CS11313 - **Spring** 2022

Design & Analysis of Algorithms

Greedy Algorithms

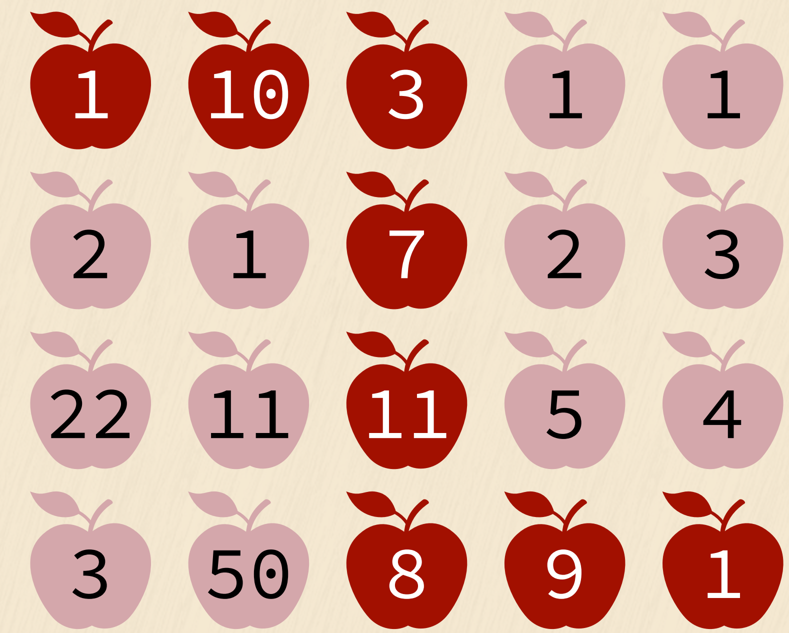
Ibrahim Albluwi

Reminder: Collecting Apples

Problem Description.

- **Goal.** Collect as many apples as possible.
- **Constraints**. Move *right* or *down* only.
- **Input**. The matrix `apples[N][M]`
  `apples[i][j]` is the number of apples at cell `[i][j]`.

start



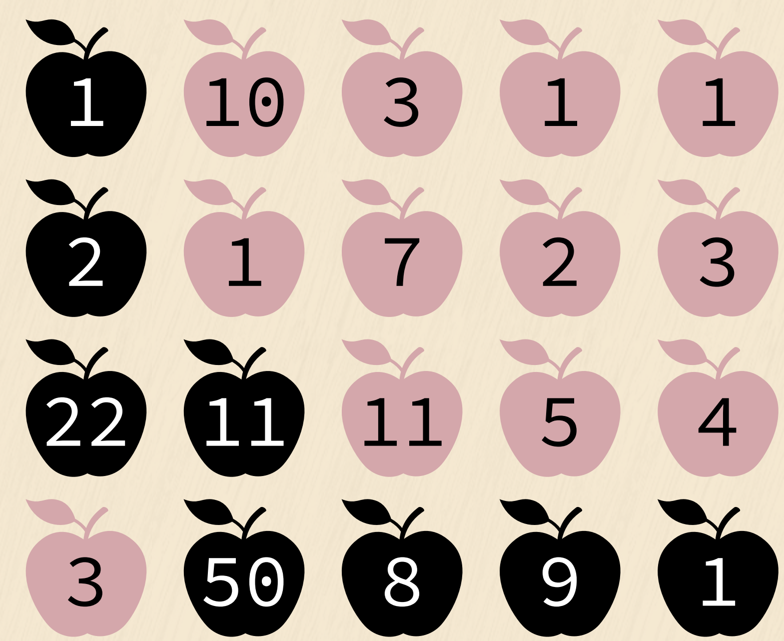
end

Total = 50

Solution # 1.

```
if apples[i+1][j] > apples[i][j+1]:
      go down.
else go right.
```

FAIL



Total = 104

Reminder: Collecting Apples

Problem Description.

- **Goal.** Collect as many apples as possible.
- **Constraints**. Move *right* or *down* only.
- **Input**. The matrix `apples[N][M]`
  `apples[i][j]` is the number of apples at cell `[i][j]`.

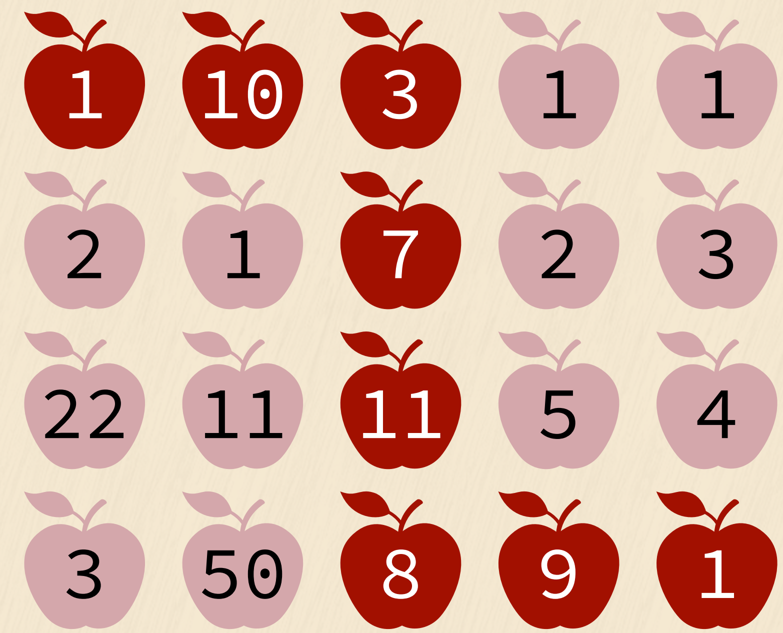
Solution # 1.

```
if apples[i+1][j] > apples[i][j+1]:
      go down.
else go right.
```

FAIL

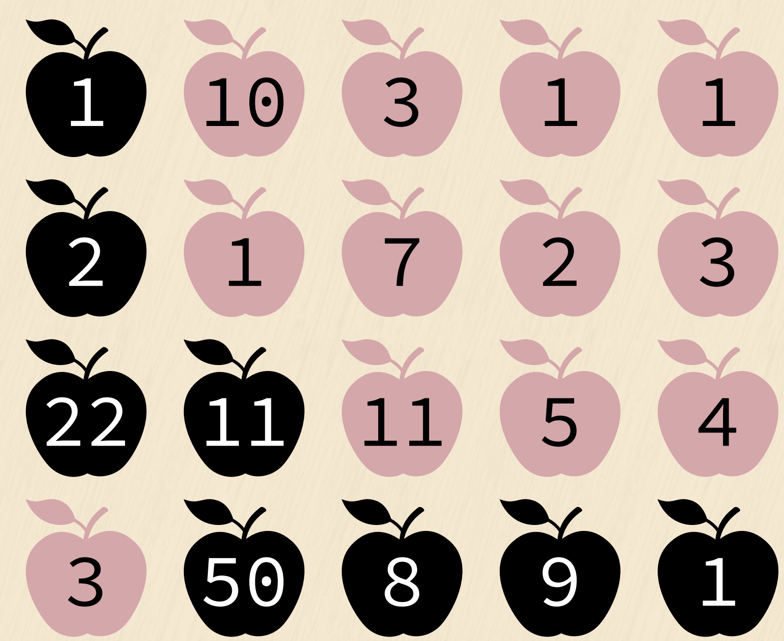
A **GREEDY** efficient solution that does not always work!

start



end

Total = 50



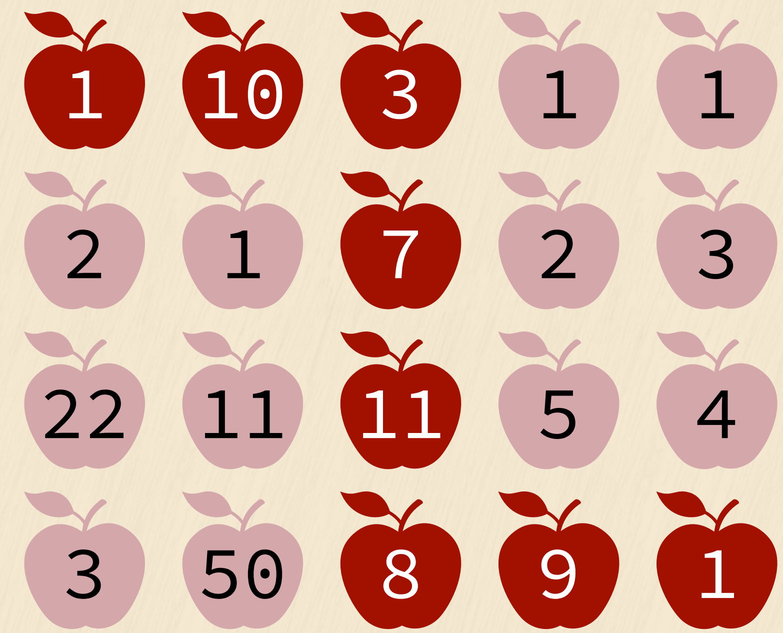
Total = 104

Reminder: Collecting Apples

Problem Description.

- **Goal.** Collect as many apples as possible.
- **Constraints**. Move *right* or *down* only.
- **Input**. The matrix `apples[N][M]`
  `apples[i][j]` is the number of apples at cell `[i][j]`.

start



end

Total = 50

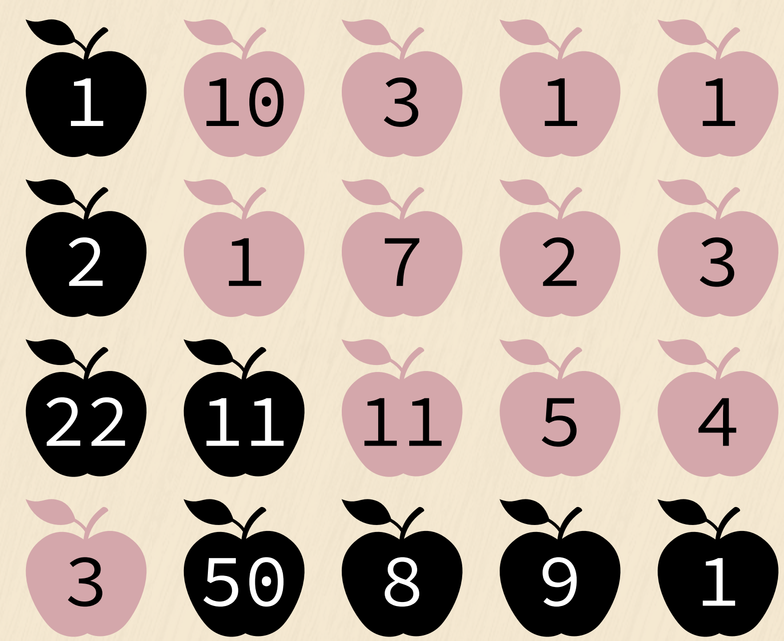
Solution # 1.

```
if apples[i+1][j] > apples[i][j+1]:
     go down.
else go right.
```

FAIL

A **GREEDY** efficient solution that does not always work!

Which problems have greedy solutions that always work?



Total = 104

Greedy Algorithms

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

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

Example. 0-1 Knapsack.

Objective Function: Choose the subset of items with the *maximum* value.
Constraints: The *total weight* of the chosen items must be *less* than the knapsack capacity.

Greedy Algorithms

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

Example. 0-1 Knapsack.

Objective Function: Choose the subset of items with the *maximum* value.
Constraints: The *total weight* of the chosen items must be *less* than the knapsack capacity.

Greedy Algorithm. Makes the best choice given the available information at the moment (without looking ahead or backtracking).

Greedy Algorithms

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

Example. 0-1 Knapsack.

Objective Function: Choose the subset of items with the *maximum* value.
Constraints: The *total weight* of the chosen items must be *less* than the knapsack capacity.

Greedy Algorithm. Makes the best choice given the available information at the moment (without looking ahead or backtracking).

Example. Should we take Item 1?

Greedy Algorithms

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

Example. 0-1 Knapsack.

Objective Function: Choose the subset of items with the *maximum* value.
Constraints: The *total weight* of the chosen items must be *less* than the knapsack capacity.

Greedy Algorithm. Makes the best choice given the available information at the moment (without looking ahead or backtracking).

Example. Should we take Item 1?

Non-Greedy. I can't tell yet! I must first know the best value I can get without Item 1 and compare it to the best value I can get with Item 1.

Greedy Algorithms

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

Example. 0-1 Knapsack.

Objective Function: Choose the subset of items with the *maximum* value.
Constraints: The *total weight* of the chosen items must be *less* than the knapsack capacity.

Greedy Algorithm. Makes the best choice given the available information at the moment (without looking ahead or backtracking).

Example. Should we take Item 1?

Non-Greedy. I can't tell yet! I must first know the best value I can get without Item 1 and compare it to the best value I can get with Item 1.

Greedy 1. Take it if it is the *most valuable* item!

Greedy 2. Take it if it is the *lightest* item!

Greedy 3. Take it if it has the *maximum value per Kg*.

Greedy Algorithms vs Dynamic Programming

Similarities. Both can be used to solve *optimization problems* and both work on problems with an *optimal substructure*.

Greedy Algorithms vs Dynamic Programming

Similarities. Both can be used to solve *optimization problems* and both work on problems with an *optimal substructure*.

Differences.

A Greedy Algorithm. Makes a decision that is guaranteed to be optimal and then solves a *single* subproblem.

Greedy Algorithms vs Dynamic Programming

Similarities. Both can be used to solve *optimization problems* and both work on problems with an *optimal substructure*.

Differences.

A Greedy Algorithm. Makes a decision that is guaranteed to be optimal and then solves a *single* subproblem.

Dynamic programming. Picks between solutions to *multiple* subproblems and stores them to avoid recomputing them again.

Greedy Algorithms vs Dynamic Programming

Similarities. Both can be used to solve *optimization problems* and both work on problems with an *optimal substructure*.

Differences.

A Greedy Algorithm. Makes a decision that is guaranteed to be optimal and then solves a *single* subproblem.

Dynamic programming. Picks between solutions to *multiple* subproblems and stores them to avoid recomputing them again.

Example. **Knapsack**(Item 1 | Item 2 | Item 3 | Item 4 | Item 5 ······ Item *n* | W)

Greedy. Decide (without much effort) whether to pick Item 1 or not and solve: `Knapsack(2 ... n, W-`w_1`)` *or* `Knapsack(2 ... n, W)`

Greedy Algorithms vs Dynamic Programming

Similarities. Both can be used to solve *optimization problems* and both work on problems with an *optimal substructure*.

Differences.

A Greedy Algorithm. Makes a decision that is guaranteed to be optimal and then solves a *single* subproblem.

Dynamic programming. Picks between solutions to *multiple* subproblems and stores them to avoid recomputing them again.

Example. **Knapsack**(Item 1 | Item 2 | Item 3 | Item 4 | Item 5 …… Item n | W)

Greedy. Decide (without much effort) whether to pick Item 1 or not and solve: **Knapsack**(2 ... n, $W-w_1$) *or* **Knapsack**(2 ... n, W)

We need a **proof** that such a decision is safe !!

Greedy Algorithms vs Dynamic Programming

Similarities. Both can be used to solve *optimization problems* and both work on problems with an *optimal substructure*.

Differences.

A Greedy Algorithm. Makes a decision that is guaranteed to be optimal and then solves a *single* subproblem.

Dynamic programming. Picks between solutions to *multiple* subproblems and stores them to avoid recomputing them again.

Example. **Knapsack**(Item 1 Item 2 Item 3 Item 4 Item 5 · · · · · · Item *n* W)

Greedy. Decide (without much effort) whether to pick Item 1 or not and solve: **Knapsack**(2 ... n, $W-w_1$) *or* **Knapsack**(2 ... n, W)

Dynamic Programming.

- Try picking Item 1 and solve **Knapsack**(2 ... n, $W-w_1$)
- Try not picking Item 1 and solve **Knapsack**(2 ... n, W)

Decide which is better.

Greedy Algorithms Don't Always Work!

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

Greedy Algorithms Don't Always Work!

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

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

Greedy Algorithms Don't Always Work!

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

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

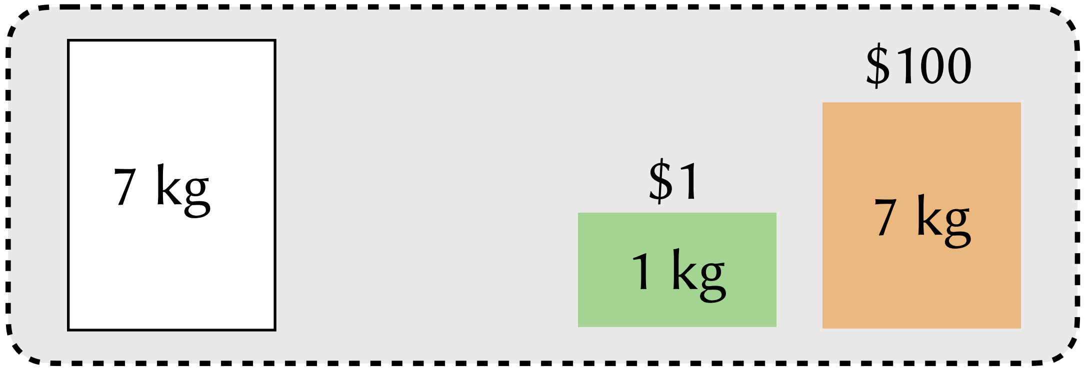
Greedy choice # 2. Take the *lightest* item first.

Greedy Algorithms Don't Always Work!

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.



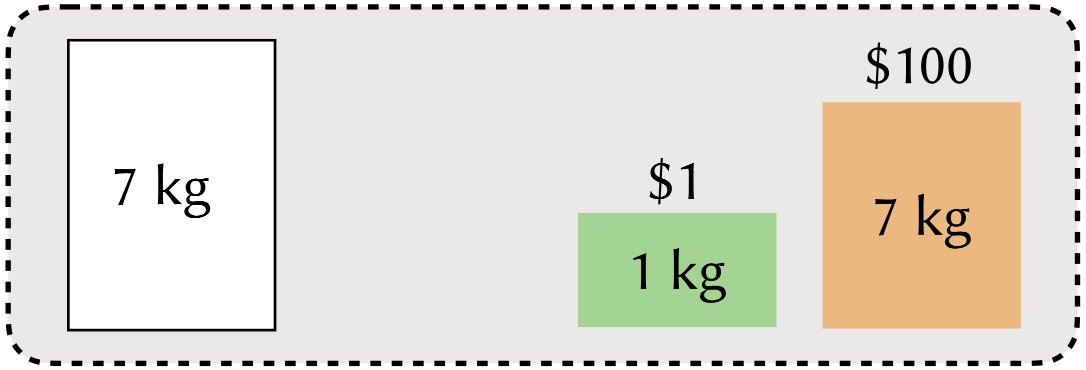
Counter Example. Greedy gives $1 but the optimal is $100.

Greedy Algorithms Don't Always Work!

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.

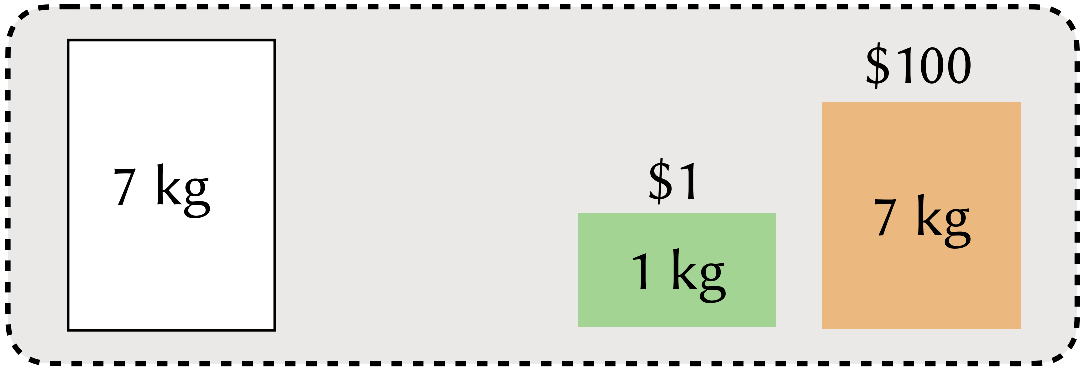
Greedy choice # 3. Take the item with the *highest value per Kg*.

Greedy Algorithms Don't Always Work!

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

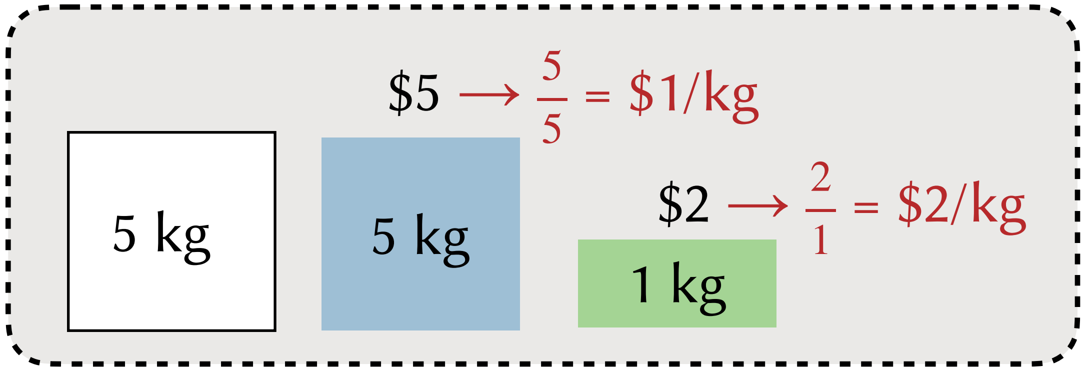
Counter Example. Greedy gives $12 but the optimal is $17.

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.

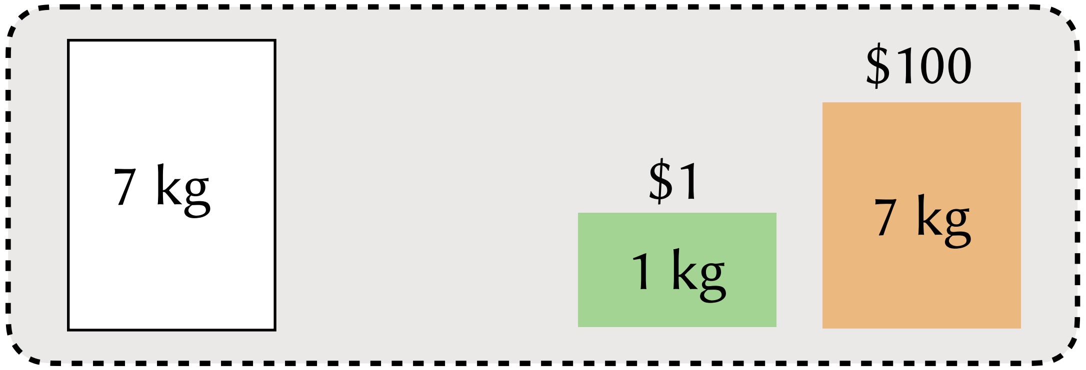


Counter Example. Greedy gives $2 but the optimal is $5.

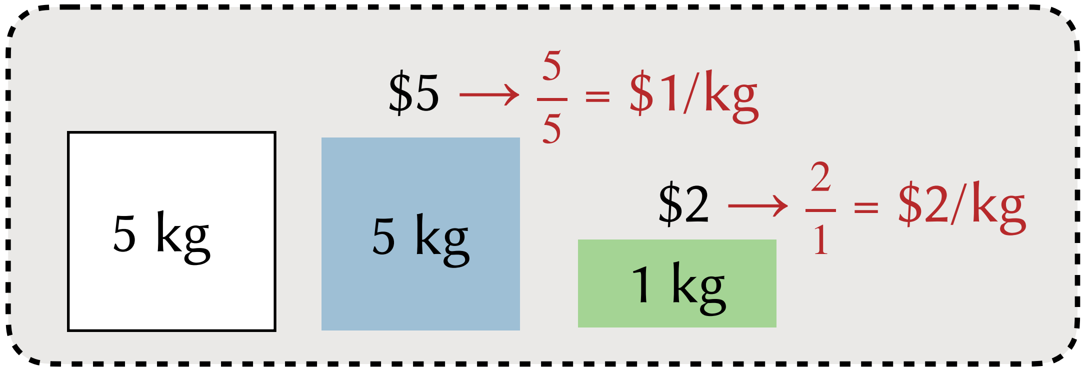
Greedy Algorithms Don't Always Work!

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

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



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

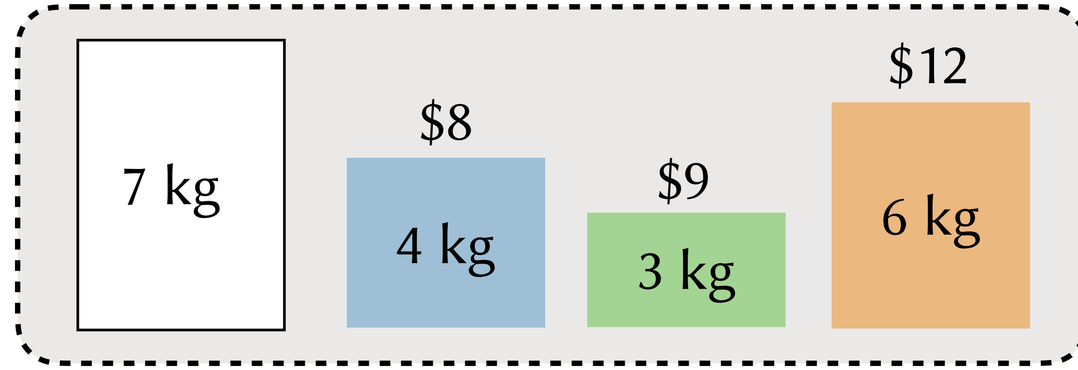


Counter Example. Greedy gives $2 but the optimal is $5.

Greedy Algorithms Don't Always Work!

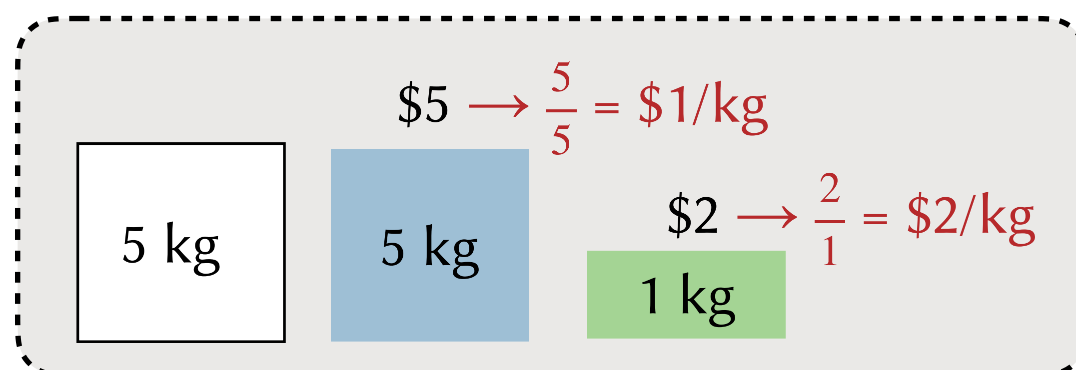
Greedy Choice Property. A *globally* optimal solution can be reached with a sequence of *locally* optimal decisions.

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.

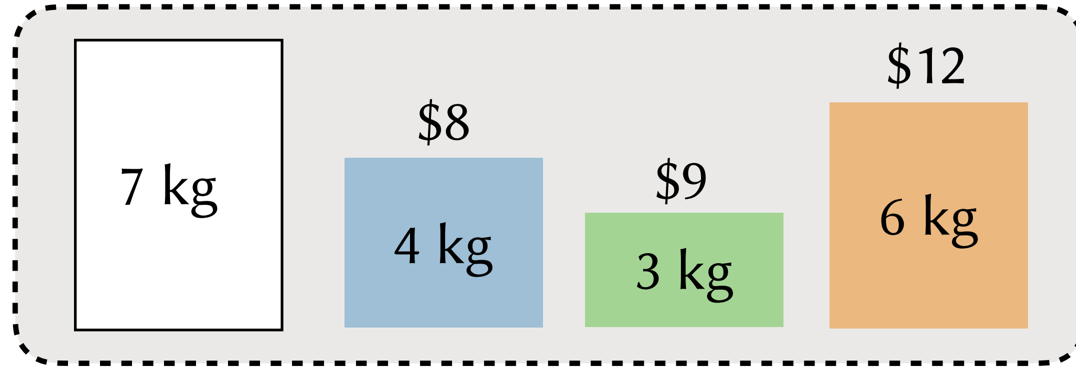
Greedy Algorithms Don't Always Work!

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

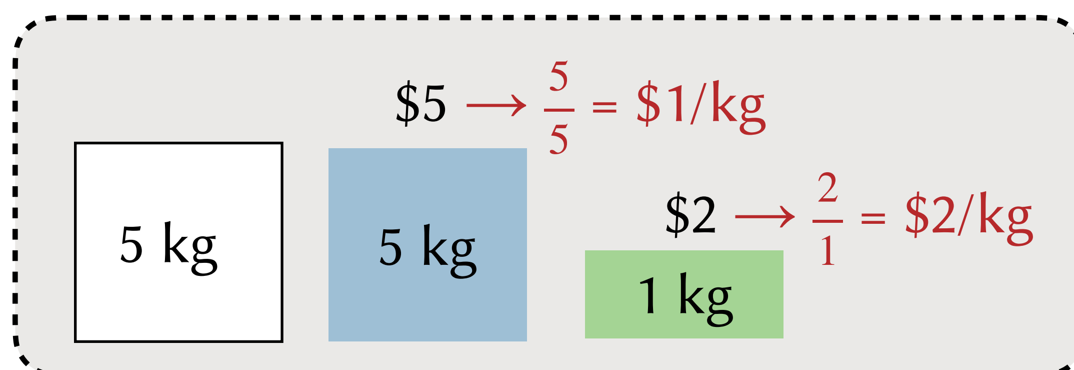
To prove that a greedy strategy is *not optimal* it is enough to find *one counter example*.

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



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.

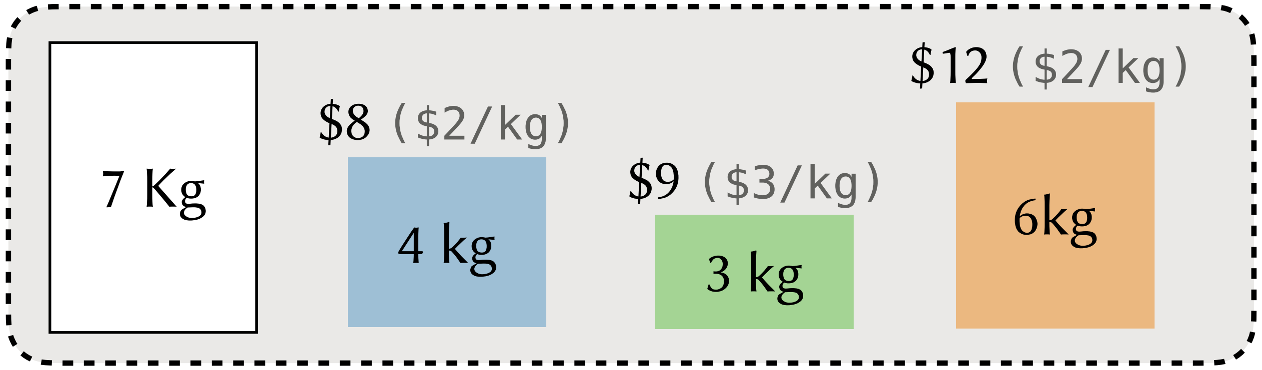
Fractional Knapsack

Fractional Knapsack. Assume that we are allowed to take fractions of the items.

Fractional Knapsack

Fractional Knapsack. Assume that we are allowed to take fractions of the items.

Greedy choice.
Start filling with the item that has *highest value per Kg*.

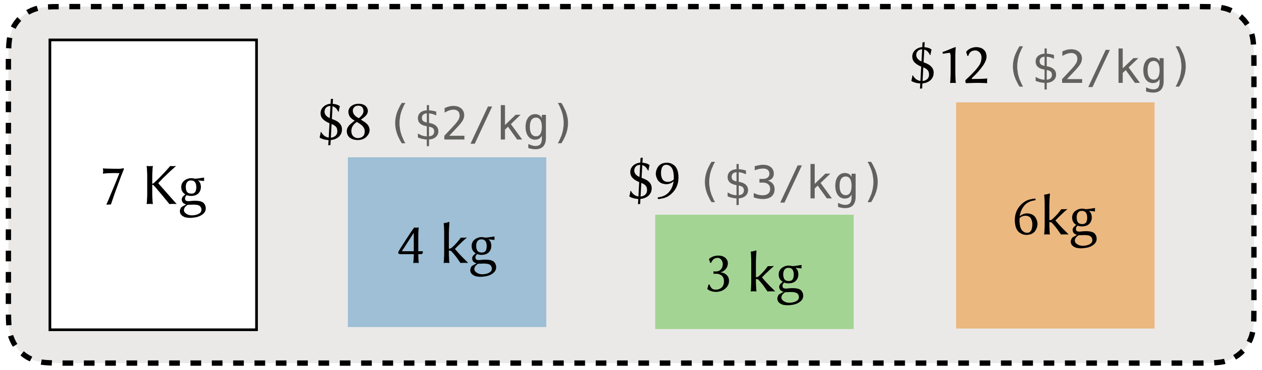


(3 x $3/kg) + (4 x $2/kg)
= $17

Fractional Knapsack

Fractional Knapsack. Assume that we are allowed to take fractions of the items.

Greedy choice.
Start filling with the item that has *highest value per Kg*.



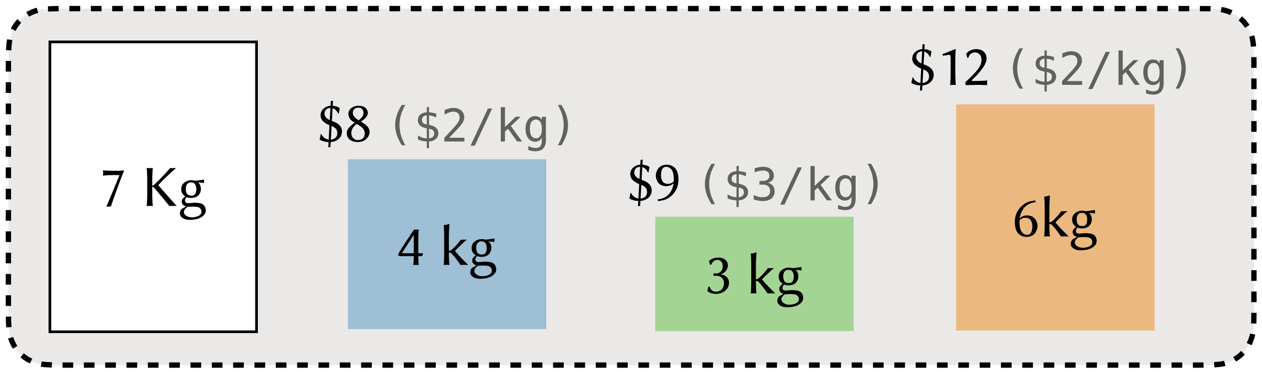
(3 x $3/kg) + (4 x $2/kg)
= $17

(5 x $20/kg) + (2 x $0.33/kg)
= $100.66

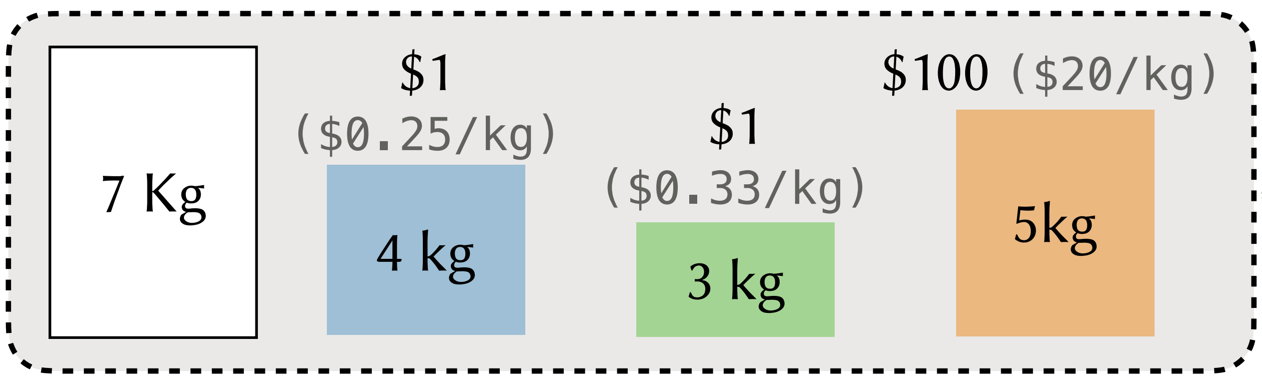
Fractional Knapsack

Fractional Knapsack. Assume that we are allowed to take fractions of the items.

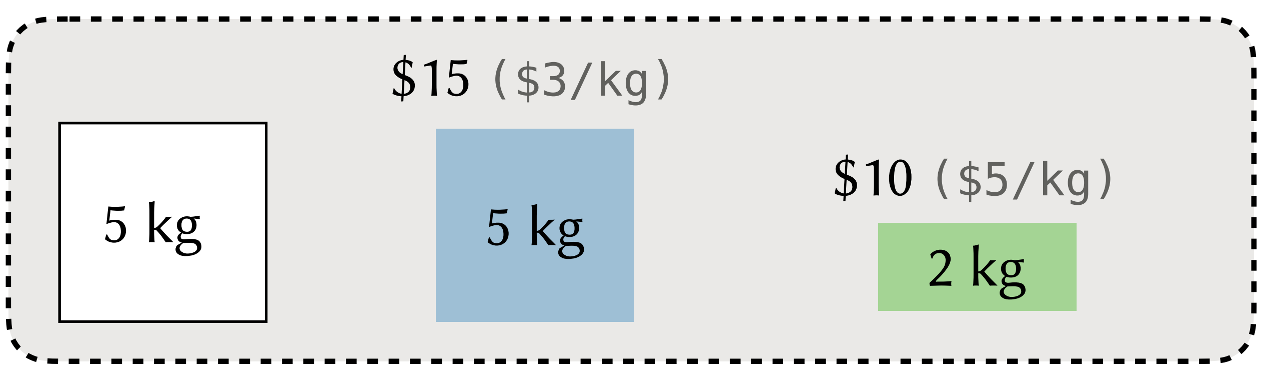
Greedy choice.
Start filling with the item that has *highest value per Kg*.



(3 x $3/kg) + (4 x $2/kg)
= $17



(5 x $20/kg) + (2 x $0.33/kg)
= $100.66



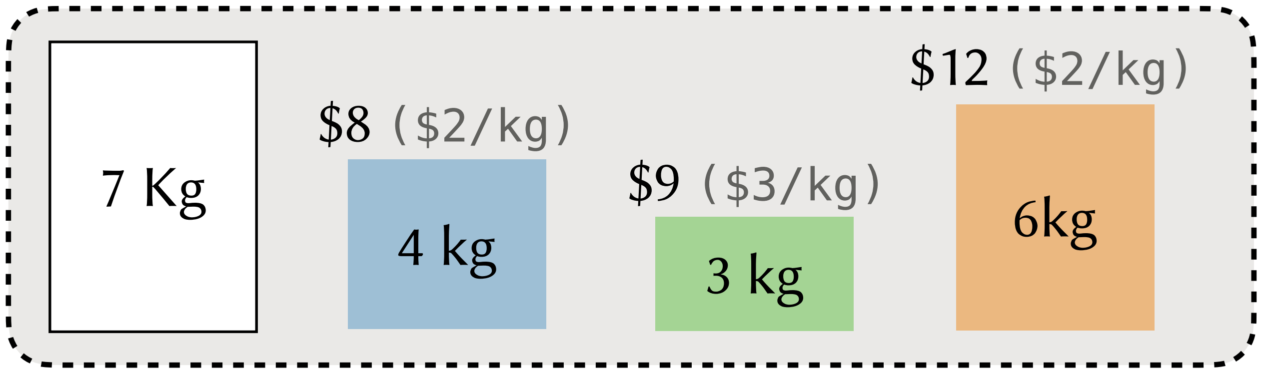
(2 x $5/kg) + (3 x $3/kg)
= $19

Fractional Knapsack

Fractional Knapsack. Assume that we are allowed to take fractions of the items.

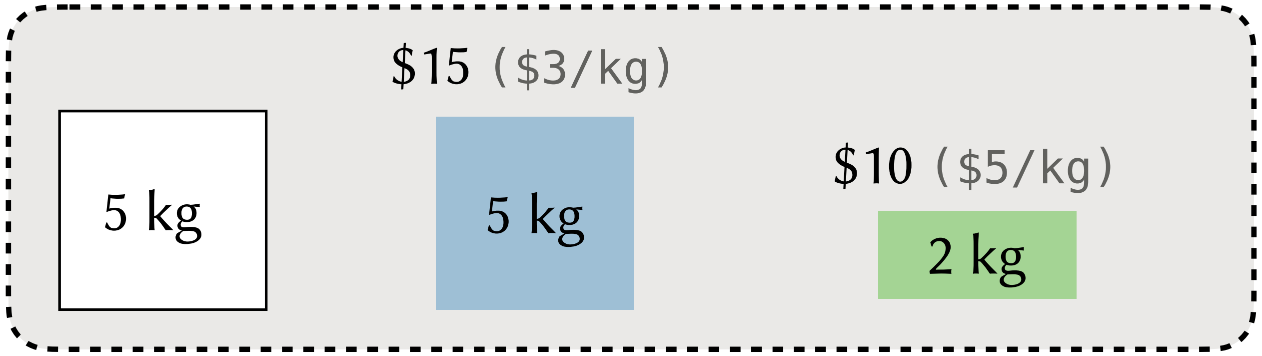
Greedy choice.
Start filling with the item that has *highest value per Kg*.

This algorithm is *optimal*.
However, we need a proof!



(3 x $3/kg) + (4 x $2/kg)
= $17

(5 x $20/kg) + (2 x $0.33/kg)
= $100.66



(2 x $5/kg) + (3 x $3/kg)
= $19

Fractional Knapsack

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

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

Item 1	Item 2	Item 3	Item 4	Item 5	·······	Item n

$$\frac{v_1}{w_1} > \frac{v_2}{w_2} > \frac{v_3}{w_3} > \frac{v_4}{w_4} > \frac{v_5}{w_5} > \cdots > \frac{v_n}{w_n}$$

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

Fractional Knapsack

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

Item 1	Item 2	Item 3	Item 4	Item 5	········	Item n

$$\frac{v_1}{w_1} > \frac{v_2}{w_2} > \frac{v_3}{w_3} > \frac{v_4}{w_4} > \frac{v_5}{w_5} > \qquad > \frac{v_n}{w_n}$$

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 (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:
```

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.

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?

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¢

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¢`

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

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¢`

Greedy Strategy. 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¢ ?

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¢`

Greedy Strategy. 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)

Interview Problem

💪 optional

Problem. Given n coin denominations $\{ d_1, d_2, \ldots, d_n \}$ and a target value V, find the fewest coins needed to make change for V (or report impossible).

Subproblems. $OPT(v)$ = fewest coins needed to make change for amount v.

Optimal value. $OPT(V)$.

Multiway choice. To compute $OPT(v)$,

- Select a coin of denomination $d_i \leq v$ for some i.
- Use fewest coins to make change for $v - d_i$.

take best

optimal substructure

Dynamic programming recurrence.

$$OPT(v) = \begin{cases} \infty & \text{if } v < 0 \\ 0 & \text{if } v = 0 \\ \min\limits_{1 \leq i \leq n} \{ 1 + OPT(v - d_i) \} & \text{if } v > 0 \end{cases}$$

slide by Kevin Wayne

Interview Problem

💪 optional

Bottom-up DP implementation.

```
create an array opt[] of size V+1

for (v = 1 to V)
    opt[v] = INFINITY
    for (i = 1 to n)
        if (d[i] > v) continue
        else opt[v] = min(opt[v], 1 + opt[v - d[i]])
```

Running time. The bottom-up DP algorithm takes $\Theta(n\ V)$ time.

Note. Not polynomial in input size (and no poly-time algorithm is known).

n, $\log V$

adapted from a slide 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.

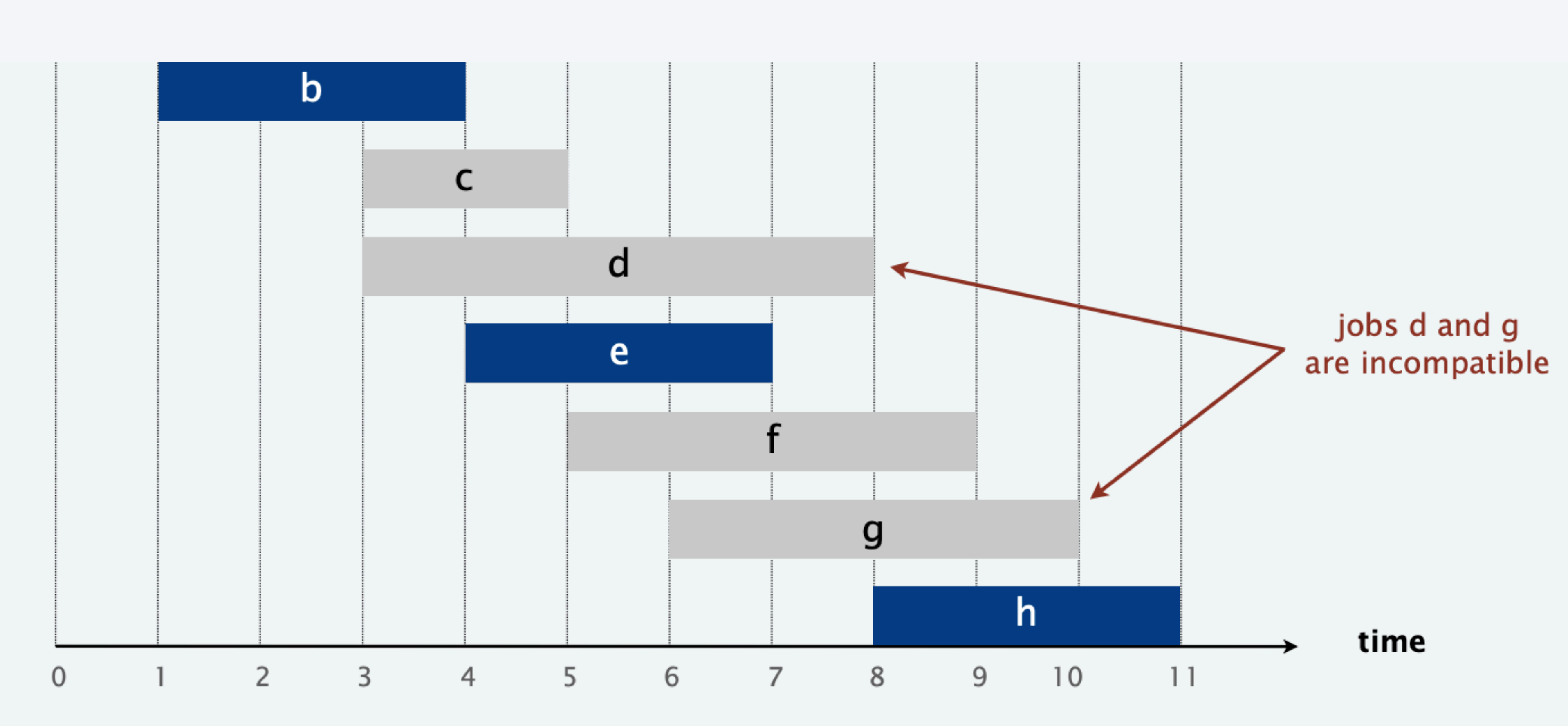


image by Kevin Wayne

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.

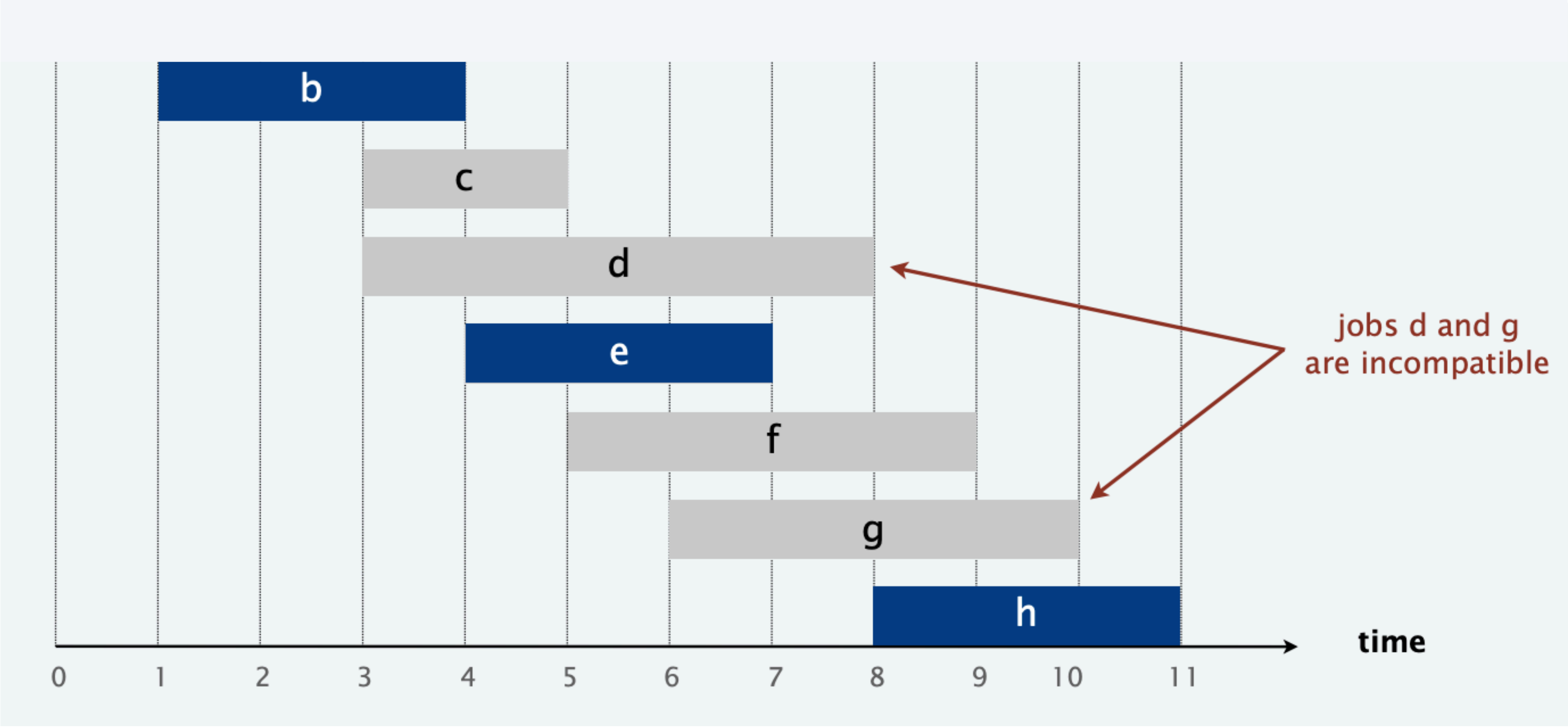


image by Kevin Wayne

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

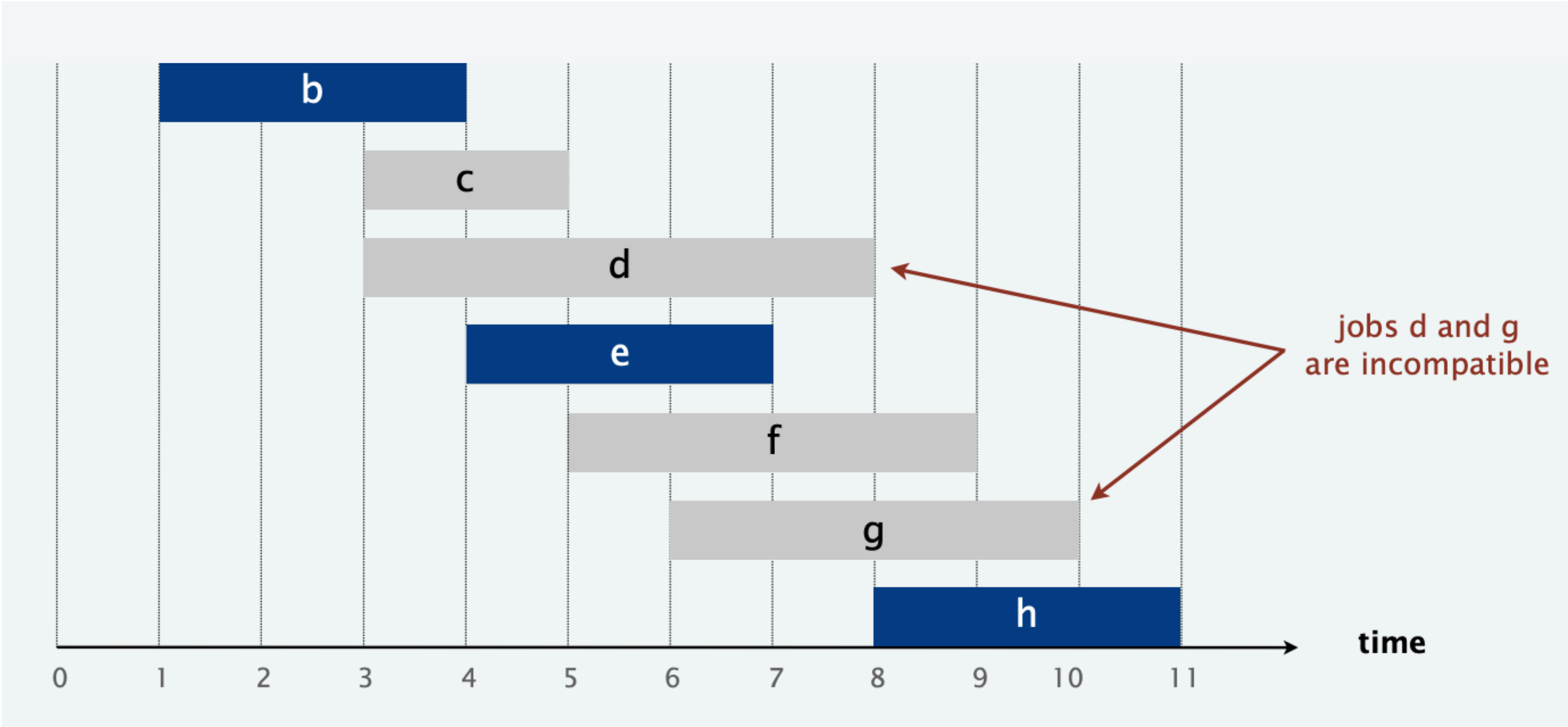


image by Kevin Wayne

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 needs the activity 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.

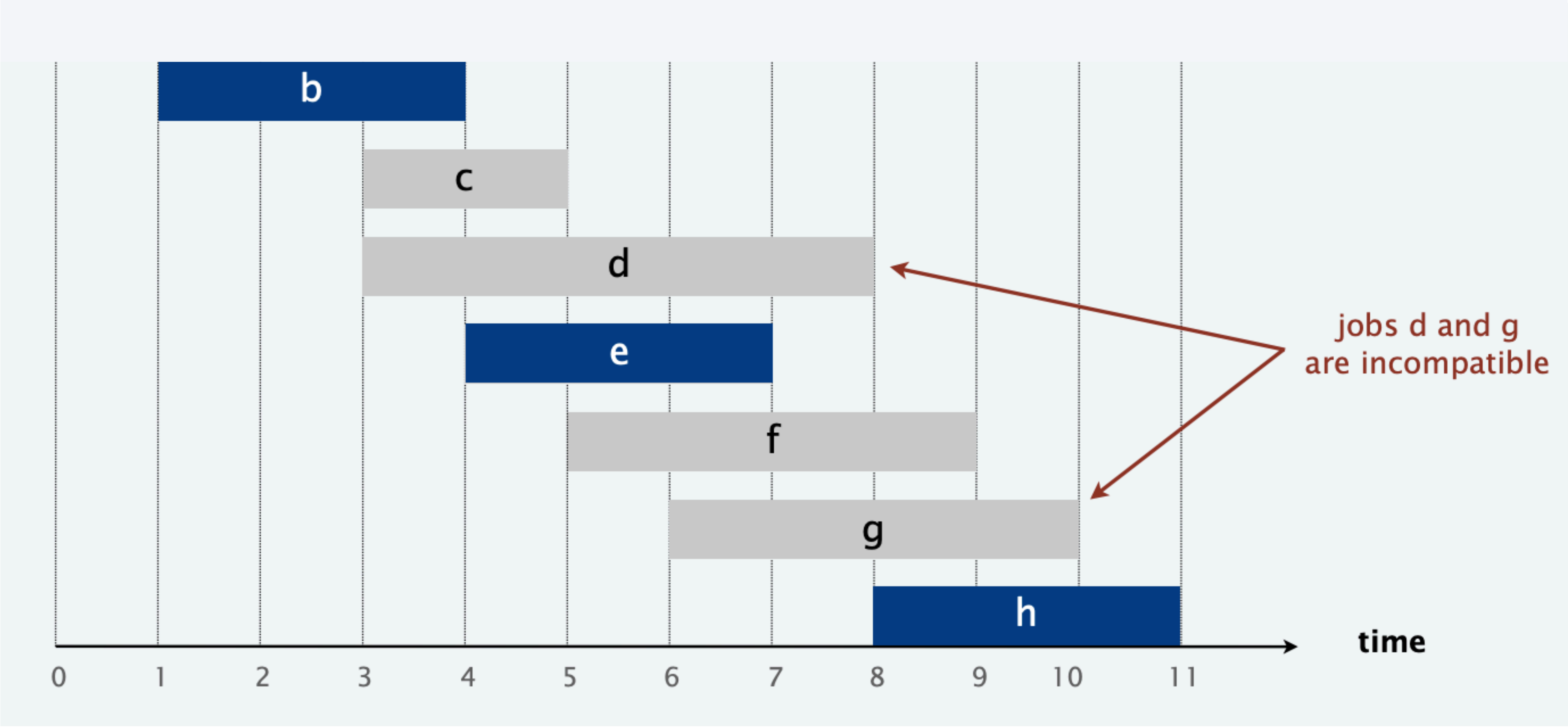


image by Kevin Wayne

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

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.

?

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.

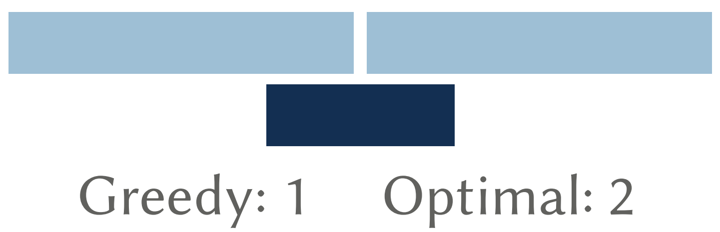
?

Activity Selection

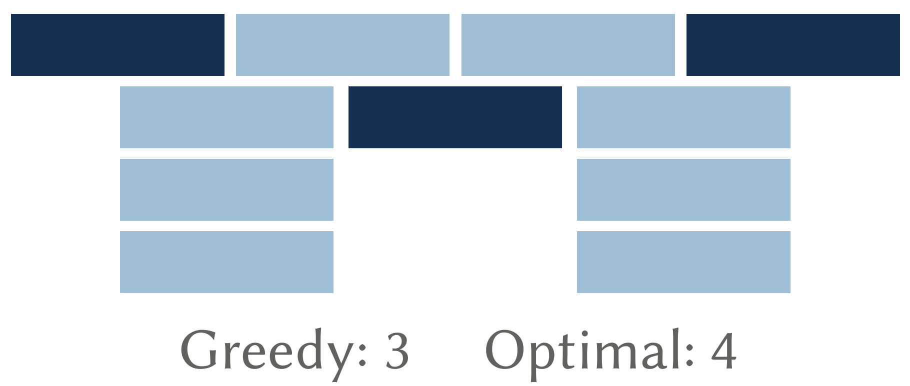
Counter Examples

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

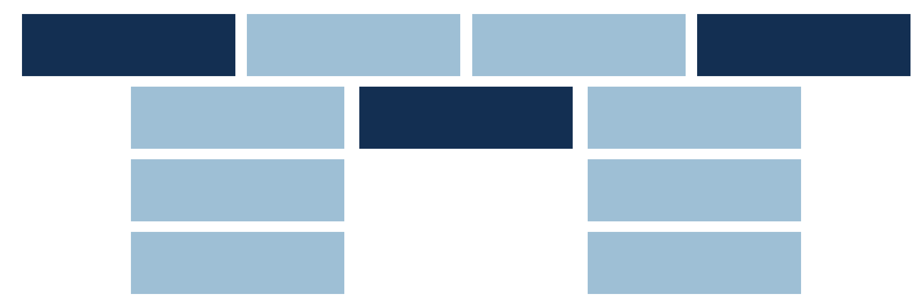
Greedy strategy # 1. Earliest start time first.
Rationale. Start the activities as early as possible.

Greedy: 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

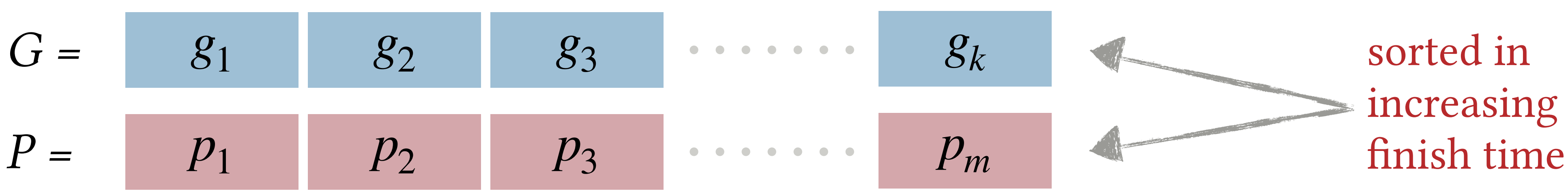
optimal

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.

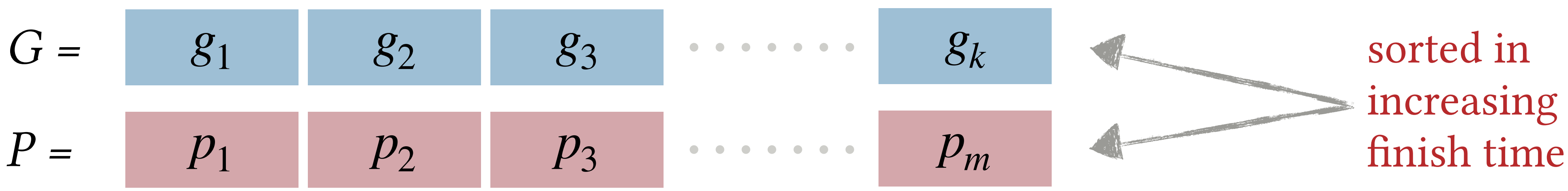
Activity Selection (Optimality of Greedy Solution)

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*).

$G = $ g_1 g_2 g_3 $\cdots\cdots$ g_k

$P = $ p_1 p_2 p_3 $\cdots\cdots$ p_m

sorted in increasing finish time

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 $p_2 \longrightarrow p_m$.

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).

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*).

$G = \boxed{g_1}\,\boxed{g_2}\,\boxed{g_3} \cdots\cdots \boxed{g_k}$

$P = \boxed{p_1}\,\boxed{p_2}\,\boxed{p_3} \cdots\cdots \boxed{p_m}$

sorted in increasing finish time

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

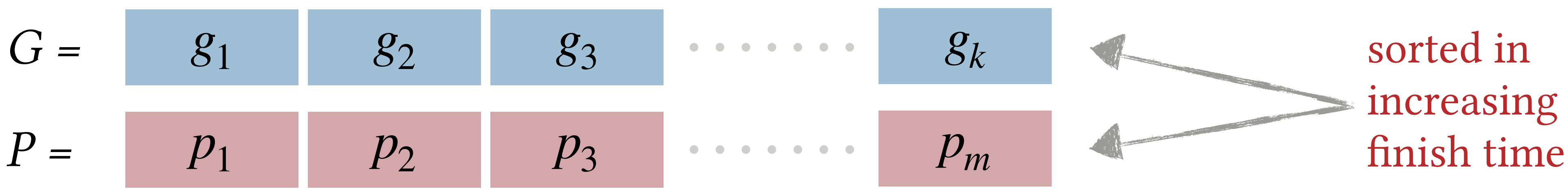
Proof. Since g_1 is guaranteed to finish before (or with) p_1, g_1 is compatible with $p_2 \longrightarrow p_m$.

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 $p_2 \longrightarrow p_m$.

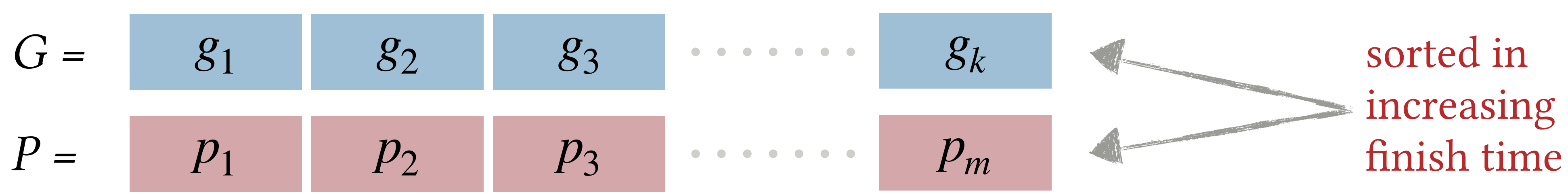
Hence, there is always an optimal solution that starts with the activity that finishes first (g_1).

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*).

$G =$	g_1	g_2	g_3	········	g_k
$P =$	p_1	p_2	p_3	········	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 $i \le k$, 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	$\cdots\cdots$	g_k	Why didn't G pick p_{i+1}?			
$P =$	p_1	p_2	p_3	$\cdots\cdots$	p_i	p_{i+1}	$\cdots\cdots$	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)

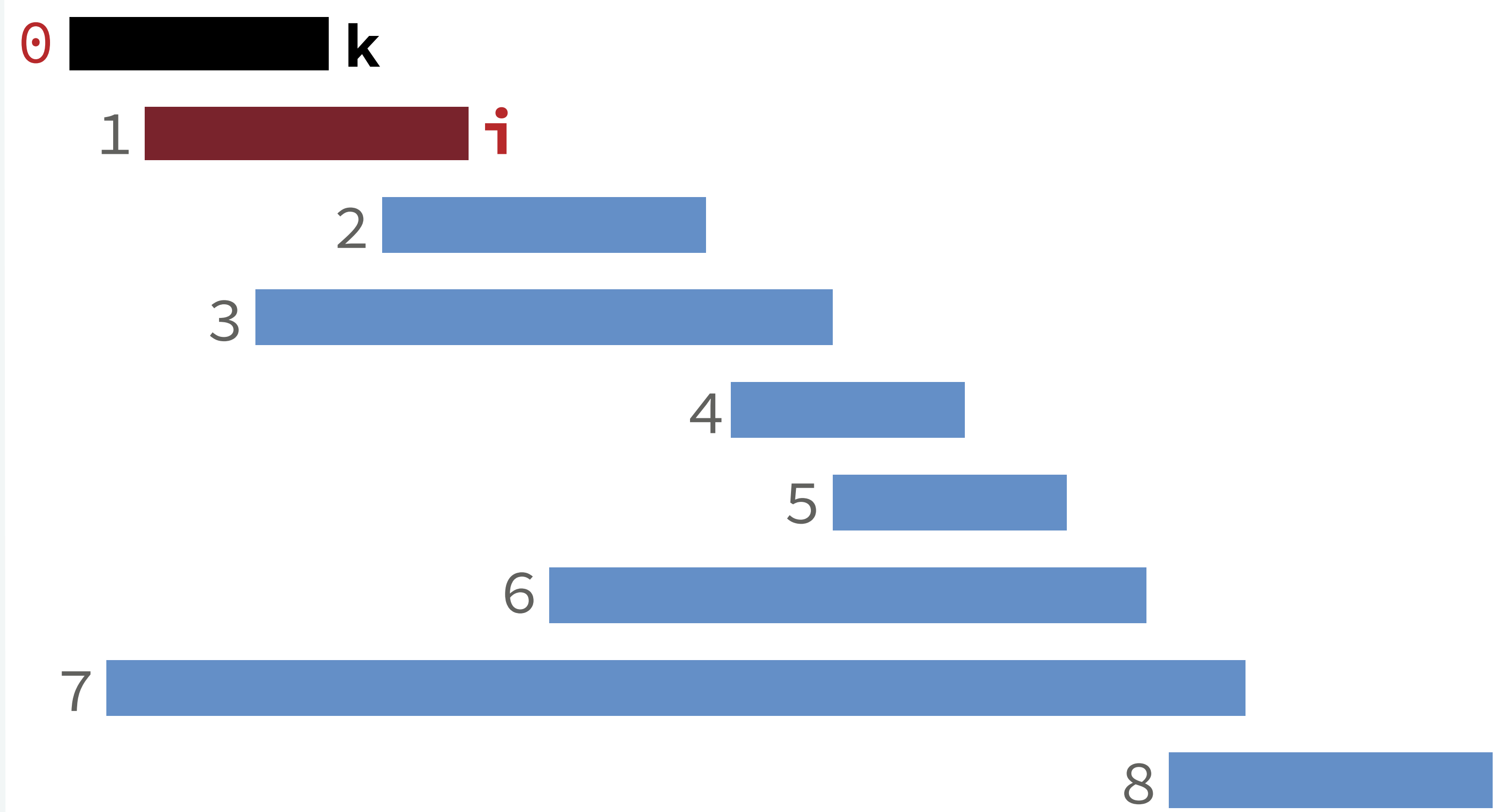
```
SELECT(s[], f[], n)

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)

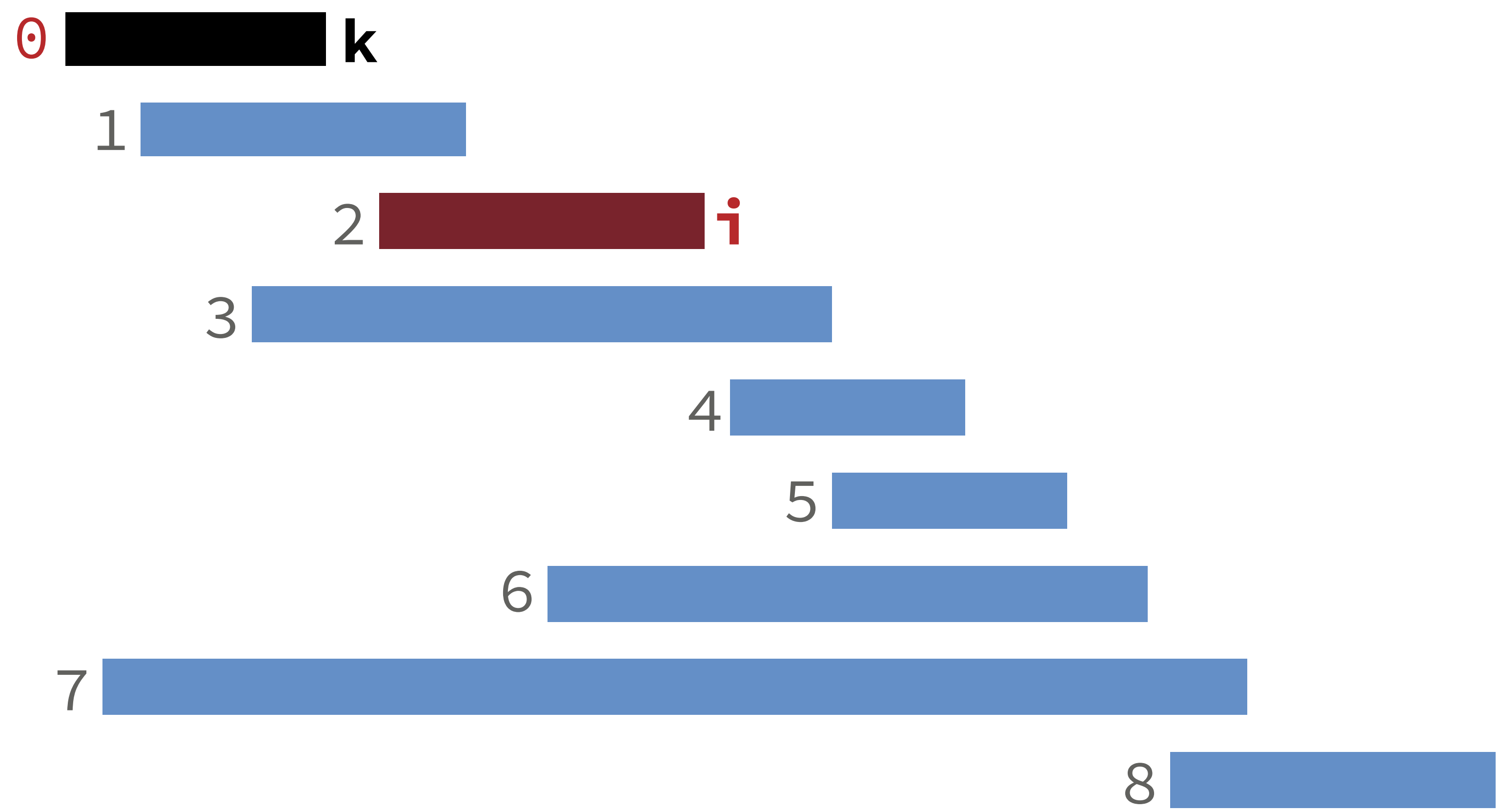
```
SELECT(s[], f[], n)

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)

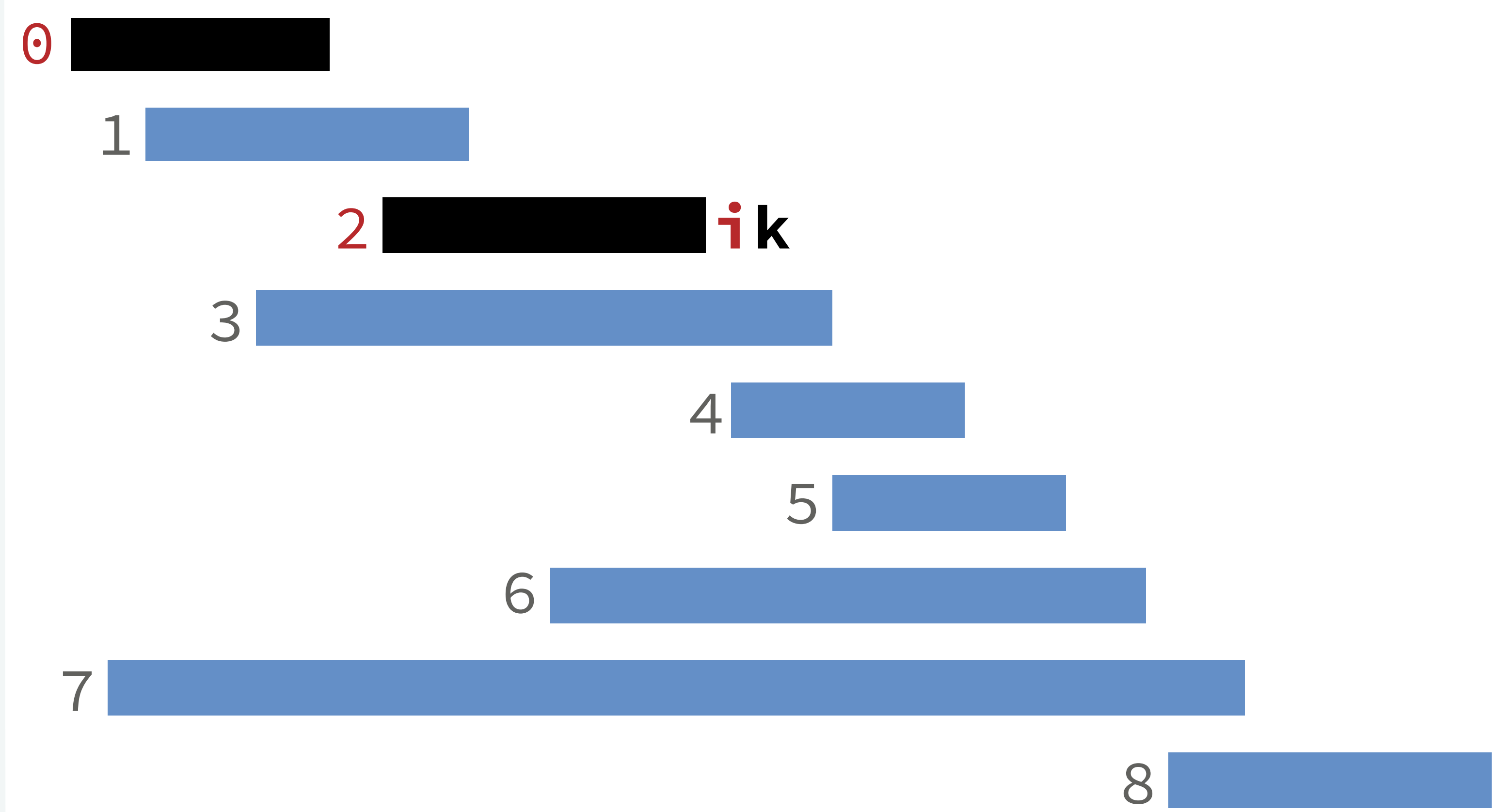
```
SELECT(s[], f[], n)

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)

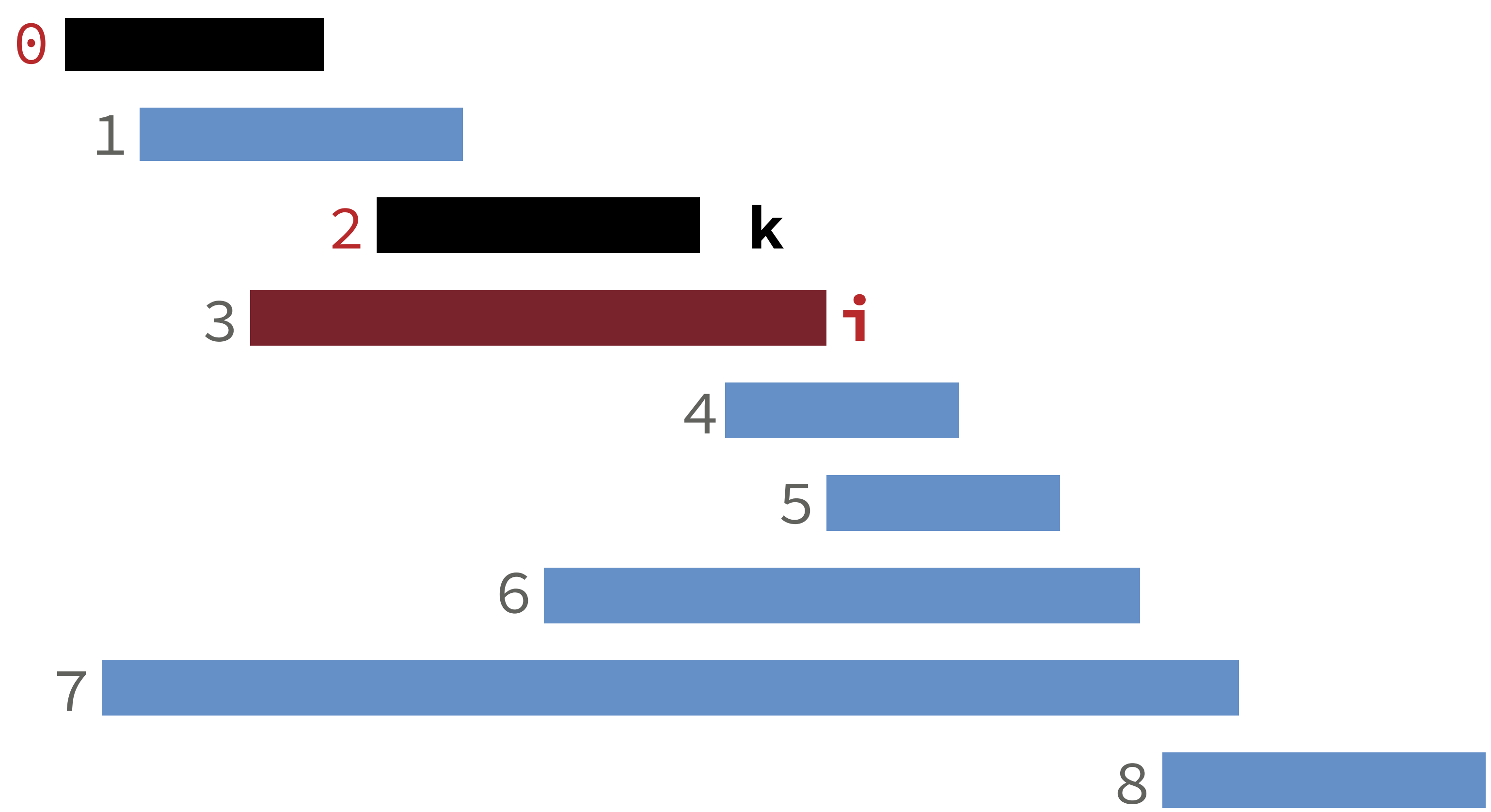
```
SELECT(s[], f[], n)

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)

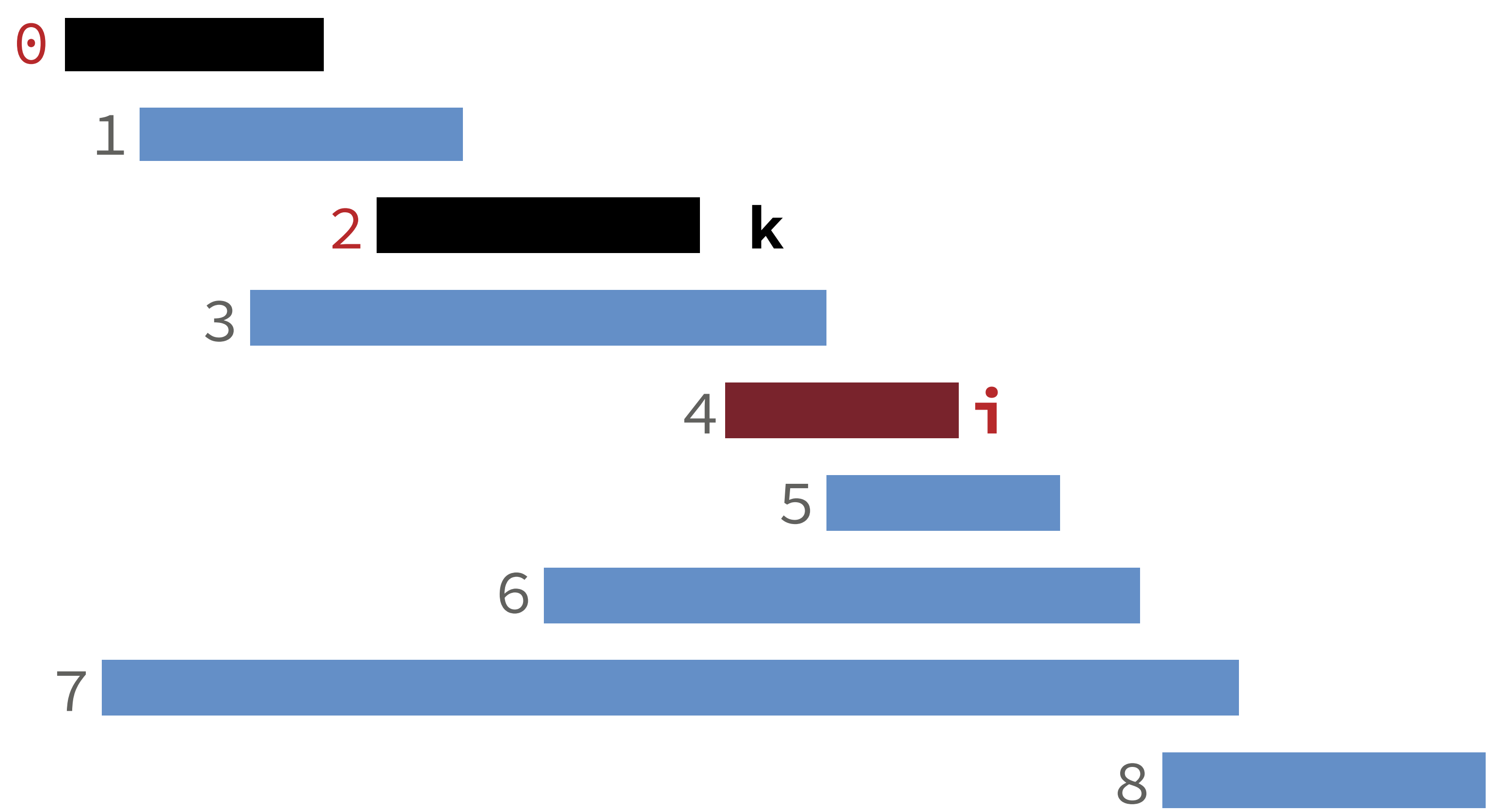
```
SELECT(s[], f[], n)

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)

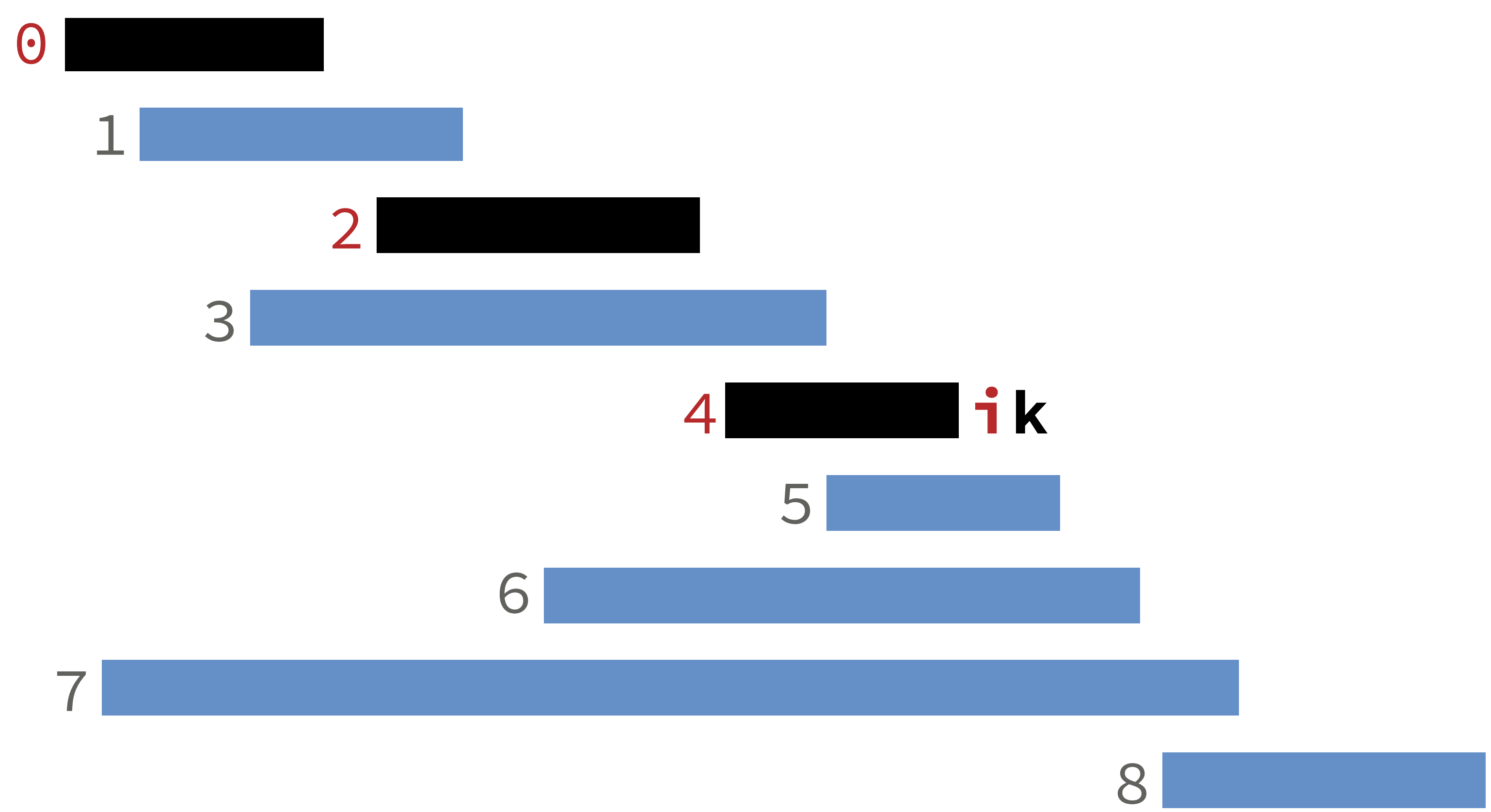
```
SELECT(s[], f[], n)

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)

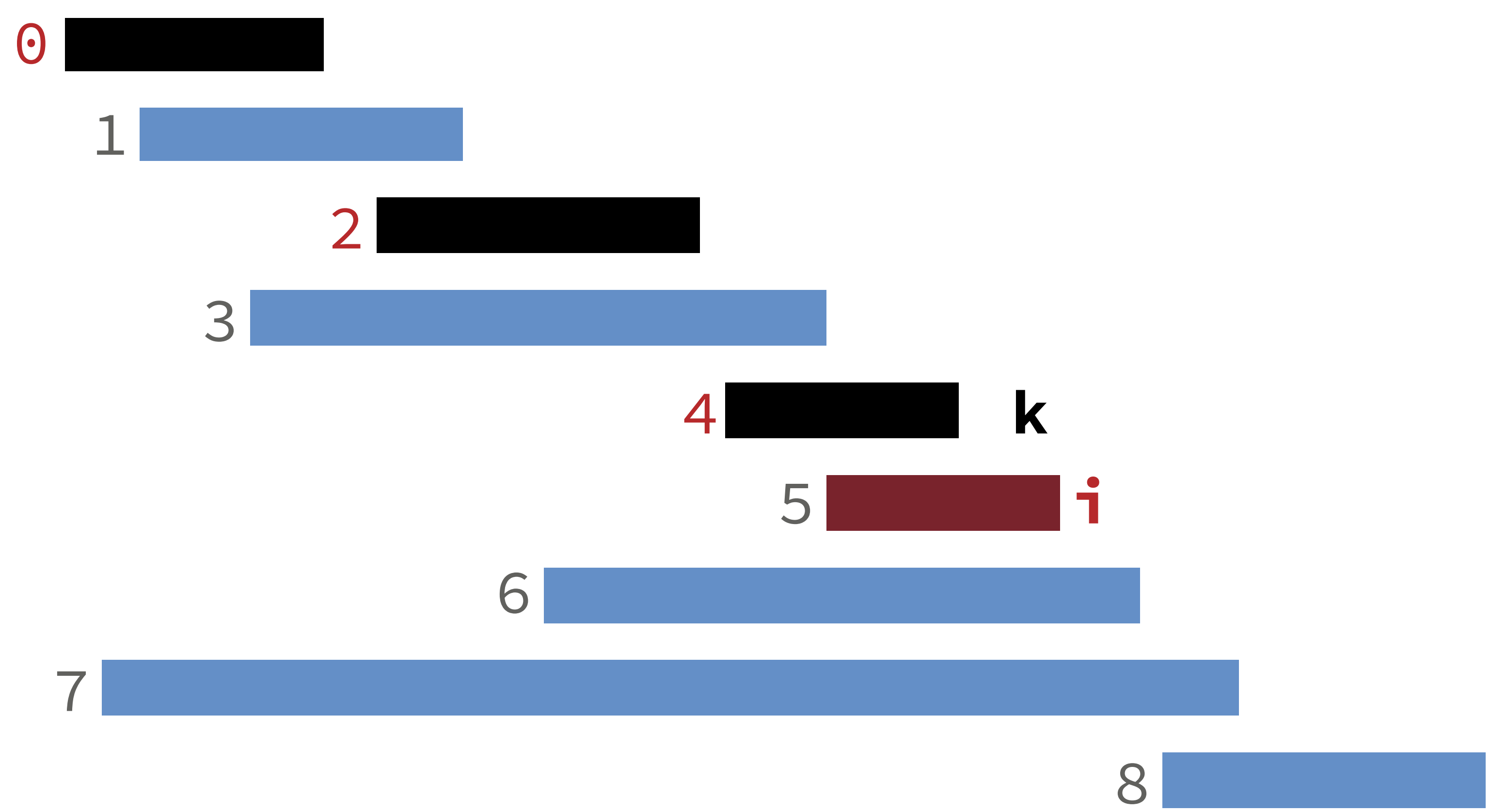
```
SELECT(s[], f[], n)

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)

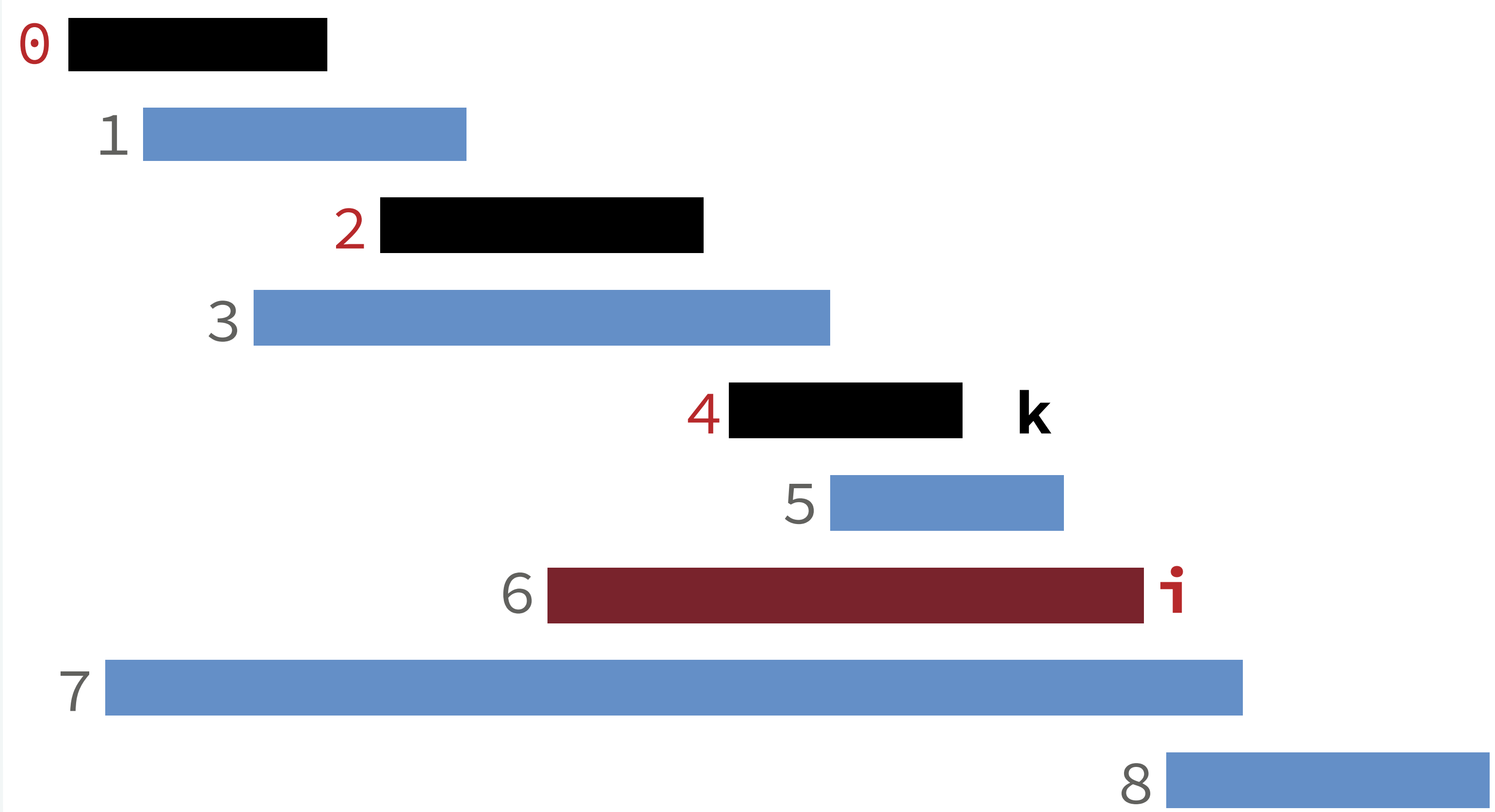
```
SELECT(s[], f[], n)

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)

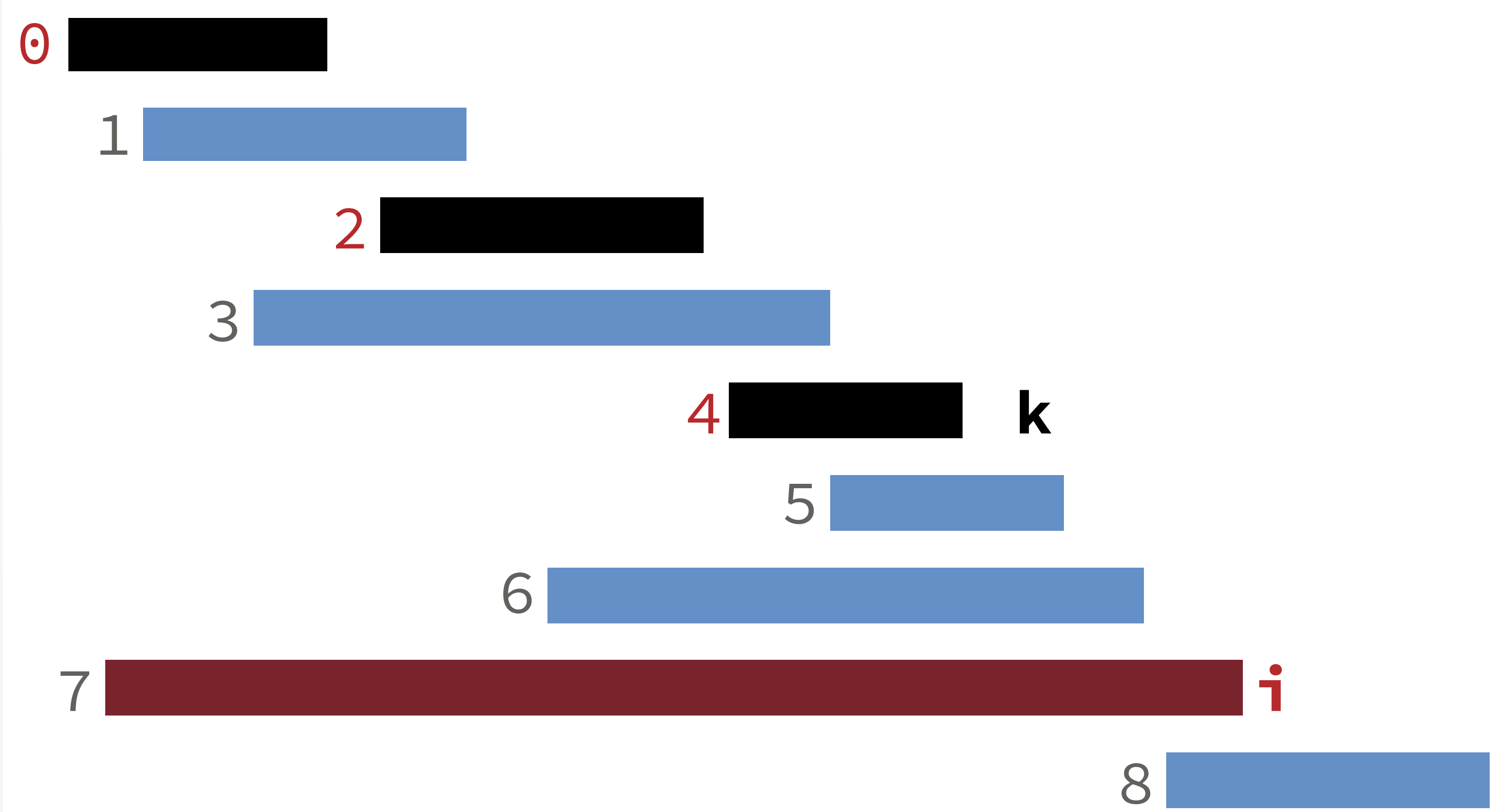
```
SELECT(s[], f[], n)

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)

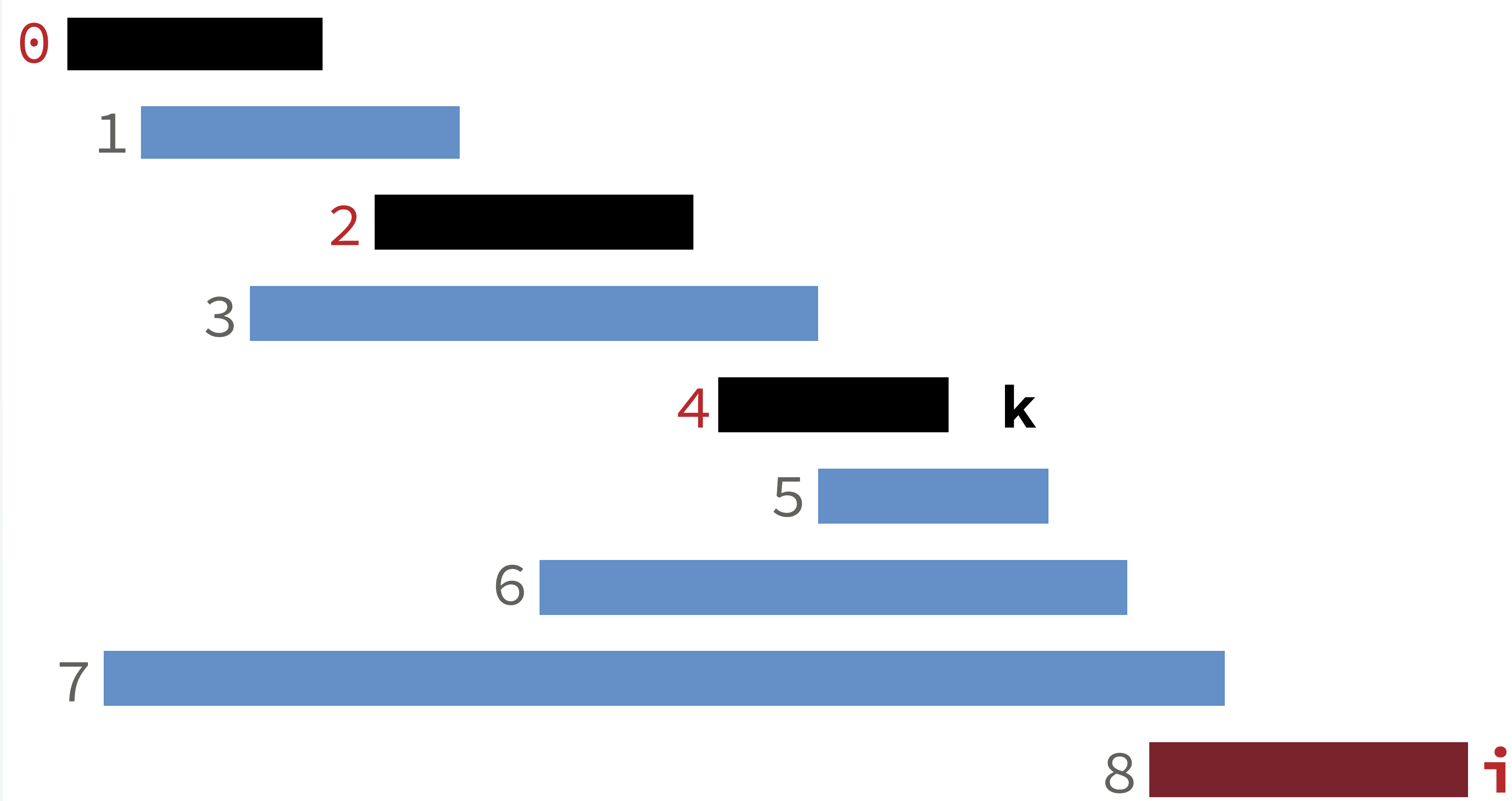
```
SELECT(s[], f[], n)

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)

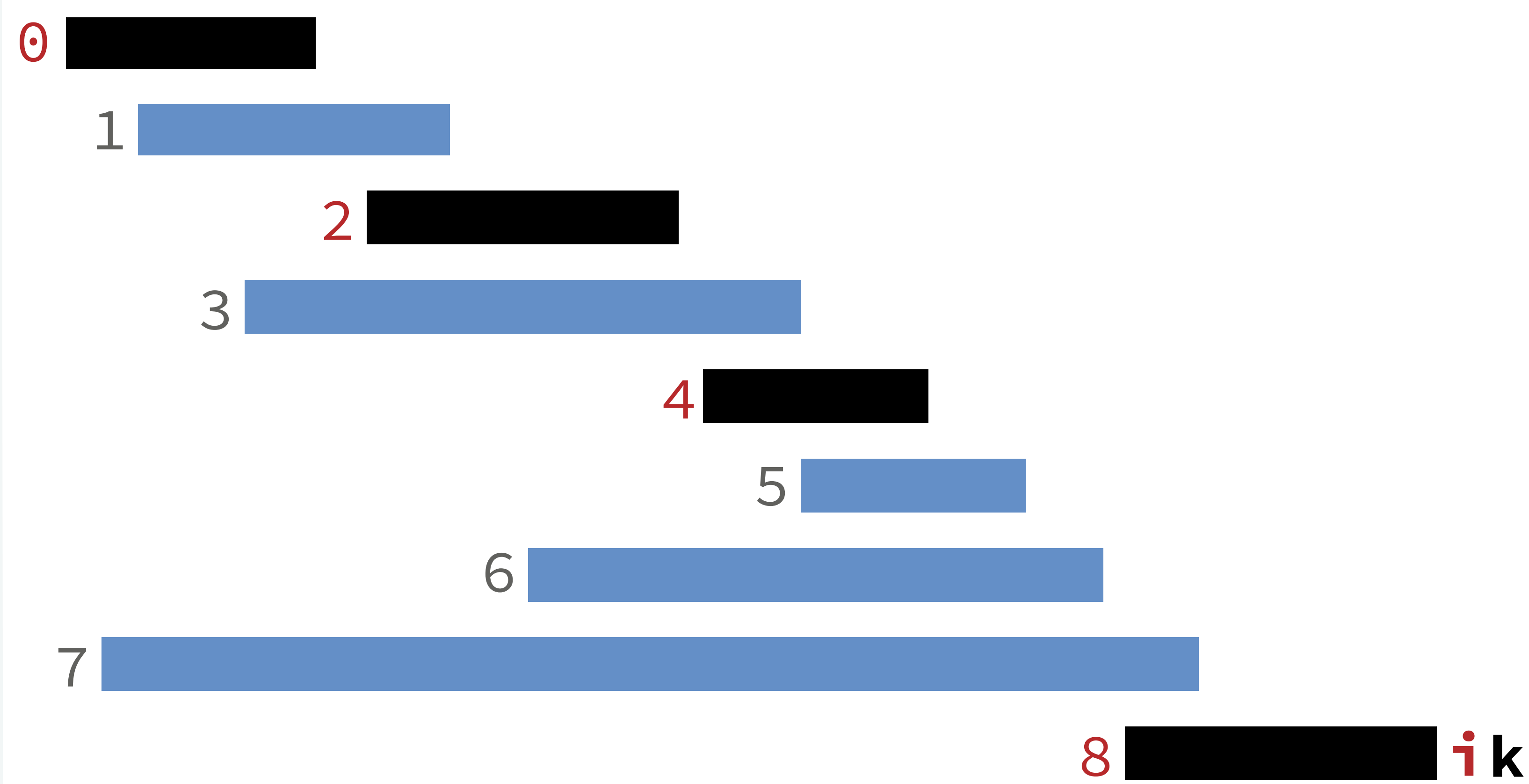
```
SELECT(s[], f[], n)

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
```

Sort step → $\Theta(n \log n)$

Loop → $\Theta(n)$

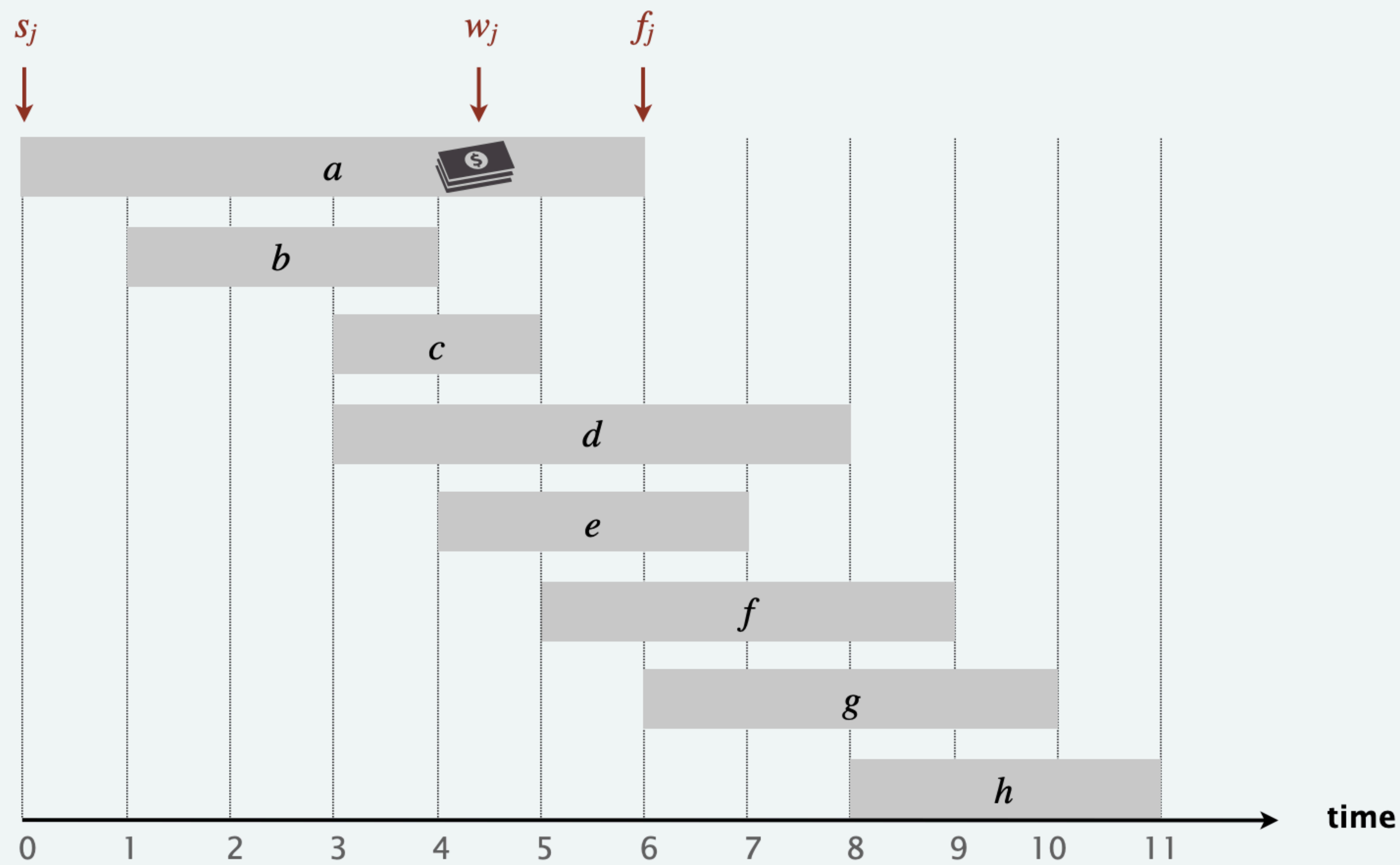
💪 optional

Interview Problem

💪 optional

Weighted interval scheduling

- 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

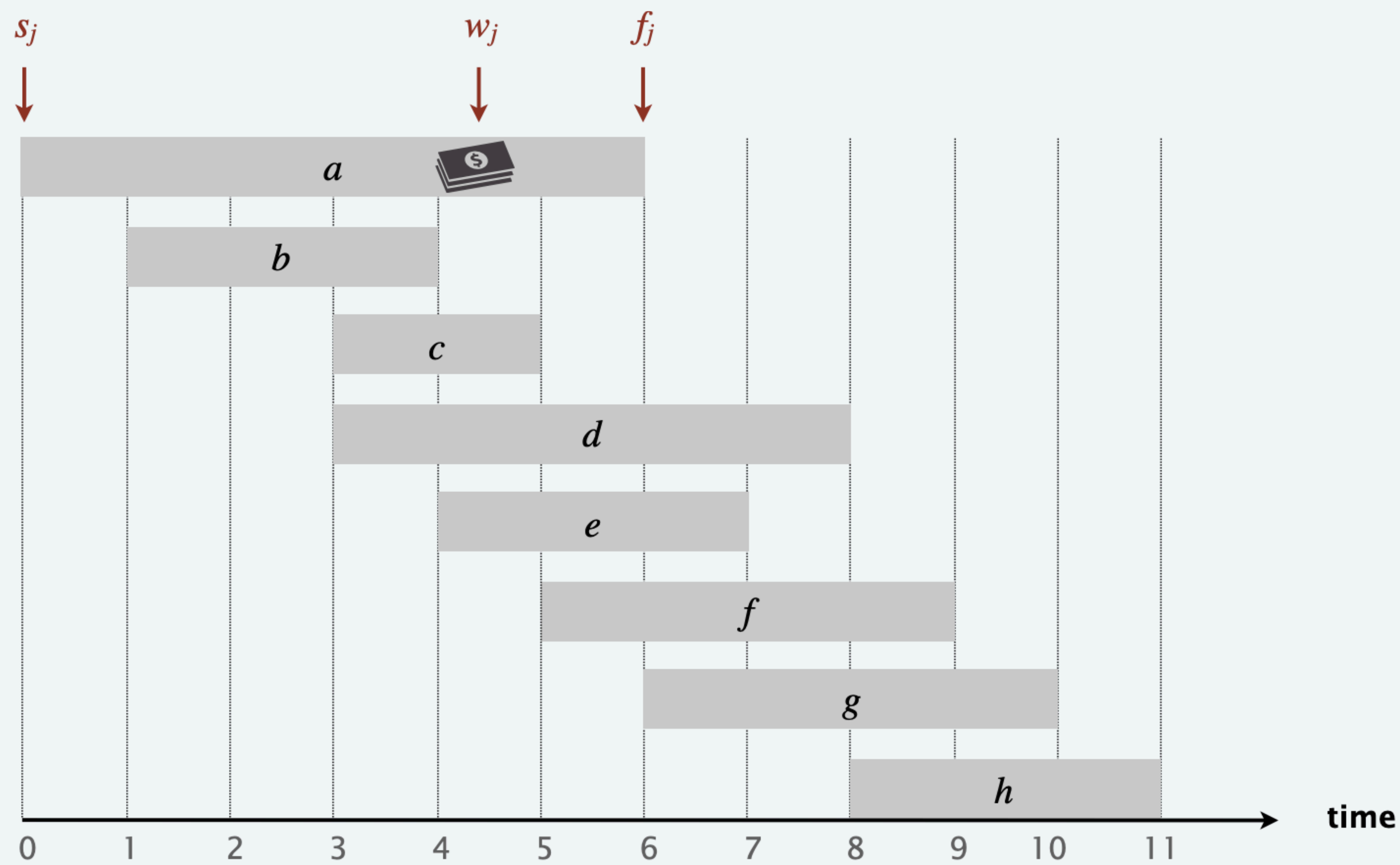
Weighted interval scheduling

Convention. Jobs are in ascending order of finish time: $f_1 \le f_2 \le \ldots \le f_n$.

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

Ex. $p(8) = 1, p(7) = 3, p(2) = 0$.

i is leftmost interval that ends before j begins



Interview Problem

💪 optional

Dynamic programming: binary choice

Def. $OPT(j)$ = max weight of any subset of mutually compatible jobs for subproblem consisting only of jobs $1, 2, \ldots, j$.

Goal. $OPT(n)$ = max weight of any subset of mutually compatible jobs.

Case 1. $OPT(j)$ does not select job j.

- Must be an optimal solution to problem consisting of remaining jobs $1, 2, \ldots, j-1$.

Case 2. $OPT(j)$ selects job j.

- Collect profit w_j.
- Can't use incompatible jobs $\{ p(j)+1, p(j)+2, \ldots, j-1 \}$.
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \ldots, p(j)$.

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

slide by Kevin Wayne