>$	$\frac{v_2}{w_2}$	$>$	$\frac{v_3}{w_3}$	$>$	$\frac{v_4}{w_4}$	$>$	$\frac{v_5}{w_5}$	$>$		$>$	$\frac{v_n}{w_n}$

Assume that there is an optimal solution for the fractional knapsack problem that *does not consume the item with the highest value per Kg.*

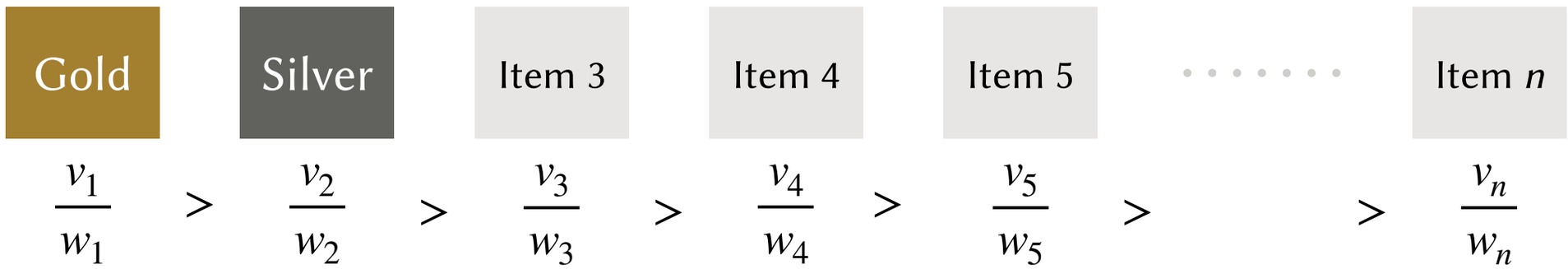


We can replace parts of this item with parts of *item 1* and get a better solution! Therefore, the solution is not optimal! (*contradiction*)

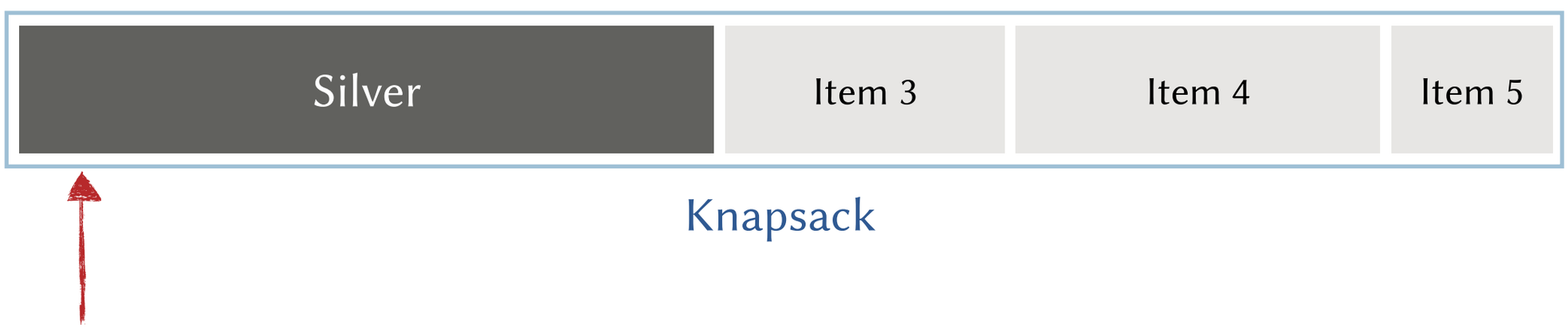
Note. This is not a proof. However, it captures the essence of the argument for why this greedy choice is optimal.

Fractional Knapsack: Proof of Correctness

Consider a set of n items sorted by value per weight.



Assume that there is an optimal solution for the fractional knapsack problem that *does not consume the item with the highest value per Kg.*



We can replace parts of the silver with parts of *gold* and get a better solution! Therefore, the solution is not optimal! (*contradiction*)

Note. This is not a proof. However, it captures the essence of the argument for why this greedy choice is optimal.

Fractional Knapsack (Algorithm)

```
FRACTIONAL_KNAPSACK(w[], v[], n, W)

Sort w and v by v[i]/w[i] in decreasing order
```

Fractional Knapsack (Algorithm)

```
FRACTIONAL_KNAPSACK(w[], v[], n, W)

Sort w and v by v[i]/w[i] in decreasing order

load = 0, i = 1
while i <= n and load < W:
```

number of items

knapsack capacity

Fractional Knapsack (Algorithm)

```
FRACTIONAL_KNAPSACK(w[], v[], n, W)

Sort w and v by v[i]/w[i] in decreasing order

load = 0, i = 1
while i <= n and load < W:

    if w[i] <= W-load:

        all of the item fits inside the
        remaining capacity

    else

        only a fraction of the item fits
        inside the remaining capacity
```

Fractional Knapsack (Algorithm)

```
FRACTIONAL_KNAPSACK(w[], v[], n, W)

Sort w and v by v[i]/w[i] in decreasing order

load = 0, i = 1
while i <= n and load < W:
    if w[i] <= W-load:
        Take all of item i
        load += w[i]
        profit = profit + v[i]
    else
        only a fraction of the item fits
        inside the remaining capacity
```

Fractional Knapsack (Algorithm)

```
FRACTIONAL_KNAPSACK(w[], v[], n, W)

Sort w and v by v[i]/w[i] in decreasing order

load = 0, i = 1
while i <= n and load < W:

    if w[i] <= W-load:
        Take all of item i
        load += w[i]
        profit = profit + v[i]
    else
        Take (W - load) of item i
        profit = profit + (W - load) * v[i]/w[i]
        load = W

    i = i+1
```

Fractional Knapsack (Algorithm)

```
FRACTIONAL_KNAPSACK(w[], v[], n, W)

Sort w and v by v[i]/w[i] in decreasing order

load = 0, i = 1
while i <= n and load < W:

    if w[i] <= W-load:
        Take all of item i
        load += w[i]
        profit = profit + v[i]
    else
        Take (W - load) of item i
        profit = profit + (W - load) * v[i]/w[i]
        load = W

    i = i+1
```

$\Theta(n \log n)$

$O(n)$

Running Time. $\Theta(n \log n)$

Lesson. Greedy algorithms are typically simple and efficient.

0 – 1 Knapsack

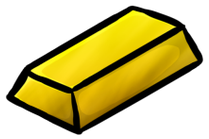
Input. N items, each of weight $w_i > 0$ and value $v_i > 0$. and a knapsack of weight capacity $W > 0$.

Goal. Find the *maximum* value the knapsack can hold.

Assumptions.

- Taking a fraction of an item is **not** allowed.
- W, w_i and v_i are integers.

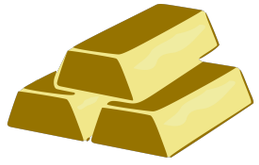
capacity = 13 kg



weight = 4 kg
value = $3000

weight = 12 kg
value = $8000

weight = 5 kg
value = $4000



weight = 6 kg
value = $6000

weight = 1 kg
value = $1000

0 – 1 Knapsack

Input. N items, each of weight $w_i > 0$ and value $v_i > 0$. and a knapsack of weight capacity $W > 0$.

Goal. Find the *maximum* value the knapsack can hold.

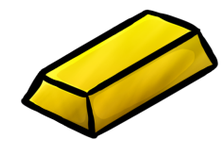
Assumptions.

- Taking a fraction of an item is **not** allowed.
- W, w_i and v_i are integers.

Examples.

- `$8000` (12 kg) + `$1000` (1 kg) = `$9000` (13 kg)

capacity = 13 kg



weight = 4 kg
value = $3000

weight = 12 kg
value = $8000

weight = 5 kg
value = $4000

weight = 6 kg
value = $6000

weight = 1 kg
value = $1000

0 – 1 Knapsack

Input. N items, each of weight $w_i > 0$ and value $v_i > 0$. and a knapsack of weight capacity $W > 0$.

Goal. Find the *maximum* value the knapsack can hold.

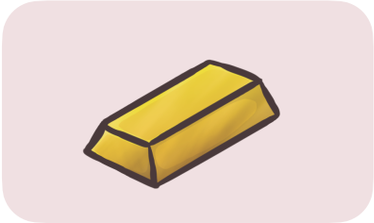
Assumptions.

- Taking a fraction of an item is **not** allowed.
- W, w_i and v_i are integers.

Examples.

- `$8000` (12 kg) + `$1000` (1 kg) = `$9000` (13 kg)
- `$6000` (6 kg) + `$3000` (4 kg) + `$1000` (1 kg) = `$10000` (11 kg)

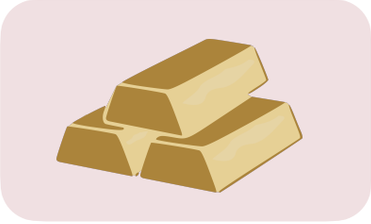
capacity = 13 kg



weight = 4 kg
value = $3000

weight = 12 kg
value = $8000

weight = 5 kg
value = $4000



weight = 6 kg
value = $6000

weight = 1 kg
value = $1000

0–1 Knapsack

Input. N items, each of weight $w_i > 0$ and value $v_i > 0$. and a knapsack of weight capacity $W > 0$.

Goal. Find the *maximum* value the knapsack can hold.

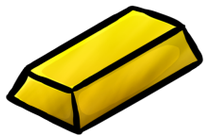
Assumptions.

- Taking a fraction of an item is **not** allowed.
- W, w_i and v_i are integers.

Examples.

- `$8000` (12 kg) + `$1000` (1 kg) = `$9000` (13 kg)
- `$6000` (6 kg) + `$3000` (4 kg) + `$1000` (1 kg) = `$10000` (11 kg)
- `$6000` (6 kg) + `$4000` (5 kg) + `$1000` (1 kg) = `$11000` (12 kg)

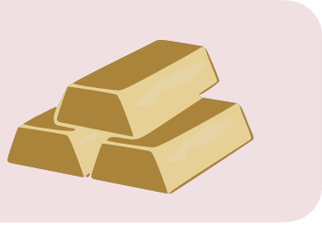
capacity = 13 kg



weight = 4 kg
value = $3000

weight = 12 kg
value = $8000

weight = 5 kg
value = $4000



weight = 6 kg
value = $6000

weight = 1 kg
value = $1000

0 – 1 **Knapsack**: Greedy Strategies

Greedy choice # 1. Take the *most valuable* item first.

Counter Example.
Greedy gives $12 but the optimal is $17.

0–1 Knapsack: Greedy Strategies

Greedy choice # 1. Take the *most valuable* item first.

Counter Example.
Greedy gives $12 but the optimal is $17.

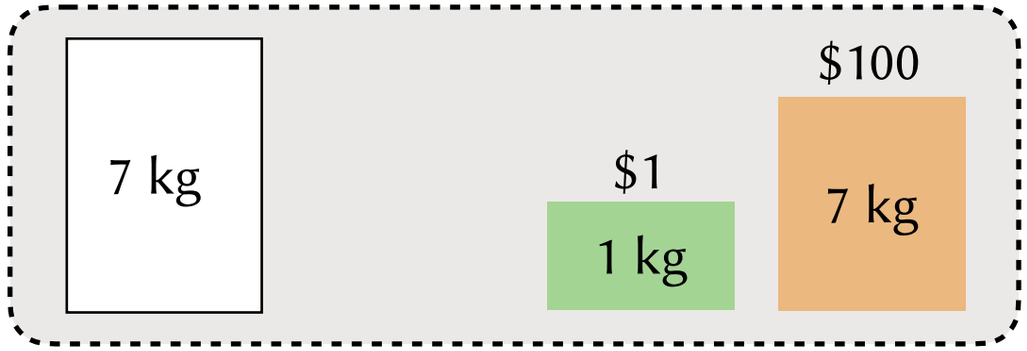
Greedy choice # 2. Take the *lightest* item first.

0–1 Knapsack: Greedy Strategies

Greedy choice # 1. Take the *most valuable* item first.

Counter Example.
Greedy gives $12 but the optimal is $17.

Greedy choice # 2. Take the *lightest* item first.



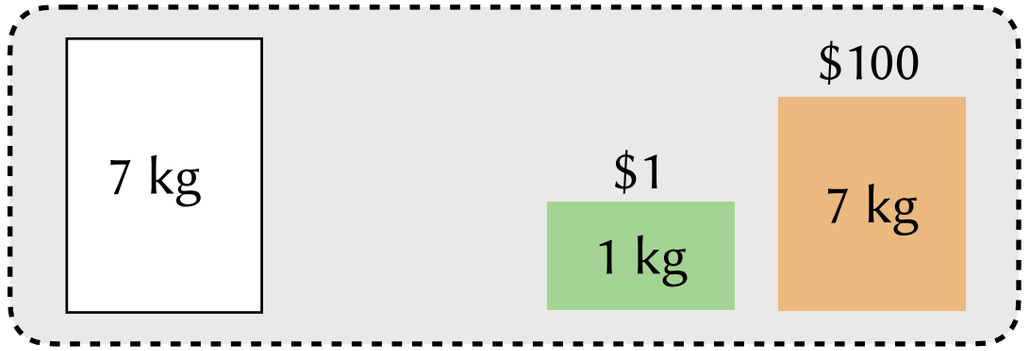
Counter Example.
Greedy gives $1 but the optimal is $100.

0-1 Knapsack: Greedy Strategies

Greedy choice # 1. Take the *most valuable* item first.

Counter Example.
Greedy gives $12 but the optimal is $17.

Greedy choice # 2. Take the *lightest* item first.



Counter Example.
Greedy gives $1 but the optimal is $100.

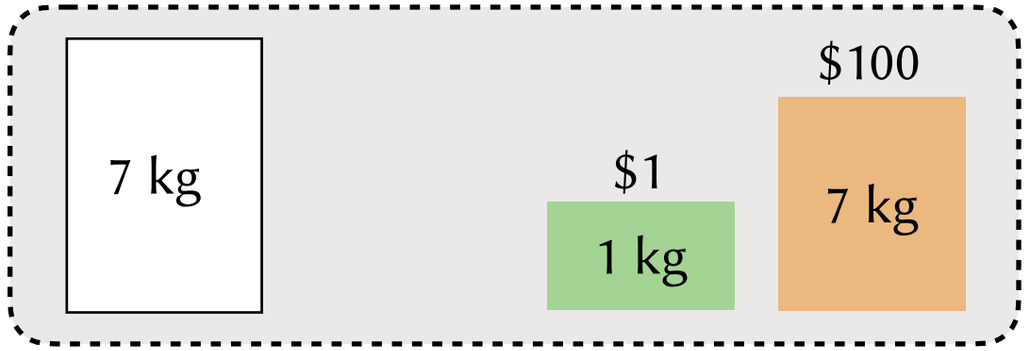
Greedy choice # 3. Take the item with the *highest value per Kg*.

0-1 Knapsack: Greedy Strategies

Greedy choice # 1. Take the *most valuable* item first.

Counter Example.
Greedy gives $12 but the optimal is $17.

Greedy choice # 2. Take the *lightest* item first.



Counter Example.
Greedy gives $1 but the optimal is $100.

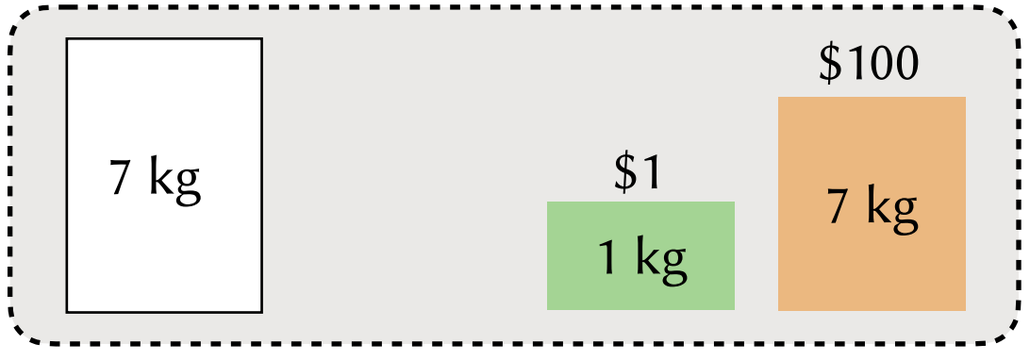
Greedy choice # 3. Take the item with the *highest value per* Kg.

Counter Example.
Greedy gives $2 but the optimal is $5.

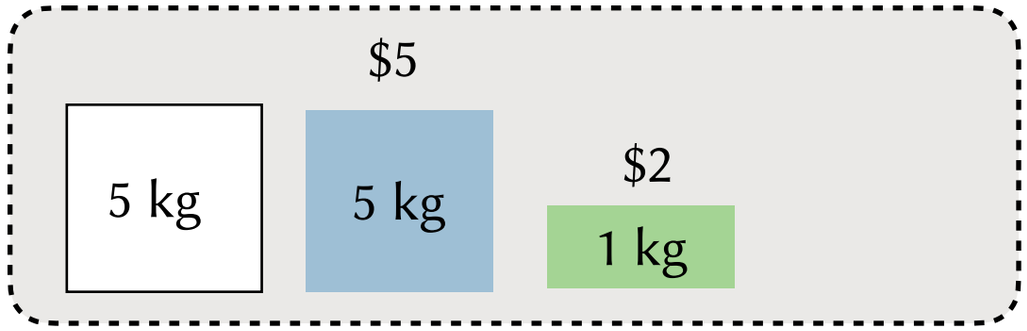
Greedy Solutions Do not Always Work

Greedy Choice Property. A *globally* optimal solution can be reached with a sequence of l*ocally* optimal decisions.

Counter Example.
Greedy gives $12 but the optimal is $17.



Counter Example.
Greedy gives $1 but the optimal is $100.



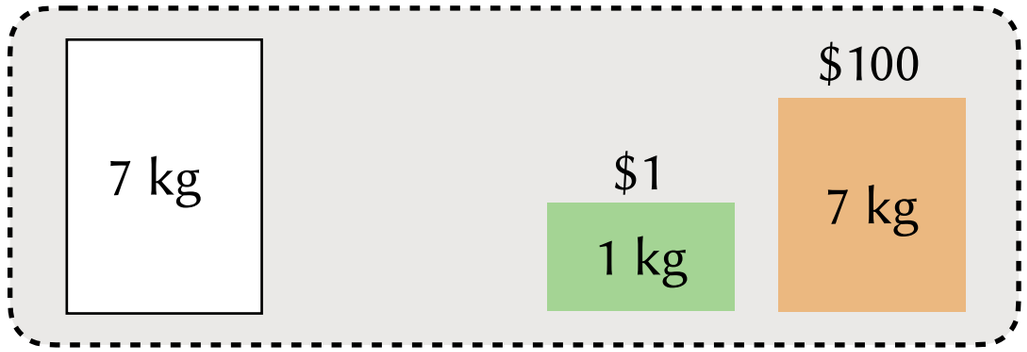
Counter Example.
Greedy gives $2 but the optimal is $5.

Greedy Solutions Do not Always Work

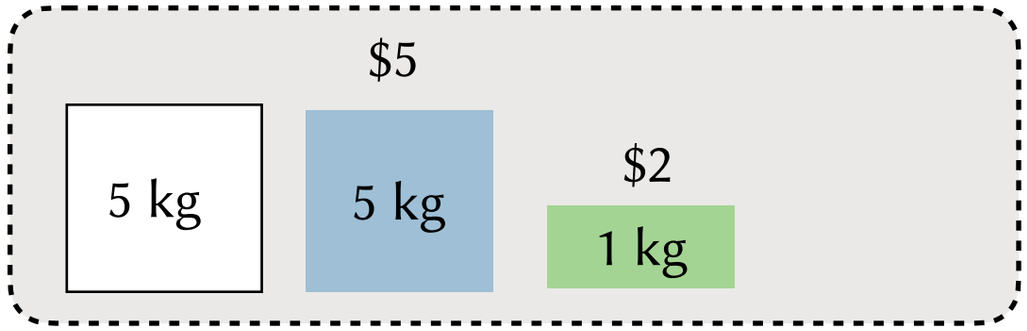
Greedy Choice Property. A *globally* optimal solution can be reached with a sequence of l*ocally* optimal decisions.

To prove that the greedy choice property holds, we must prove that a greedy strategy produces an optimal solution for *every instance* of the problem.

Counter Example.
Greedy gives $12 but the optimal is $17.



Counter Example.
Greedy gives $1 but the optimal is $100.



Counter Example.
Greedy gives $2 but the optimal is $5.

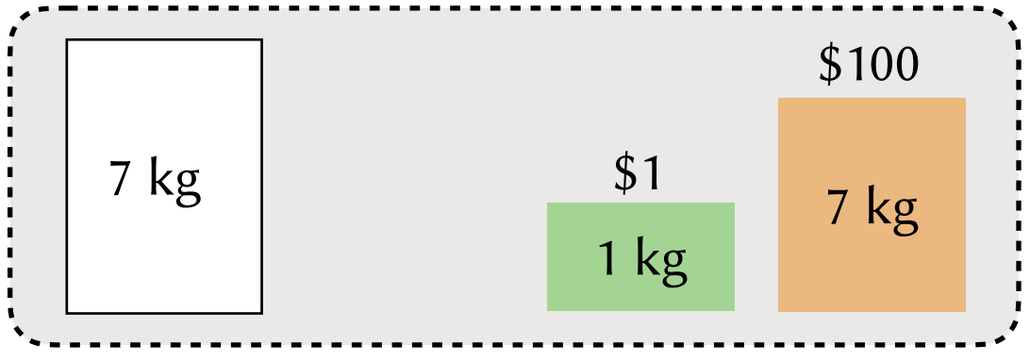
Greedy Solutions Do not Always Work

Greedy Choice Property. A *globally* optimal solution can be reached with a sequence of l*ocally* optimal decisions.

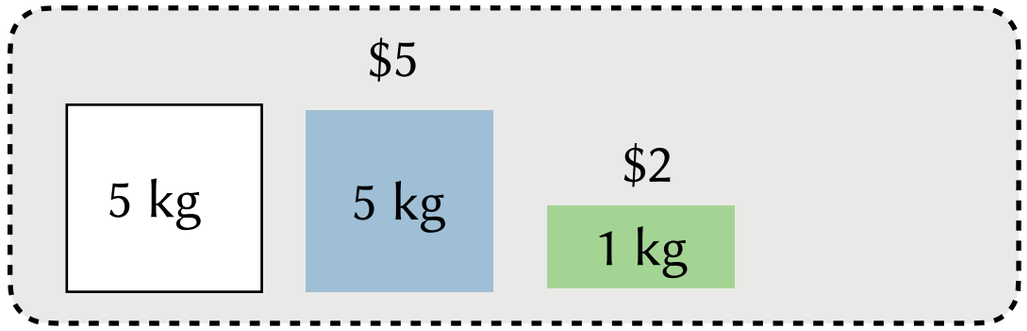
To prove that the greedy choice property holds, we must prove that a greedy strategy produces an optimal solution for *every instance* of the problem.

To prove that a greedy strategy is *not optimal* it is enough to find *one counter example*.

Counter Example.
Greedy gives $12 but the optimal is $17.



Counter Example.
Greedy gives $1 but the optimal is $100.



Counter Example.
Greedy gives $2 but the optimal is $5.

Greedy Solutions Do not Always Work

Greedy Choice Property. A *globally* optimal solution can be reached with a sequence of l*ocally* optimal decisions.

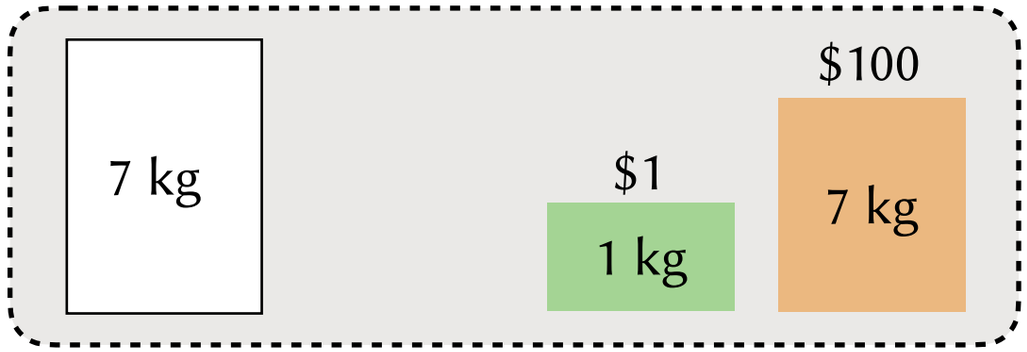
To prove that the greedy choice property holds, we must prove that a greedy strategy produces an optimal solution for *every instance* of the problem.

To prove that a greedy strategy is *not optimal* it is enough to find *one counter example*.

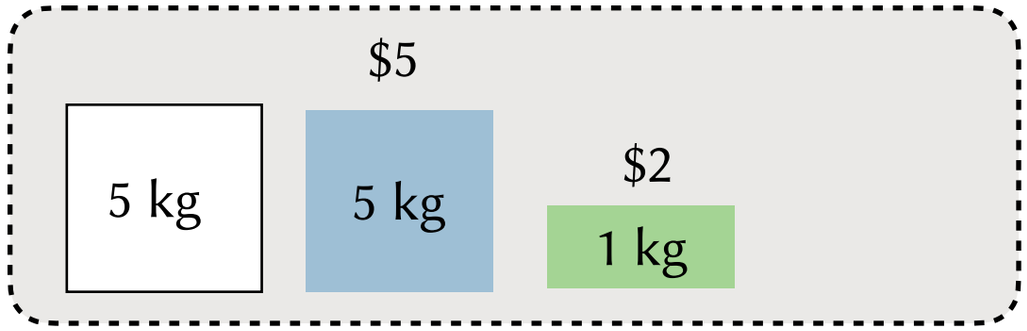
There is no known optimal greedy strategy for the 0-1 Knapsack problem

We will learn a
dynamic programming
solution for the problem later!

Counter Example.
Greedy gives $12 but the optimal is $17.



Counter Example.
Greedy gives $1 but the optimal is $100.

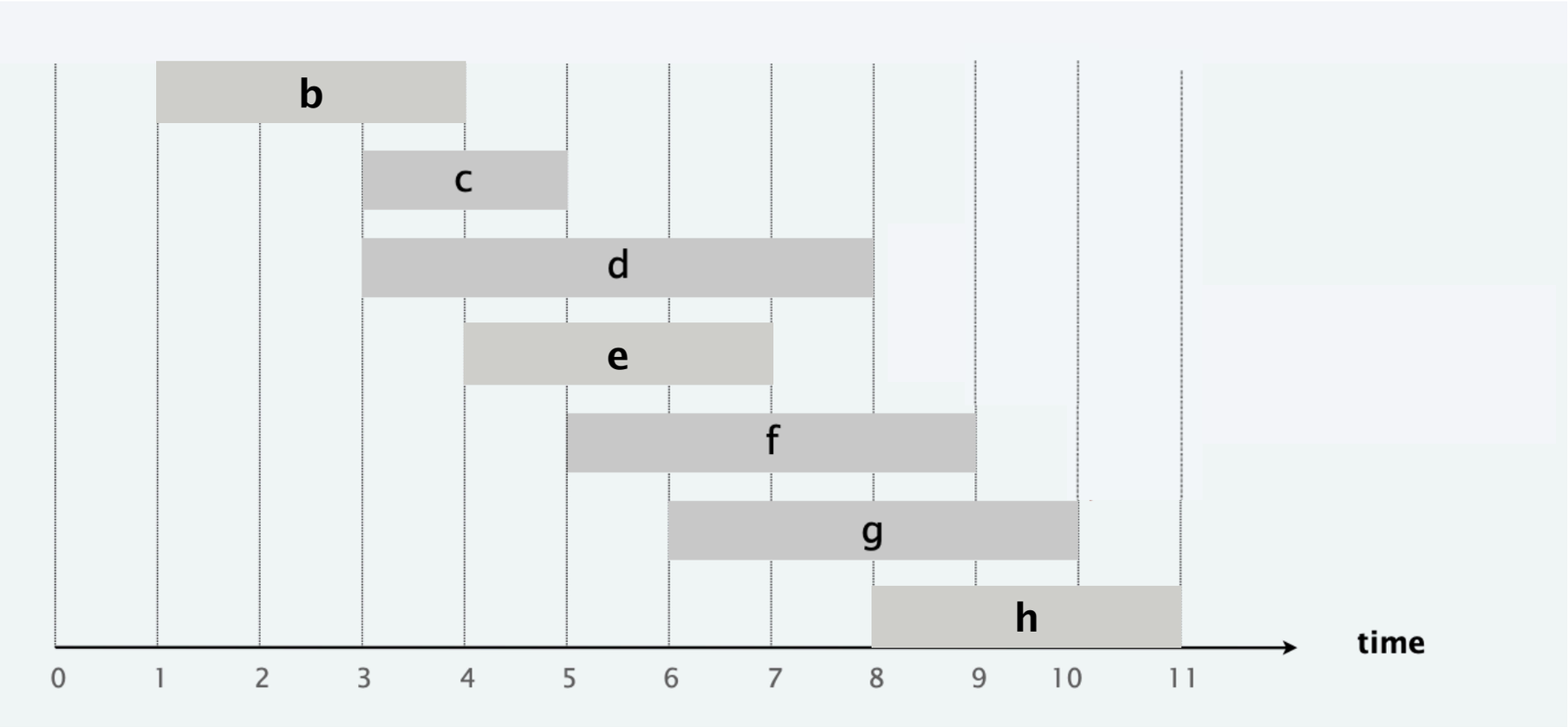


Counter Example.
Greedy gives $2 but the optimal is $5.

Activity Selection

Problem. Given n activities that require an exclusive use of a common resource, find the *maximum* number of non-overlapping activities.

Examples. Maximum number of jobs that can be done by a person.
Maximum number of events that can be held in a room.

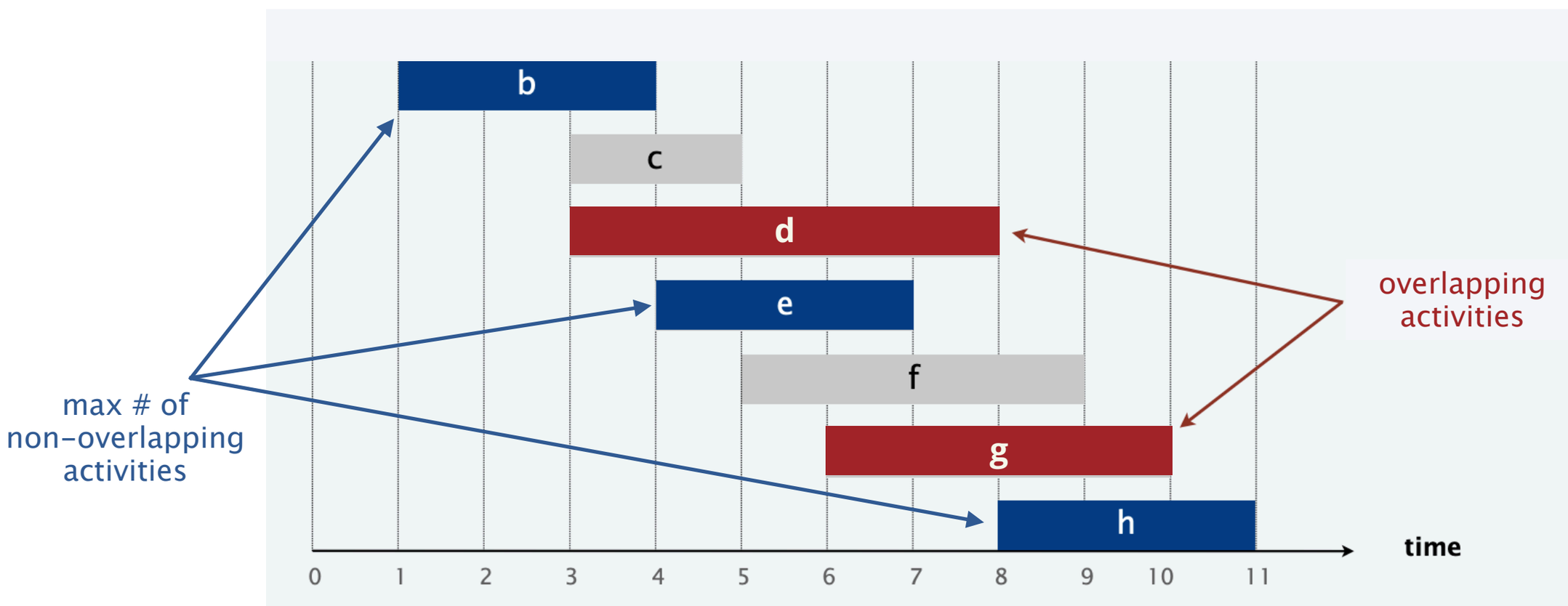


adapted from an image by Kevin Wayne

Activity Selection

Problem. Given n activities that require an exclusive use of a common resource, find the *maximum* number of non-overlapping activities.

Examples. Maximum number of jobs that can be done by a person.
Maximum number of events that can be held in a room.



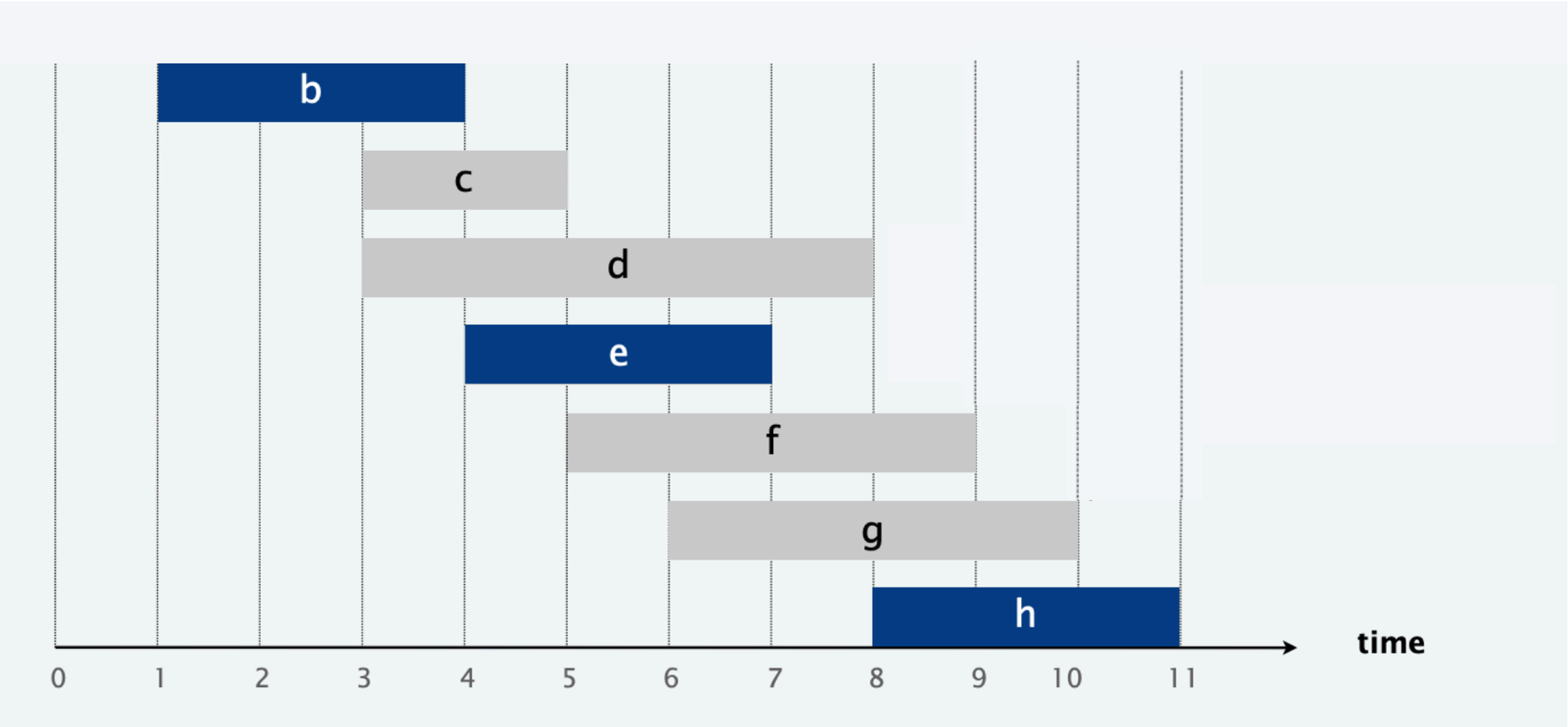
adapted from an image by Kevin Wayne

Activity Selection

Problem. Given n activities that require an exclusive use of a common resource, find the *maximum* number of non-overlapping activities.

Examples. Maximum number of jobs that can be done by a person.
Maximum number of events that can be held in a room.

Notation. - $S = \{a_1, a_2, a_3, \ldots, a_n\}$ is the set of activities.



adapted from an image by Kevin Wayne

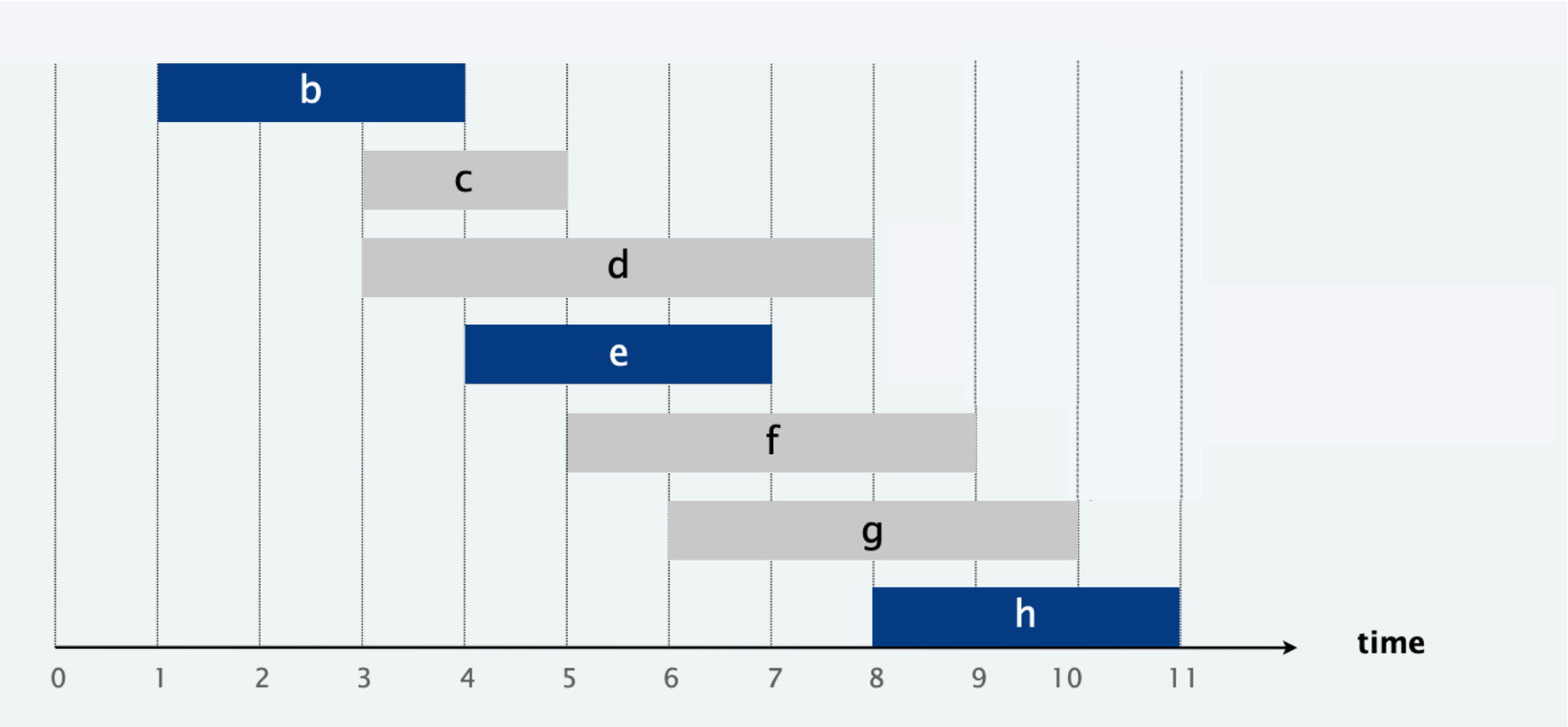
Activity Selection

Problem. Given n activities that require an exclusive use of a common resource, find the *maximum* number of non-overlapping activities.

Examples. Maximum number of jobs that can be done by a person.
Maximum number of events that can be held in a room.

Notation. - $S = \{a_1, a_2, a_3, \ldots, a_n\}$ is the set of activities.

- a_i needs the resource during $[\, s_i, f_i \,)$ (s_i = start time and f_i = finish time) where $0 \leq s_i < f_i < \infty$.



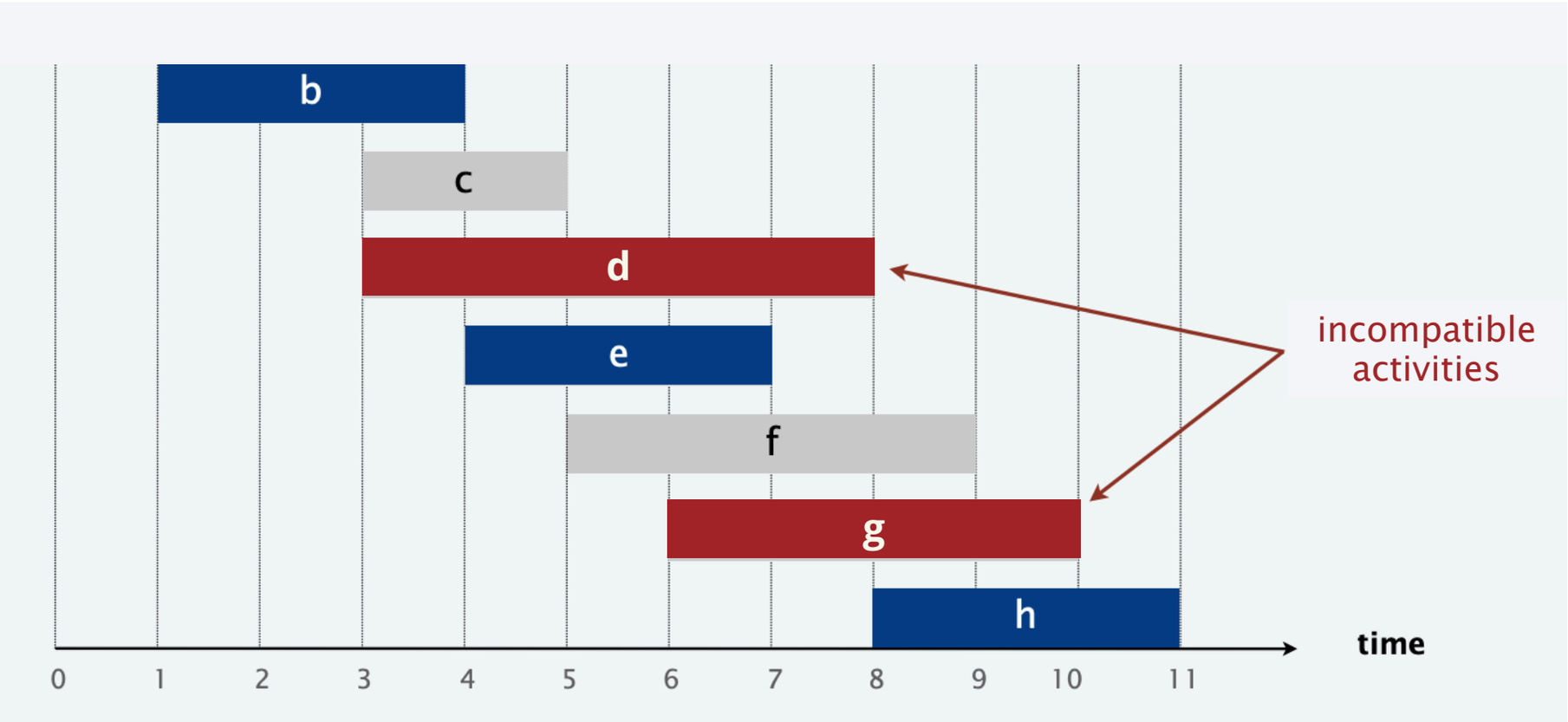
adapted from an image by Kevin Wayne

Activity Selection

Problem. Given n activities that require an exclusive use of a common resource, find the *maximum* number of non-overlapping activities.

Examples. Maximum number of jobs that can be done by a person.
Maximum number of events that can be held in a room.

Notation.
- $S = \{a_1, a_2, a_3, \ldots, a_n\}$ is the set of activities.
- a_i needs the resource during $[\, s_i, f_i \,)$ (s_i = start time and f_i = finish time) where $0 \leq s_i < f_i < \infty$.
- a_i and a_j are compatible if $[\, s_i, f_i \,)$ and $[\, s_j, f_j \,)$ do not overlap.



adapted from an image by Kevin Wayne

Activity Selection

Greedy strategy # 1. Earliest start time first.
Rationale. Start the activities as early as possible.

Activity Selection

Greedy strategy # 1. Earliest start time first.
Rationale. Start the activities as early as possible.

Counter Examples

Greedy: 1 Optimal: 2

Activity Selection

Greedy strategy # 1. Earliest start time first.
Rationale. Start the activities as early as possible.

Greedy strategy # 2. Shortest activity first.
Rationale. Shorter activities tend to have less conflicts.

Counter Examples

?

Activity Selection

Counter Examples

Greedy strategy # 1. Earliest start time first.
Rationale. Start the activities as early as possible.

Greedy strategy # 2. Shortest activity first.
Rationale. Shorter activities tend to have less conflicts.

Greedy strategy # 3. Least number of conflicts first.
Rationale. Less conflicts means more compatible activities.

?

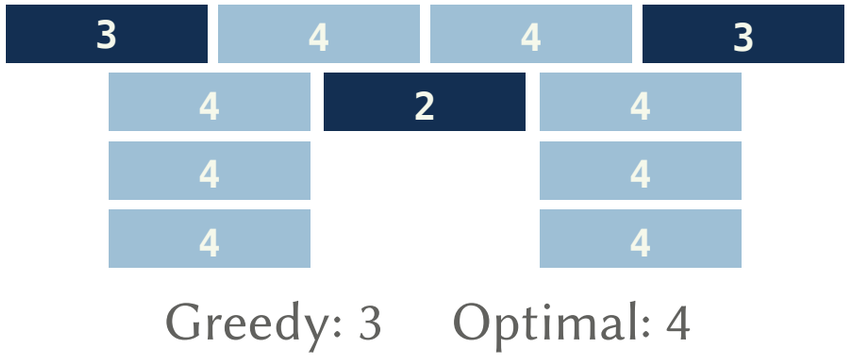
Activity Selection

Counter Examples

Greedy strategy # 1. Earliest start time first.
Rationale. Start the activities as early as possible.

Greedy strategy # 2. Shortest activity first.
Rationale. Shorter activities tend to have less conflicts.

Greedy strategy # 3. Least number of conflicts first.
Rationale. Less conflicts means more compatible activities.



Activity Selection

Counter Examples

Greedy strategy # 1. Earliest start time first.
Rationale. Start the activities as early as possible.

Greedy: 1 Optimal: 2

Greedy strategy # 2. Shortest activity first.
Rationale. Shorter activities tend to have less conflicts.

Greedy: 1 Optimal: 2

Greedy strategy # 3. Least number of conflicts first.
Rationale. Less conflicts means more compatible activities.

Greedy: 3 Optimal: 4

Greedy strategy # 4. Earliest finish time first.
Rationale. Activities that finish early leave more time to be filled with other activities later.

Greedy strategy # 5. Latest start time first.
Rationale. Activities that start late leave more time to be filled with other activities earlier.

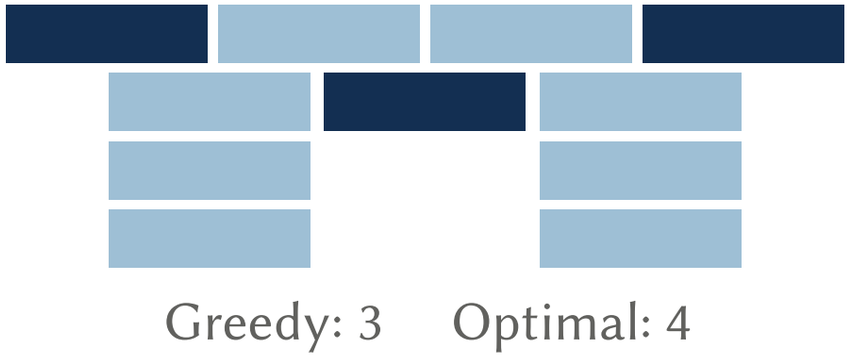
Activity Selection

Counter Examples

Greedy strategy # 1. Earliest start time first.
Rationale. Start the activities as early as possible.

Greedy strategy # 2. Shortest activity first.
Rationale. Shorter activities tend to have less conflicts.

Greedy strategy # 3. Least number of conflicts first.
Rationale. Less conflicts means more compatible activities.



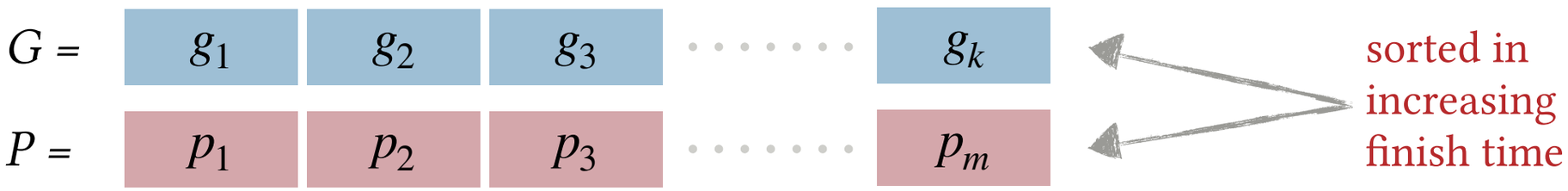
optimal

Greedy strategy # 4. Earliest finish time first.
Rationale. Activities that finish early leave more time to be filled with other activities later.

Greedy strategy # 5. Latest start time first.
Rationale. Activities that start late leave more time to be filled with other activities earlier.

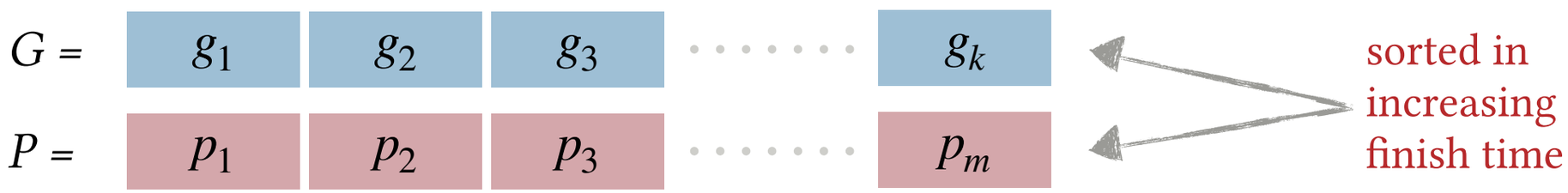
Activity Selection (Optimality of Greedy Solution)

Assume P is an optimal solution with m activities and G is the greedy solution with k activities. We would like to show that $m = k$ (*i.e. G is as good as the optimal solution*).



Activity Selection (Optimality of Greedy Solution)

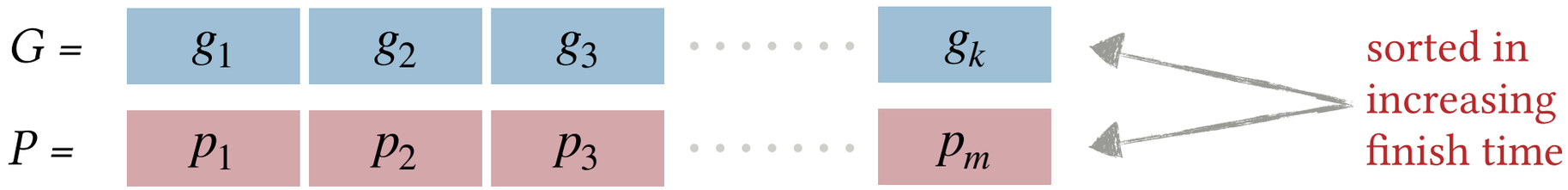
Assume P is an optimal solution with m activities and G is the greedy solution with k activities. We would like to show that $m = k$ (*i.e. G is as good as the optimal solution*).



Observation 1. We can always replace p_1 in P by g_1 from G.

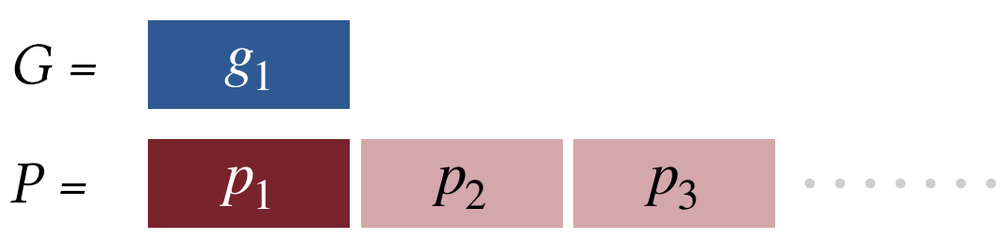
Activity Selection (Optimality of Greedy Solution)

Assume P is an optimal solution with m activities and G is the greedy solution with k activities. We would like to show that $m = k$ (*i.e. G is as good as the optimal solution*).



Observation 1. We can always replace p_1 in P by g_1 from G.

Proof. Since g_1 is guaranteed to finish before (or with) p_1, g_1 is compatible with $p_2 \longrightarrow p_m$. Hence, there is always an optimal solution that starts with the activity that finishes first (g_1).

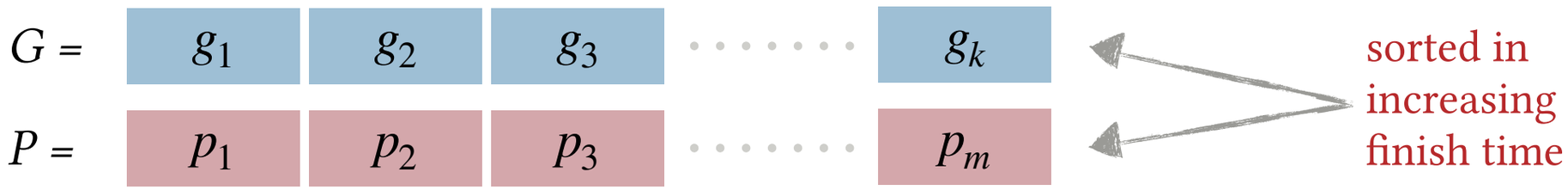


If g_1 ends with p_1
it is compatible with p_2

If g_1 ends before p_1
it is compatible with p_2

Activity Selection (Optimality of Greedy Solution)

Assume P is an optimal solution with m activities and G is the greedy solution with k activities. We would like to show that $m = k$ (*i.e. G is as good as the optimal solution*).



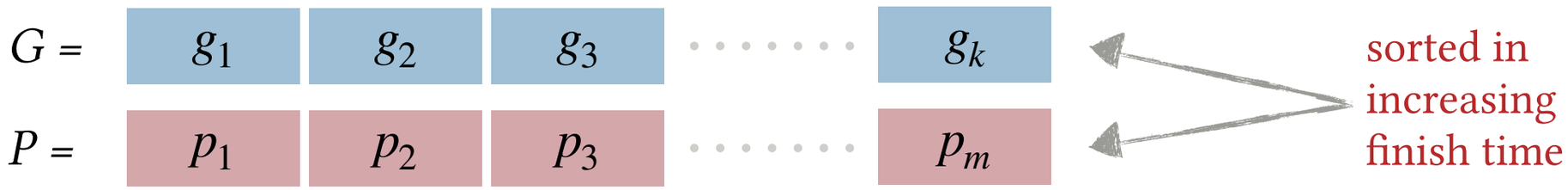
Observation 1. We can always replace p_1 in P by g_1 from G.

Proof. Since g_1 is guaranteed to finish before (or with) p_1, g_1 is compatible with $p_2 \longrightarrow p_m$. Hence, there is always an optimal solution that starts with the activity that finishes first (g_1).

Observation 2. P is made of p_1 + an optimal solution to the activity selection problem considering only activities that start after the finish time of g_1 and p_1.

Activity Selection (Optimality of Greedy Solution)

Assume P is an optimal solution with m activities and G is the greedy solution with k activities. We would like to show that $m = k$ (*i.e. G is as good as the optimal solution*).



Observation 1. We can always replace p_1 in P by g_1 from G.

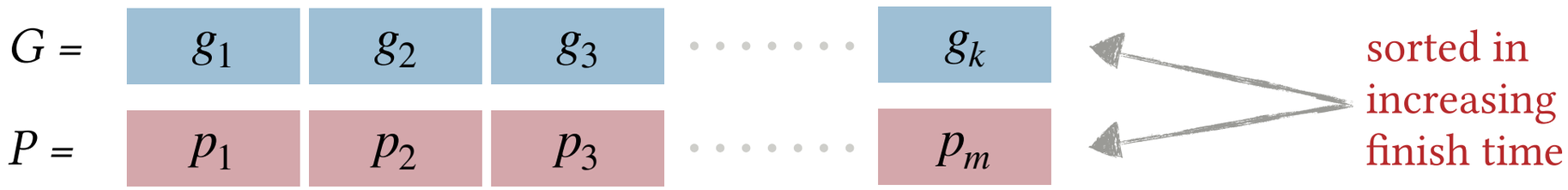
Proof. Since g_1 is guaranteed to finish before (or with) p_1, g_1 is compatible with $p_2 \longrightarrow p_m$. Hence, there is always an optimal solution that starts with the activity that finishes first (g_1).

Observation 2. P is made of p_1 + an optimal solution to the activity selection problem considering only activities that start after the finish time of g_1 and p_1.

Proof. If this is not true, then P is not optimal (contradiction).

Activity Selection (Optimality of Greedy Solution)

Assume P is an optimal solution with m activities and G is the greedy solution with k activities. We would like to show that $m = k$ (*i.e. G is as good as the optimal solution*).



Observation 1. We can always replace p_1 in P by g_1 from G.

Proof. Since g_1 is guaranteed to finish before (or with) p_1, g_1 is compatible with $p_2 \longrightarrow p_m$. Hence, there is always an optimal solution that starts with the activity that finishes first (g_1).

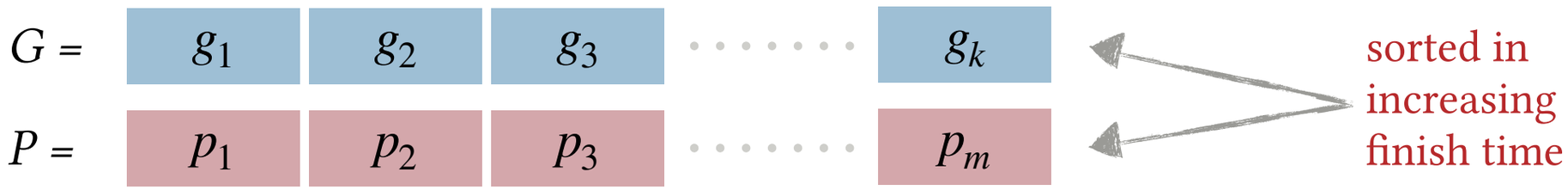
Observation 2. P is made of p_1 + an optimal solution to the activity selection problem considering only activities that start after the finish time of g_1 and p_1.

Proof. If this is not true, then P is not optimal (contradiction).

Hence, we can apply the same argument used in Observation 1 to show that the first activity (p_2) in the optimal solution for the new subproblem can be replaced by g_2.

Activity Selection (Optimality of Greedy Solution)

Assume P is an optimal solution with m activities and G is the greedy solution with k activities. We would like to show that $m = k$ (*i.e. G is as good as the optimal solution*).



Observation 1. We can always replace p_1 in P by g_1 from G.

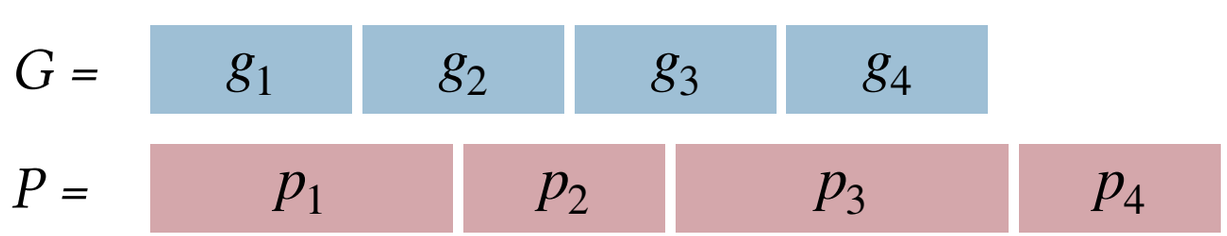
Proof. Since g_1 is guaranteed to finish before (or with) p_1, g_1 is compatible with $p_2 \longrightarrow p_m$. Hence, there is always an optimal solution that starts with the activity that finishes first (g_1).

Observation 2. P is made of p_1 + an optimal solution to the activity selection problem considering only activities that start after the finish time of g_1 and p_1.

Proof. If this is not true, then P is not optimal (contradiction).

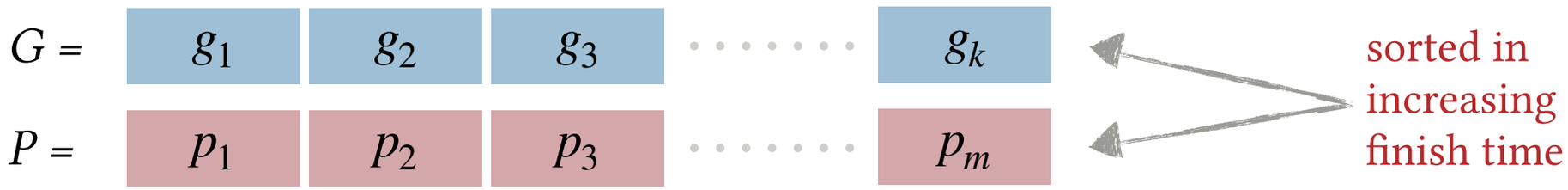
Hence, we can apply the same argument used in Observation 1 to show that the first activity (p_2) in the optimal solution for the new subproblem can be replaced by g_2.

Example.



Activity Selection (Optimality of Greedy Solution)

Assume P is an optimal solution with m activities and G is the greedy solution with k activities. We would like to show that $m = k$ (*i.e. G is as good as the optimal solution*).



Observation 1. We can always replace p_1 in P by g_1 from G.

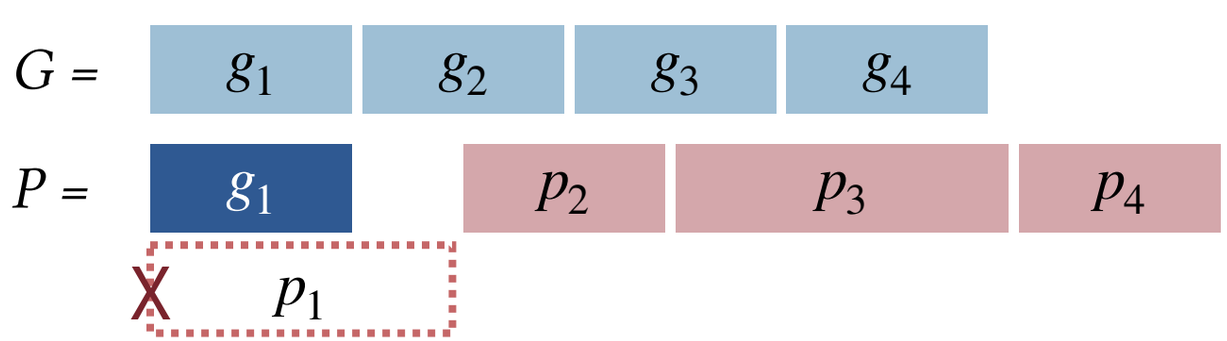
Proof. Since g_1 is guaranteed to finish before (or with) p_1, g_1 is compatible with $p_2 \longrightarrow p_m$. Hence, there is always an optimal solution that starts with the activity that finishes first (g_1).

Observation 2. P is made of p_1 + an optimal solution to the activity selection problem considering only activities that start after the finish time of g_1 and p_1.

Proof. If this is not true, then P is not optimal (contradiction).

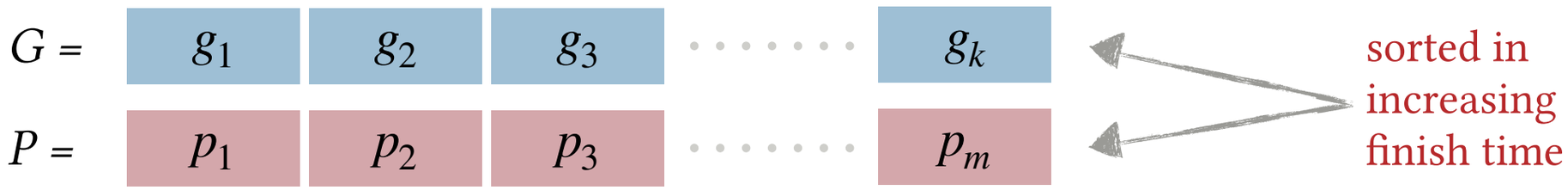
Hence, we can apply the same argument used in Observation 1 to show that the first activity (p_2) in the optimal solution for the new subproblem can be replaced by g_2.

Example.



Activity Selection (Optimality of Greedy Solution)

Assume P is an optimal solution with m activities and G is the greedy solution with k activities. We would like to show that $m = k$ (*i.e. G is as good as the optimal solution*).



Observation 1. We can always replace p_1 in P by g_1 from G.

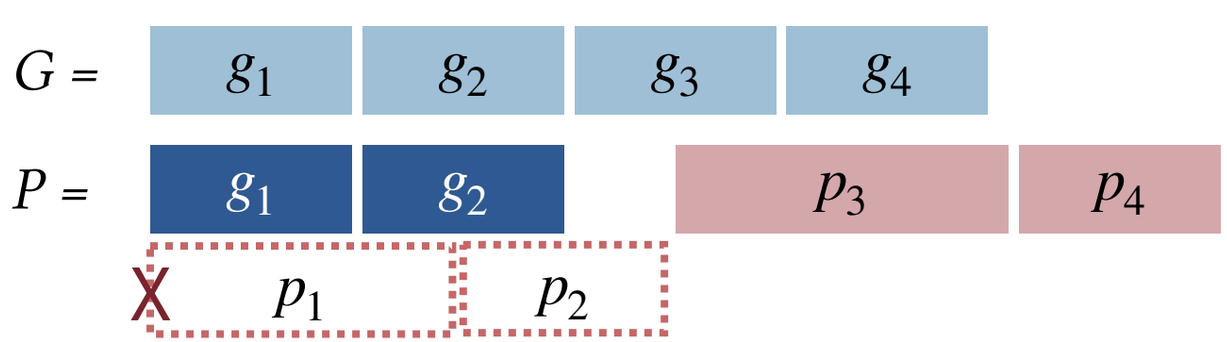
Proof. Since g_1 is guaranteed to finish before (or with) p_1, g_1 is compatible with $p_2 \longrightarrow p_m$. Hence, there is always an optimal solution that starts with the activity that finishes first (g_1).

Observation 2. P is made of p_1 + an optimal solution to the activity selection problem considering only activities that start after the finish time of g_1 and p_1.

Proof. If this is not true, then P is not optimal (contradiction).

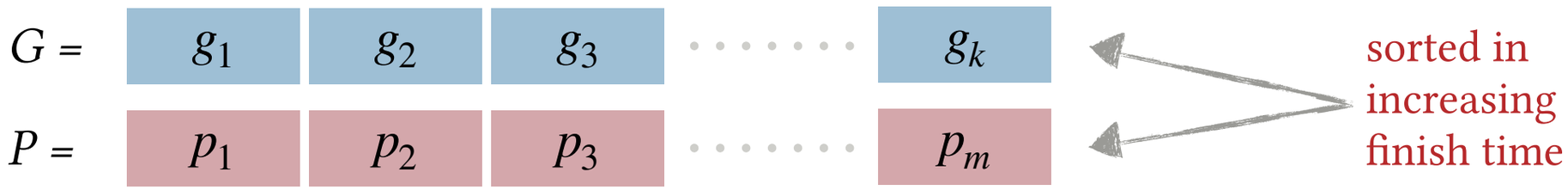
Hence, we can apply the same argument used in Observation 1 to show that the first activity (p_2) in the optimal solution for the new subproblem can be replaced by g_2.

Example.



Activity Selection (Optimality of Greedy Solution)

Assume P is an optimal solution with m activities and G is the greedy solution with k activities. We would like to show that $m = k$ (*i.e. G is as good as the optimal solution*).



Observation 1. We can always replace p_1 in P by g_1 from G.

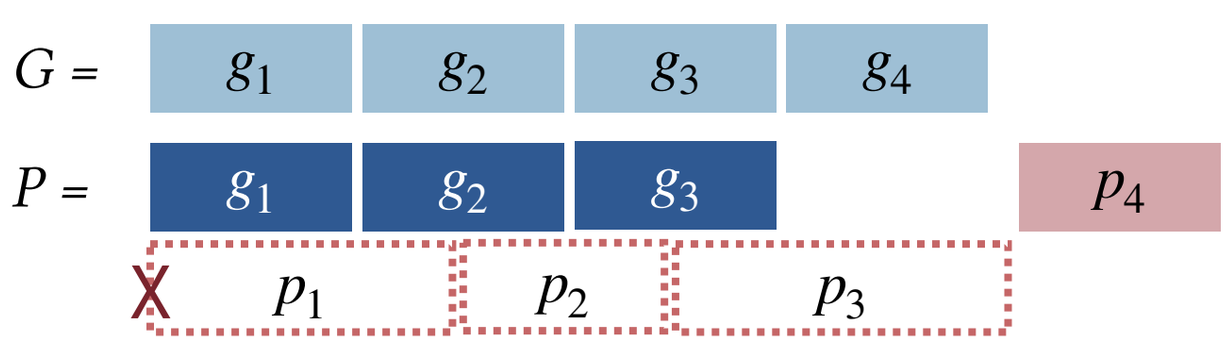
Proof. Since g_1 is guaranteed to finish before (or with) p_1, g_1 is compatible with $p_2 \longrightarrow p_m$. Hence, there is always an optimal solution that starts with the activity that finishes first (g_1).

Observation 2. P is made of p_1 + an optimal solution to the activity selection problem considering only activities that start after the finish time of g_1 and p_1.

Proof. If this is not true, then P is not optimal (contradiction).

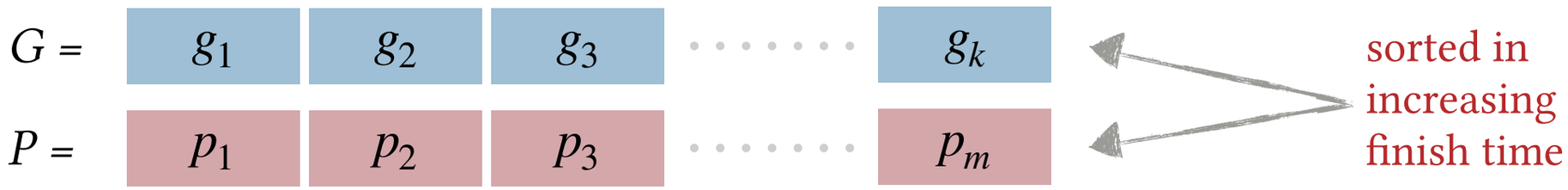
Hence, we can apply the same argument used in Observation 1 to show that the first activity (p_2) in the optimal solution for the new subproblem can be replaced by g_2.

Example.



Activity Selection (Optimality of Greedy Solution)

Assume P is an optimal solution with m activities and G is the greedy solution with k activities. We would like to show that $m = k$ (*i.e. G is as good as the optimal solution*).



Observation 1. We can always replace p_1 in P by g_1 from G.

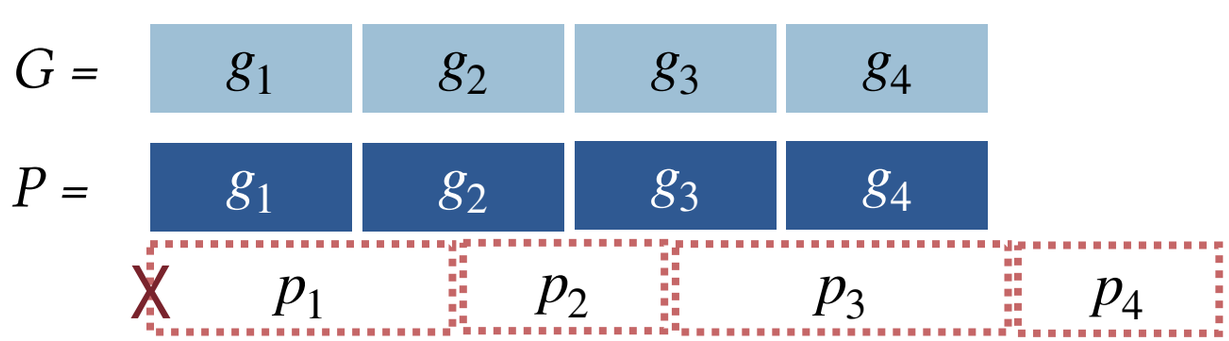
Proof. Since g_1 is guaranteed to finish before (or with) p_1, g_1 is compatible with $p_2 \longrightarrow p_m$. Hence, there is always an optimal solution that starts with the activity that finishes first (g_1).

Observation 2. P is made of p_1 + an optimal solution to the activity selection problem considering only activities that start after the finish time of g_1 and p_1.

Proof. If this is not true, then P is not optimal (contradiction).

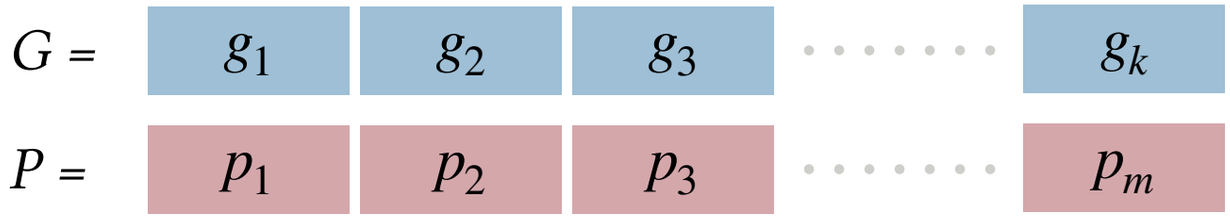
Hence, we can apply the same argument used in Observation 1 to show that the first activity (p_2) in the optimal solution for the new subproblem can be replaced by g_2.

Example.



Activity Selection (Optimality of Greedy Solution)

Assume P is an optimal solution with m activities and G is the greedy solution with k activities. We would like to show that $m = k$ (*i.e. G is as good as the optimal solution*).



Observation 1. We can always replace p_1 in P by g_1 from G.

Proof. Since g_1 is guaranteed to finish before (or with) p_1, g_1 is compatible with $p_2 \longrightarrow p_m$. Hence, there is always an optimal solution that starts with the activity that finishes first (g_1).

Observation 2. P is made of p_1 + an optimal solution to the activity selection problem considering only activities that start after the finish time of g_1 and p_1.

Proof. If this is not true, then P is not optimal (contradiction).

Hence, we can apply the same argument used in Observation 1 to show that the first activity (p_2) in the optimal solution for the new subproblem can be replaced by g_2.

Rolling this out, we conclude that for all $i \leq k$, every p_i can be replaced by g_i.

- If $k = m$, then G is optimal.

Activity Selection (Optimality of Greedy Solution)

Assume P is an optimal solution with m activities and G is the greedy solution with k activities. We would like to show that $m = k$ (*i.e. G is as good as the optimal solution*).

$G =$	g_1	g_2	g_3	.....	g_i	.....	g_k
$P =$	p_1	p_2	p_3	.....	p_m		

If G has more then P is not optimal!

Observation 1. We can always replace p_1 in P by g_1 from G.

Proof. Since g_1 is guaranteed to finish before (or with) p_1, g_1 is compatible with $p_2 \longrightarrow p_m$. Hence, there is always an optimal solution that starts with the activity that finishes first (g_1).

Observation 2. P is made of p_1 + an optimal solution to the activity selection problem considering only activities that start after the finish time of g_1 and p_1.

Proof. If this is not true, then P is not optimal (contradiction).

Hence, we can apply the same argument used in Observation 1 to show that the first activity (p_2) in the optimal solution for the new subproblem can be replaced by g_2.

Rolling this out, we conclude that for all $i \leq k$, every p_i can be replaced by g_i.

- If $k = m$, then G is optimal.
- If $k > m$, then P is not optimal.

Activity Selection (Optimality of Greedy Solution)

Assume P is an optimal solution with m activities and G is the greedy solution with k activities. We would like to show that $m = k$ (*i.e. G is as good as the optimal solution*).

$G =$	g_1	g_2	g_3	·····	g_k	Why didn't G pick p_{i+1}?		
$P =$	p_1	p_2	p_3	·····	p_i	p_{i+1}	·····	p_m

Observation 1. We can always replace p_1 in P by g_1 from G.

Proof. Since g_1 is guaranteed to finish before (or with) p_1, g_1 is compatible with $p_2 \longrightarrow p_m$.

Hence, there is always an optimal solution that starts with the activity that finishes first (g_1).

Observation 2. P is made of p_1 + an optimal solution to the activity selection problem considering only activities that start after the finish time of g_1 and p_1.

Proof. If this is not true, then P is not optimal (contradiction).

Hence, we can apply the same argument used in Observation 1 to show that the first activity (p_2) in the optimal solution for the new subproblem can be replaced by g_2.

Rolling this out, we conclude that for all $i \leq k$, every p_i can be replaced by g_i.

- If $k = m$, then G is optimal.
- If $k > m$, then P is not optimal.
- If $k < m$, then there is an activity that starts after g_k that G can still pick!

Greedy Proofs (General Pattern)

This pattern in proving the optimality of greedy solutions is commonly used:

- Assume that there is an optimal solution P and a greedy solution G.
- Show that we can always *exchange* the first choice in P with the first choice in G without making the solution worse.
- Show that the problem has an *optimal substructure* and thus the same argument applies to the solution of the subproblem after making the first choice.
- Show that P can't be better than G.

Activity Selection (Algorithm)

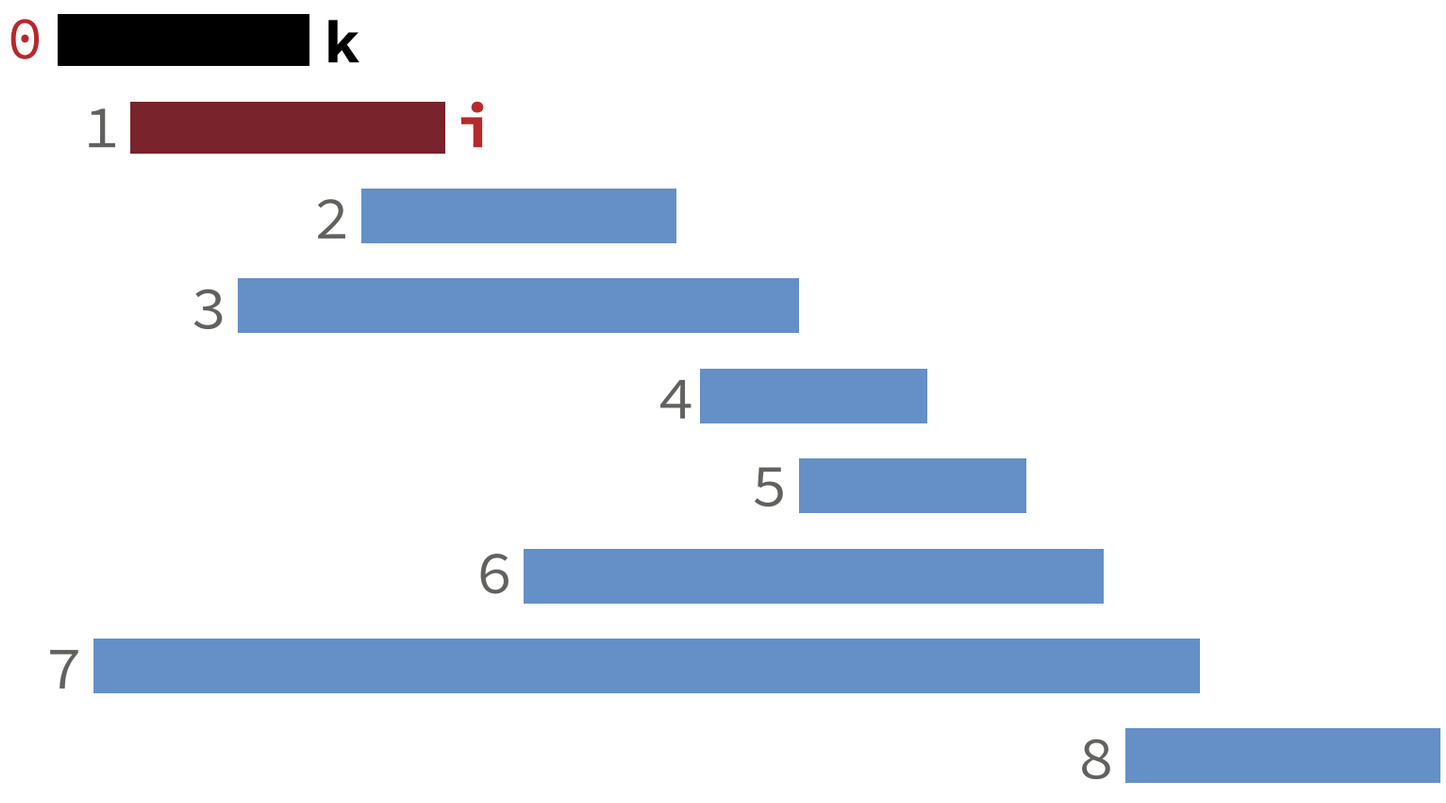
SELECT(s[], f[], n)

```
Sort the activities by
increasing finish time

// Let k = the index of
// the last taken activity
// Let A = the indices of
// the taken activities
k = 0
Add k to A

for i=1 to n-1:
   if s[i] > f[k]:
       k = i
       Add k to A

return A
```



Activity Selection (Algorithm)

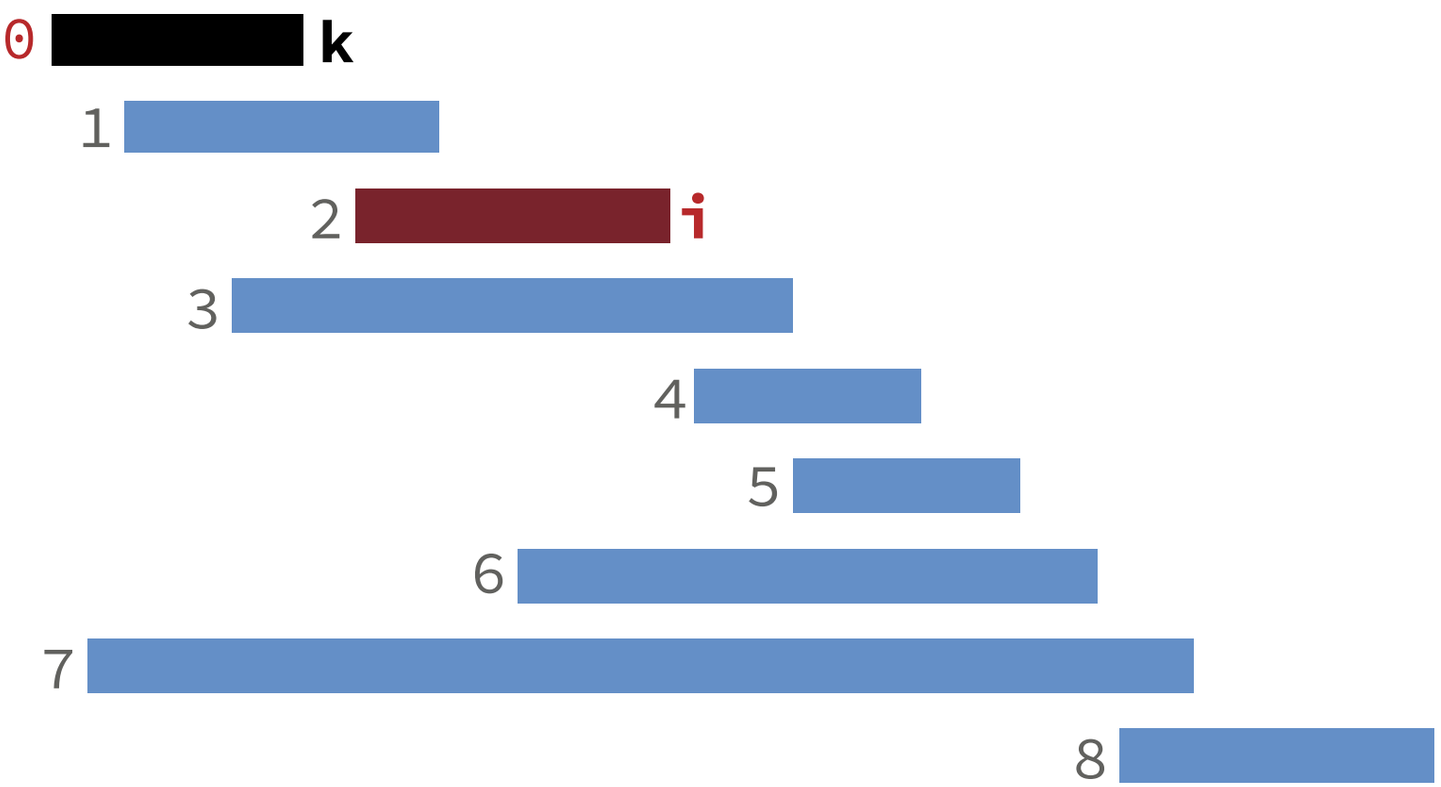
SELECT(s[], f[], n)

```
Sort the activities by
increasing finish time

// Let k = the index of
// the last taken activity
// Let A = the indices of
// the taken activities
k = 0
Add k to A

for i=1 to n-1:
   if s[i] > f[k]:
       k = i
       Add k to A

return A
```



Activity Selection (Algorithm)

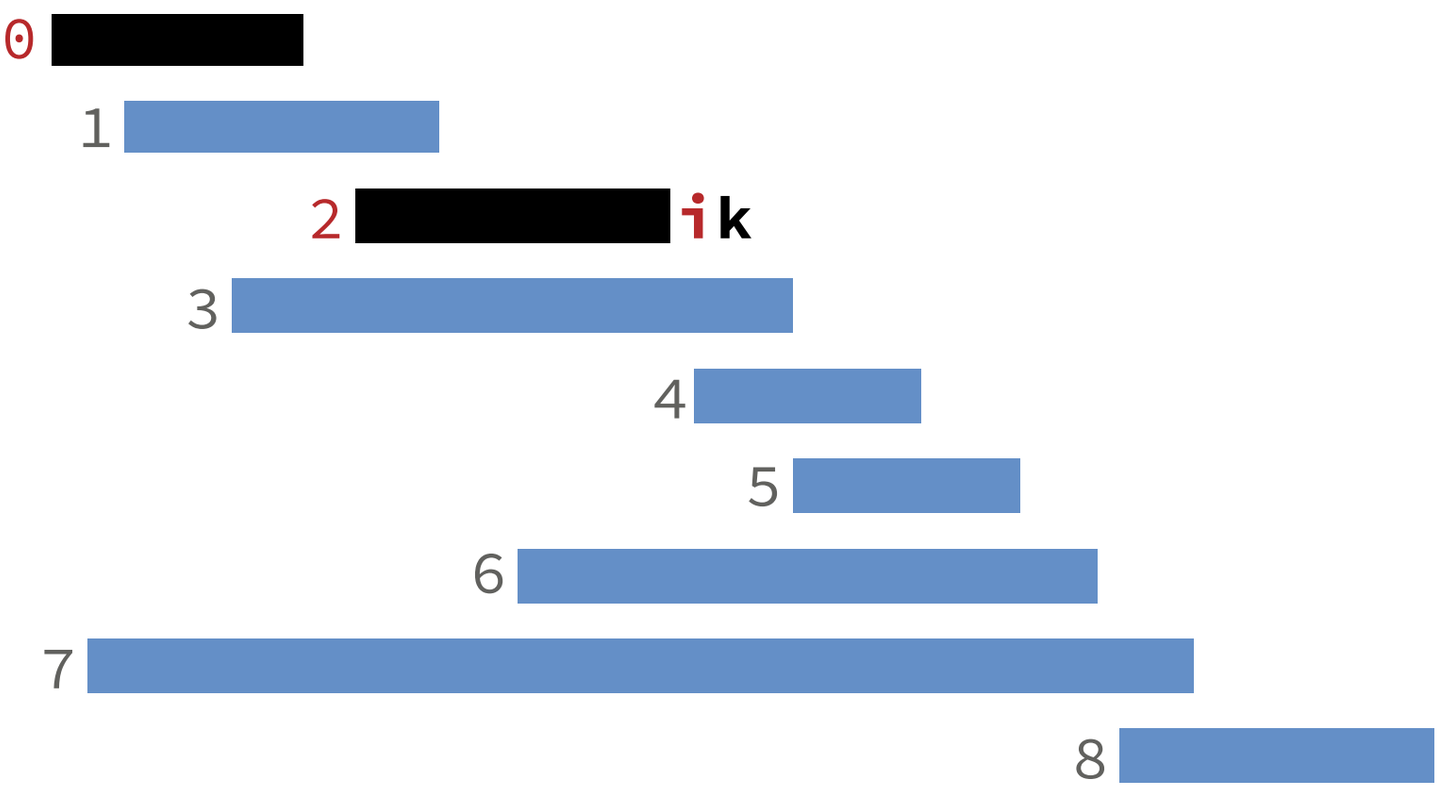
SELECT(s[], f[], n)

```
Sort the activities by
increasing finish time

// Let k = the index of
// the last taken activity
// Let A = the indices of
// the taken activities
k = 0
Add k to A

for i=1 to n-1:
    if s[i] > f[k]:
        k = i
        Add k to A

return A
```



Activity Selection (Algorithm)

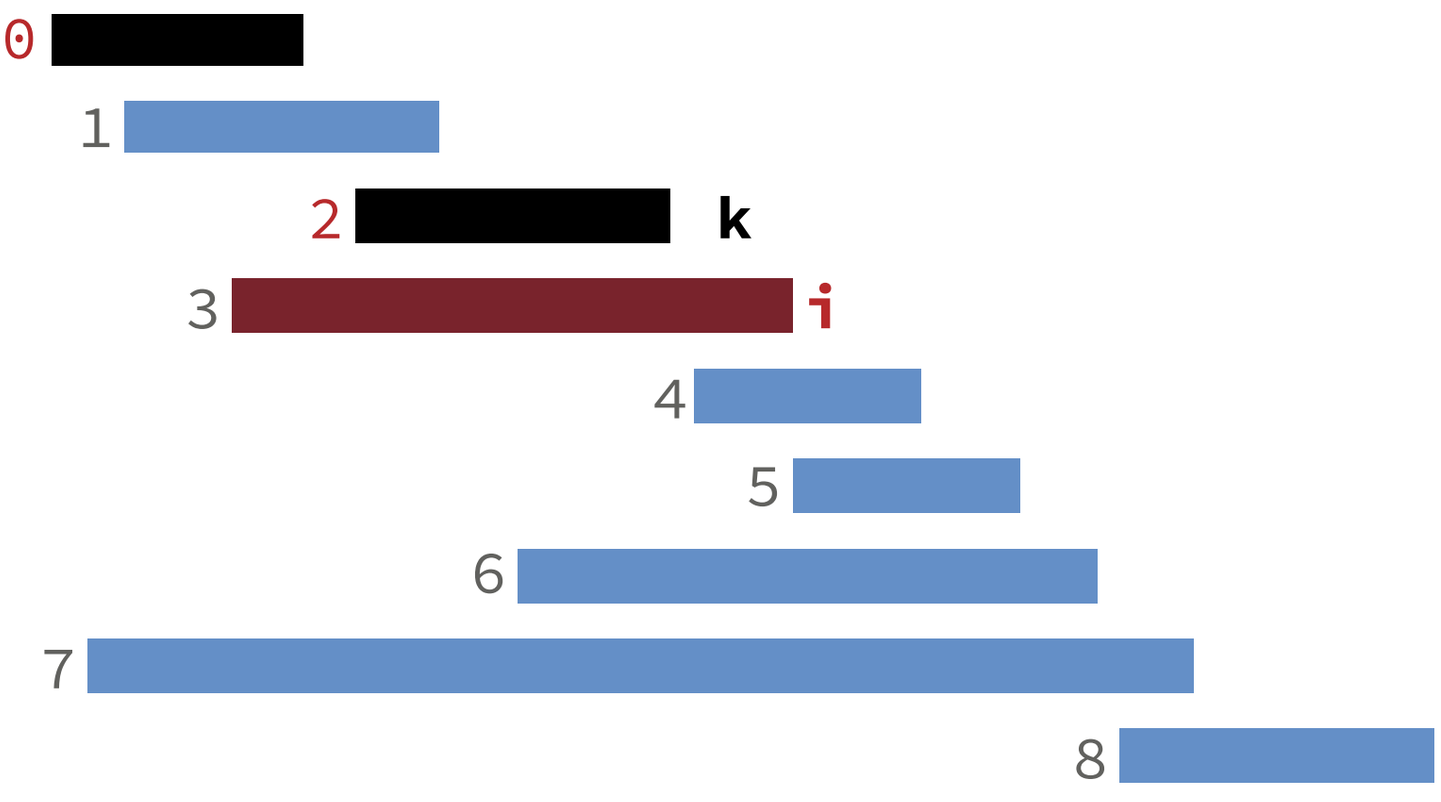
```
SELECT(s[], f[], n)

Sort the activities by
increasing finish time

// Let k = the index of
// the last taken activity
// Let A = the indices of
// the taken activities
k = 0
Add k to A

for i=1 to n-1:
   if s[i] > f[k]:
       k = i
       Add k to A

return A
```



Activity Selection (Algorithm)

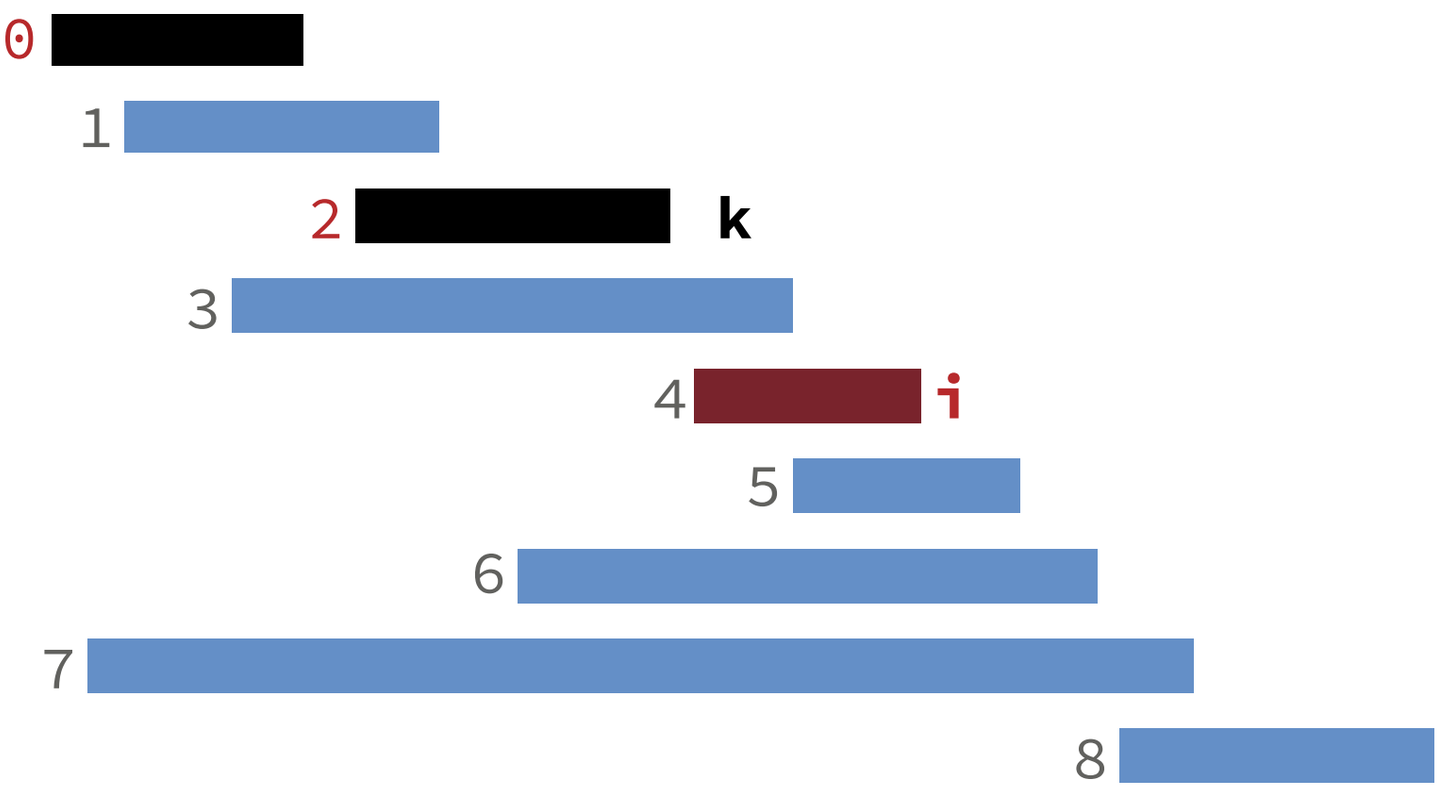
SELECT(s[], f[], n)

```
Sort the activities by
increasing finish time

// Let k = the index of
// the last taken activity
// Let A = the indices of
// the taken activities
k = 0
Add k to A

for i=1 to n-1:
   if s[i] > f[k]:
      k = i
      Add k to A

return A
```



Activity Selection (Algorithm)

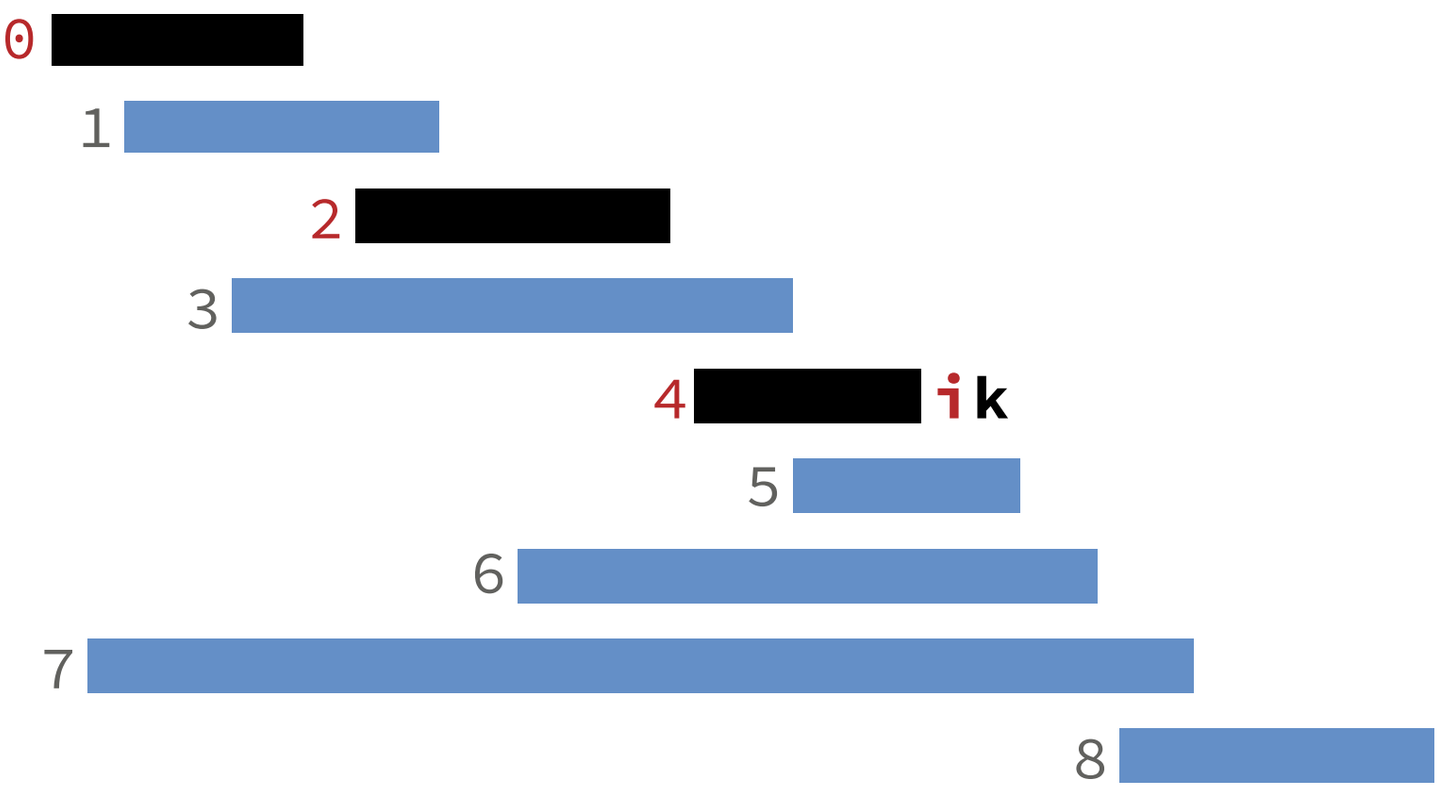
SELECT(s[], f[], n)

```
Sort the activities by
increasing finish time

// Let k = the index of
// the last taken activity
// Let A = the indices of
// the taken activities
k = 0
Add k to A

for i=1 to n-1:
    if s[i] > f[k]:
        k = i
        Add k to A

return A
```



Activity Selection (Algorithm)

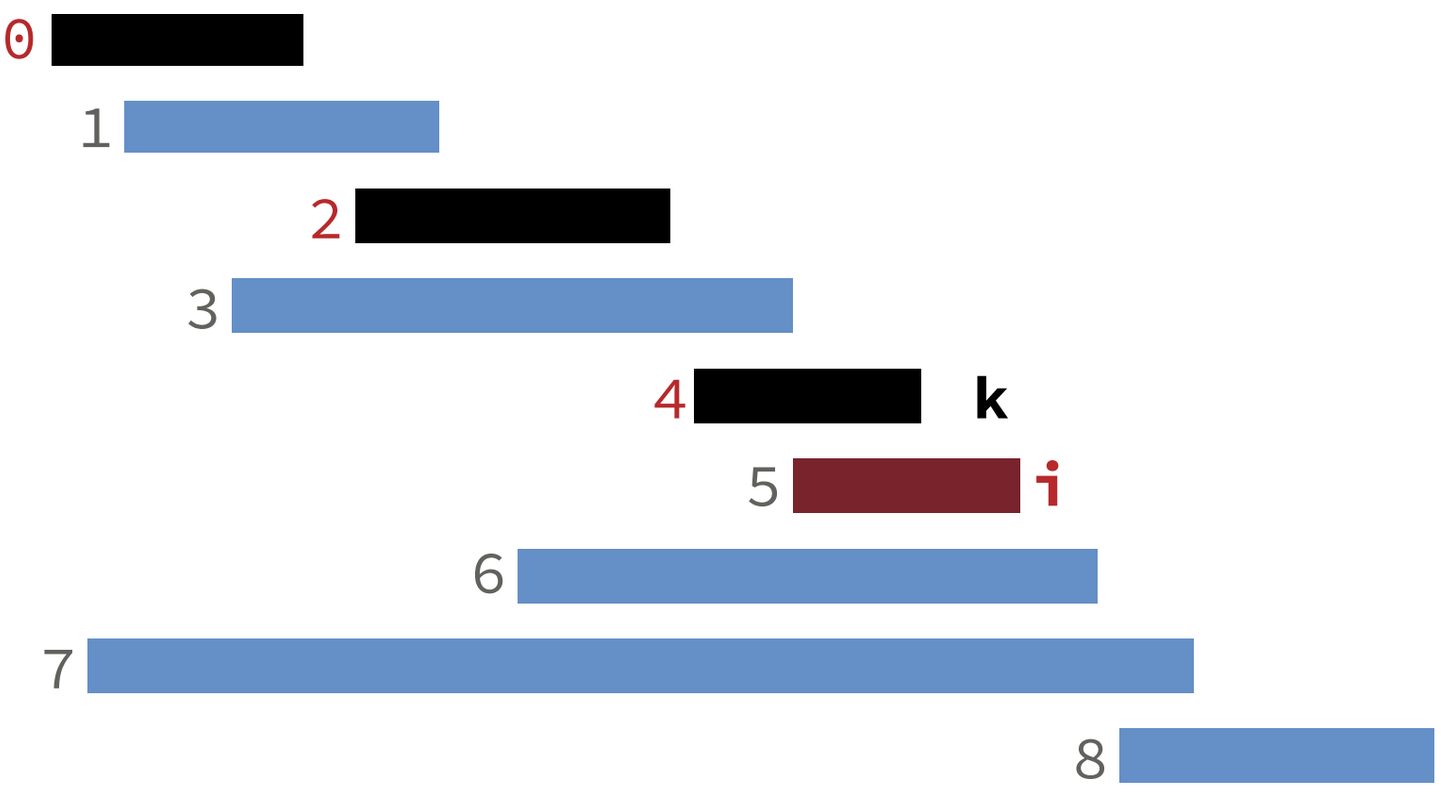
SELECT(s[], f[], n)

```
Sort the activities by
increasing finish time

// Let k = the index of
// the last taken activity
// Let A = the indices of
// the taken activities
k = 0
Add k to A

for i=1 to n-1:
    if s[i] > f[k]:
        k = i
        Add k to A

return A
```



Activity Selection (Algorithm)

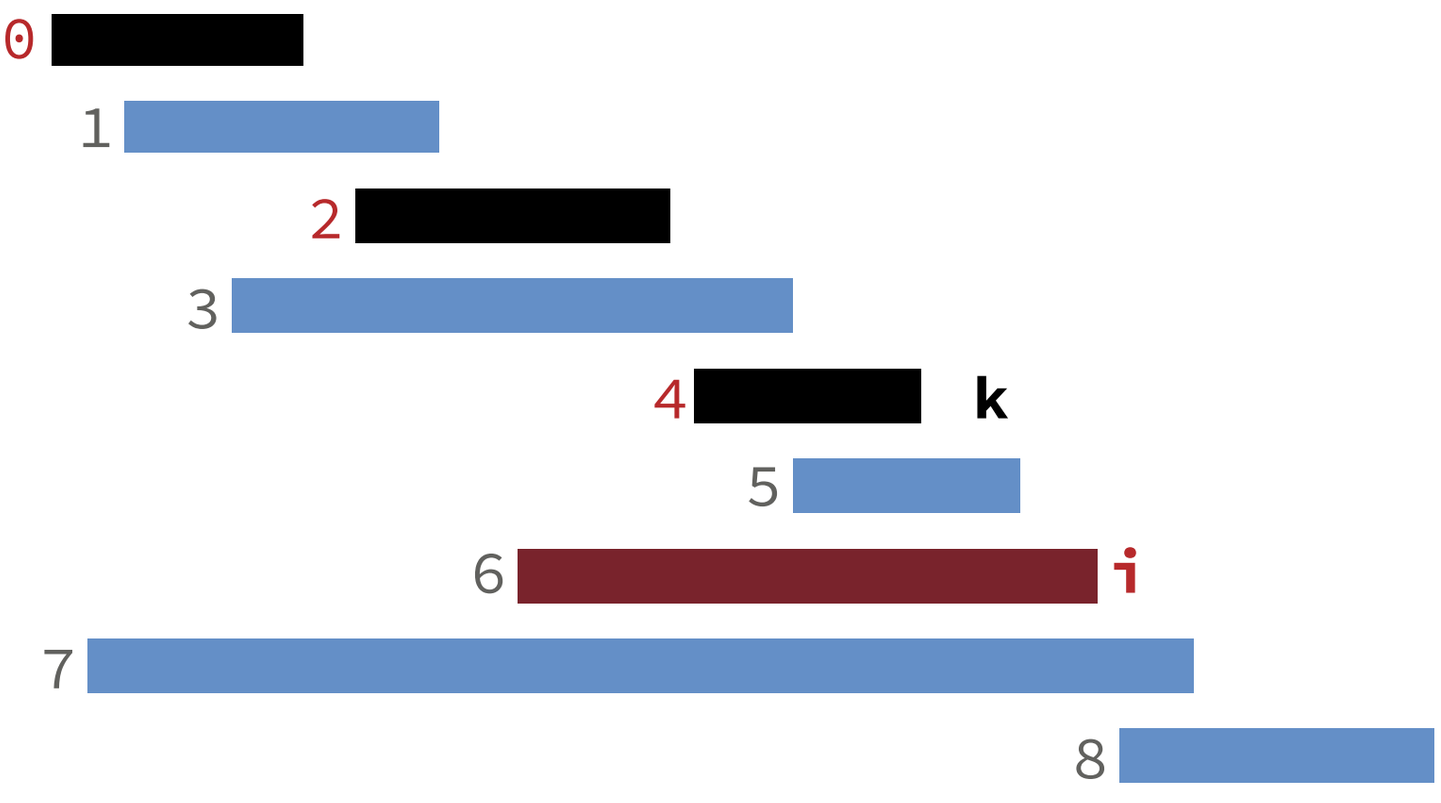
SELECT(s[], f[], n)

```
Sort the activities by
increasing finish time

// Let k = the index of
// the last taken activity
// Let A = the indices of
// the taken activities
k = 0
Add k to A

for i=1 to n-1:
    if s[i] > f[k]:
        k = i
        Add k to A

return A
```



Activity Selection (Algorithm)

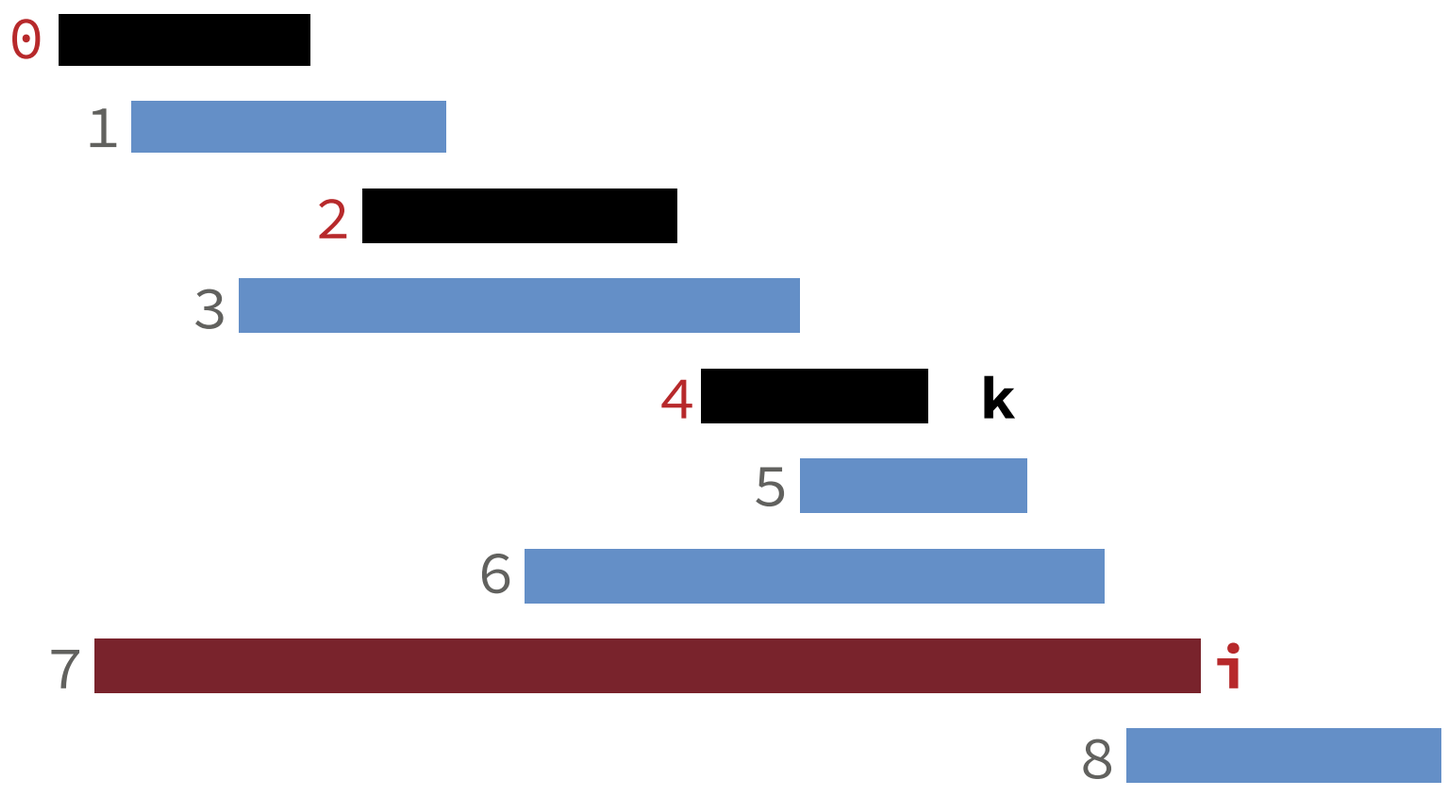
SELECT(s[], f[], n)

```
Sort the activities by
increasing finish time

// Let k = the index of
// the last taken activity
// Let A = the indices of
// the taken activities
k = 0
Add k to A

for i=1 to n-1:
    if s[i] > f[k]:
        k = i
        Add k to A

return A
```



Activity Selection (Algorithm)

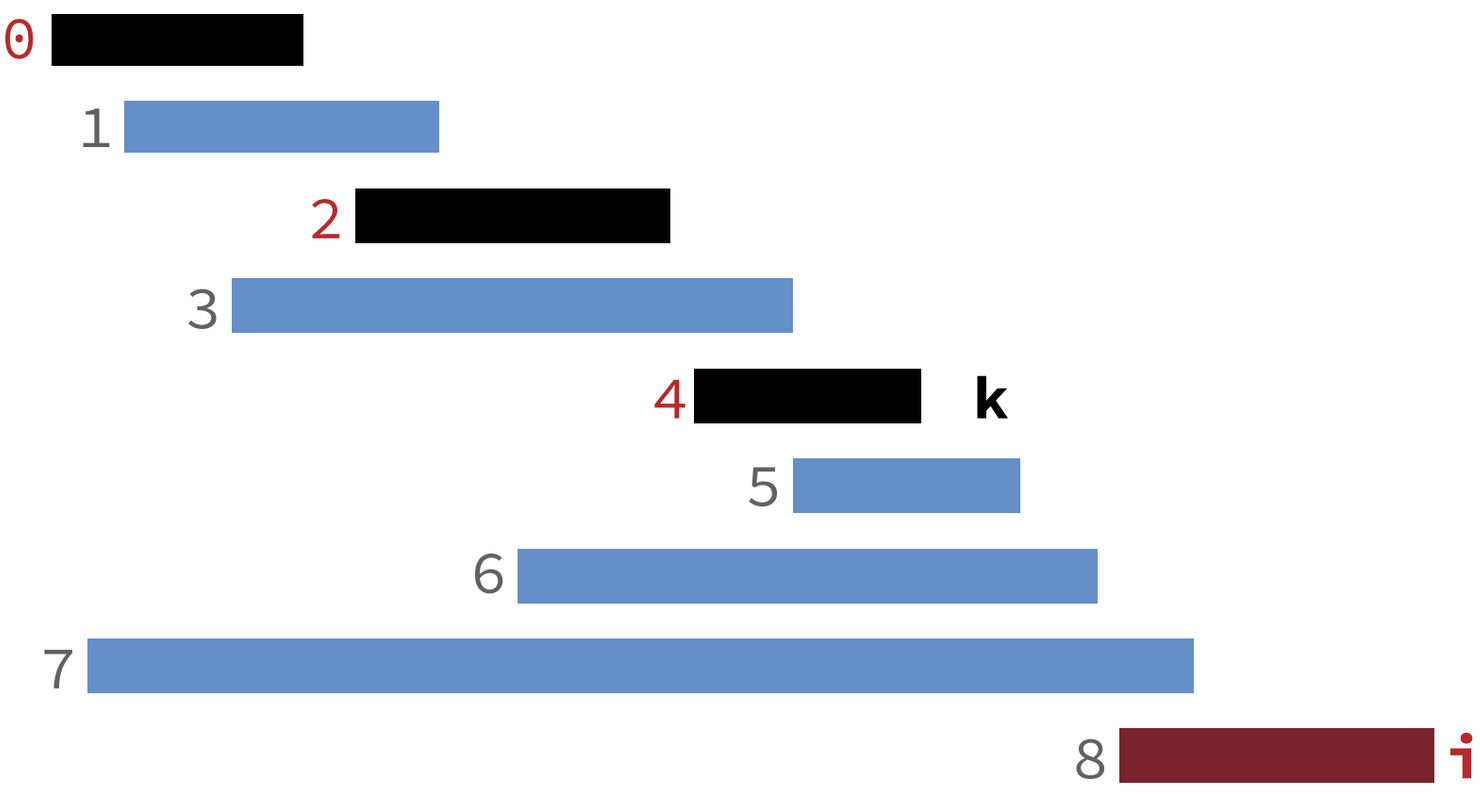
SELECT(s[], f[], n)

```
Sort the activities by
increasing finish time

// Let k = the index of
// the last taken activity
// Let A = the indices of
// the taken activities
k = 0
Add k to A

for i=1 to n-1:
    if s[i] > f[k]:
        k = i
        Add k to A

return A
```



Activity Selection (Algorithm)

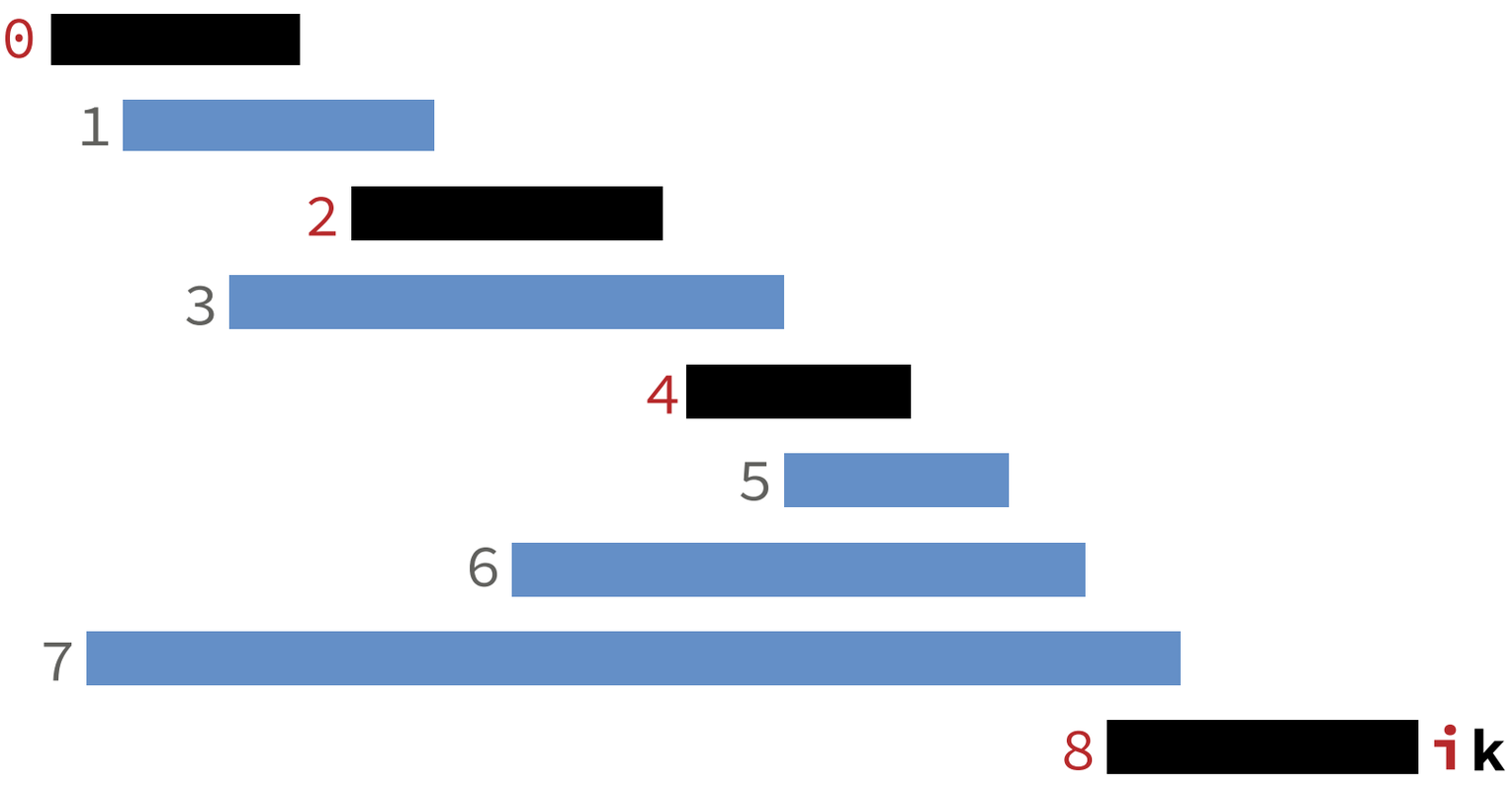
SELECT(s[], f[], n)

```
Sort the activities by
increasing finish time

// Let k = the index of
// the last taken activity
// Let A = the indices of
// the taken activities
k = 0
Add k to A

for i=1 to n-1:
    if s[i] > f[k]:
        k = i
        Add k to A

return A
```



Activity Selection (Algorithm)

```
SELECT(s[], f[], n)

Sort the activities by
increasing finish time

// Let k = the index of
// the last taken activity
// Let A = the indices of
// the taken activities
k = 0
Add k to A

for i=1 to n-1:
   if s[i] > f[k]:
       k = i
       Add k to A

return A
```

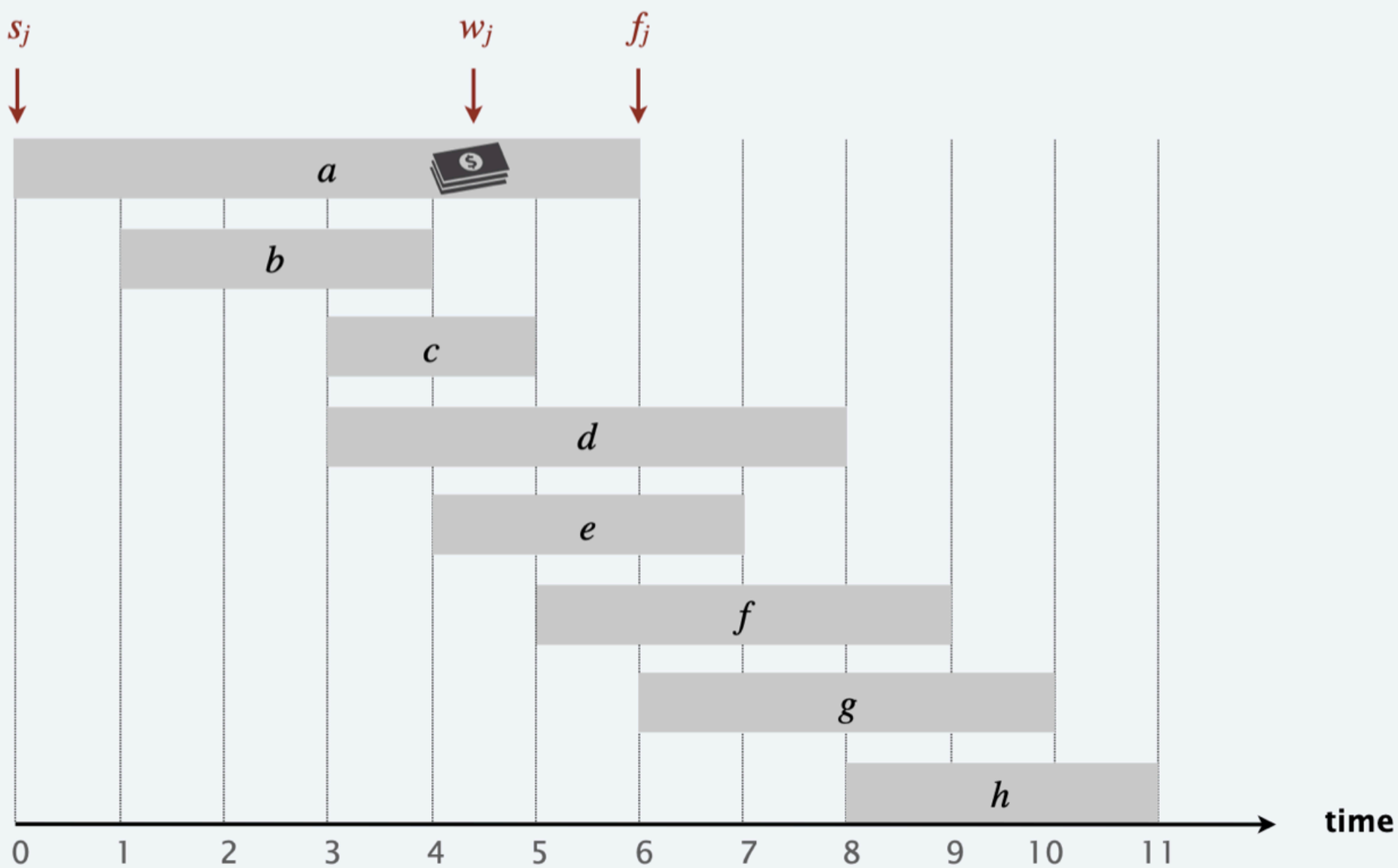
$\Theta(n \log n)$

$\Theta(n)$

Activity Selection: A Variant

Weighted Activity Selection

- Job j starts at s_j, finishes at f_j, and has weight $w_j > 0$.
- Two jobs are compatible if they don't overlap.
- Goal: find max-weight subset of mutually compatible jobs.



slide by Kevin Wayne

Exercise

Find a counterexample for each of the following greedy strategies

a. Highest value first

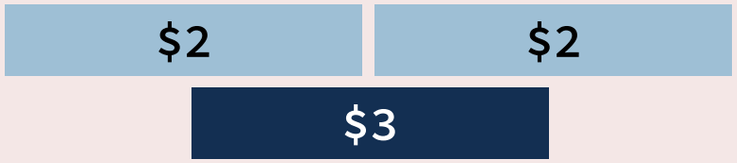
b. Highest value per time unit first

c. Earliest finish time first

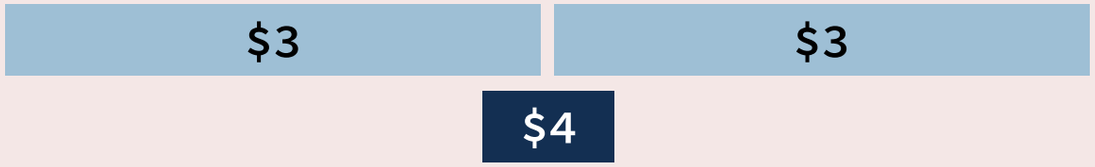
Exercise

Find a counterexample for each of the following greedy strategies

a. Highest value first



b. Highest value per time unit first

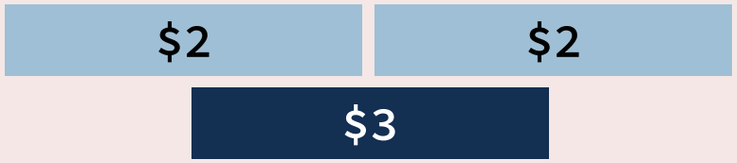


c. Earliest finish time first

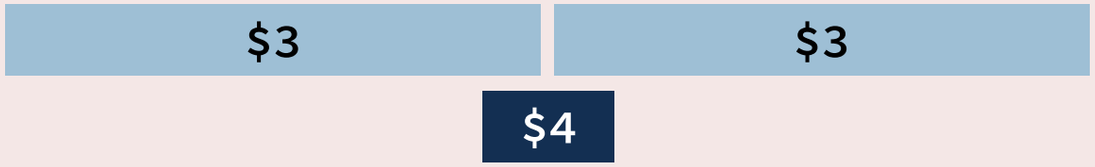
Exercise

Find a counterexample for each of the following greedy strategies

a. Highest value first



b. Highest value per time unit first



c. Earliest finish time first

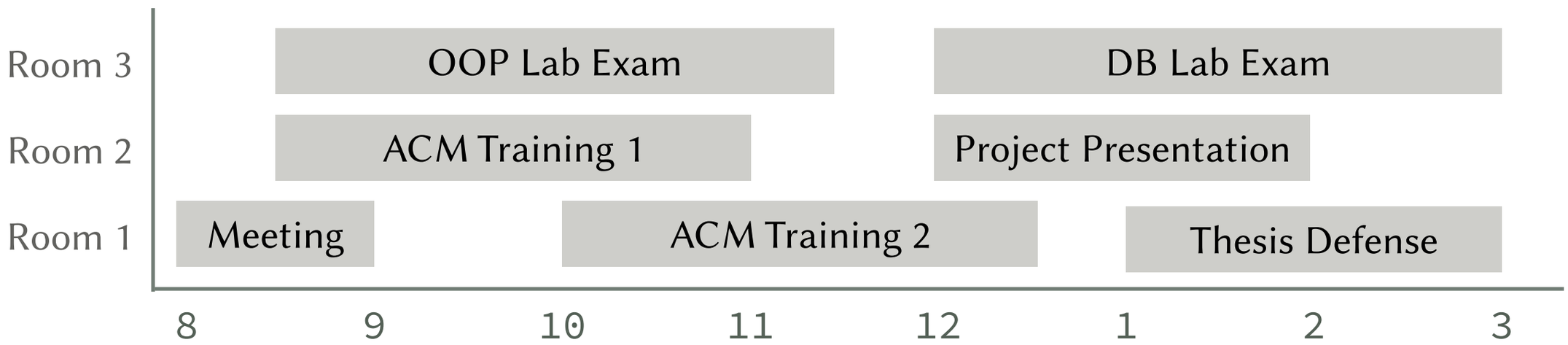
There is no known optimal greedy strategy for the weighted activity selection problem

We will learn a **dynamic programming** solution for the problem later!

Interval Partitioning

Problem. Given n activities each requiring an exclusive use of a room, find the *minimum number of rooms* such that no two activities taking place in the same room have overlapping times.

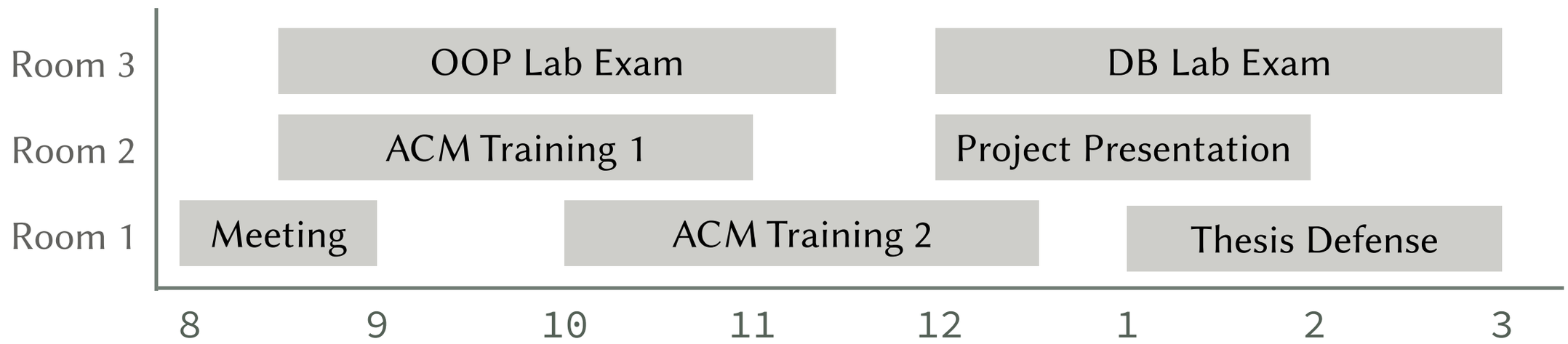
Example. At least three rooms are needed for the following activities.



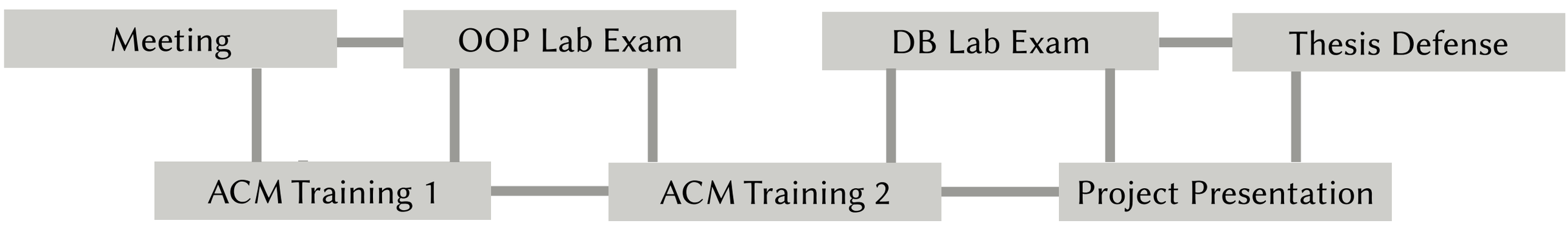
Interval Partitioning

Problem. Given n activities each requiring an exclusive use of a room, find the *minimum number of rooms* such that no two activities taking place in the same room have overlapping times.

Example. At least three rooms are needed for the following activities.



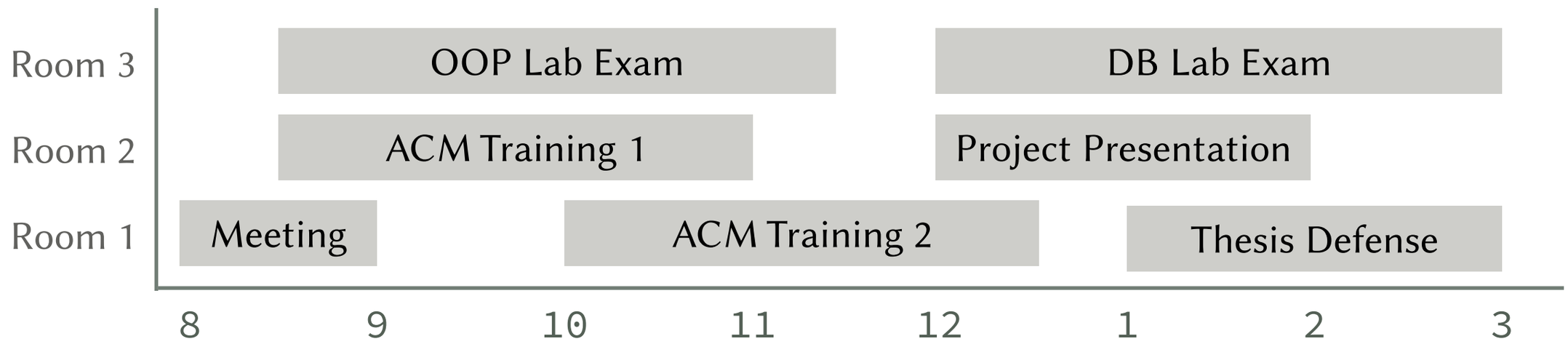
Another Formulation. Color the activities such that activities with links have different colors. What is the minimum number of colors needed?



Interval Partitioning (AKA Interval Graph Coloring)

Problem. Given n activities each requiring an exclusive use of a room, find the *minimum number of rooms* such that no two activities taking place in the same room have overlapping times.

Example. At least three rooms are needed for the following activities.



Another Formulation. Color the activities such that activities with links have different colors. What is the minimum number of colors needed?

Interval Partitioning: Greedy Algorithm

Algorithm Sketch.

- The greedy algorithm **orders** the activities based on some strategy and then processes activities one by one.

Interval Partitioning: Greedy Algorithm

Algorithm Sketch.

- The greedy algorithm **orders** the activities based on some strategy and then processes activities one by one.
- Each activity is assigned to the **first room that can take it**.

Interval Partitioning: Greedy Algorithm

Algorithm Sketch.

- The greedy algorithm **orders** the activities based on some strategy and then processes activities one by one.
- Each activity is assigned to the **first room that can take it**.
- If no room can take the activity, a new room is opened.

Interval Partitioning: Greedy Algorithm

Algorithm Sketch.

- The greedy algorithm **orders** the activities based on some strategy and then processes activities one by one.
- Each activity is assigned to the **first room that can take it**.
- If no room can take the activity, a new room is opened.

```
ASSIGN-ROOMS(a[])

Sort the activities in a[] using some criterion
Let rooms[] be an array of lists
```

Note. This pseudocode is meant to explain the idea of the greedy algorithm only. The actual implementation of the algorithm will be different.

Interval Partitioning: Greedy Algorithm

Algorithm Sketch.

- The greedy algorithm **orders** the activities based on some strategy and then processes activities one by one.
- Each activity is assigned to the **first room that can take it**.
- If no room can take the activity, a new room is opened.

```
ASSIGN-ROOMS(a[])

Sort the activities in a[] using some criterion
Let rooms[] be an array of lists

for i=0 to a.size-1:  ← for each activity
```

Note. This pseudocode is meant to explain the idea of the greedy algorithm only. The actual implementation of the algorithm will be different.

Interval Partitioning: Greedy Algorithm

Algorithm Sketch.

- The greedy algorithm **orders** the activities based on some strategy and then processes activities one by one.
- Each activity is assigned to the **first room that can take it**.
- If no room can take the activity, a new room is opened.

```
ASSIGN-ROOMS(a[])

Sort the activities in a[] using some criterion
Let rooms[] be an array of lists

for i=0 to a.size-1:
   for r=0 to rooms.size-1:  ← for each room
```

Note. This pseudocode is meant to explain the idea of the greedy algorithm only. The actual implementation of the algorithm will be different.

Interval Partitioning: Greedy Algorithm

Algorithm Sketch.

- The greedy algorithm **orders** the activities based on some strategy and then processes activities one by one.
- Each activity is assigned to the **first room that can take it**.
- If no room can take the activity, a new room is opened.

```
ASSIGN-ROOMS(a[])

Sort the activities in a[] using some criterion
Let rooms[] be an array of lists

for i=0 to a.size-1:
   for r=0 to rooms.size-1:
      if a[i] does not overlap with
         any activity in rooms[r]:
         break
```

Note. This pseudocode is meant to explain the idea of the greedy algorithm only. The actual implementation of the algorithm will be different.

Interval Partitioning: Greedy Algorithm

Algorithm Sketch.

- The greedy algorithm **orders** the activities based on some strategy and then processes activities one by one.
- Each activity is assigned to the **first room that can take it**.
- If no room can take the activity, a new room is opened.

ASSIGN-ROOMS(a[])

```
Sort the activities in a[] using some criterion
Let rooms[] be an array of lists

for i=0 to a.size-1:
   for r=0 to rooms.size-1:
      if a[i] does not overlap with
         any activity in rooms[r]:
         break
   Add a[i] to rooms[r]
```

Assuming r = a.size if the break statement does not execute

Note. This pseudocode is meant to explain the idea of the greedy algorithm only. The actual implementation of the algorithm will be different.

Interval Partitioning: Unsuccessful Greedy Strategies

Example

Optimal

Greedy Strategy 1. Shortest duration first.

Interval Partitioning: Unsuccessful Greedy Strategies

Example

Optimal

Greedy Strategy 1. Shortest duration first.

Interval Partitioning: Unsuccessful Greedy Strategies

Example

Optimal

Greedy Strategy 1. Shortest duration first.

Interval Partitioning: Unsuccessful Greedy Strategies

Example

Optimal

Greedy Strategy 1. Shortest duration first.

Interval Partitioning: Unsuccessful Greedy Strategies

Example

Optimal

Greedy Strategy 1. Shortest duration first.

Interval Partitioning: Unsuccessful Greedy Strategies

Example

Optimal

Greedy Strategy 1. Shortest duration first.

Greedy Strategy 2. Earliest finish time first.

Interval Partitioning: Unsuccessful Greedy Strategies

Example

Optimal

Greedy Strategy 1. Shortest duration first.

Greedy Strategy 2. Earliest finish time first.

Interval Partitioning: Unsuccessful Greedy Strategies

Example

Optimal

Greedy Strategy 1. Shortest duration first.

Greedy Strategy 2. Earliest finish time first.

Interval Partitioning: Unsuccessful Greedy Strategies

Example

Optimal

Greedy Strategy 1. Shortest duration first.

Greedy Strategy 2. Earliest finish time first.

This could have been colored red also, but according to our algorithm, the first fitting room is picked.

Interval Partitioning: Unsuccessful Greedy Strategies

Example

Optimal

Greedy Strategy 1. Shortest duration first.

Greedy Strategy 2. Earliest finish time first.

Interval Partitioning: Unsuccessful Greedy Strategies

Example

Optimal

Greedy Strategy 1. Shortest duration first.

Greedy Strategy 2. Earliest finish time first.

Greedy Strategy 3. Fewest conflicts first.

Interval Partitioning: Unsuccessful Greedy Strategies

Example

Optimal

Greedy Strategy 1. Shortest duration first.

Greedy Strategy 2. Earliest finish time first.

Greedy Strategy 3. Fewest conflicts first.

Interval Partitioning: Unsuccessful Greedy Strategies

Example

Optimal

Greedy Strategy 1. Shortest duration first.

Greedy Strategy 2. Earliest finish time first.

Greedy Strategy 3. Fewest conflicts first.

Interval Partitioning: Unsuccessful Greedy Strategies

Example

Optimal

Greedy Strategy 1. Shortest duration first.

Greedy Strategy 2. Earliest finish time first.

Greedy Strategy 3. Fewest conflicts first.

Interval Partitioning: Unsuccessful Greedy Strategies

Example

Optimal

Greedy Strategy 1. Shortest duration first.

Greedy Strategy 2. Earliest finish time first.

Greedy Strategy 3. Fewest conflicts first.

Interval Partitioning: Unsuccessful Greedy Strategies

Example

Optimal

Greedy Strategy 1. Shortest duration first.

Greedy Strategy 2. Earliest finish time first.

Greedy Strategy 3. Fewest conflicts first.

Interval Partitioning: Unsuccessful Greedy Strategies

Example

Optimal

Greedy Strategy 1. Shortest duration first.

Greedy Strategy 2. Earliest finish time first.

Greedy Strategy 3. Fewest conflicts first.

Interval Partitioning: A successful Greedy Strategy

Example

Optimal

Greedy Strategy 4. Earliest start time first.

Interval Partitioning: A successful Greedy Strategy

Example

Optimal

Greedy Strategy 4. Earliest start time first.

Interval Partitioning: A successful Greedy Strategy

Example

Optimal

Greedy Strategy 4. Earliest start time first.

Interval Partitioning: A successful Greedy Strategy

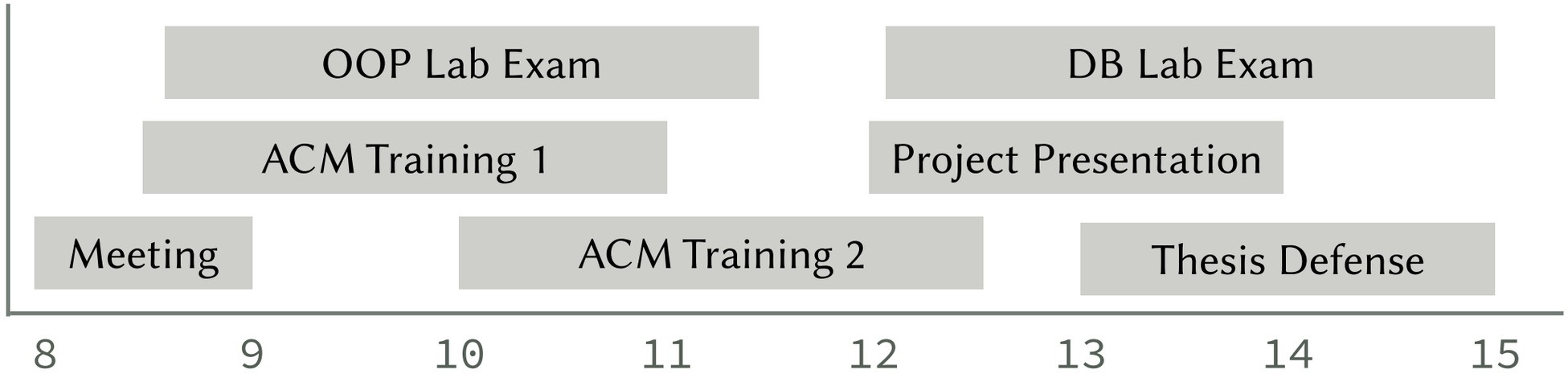
Example

Optimal

Greedy Strategy 4. Earliest start time first.

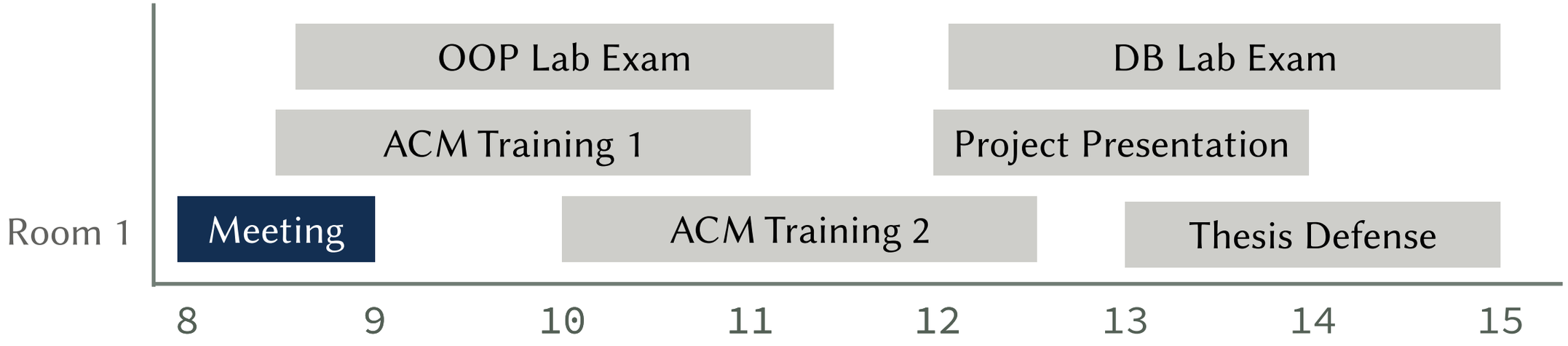
Interval Partitioning: A successful Greedy Strategy

Greedy Strategy 4. Earliest start time first.



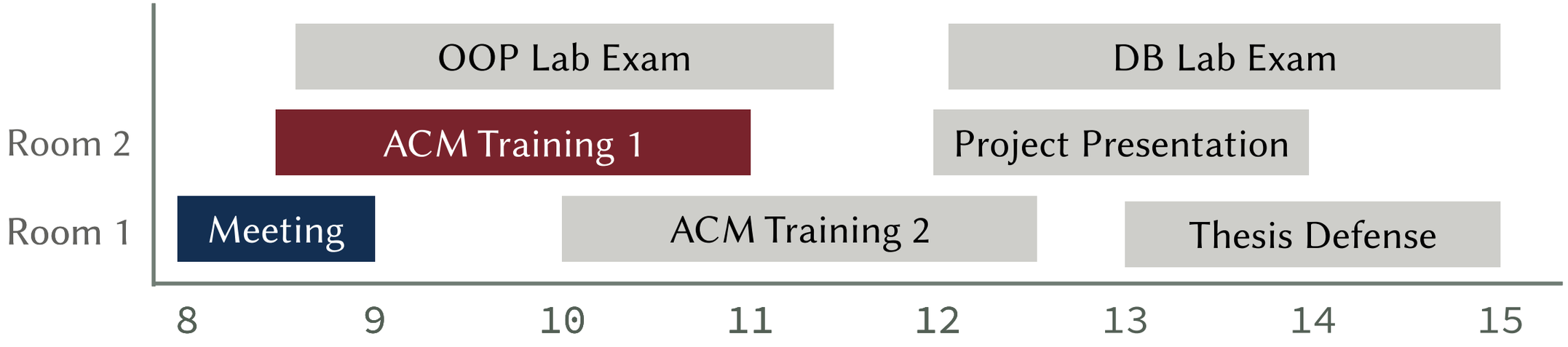
Interval Partitioning: A successful Greedy Strategy

Greedy Strategy 4. Earliest start time first.



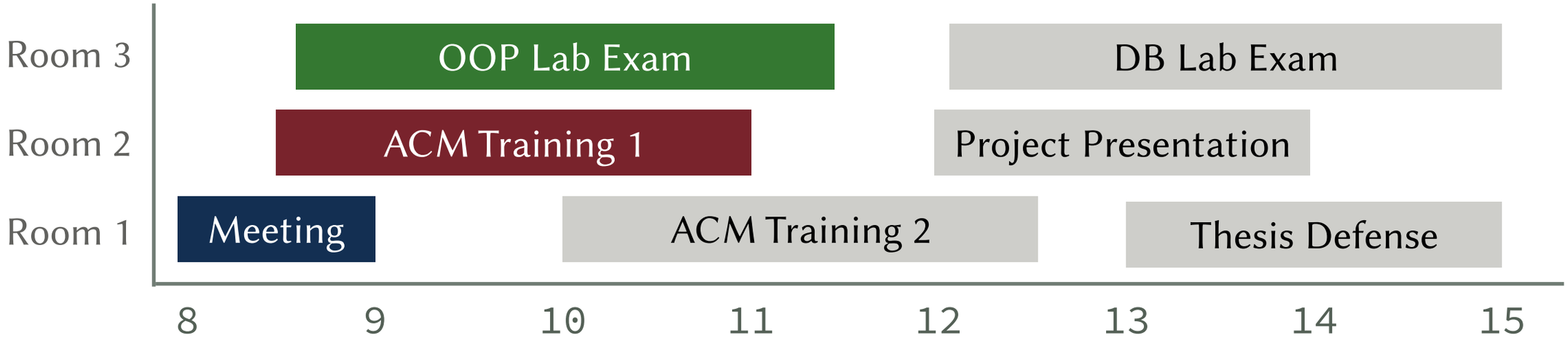
Interval Partitioning: A successful Greedy Strategy

Greedy Strategy 4. Earliest start time first.



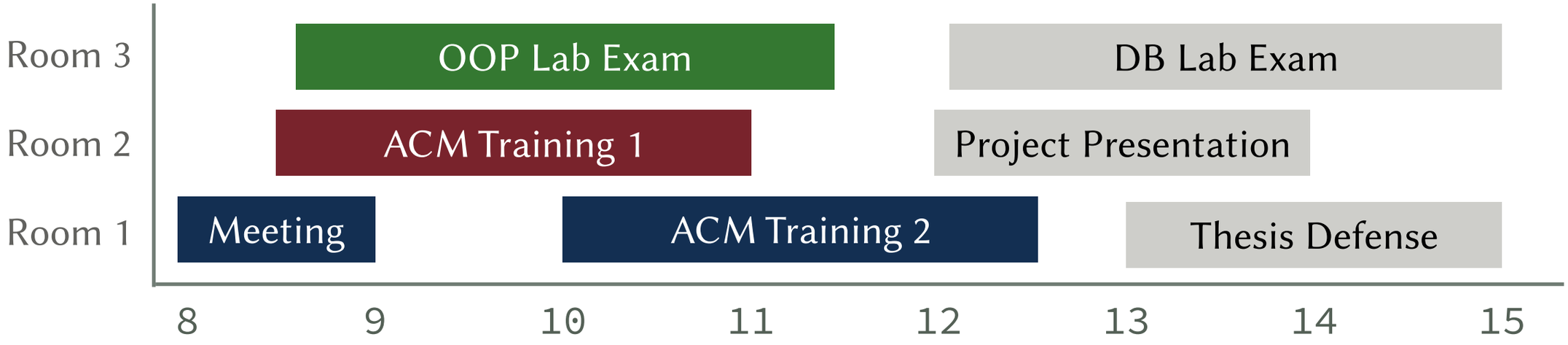
Interval Partitioning: A successful Greedy Strategy

Greedy Strategy 4. Earliest start time first.



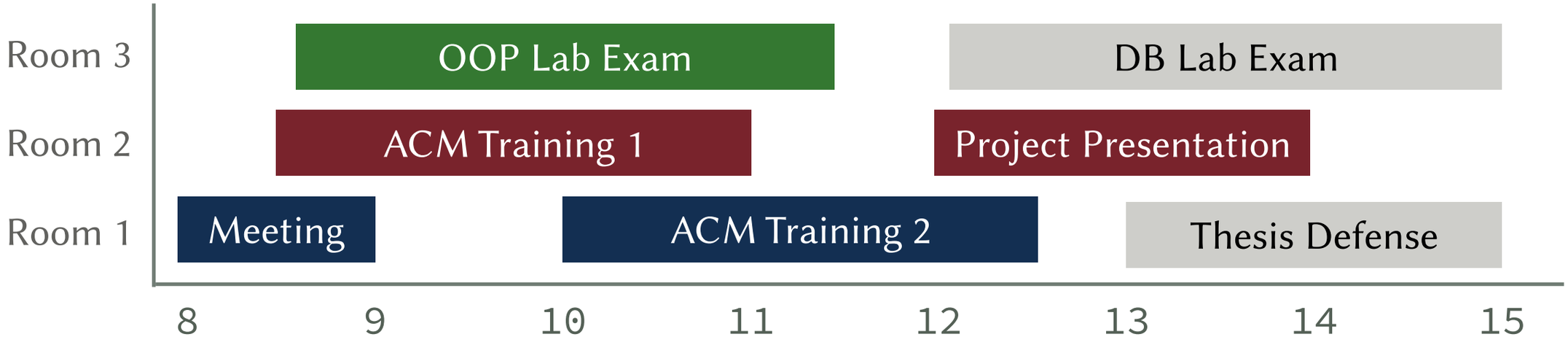
Interval Partitioning: A successful Greedy Strategy

Greedy Strategy 4. Earliest start time first.



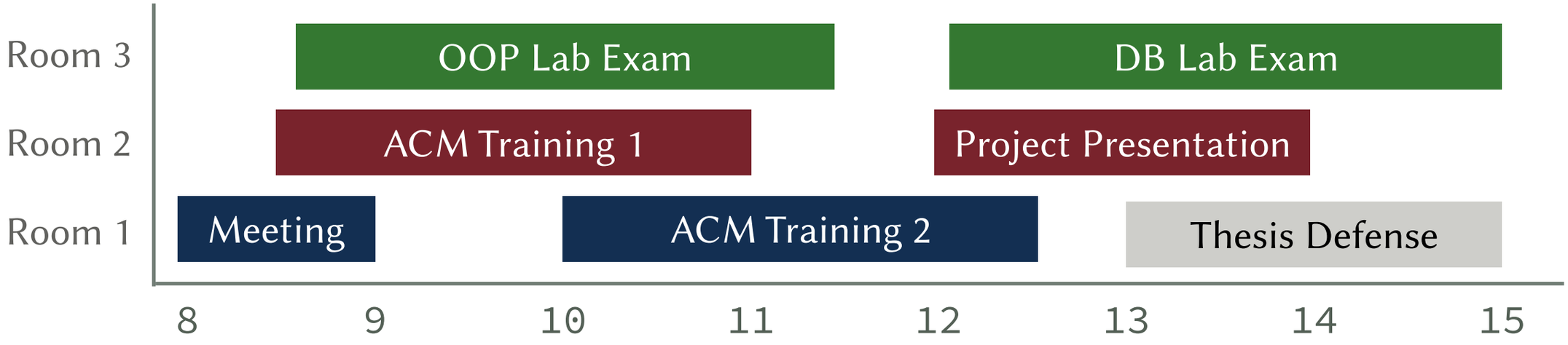
Interval Partitioning: A successful Greedy Strategy

Greedy Strategy 4. Earliest start time first.



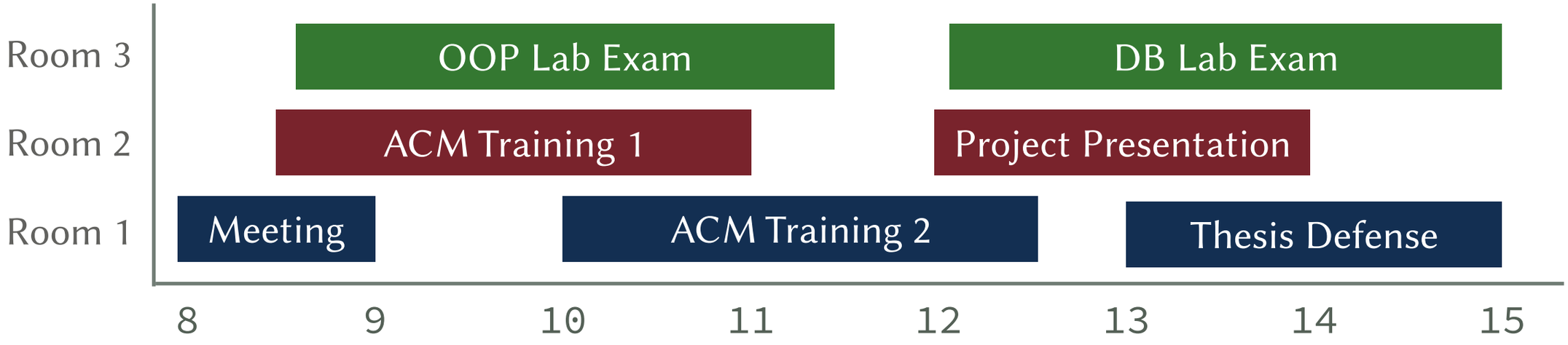
Interval Partitioning: A successful Greedy Strategy

Greedy Strategy 4. Earliest start time first.



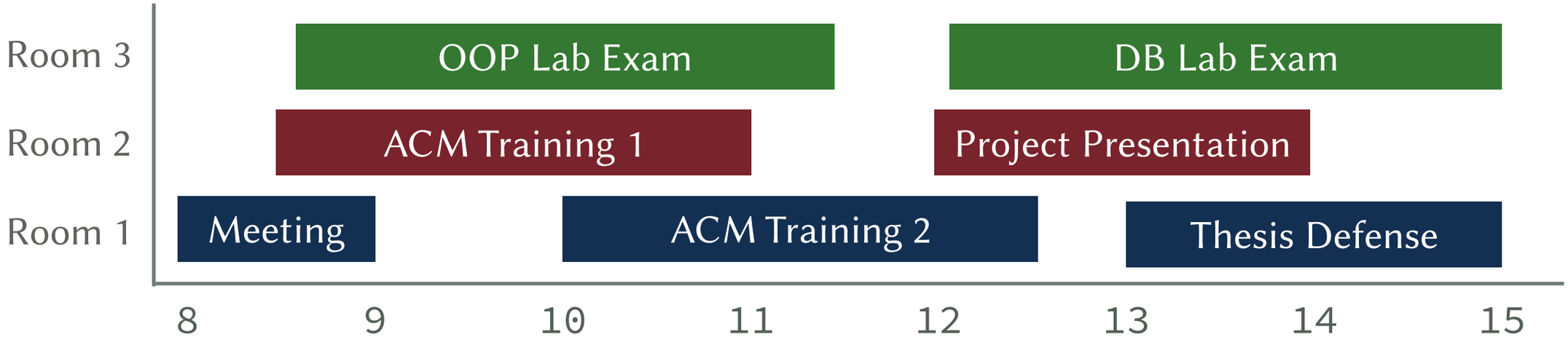
Interval Partitioning: A successful Greedy Strategy

Greedy Strategy 4. Earliest start time first.



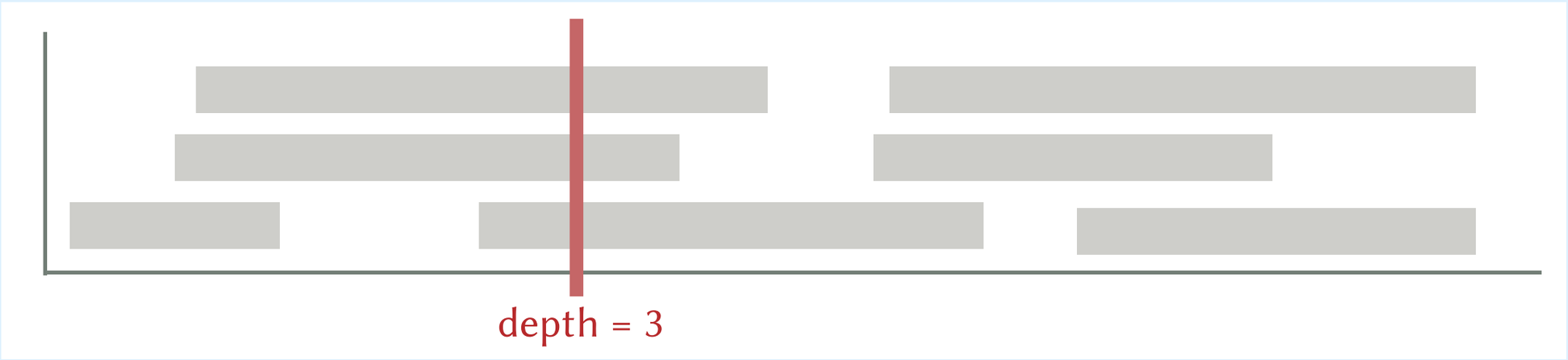
Interval Partitioning: A successful Greedy Strategies

Greedy Strategy 4. Earliest start time first.



Depth. Maximum number of activities at a certain point in time.

key Observation. Number of classrooms needed ≥ depth.

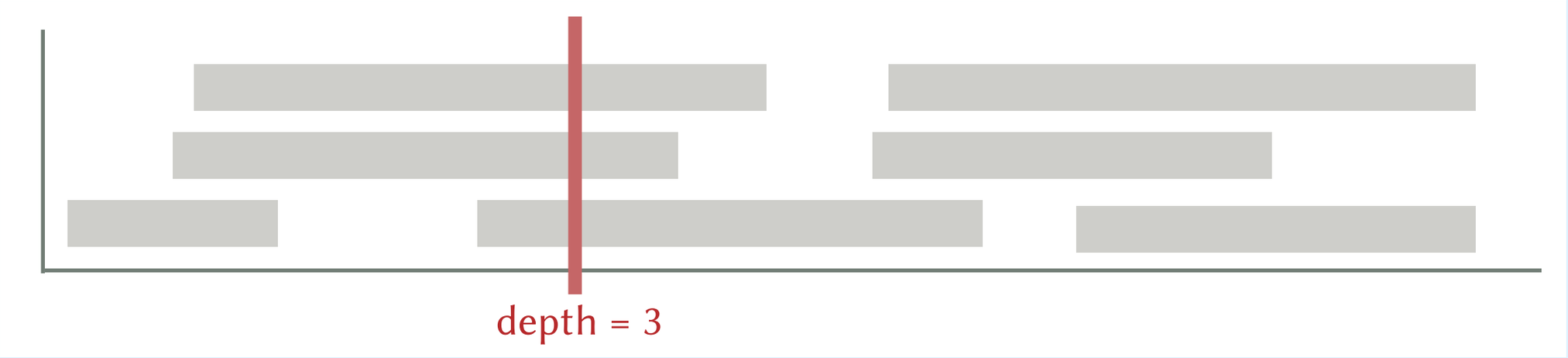


Interval Partitioning: Earliest Start Time Correctness

Observation. The greedy algorithm does not schedule overlapping activities in the same room.

Depth. Maximum number of activities at a certain point in time.

key Observation. Number of classrooms needed ≥ depth.



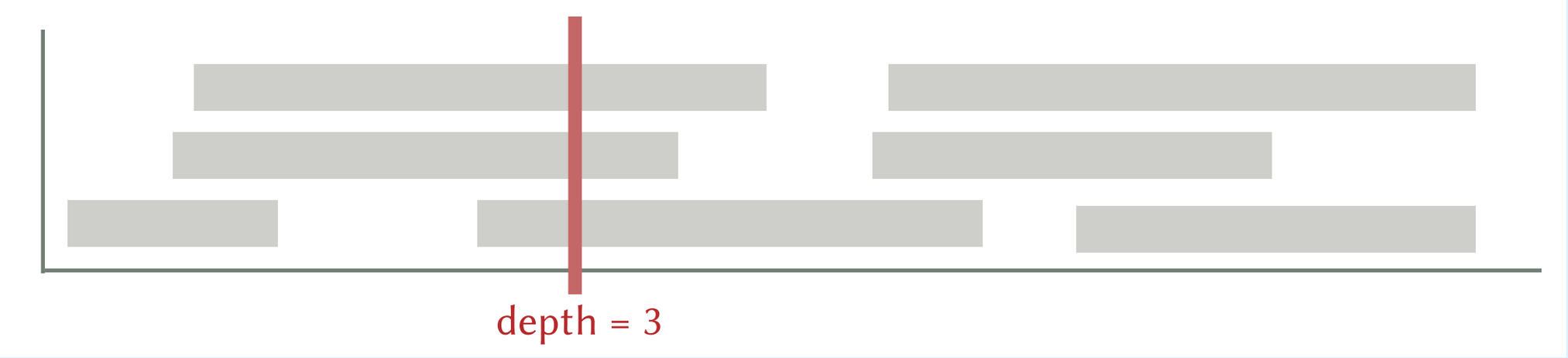
Interval Partitioning: Earliest Start Time Correctness

Observation. The greedy algorithm does not schedule overlapping activities in the same room.

Claim. The number of rooms used by the earliest-start-time strategy = **depth**.

Depth. Maximum number of activities at a certain point in time.

key Observation. Number of classrooms needed ≥ depth.

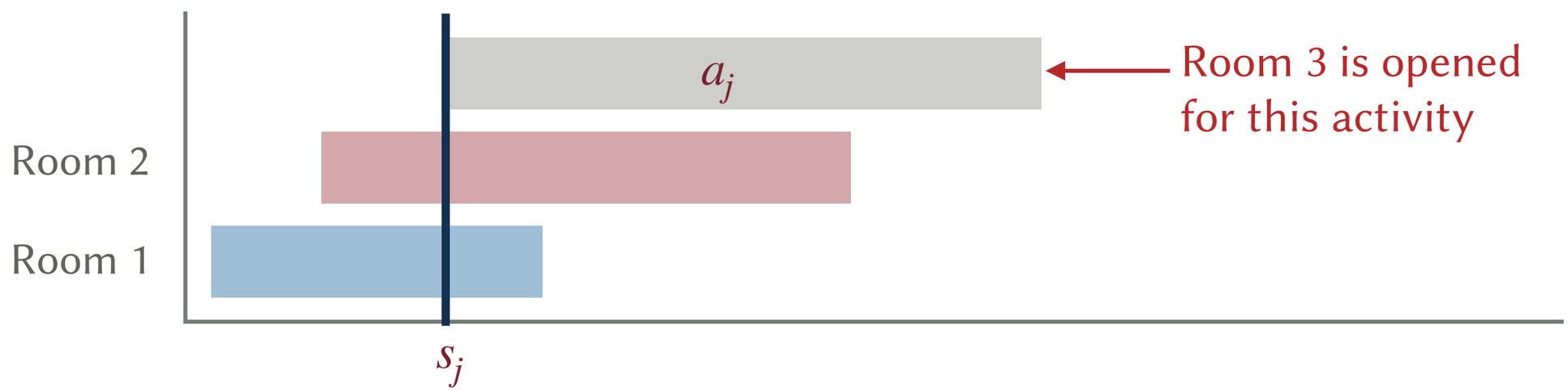


Interval Partitioning: Earliest Start Time Correctness

Observation. The greedy algorithm does not schedule overlapping activities in the same room.

Claim. The number of rooms used by the earliest-start-time strategy = **depth**.

Proof. The algorithm opens a new room k for a new activity a_j with start time s_j only if :



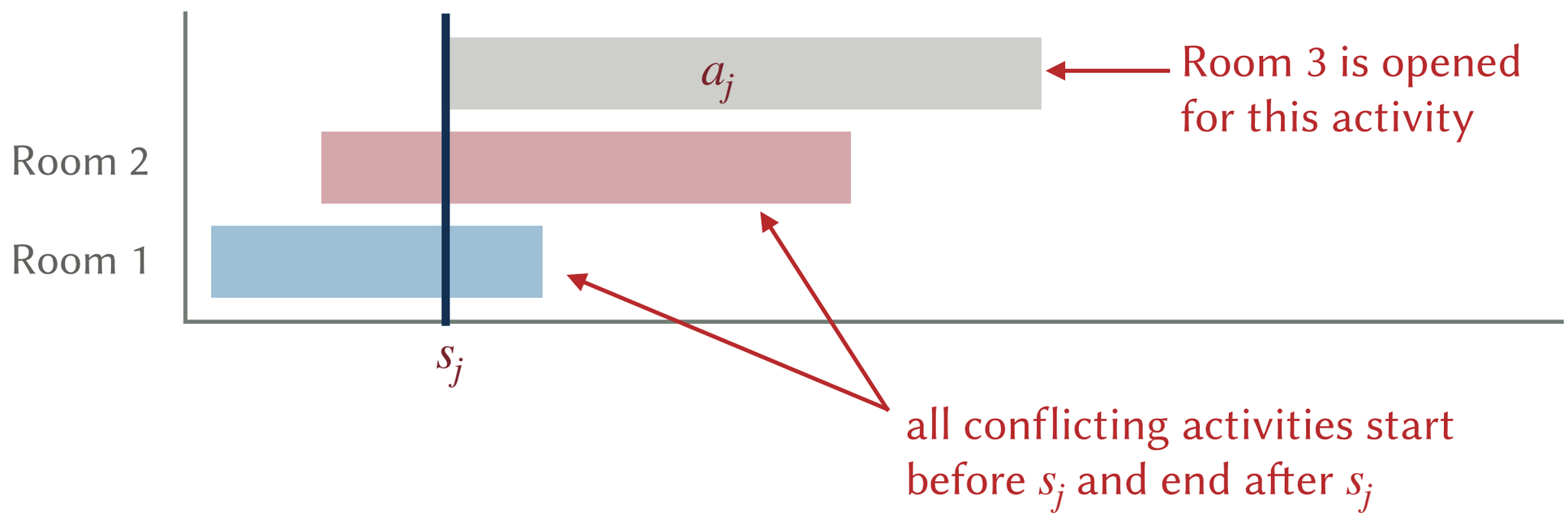
Interval Partitioning: Earliest Start Time Correctness

Observation. The greedy algorithm does not schedule overlapping activities in the same room.

Claim. The number of rooms used by the earliest-start-time strategy = **depth**.

Proof. The algorithm opens a new room k for a new activity a_j with start time s_j only if :

- Each of the $k - 1$ open rooms contains at least one activity that:
  - starts before s_j (because of the earliest-start-time strategy)
  - ends after s_j (because there is a conflict)



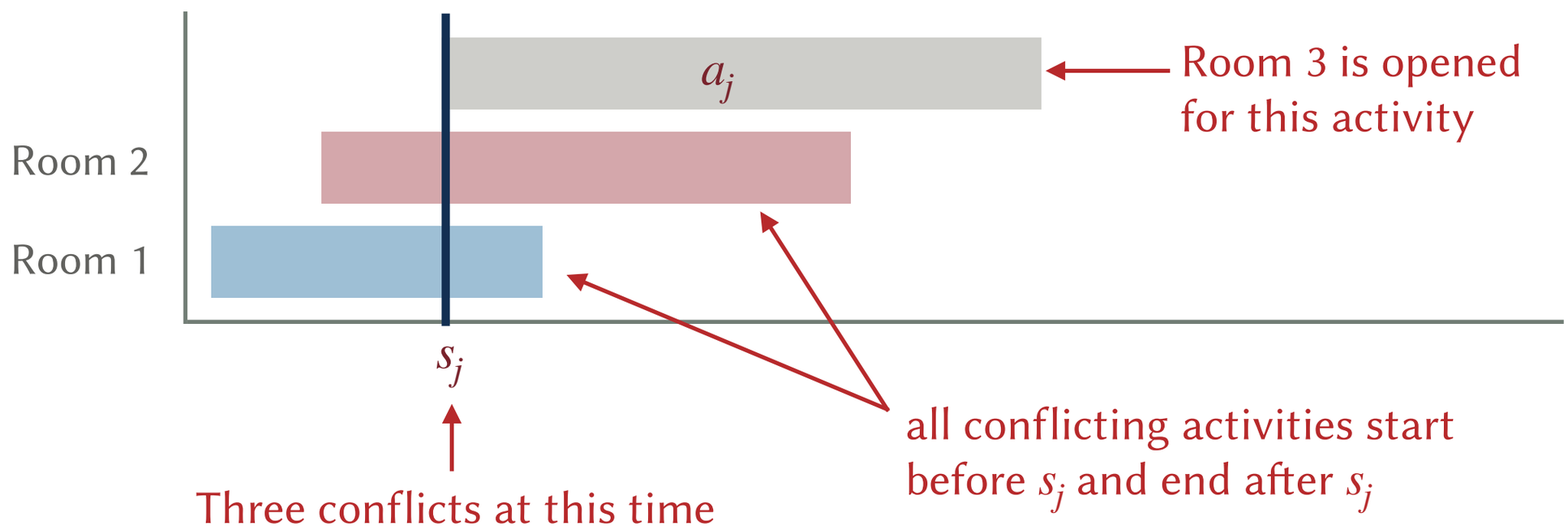
Interval Partitioning: Earliest Start Time Correctness

Observation. The greedy algorithm does not schedule overlapping activities in the same room.

Claim. The number of rooms used by the earliest-start-time strategy = **depth**.

Proof. The algorithm opens a new room k for a new activity a_j with start time s_j only if :

- Each of the $k-1$ open rooms contains at least one activity that:
  - starts before s_j (because of the earliest-start-time strategy)
  - ends after s_j (because there is a conflict)
- There are k conflicts at s_j and thus having less than k rooms is impossible.



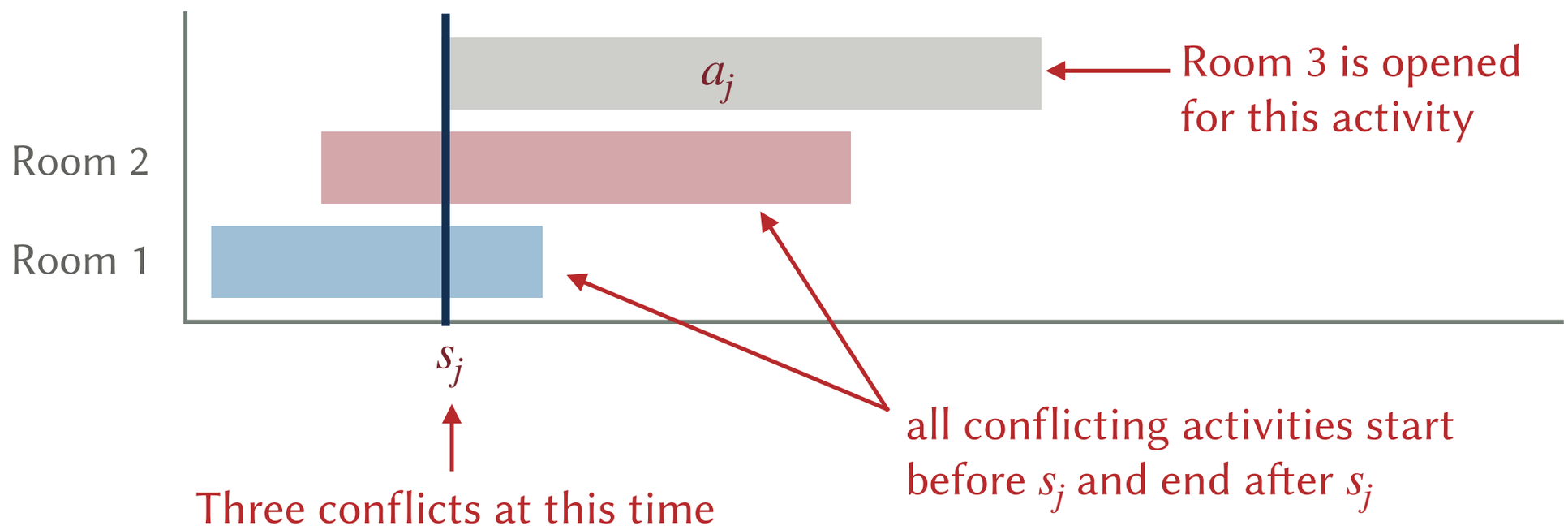
Interval Partitioning: Earliest Start Time Correctness

Observation. The greedy algorithm does not schedule overlapping activities in the same room.

Claim. The number of rooms used by the earliest-start-time strategy = **depth**.

Proof. The algorithm opens a new room k for a new activity a_j with start time s_j only if :

- Each of the $k-1$ open rooms contains at least one activity that:
  - starts before s_j (because of the earliest-start-time strategy)
  - ends after s_j (because there is a conflict)
- There are k conflicts at s_j and thus having less than k rooms is impossible.

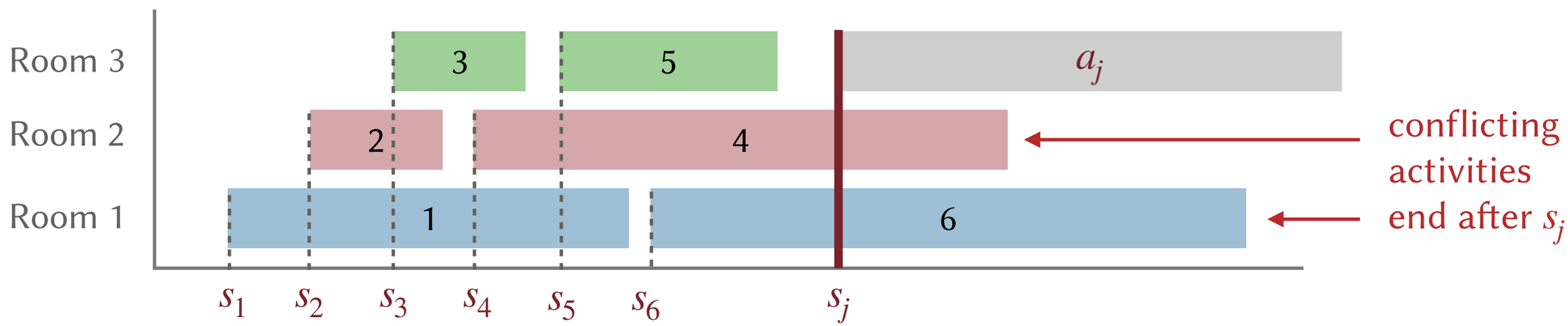


Note. Other greedy strategies do not guarantee that the k conflicts (that caused room k to open) are all at the same point in time.

Interval Partitioning: Efficient Implementation

Observation. To check if a new activity a_j conflicts with any of the activities in a room, we only need to compare s_j to the latest finish time in the room, because:

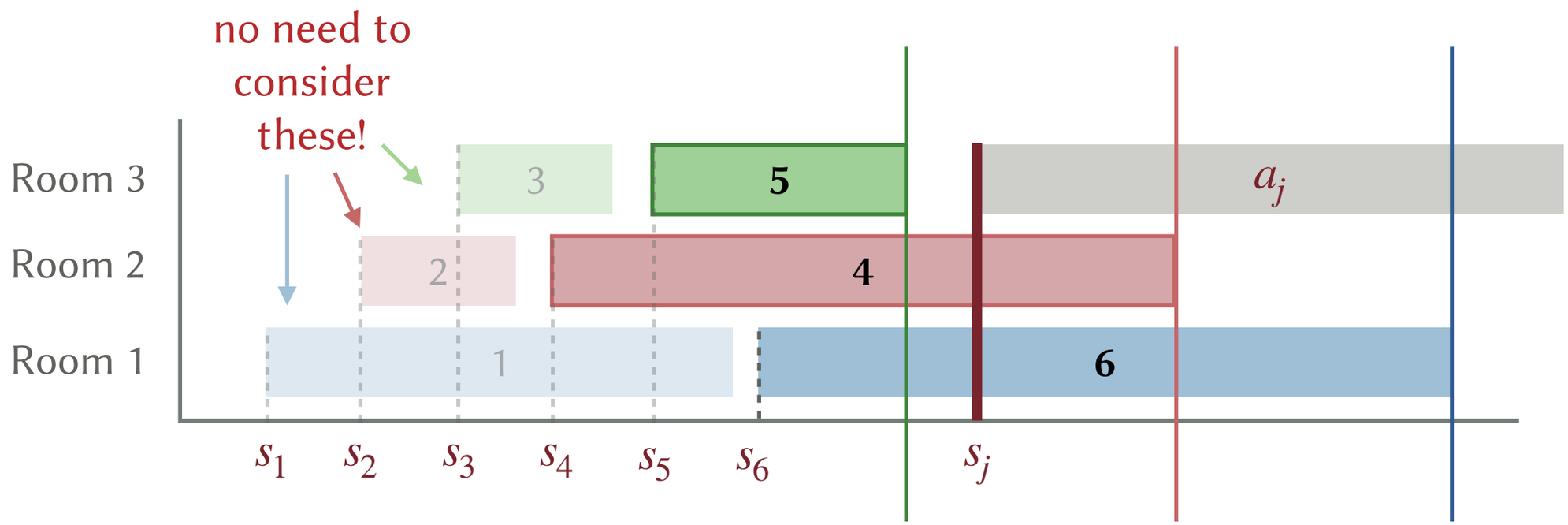
- all activities in the room start before s_j
  (because of the earliest-start-time strategy)
- a_j conflicts with an activity in the room only if a finish time is greater than s_j.
  (if $s_j >$ the latest finish time in the room, then there are no conflicts)



Interval Partitioning: Efficient Implementation

Observation. To check if a new activity a_j conflicts with any of the activities in a room, we only need to compare s_j to the latest finish time in the room, because:

- all activities in the room start before s_j
  (because of the earliest-start-time strategy)
- a_j conflicts with an activity in the room only if a finish time is greater than s_j.
  (if $s_j >$ the latest finish time in the room, then there are no conflicts)

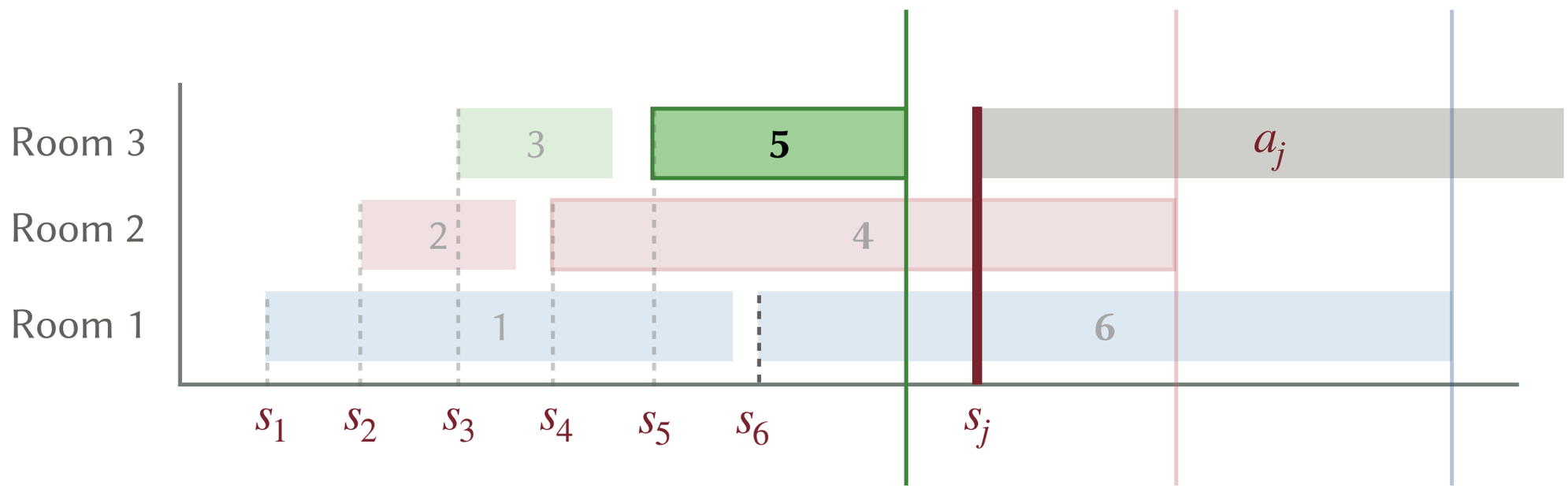


If there is a conflict with a_j in a room, then it is with the activity that has the latest finish time

Interval Partitioning: Efficient Implementation

Observation. To check if a new activity a_j conflicts with any of the activities in a room, we only need to compare s_j to the latest finish time in the room, because:

- all activities in the room start before s_j
  (because of the earliest-start-time strategy)
- a_j conflicts with an activity in the room only if a finish time is greater than s_j.
  (if $s_j >$ the latest finish time in the room, then there are no conflicts)



Note. We only need to check the room that **finishes first** among all the rooms!

Interval Partitioning: Efficient Implementation

COUNT-ROOMS(a[])

```
Sort the activities in a[] by start time
```

Assume each activity is an object that stores its start time and finish time

Interval Partitioning: Efficient Implementation

COUNT-ROOMS(a[])

```
Sort the activities in a[] by start time
Let rooms be a min-priority queue
    of objects of type Room
```

The key is the latest finish time in the room

Assume each room stores:
- a list of objects of type "activity"
- the latest finish time of its activities

Interval Partitioning: Efficient Implementation

COUNT-ROOMS(a[])

```
Sort the activities in a[] by start time
Let rooms be a min-priority queue
    of objects of type Room

Create a new room, insert a[0] into it and
insert the new room into rooms.
```

The key is the latest finish time in the room

Interval Partitioning: Efficient Implementation

```
COUNT-ROOMS(a[])

Sort the activities in a[] by start time
Let rooms be a min-priority queue
    of objects of type Room

Create a new room, insert a[0] into it and
insert the new room into rooms.

for i=1 to a.size-1:
    minRoom = rooms.MIN()
    latest = minRoom.LATEST-FINISH-TIME()
```

The key is the latest finish time in the room

compare only to the earliest finishing room

Interval Partitioning: Efficient Implementation

```
COUNT-ROOMS(a[])

Sort the activities in a[] by start time
Let rooms be a min-priority queue
    of objects of type Room

Create a new room, insert a[0] into it and
insert the new room into rooms.

for i=1 to a.size-1:
    minRoom = rooms.MIN()
    latest = minRoom.LATEST-FINISH-TIME()

    if a[i].start >= latest:
        minRoom.INSERT(a[i])
        rooms.UPDATE(minRoom)
```

The key is the latest finish time in the room

compare only to the earliest finishing room

a room with no conflicts is found

the heap needs to be updated based on the new latest finish time in the room

Interval Partitioning: Efficient Implementation

```
COUNT-ROOMS(a[])

Sort the activities in a[] by start time
Let rooms be a min-priority queue
    of objects of type Room

Create a new room, insert a[0] into it and
insert the new room into rooms.

for i=1 to a.size-1:
    minRoom = rooms.MIN()
    latest = minRoom.LATEST-FINISH-TIME()

    if a[i].start >= latest:
        minRoom.INSERT(a[i])
        rooms.UPDATE(minRoom)
    else
        Create a new room, insert a[i] into it
        and insert the new room into rooms.
```

The key is the latest finish time in the room

compare only to the earliest finishing room

a room with no conflicts is found

if a[i] does not fit in the earliest finishing room, then it does not fit in any room

Interval Partitioning: Efficient Implementation

```
COUNT-ROOMS(a[])

Sort the activities in a[] by start time
Let rooms be a min-priority queue
    of objects of type Room

Create a new room, insert a[0] into it and
insert the new room into rooms.

for i=1 to a.size-1:
    minRoom = rooms.MIN()
    latest = minRoom.LATEST-FINISH-TIME()

    if a[i].start >= latest:
        minRoom.INSERT(a[i])
        rooms.UPDATE(minRoom)

    else
        Create a new room, insert a[i] into it
        and insert the new room into rooms.

return rooms.SIZE()
```

The key is the latest finish time in the room

compare only to the earliest finishing room

a room with no conflicts is found

if a[i] does not fit in the earliest finishing room, then it does not fit in any room

Interval Partitioning: Efficient Implementation

COUNT-ROOMS(a[])

$O(n \log n)$

```
Sort the activities in a[] by start time          ← O(n log n)
Let rooms be a min-priority queue
    of objects of type Room

Create a new room, insert a[0] into it and
insert the new room into rooms.

for i=1 to a.size-1:                                ← Θ(n)
    minRoom = rooms.MIN()                           ← O(1)
    latest = minRoom.LATEST-FINISH-TIME()

    if a[i].start >= latest:
        minRoom.INSERT(a[i])                        ← O(1)
        rooms.UPDATE(minRoom)                       ← O(log n)
    else
        Create a new room, insert a[i] into it
        and insert the new room into rooms.         ← O(log n)
return rooms.SIZE()
```

in the worst case there are O(n) rooms (if d = O(n))