Design and Analysis of Algorithms
Part V: Greedy Algorithms

Lecture 10: The Fraction Knapsack Problem and The Huffman Coding Problem

Yongxin Tong (童咏昕)

School of CSE, Beihang University

yxtong@buaa.edu.cn

Outline

- Introduction to Part V

- The Fraction Knapsack Problem
  - Problem Definition
  - A Greedy Algorithm
  - Correctness

- Interval Scheduling and Interval Partitioning
  - Interval Scheduling
  - Interval Partitioning

Introduction to Greedy Algorithm

- A greedy algorithm for an optimization problem always makes the choice that looks best at the moment and adds it to the current subsolution.

Introduction to Greedy Algorithm

- A greedy algorithm for an optimization problem always makes the choice that looks best at the moment and adds it to the current subsolution.
- Examples already seen
  - Dijkstra’s shortest path algorithm:

Introduction to Greedy Algorithm

- A greedy algorithm for an optimization problem always makes the choice that looks best at the moment and adds it to the current subsolution.
- Examples already seen
  - Dijkstra’s shortest path algorithm: Select the node, among all "candidate" nodes, that is closest to the source according to estimation d[u].

An Example of Dijkstra’s Algorithm

color

W	W	W	W	W

pred

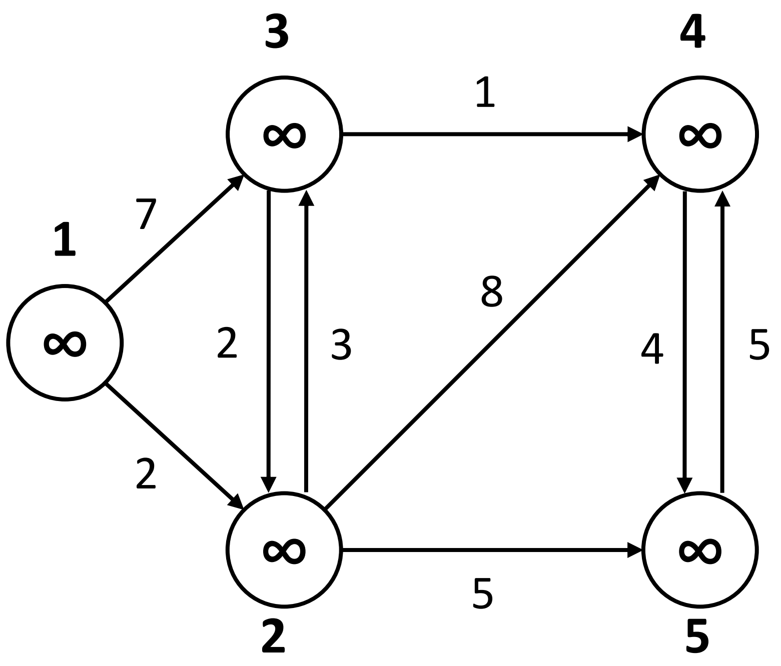
N	N	N	N	N

d

∞	∞	∞	∞	∞

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra’s Algorithm

color

W	W	W	W	W

pred

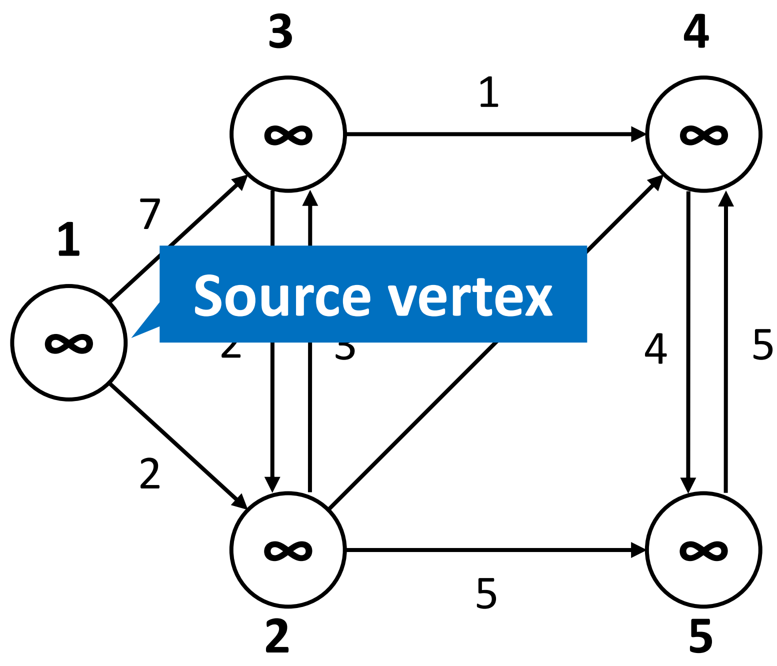
N	N	N	N	N

d

∞	∞	∞	∞	∞

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

W	W	W	W	W

pred

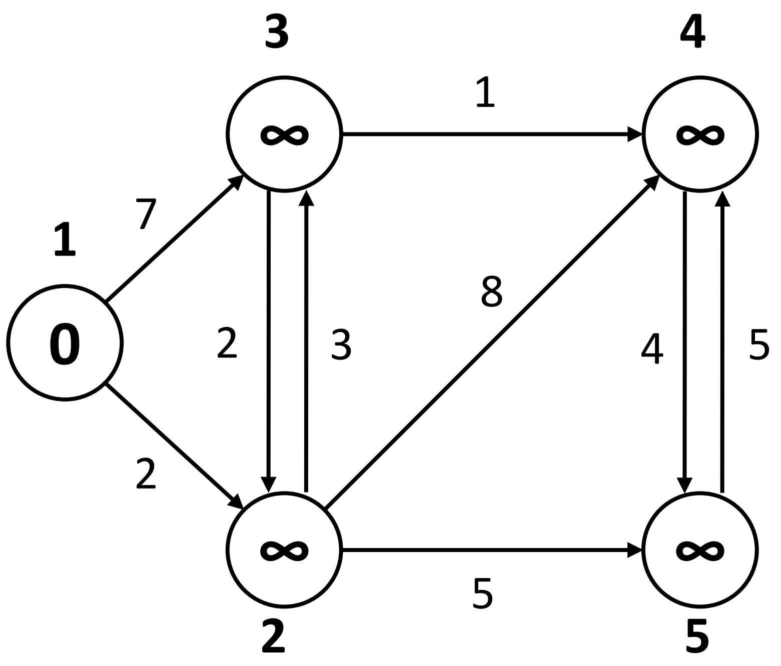
N	N	N	N	N

d

0	∞	∞	∞	∞

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra’s Algorithm

color

W	W	W	W	W

d

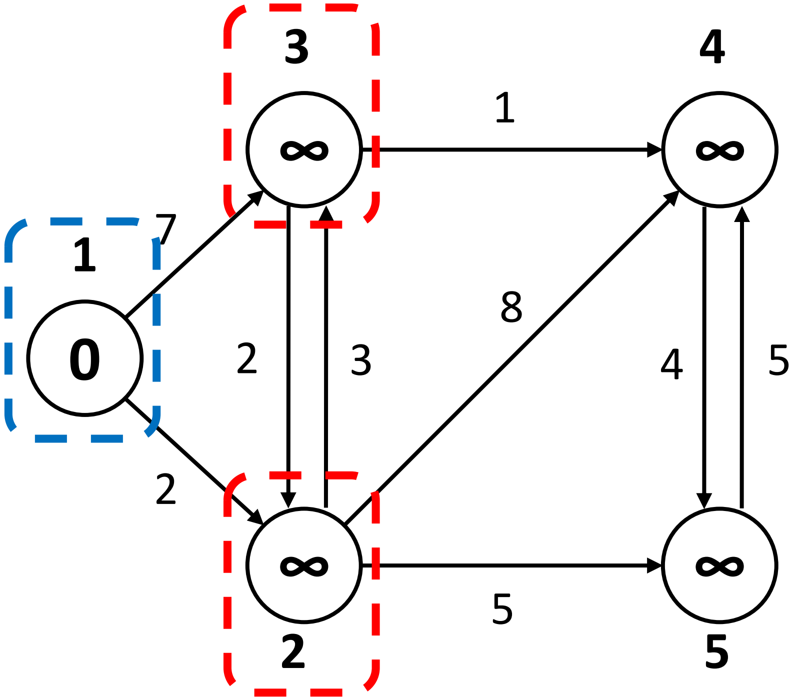
0	∞	∞	∞	∞

pred

N	N	N	N	N

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

W	W	W	W	W

pred

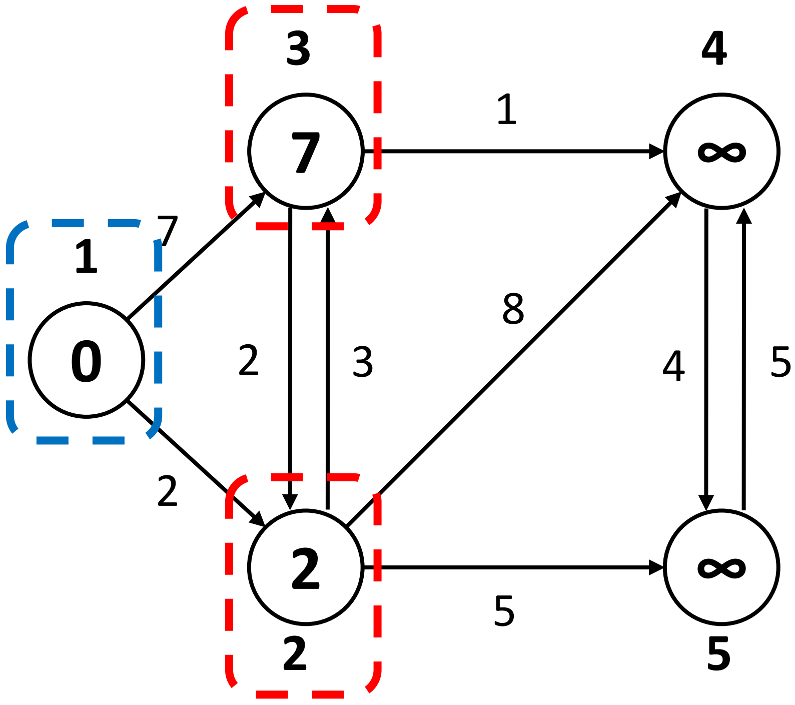
N	1	1	N	N

d

0	2	7	∞	∞

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	W	W	W	W

d

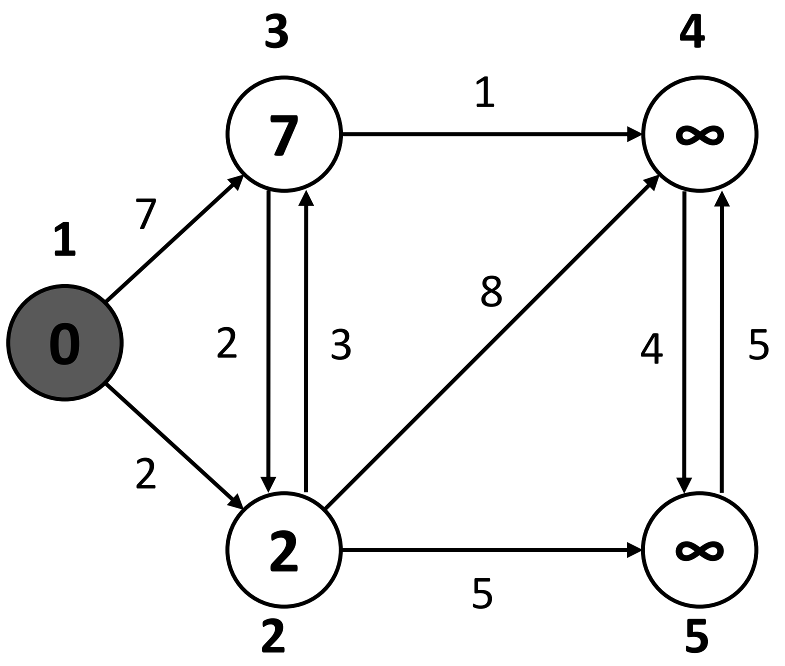
0	2	7	∞	∞

pred

N	1	1	N	N

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	W	W	W	W

pred

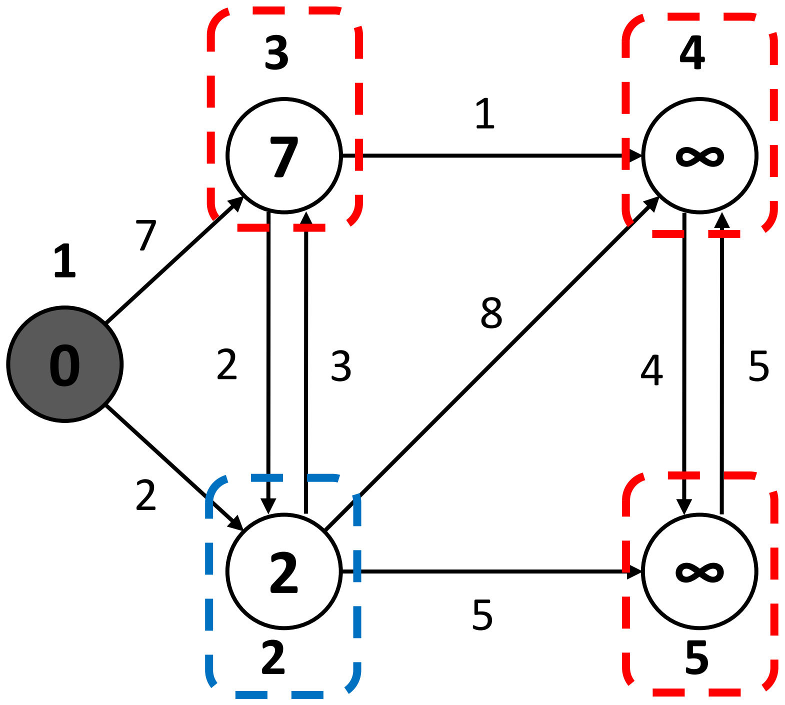
N	1	1	N	N

d

0	2	7	∞	∞

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	W	W	W	W

pred

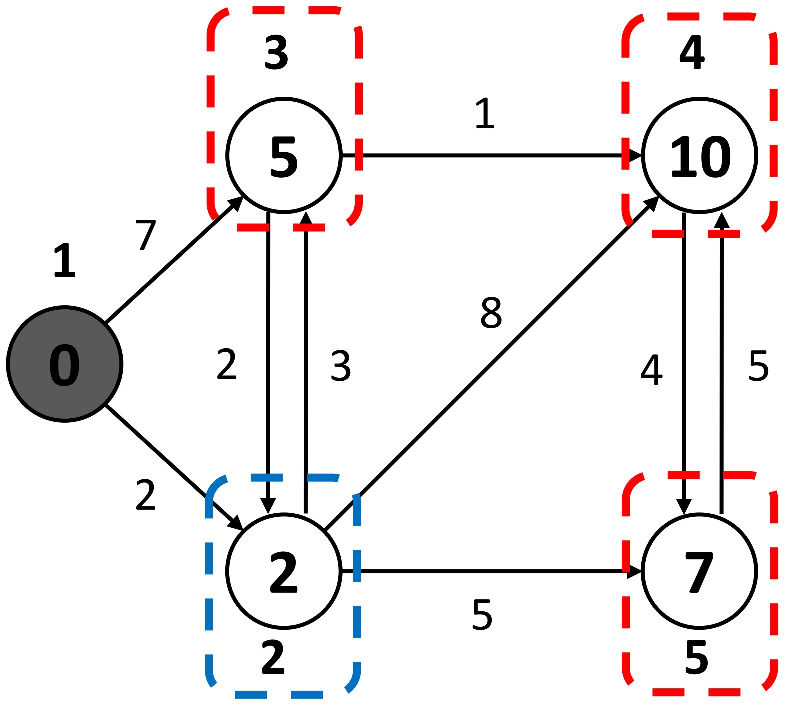
N	1	2	2	2

d

0	2	5	10	7

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	B	W	W	W

pred

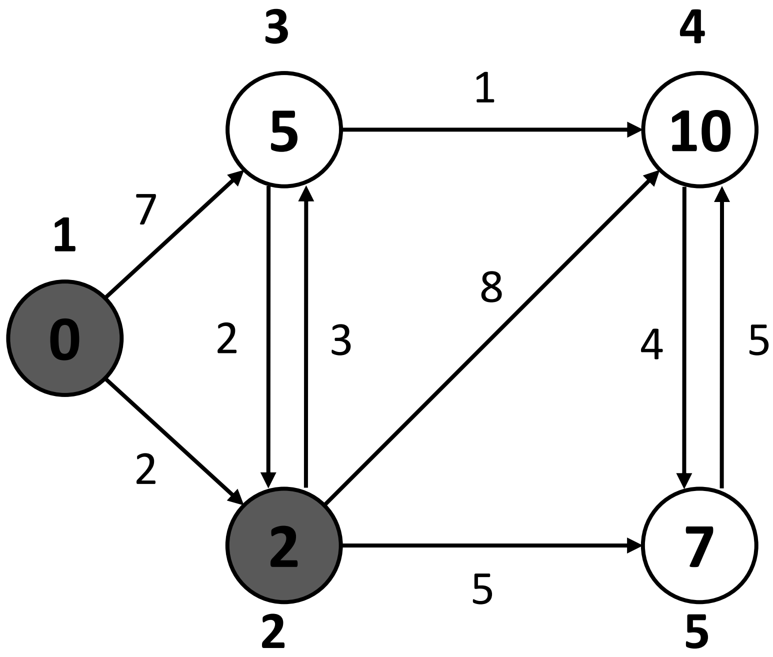
N	1	2	2	2

d

0	2	5	10	7

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	B	W	W	W

pred

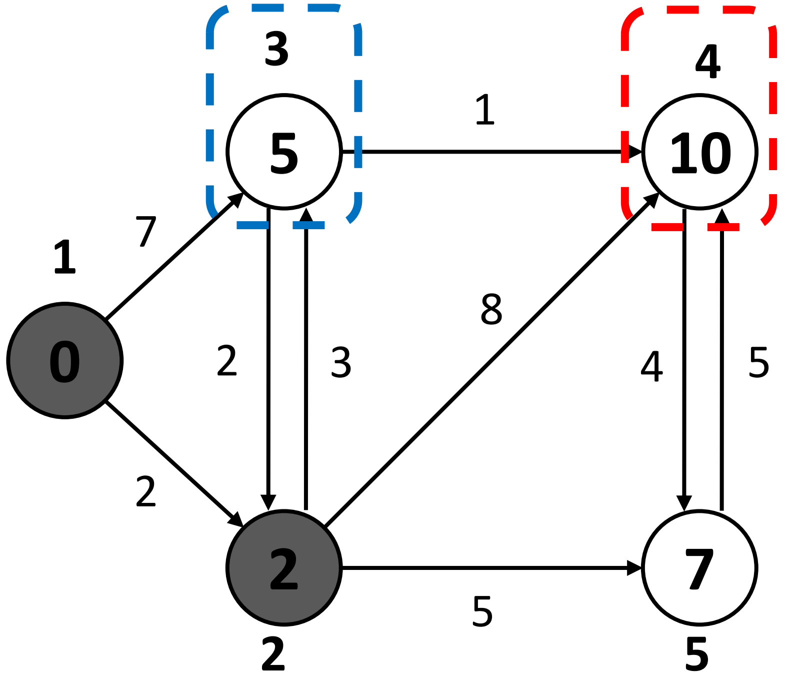
N	1	2	2	2

d

0	2	5	10	7

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	B	W	W	W

pred

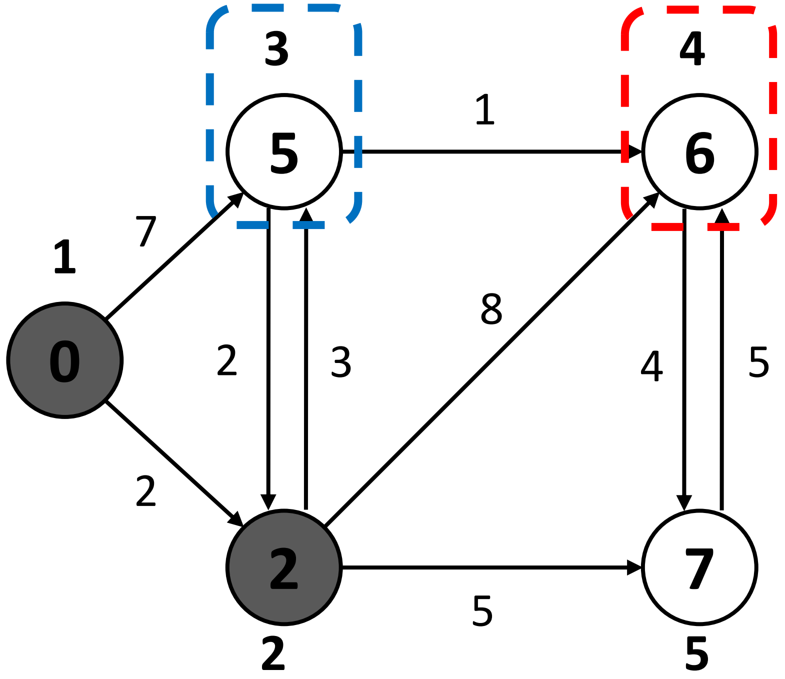
N	1	2	3	2

d

0	2	5	6	7

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	B	B	W	W

pred

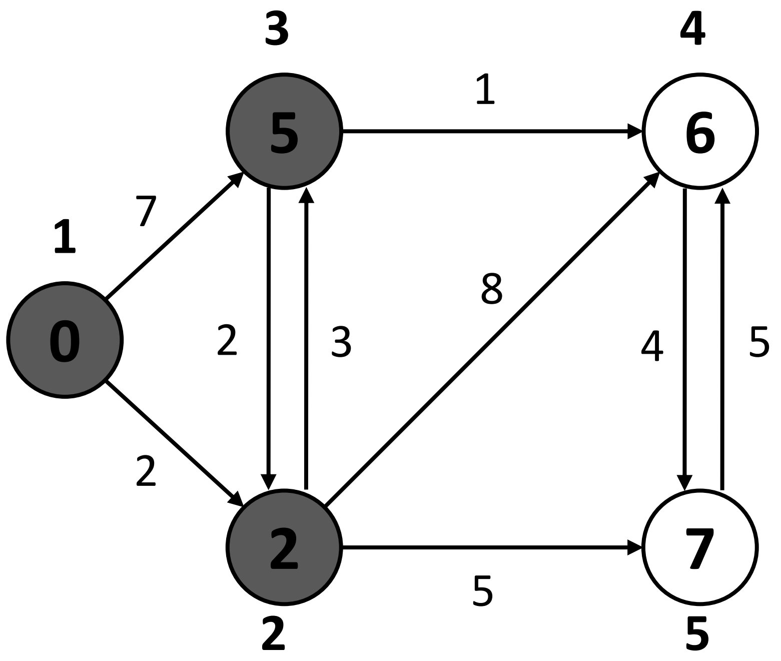
N	1	2	3	2

d

0	2	5	6	7

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	**B**	**B**	**W**	**W**

pred

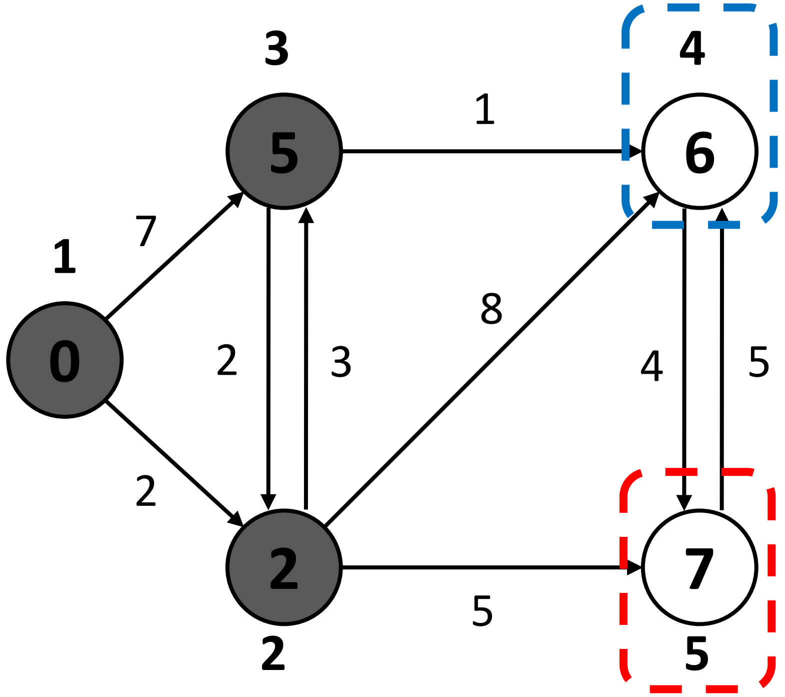
N	**1**	**2**	**3**	**2**

d

0	**2**	**5**	**6**	**7**

Q(Priority Queue)

1	**2**	**3**	**4**	**5**



An Example of Dijkstra's Algorithm

color

B	B	B	W	W

pred

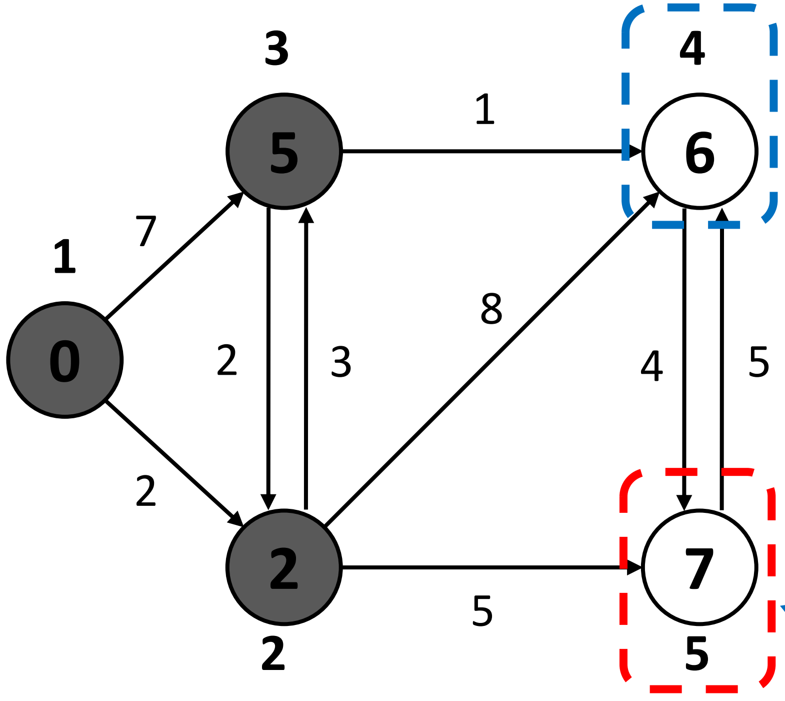
N	1	2	3	2

d

0	2	5	6	7

Q(Priority Queue)

1	2	3	4	5



d[5] is not needed to be updated because d[4]+4 > d[5]

An Example of Dijkstra’s Algorithm

color

B	**B**	**B**	**B**	**W**

pred

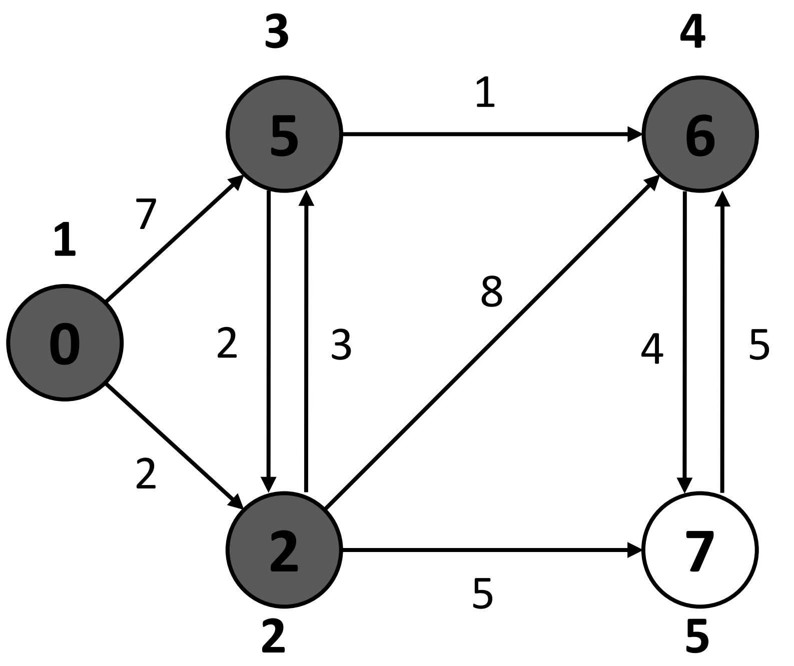
N	**1**	**2**	**3**	**2**

d

0	**2**	**5**	**6**	**7**

Q(Priority Queue)

~~**1**~~	~~**2**~~	~~**3**~~	~~**4**~~	**5**



An Example of Dijkstra's Algorithm

color

B	B	B	B	W

pred

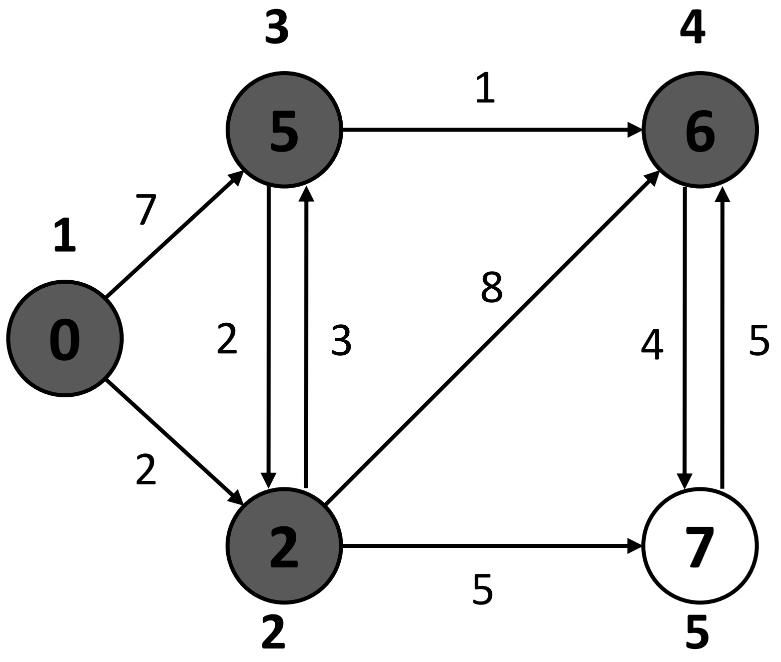
N	1	2	3	2

d

0	2	5	6	7

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	B	B	B	B

pred

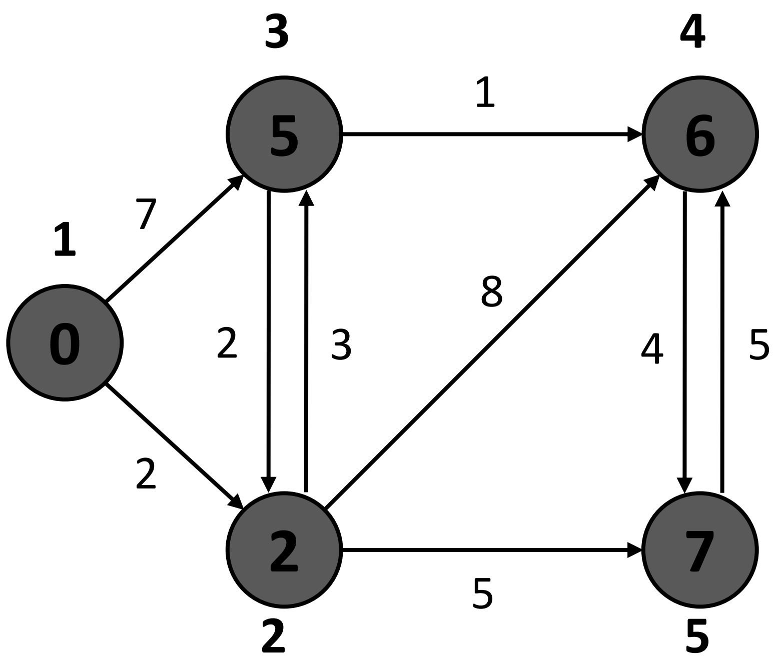
N	1	2	3	2

d

0	2	5	6	7

Q(Priority Queue)

1	2	3	4	5



An Example of Dijkstra's Algorithm

color

B	**B**	**B**	**B**	**B**

d

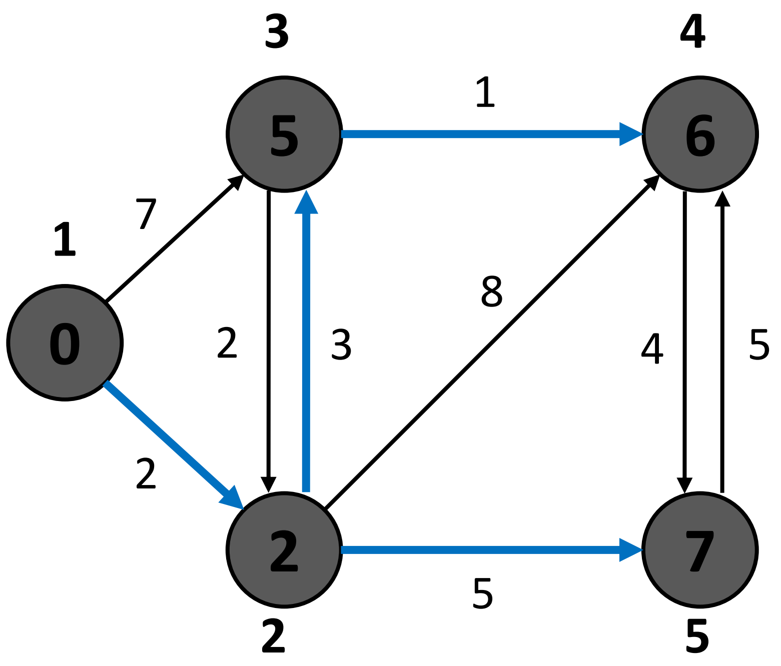
0	**2**	**5**	**6**	**7**

pred

N	**1**	**2**	**3**	**2**

Q(Priority Queue)

1	2	3	4	5

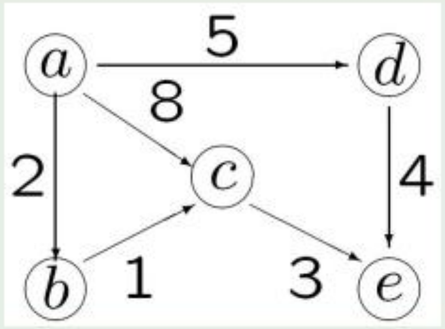


Optimal Substructure Property

Lemma

Any sub-path of a shortest path must also be a shortest path

Example



$\langle a, b, c, e \rangle$ is a shortest path; sub-path $\langle a, b, c \rangle$ is also a shortest path.

Introduction to Greedy Algorithm

- A greedy algorithm for an optimization problem always makes the choice that looks best at the moment and adds it to the current subsolution.
- Examples already seen
  - Dijkstra’s shortest path algorithm: Select the node, among all "candidate" nodes, that is closest to the source according to estimation d[u].
  - Prim/Kruskal’s MST algorithms: Select the edge, among all "candidate" edges, that is the lightest.

Prim's Example

color

W	W	W	W	W	W	W

pred

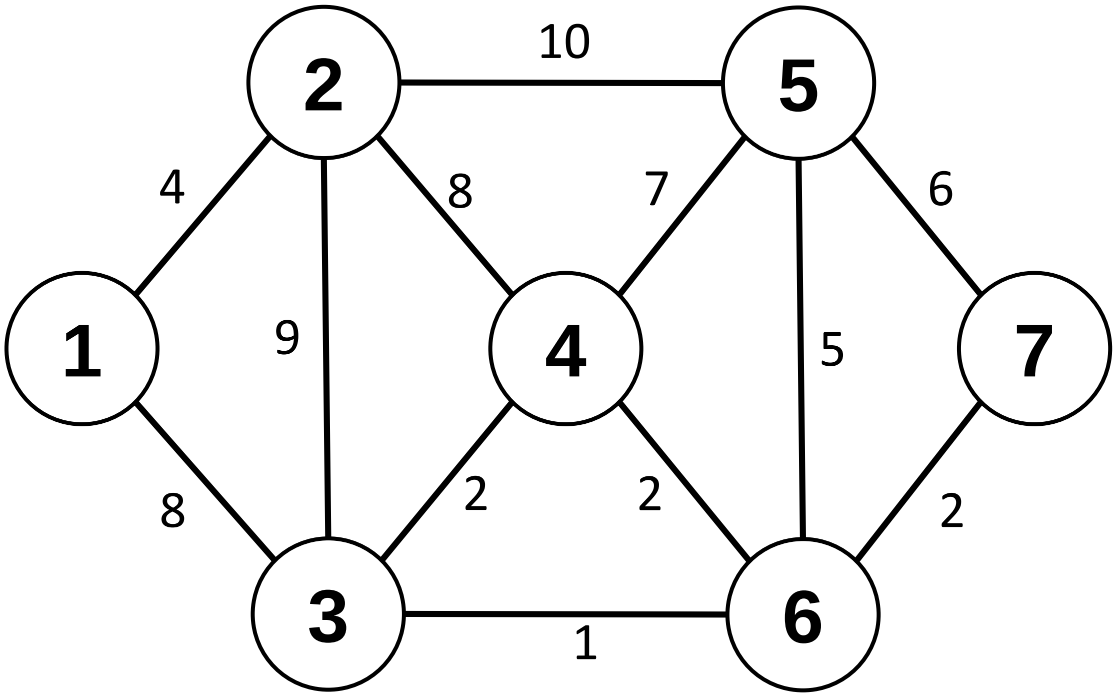
N	N	N	N	N	N	N

key

∞	∞	∞	∞	∞	∞	∞

Q

1,∞	2,∞	3,∞	4,∞	5,∞	6,∞	7,∞



Prim's Example

color

W	W	W	W	W	W	W

pred

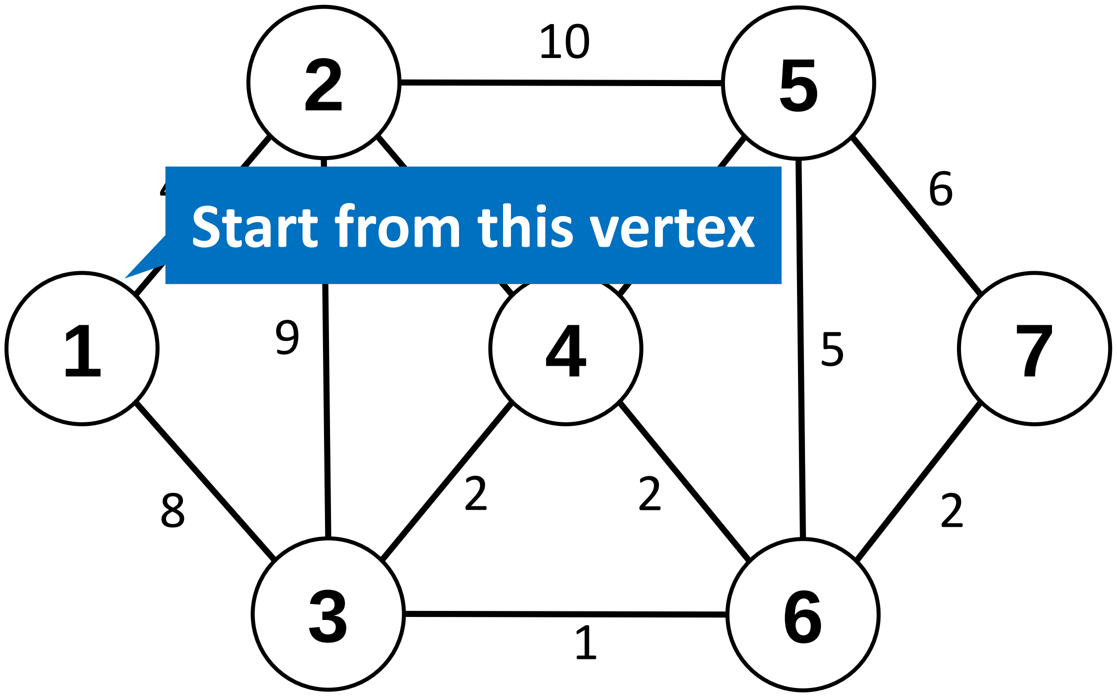
N	N	N	N	N	N	N

key

∞	∞	∞	∞	∞	∞	∞

Q

1,∞	2,∞	3,∞	4,∞	5,∞	6,∞	7,∞



Prim's Example

color

B	W	W	W	W	W	W

pred

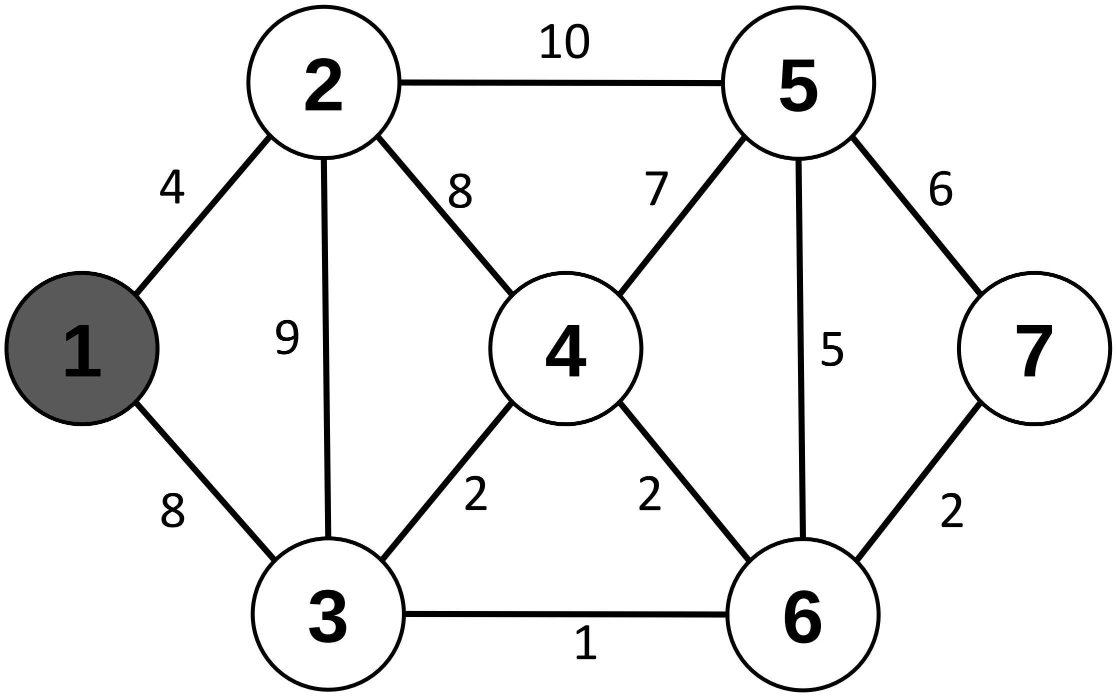
N	N	N	N	N	N	N

key

0	∞	∞	∞	∞	∞	∞

Q

1,0	2,∞	3,∞	4,∞	5,∞	6,∞	7,∞



Prim's Example

color

B	W	W	W	W	W	W

pred

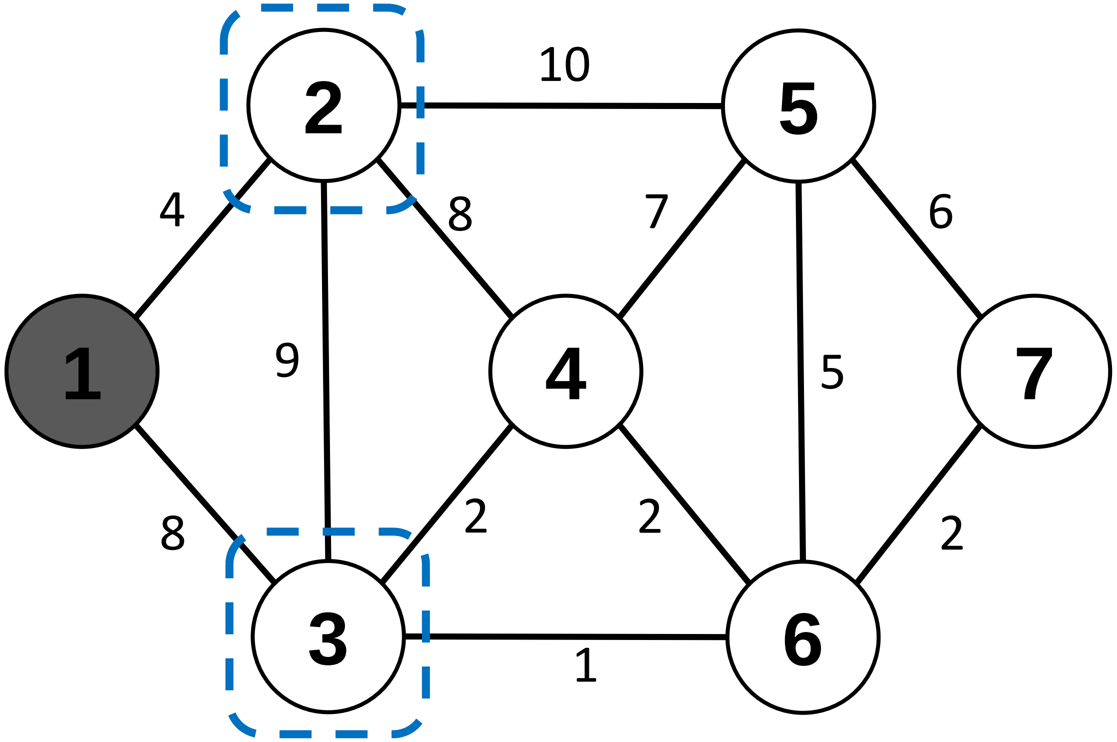
N	N	N	N	N	N	N

key

0	∞	∞	∞	∞	∞	∞

Q

7,∞	2,∞	3,∞	4,∞	5,∞	6,∞



Prim's Example

color

B	W	W	W	W	W	W

pred

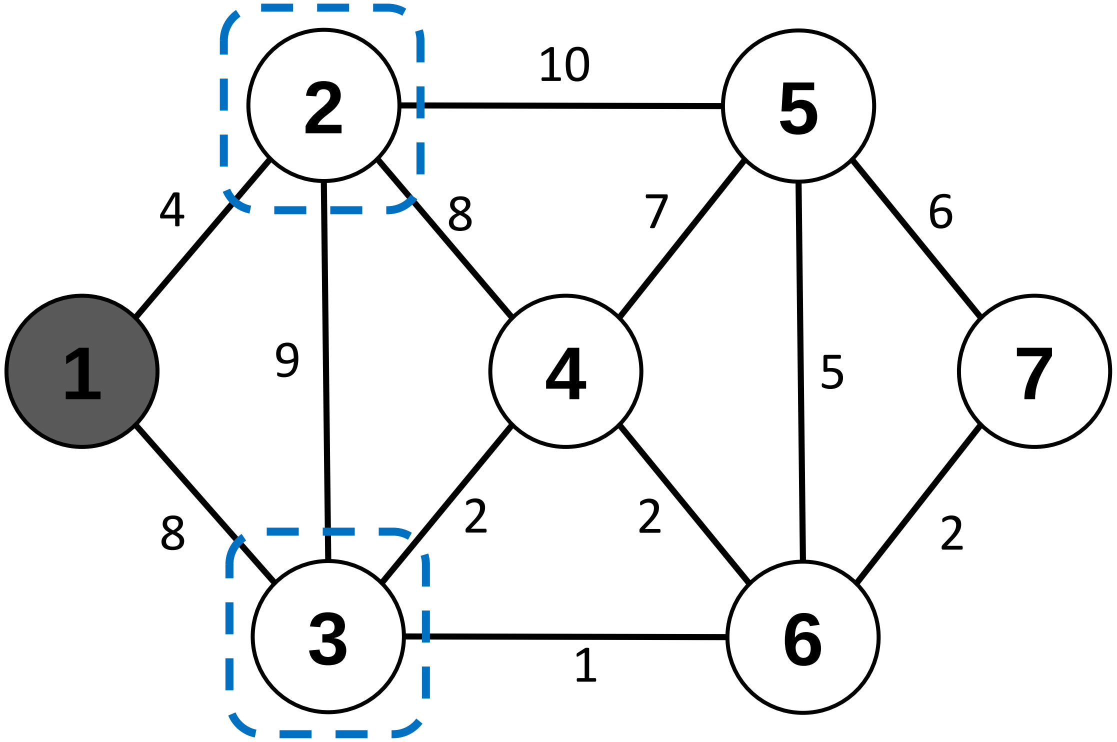
N	1	1	N	N	N	N

key

0	4	8	∞	∞	∞	∞

Q

7,∞	2,4	3,8	4,∞	5,∞	6,∞



Prim's Example

color

B	W	W	W	W	W	W

pred

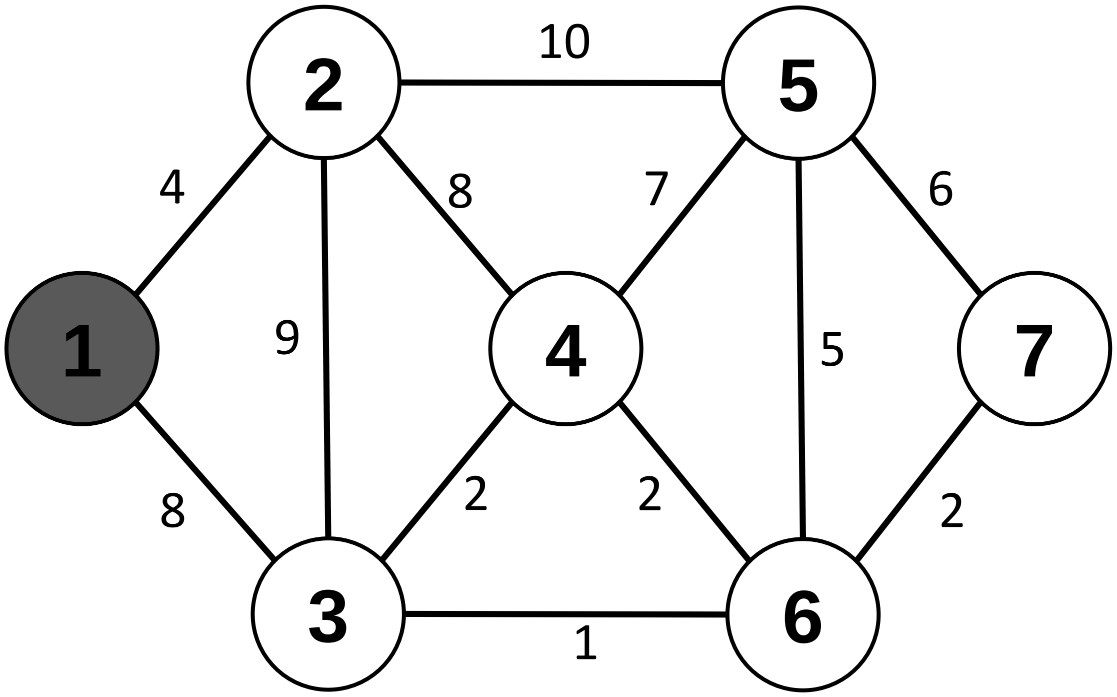
N	1	1	N	N	N	N

key

0	4	8	∞	∞	∞	∞

Q

2,4	7,∞	3,8	4,∞	5,∞	6,∞



Prim's Example

color

B	B	W	W	W	W	W

pred

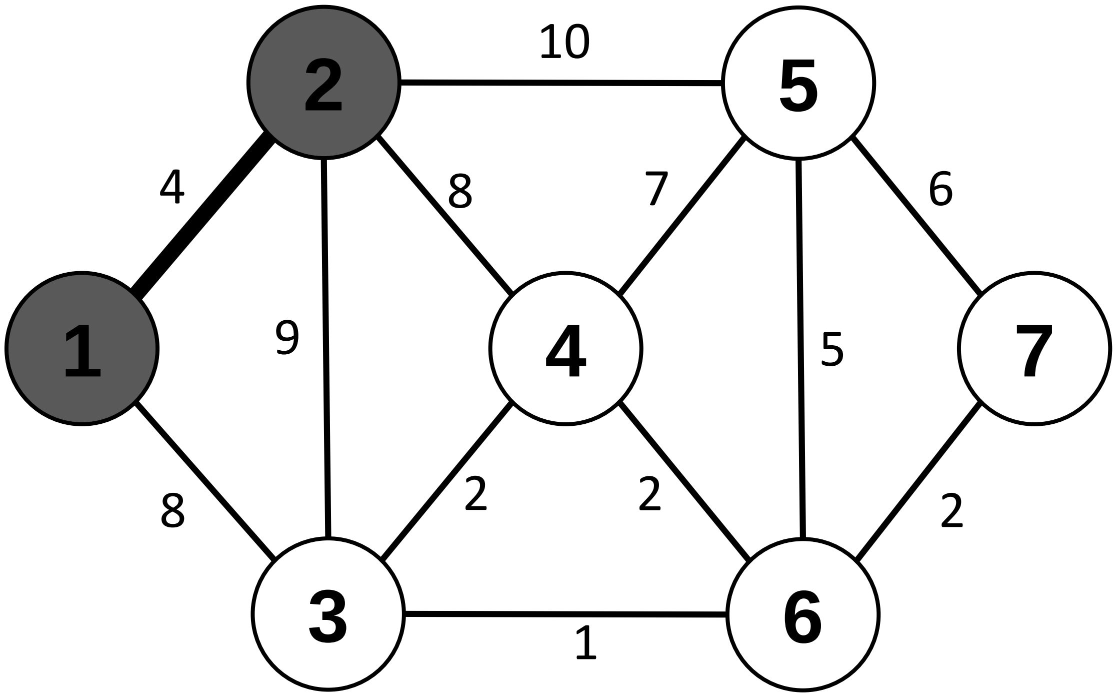
N	1	1	N	N	N	N

key

0	4	8	∞	∞	∞	∞

Q

3,8	7,∞	6,∞	4,∞	5,∞



Prim's Example

color

B	B	W	W	W	W	W

pred

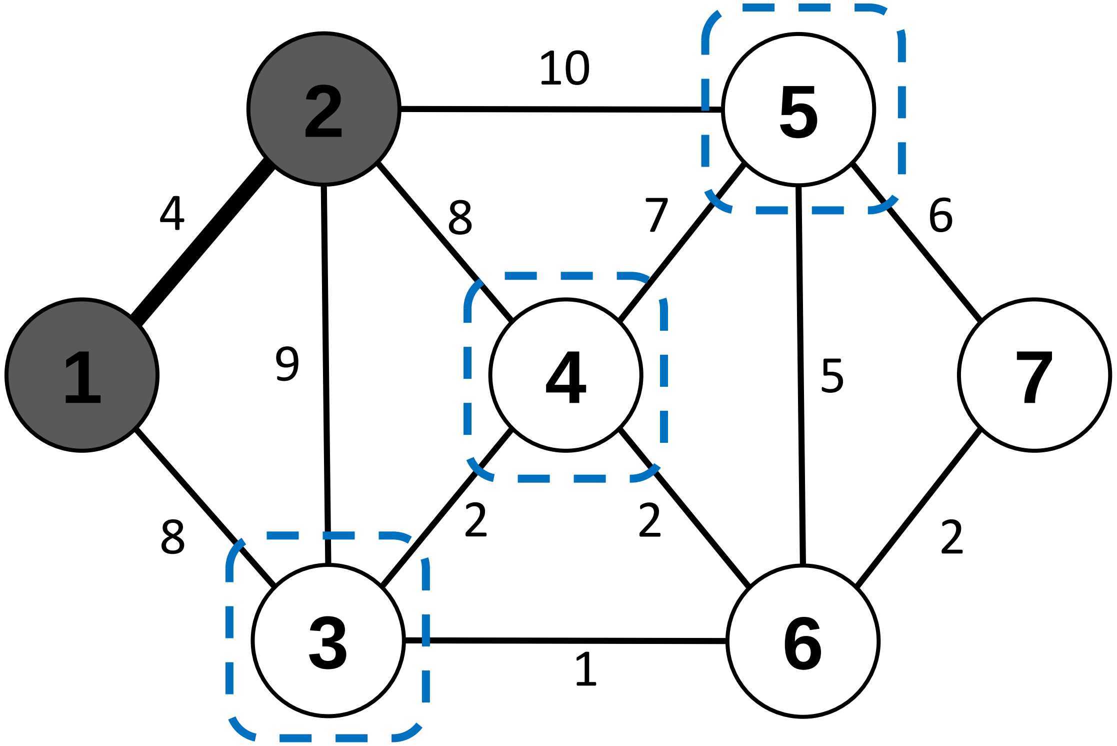
N	1	1	N	N	N	N

key

0	4	8	∞	∞	∞	∞

Q

3,8	7,∞	6,∞	4,∞	5,∞



Prim's Example

color

B	B	W	W	W	W	W

pred

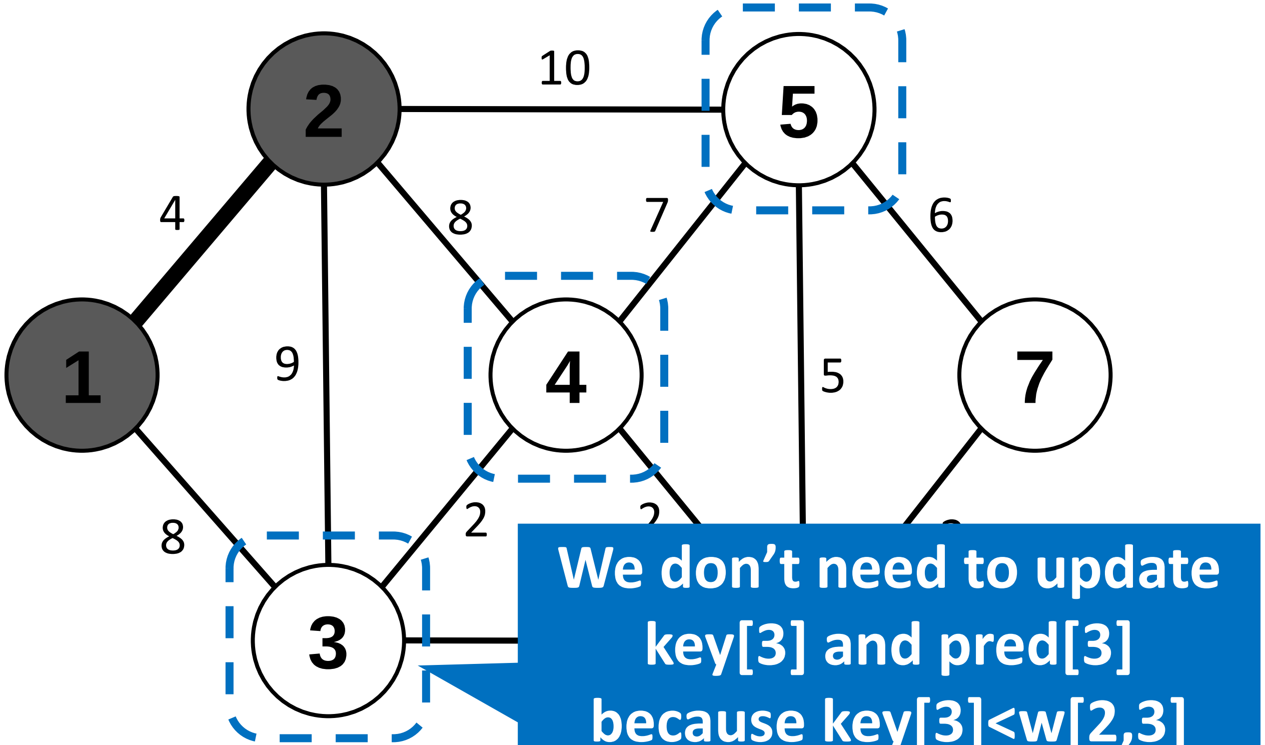
N	1	1	2	2	N	N

key

0	4	8	8	10	∞	∞

Q

3,8	7,∞	6,∞	4,8	5,10



We don’t need to update key[3] and pred[3] because key[3]<w[2,3]

Prim's Example

color

B	B	W	W	W	W	W

pred

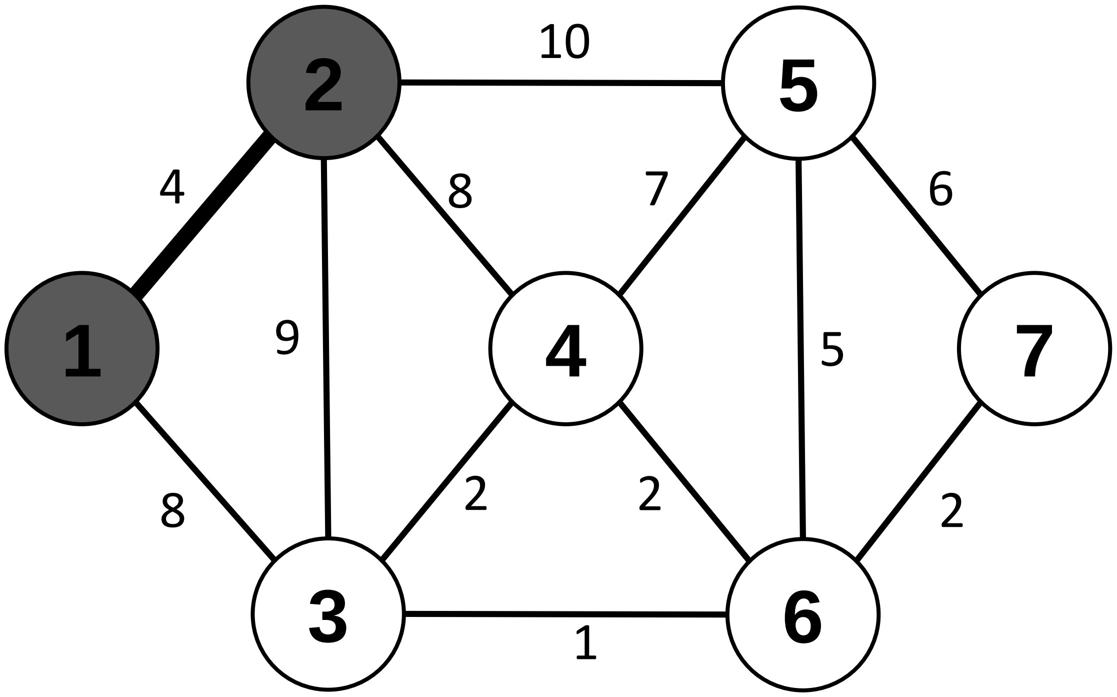
N	1	1	2	2	N	N

key

0	4	8	8	10	∞	∞

Q

3,8	4,8	6,∞	7,∞	5,10



Prim's Example

color

B	B	B	W	W	W	W

pred

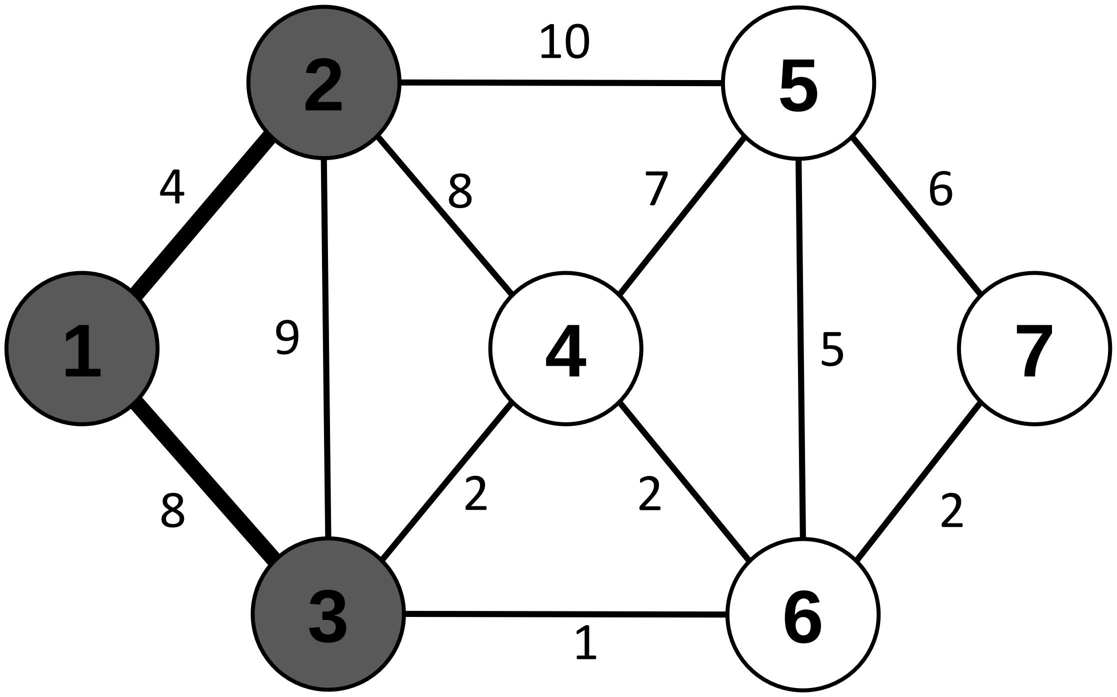
N	1	1	2	2	N	N

key

0	4	8	8	10	∞	∞

Q

4,8	5,10	6,∞	7,∞



Prim's Example

color

B	B	B	W	W	W	W

pred

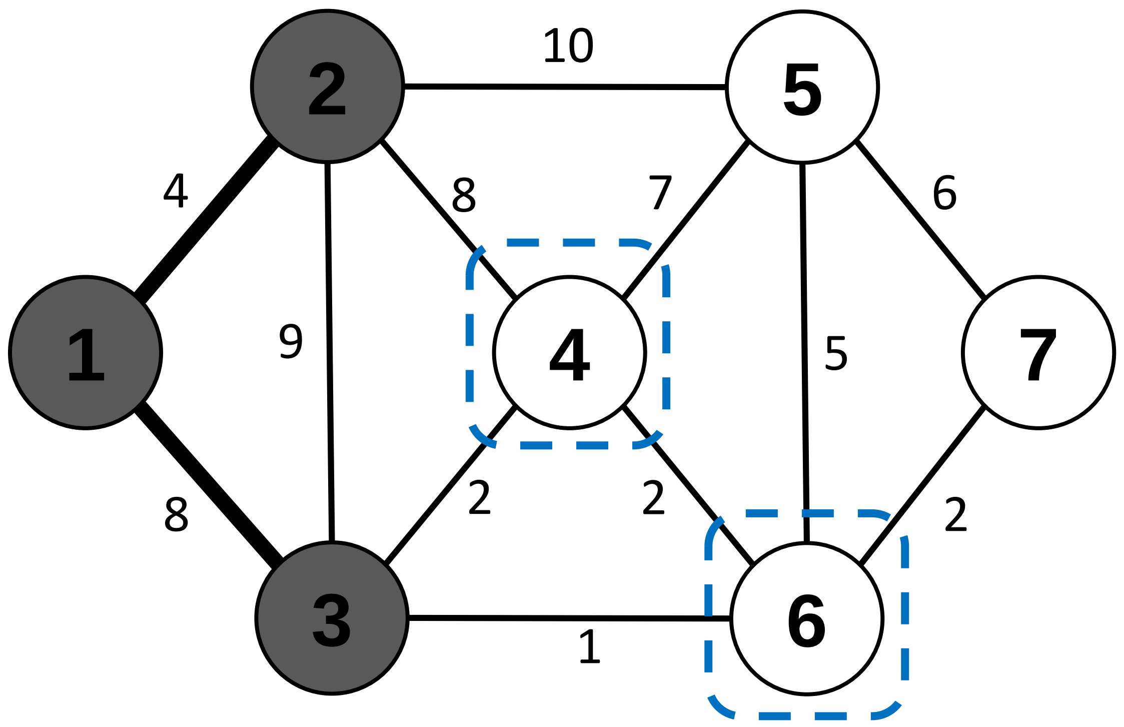
N	1	1	2	2	N	N

key

0	4	8	8	10	∞	∞

Q

4,8	5,10	6,∞	7,∞



Prim's Example

color

B	B	B	W	W	W	W

pred

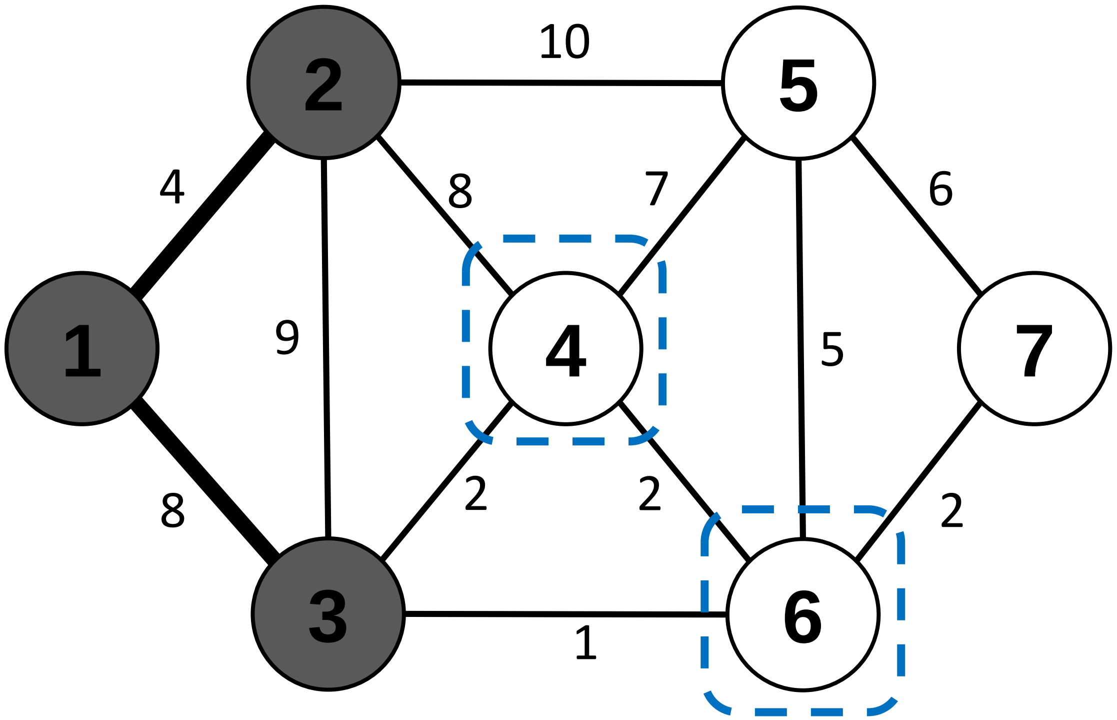
N	1	1	3	2	3	N

key

0	4	8	2	10	1	∞

Q

4,2	5,10	6,1	7,∞



Prim's Example

color

B	B	B	W	W	W	W

pred

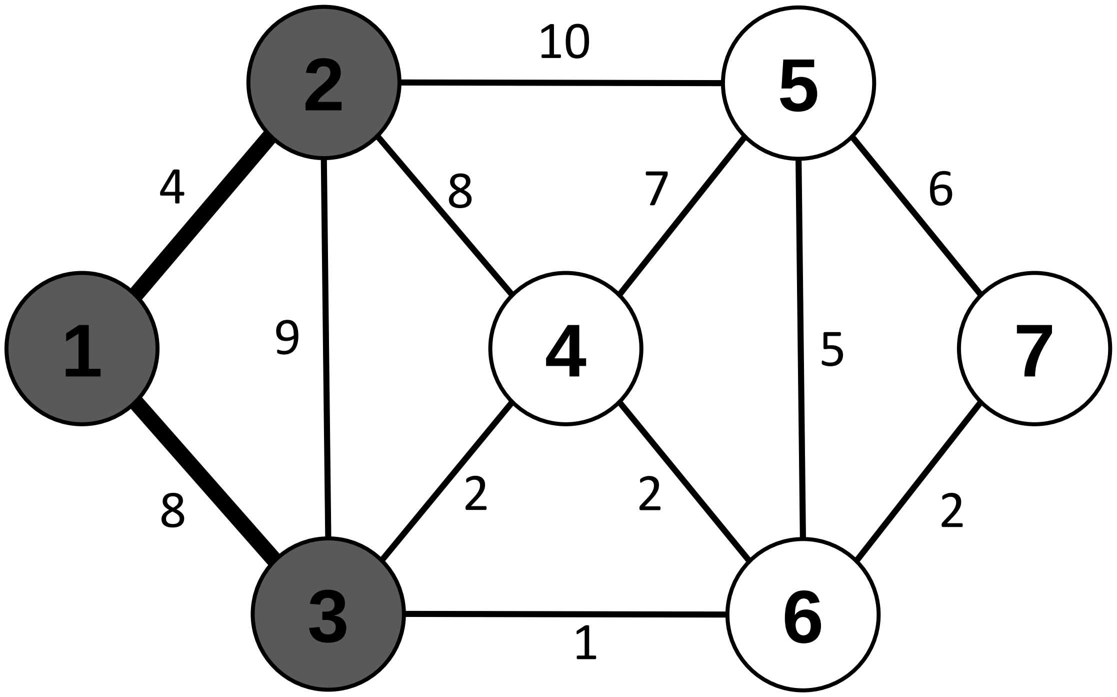
N	1	1	3	2	3	N

key

0	4	8	2	10	1	∞

Q

6,1	5,10	4,2	7,∞



Prim's Example

color

B	B	B	W	W	B	W

pred

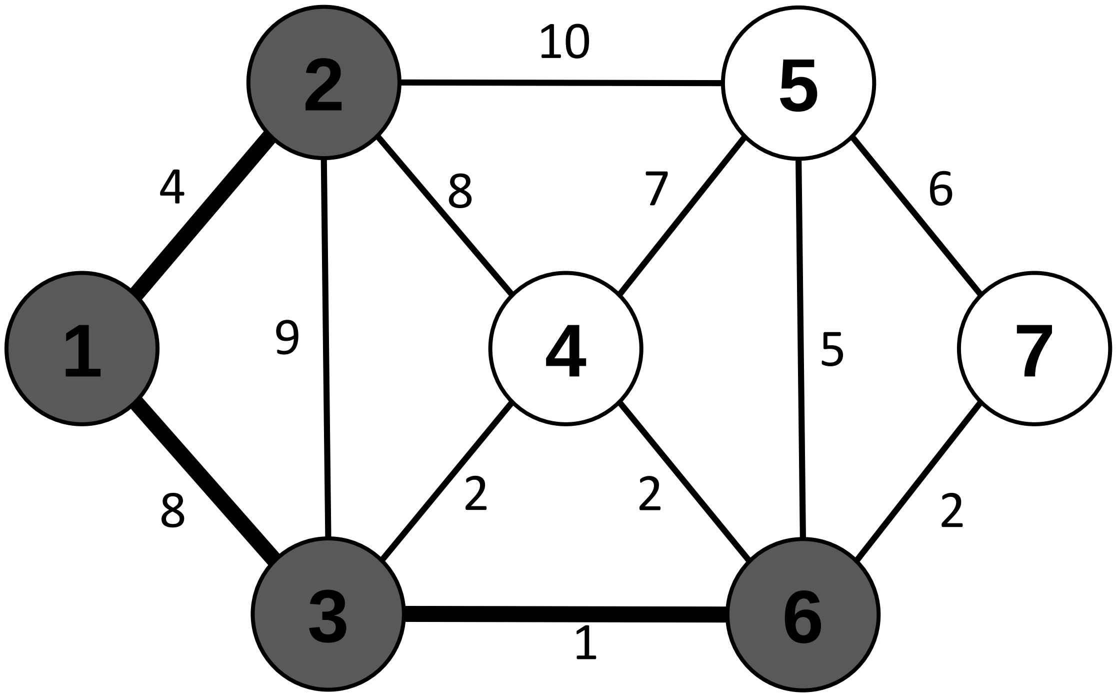
N	1	1	3	2	3	N

key

0	4	8	2	10	1	∞

Q

4,2	5,10	7,∞



Prim's Example

color

B	B	B	W	W	B	W

pred

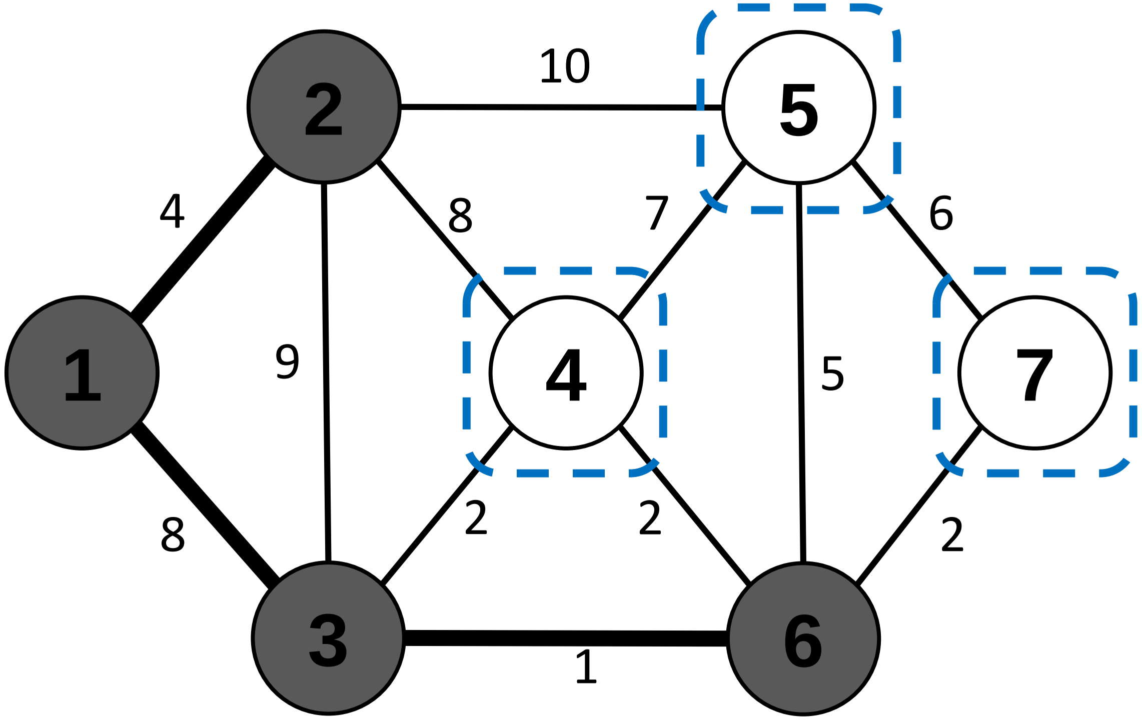
N	1	1	3	2	3	N

key

0	4	8	2	10	1	∞

Q

4,2	5,10	7,∞



Prim's Example

color

B	B	B	W	W	B	W

pred

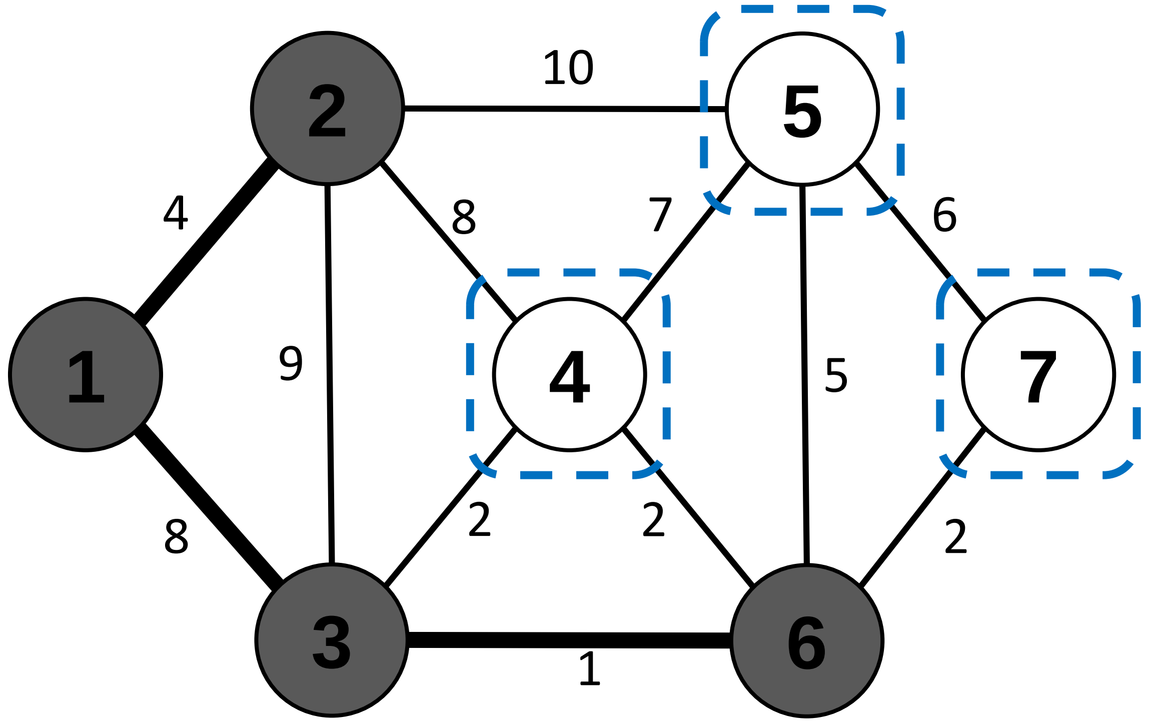
N	1	1	3	6	3	6

key

0	4	8	2	5	1	2

Q

4,2	5,5	7,2



Prim's Example

color

B	B	B	W	W	B	W

pred

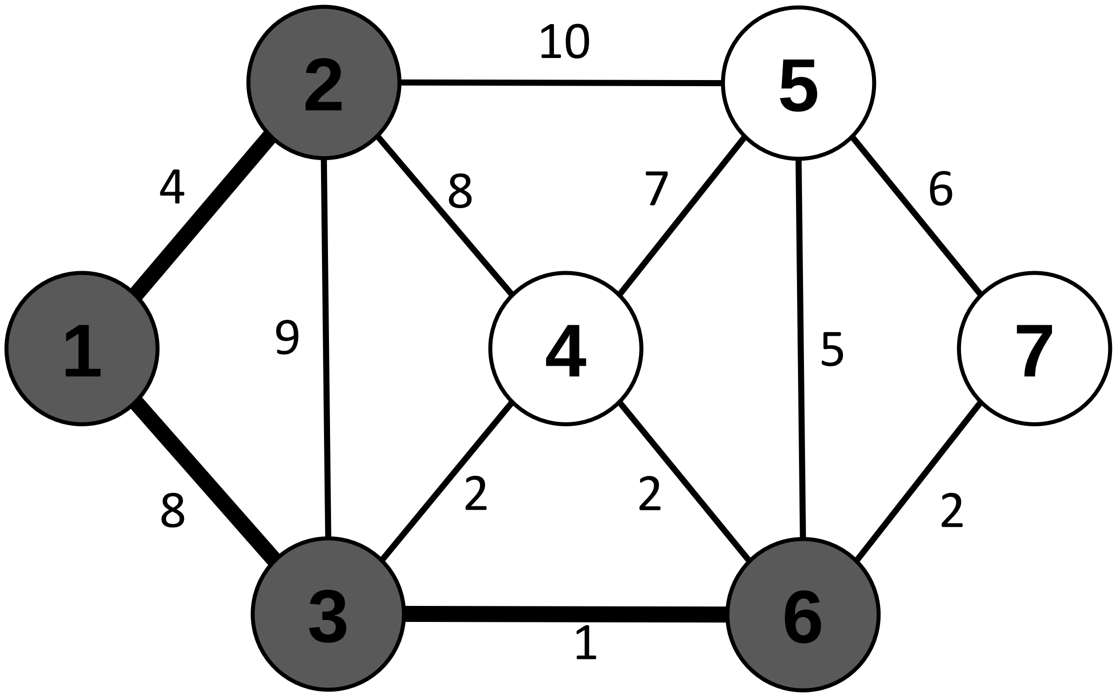
N	1	1	3	6	3	6

key

0	4	8	2	5	1	2

Q

4,2	5,5	7,2



Prim's Example

color

B	B	B	B	W	B	W

pred

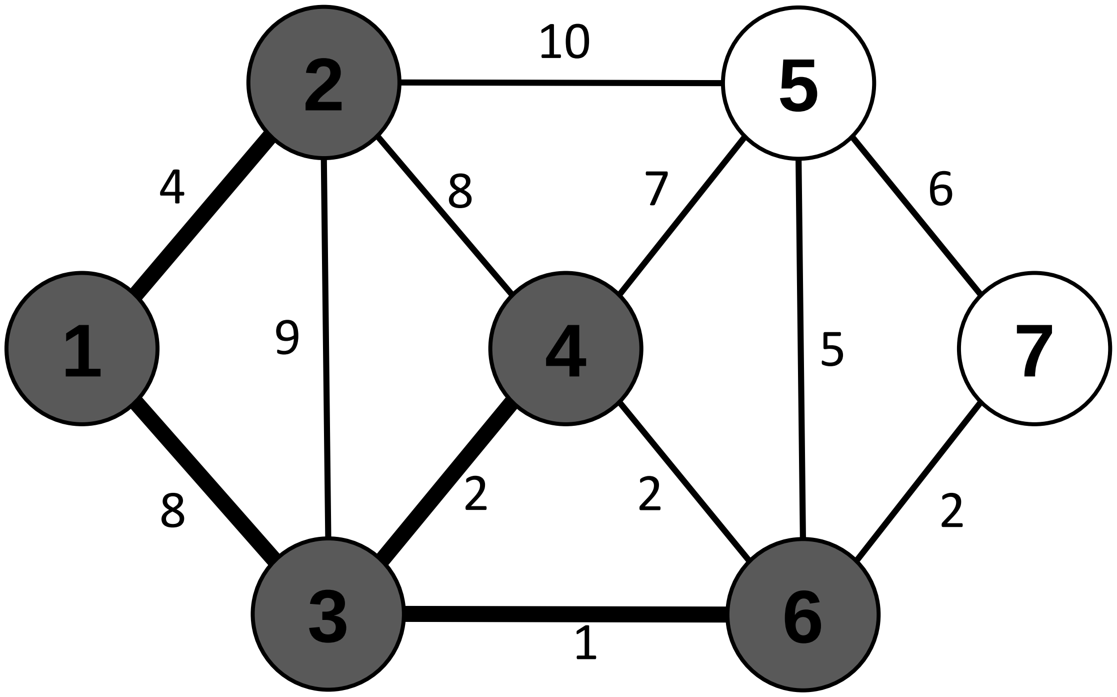
N	1	1	3	6	3	6

key

0	4	8	2	5	1	2

Q

7,2	5,5



Prim's Example

color

B	B	B	B	W	B	W

pred

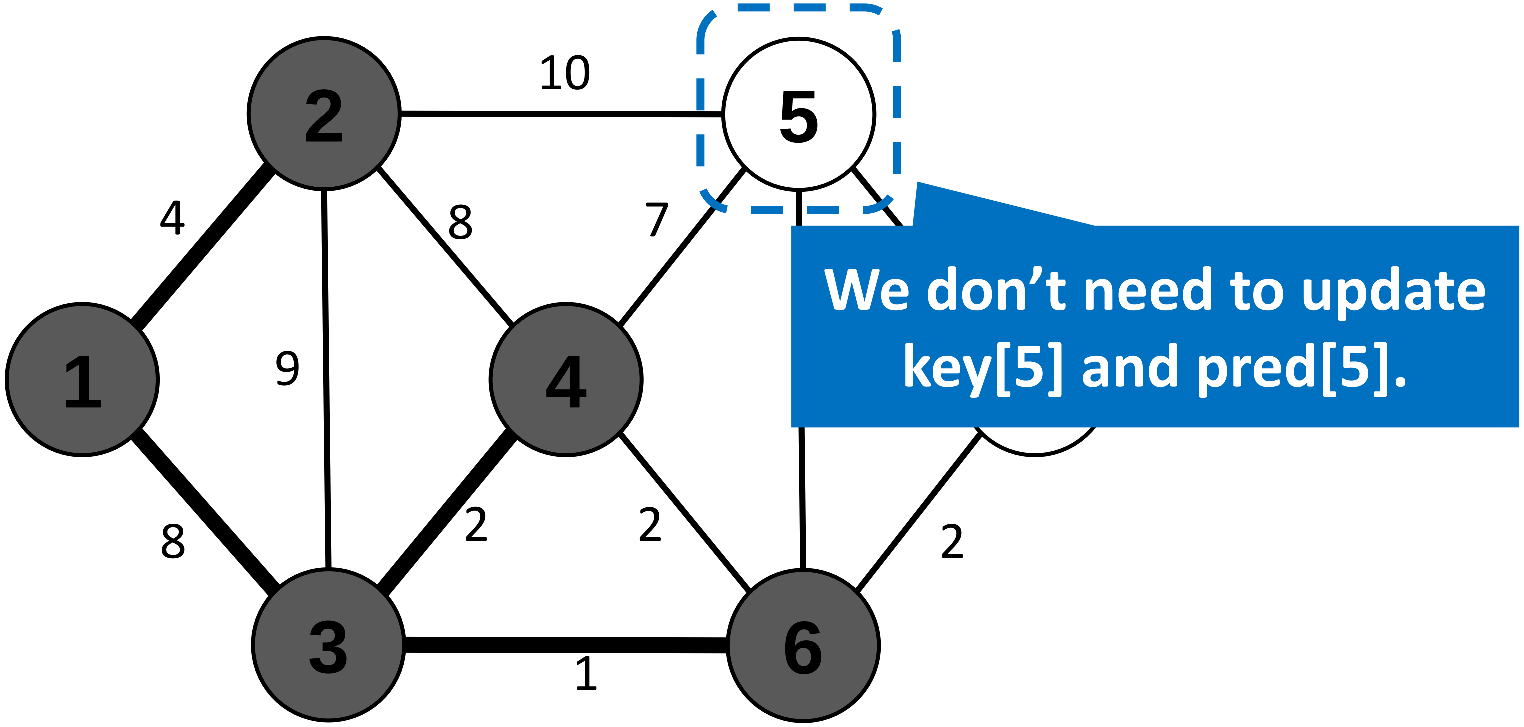
N	1	1	3	6	3	6

key

0	4	8	2	5	1	2

Q

7,2	5,5



We don't need to update key[5] and pred[5].

Prim's Example

color

B	B	B	B	W	B	W

pred

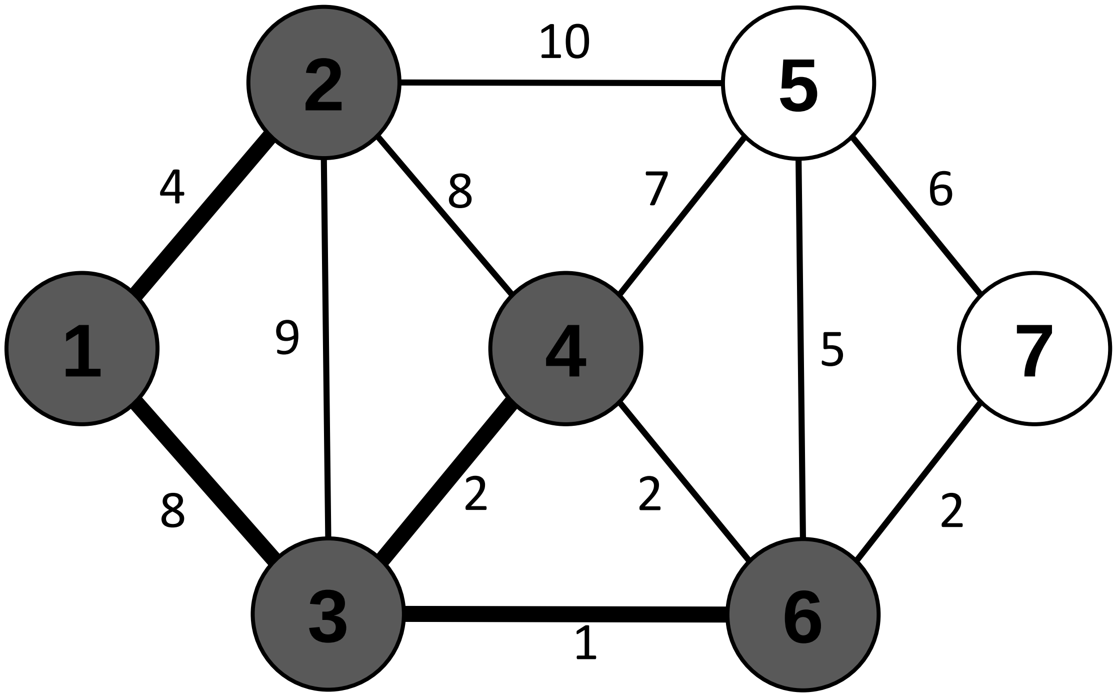
N	1	1	3	6	3	6

key

0	4	8	2	5	1	2

Q

7,2	5,5



Prim's Example

color

B	B	B	B	W	B	B

pred

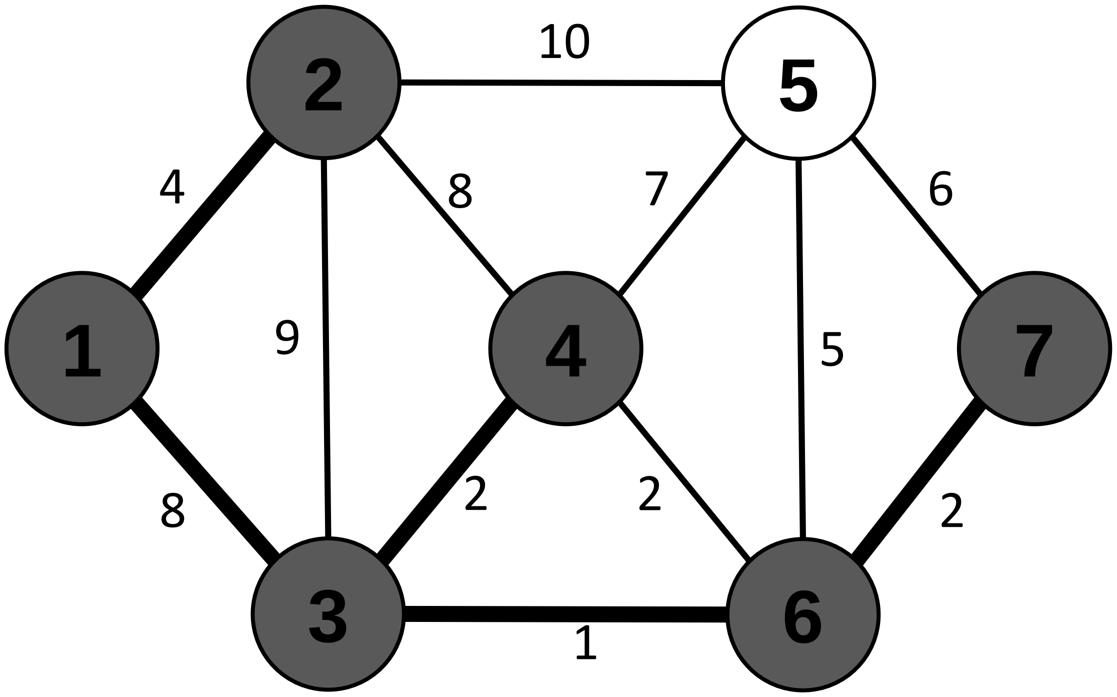
N	1	1	3	6	3	6

key

0	4	8	2	5	1	2

Q

5,5



Prim's Example

color

B	B	B	B	W	B	B

pred

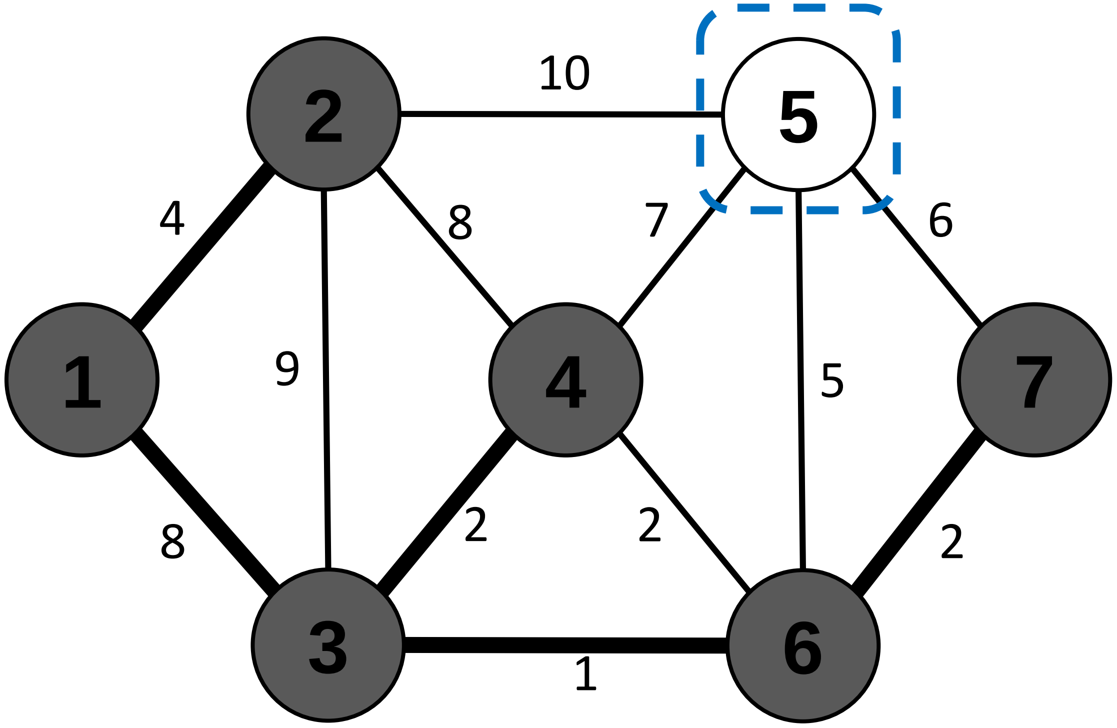
N	1	1	3	6	3	6

key

0	4	8	2	5	1	2

Q

5,5



Prim's Example

color

B	B	B	B	W	B	B

pred

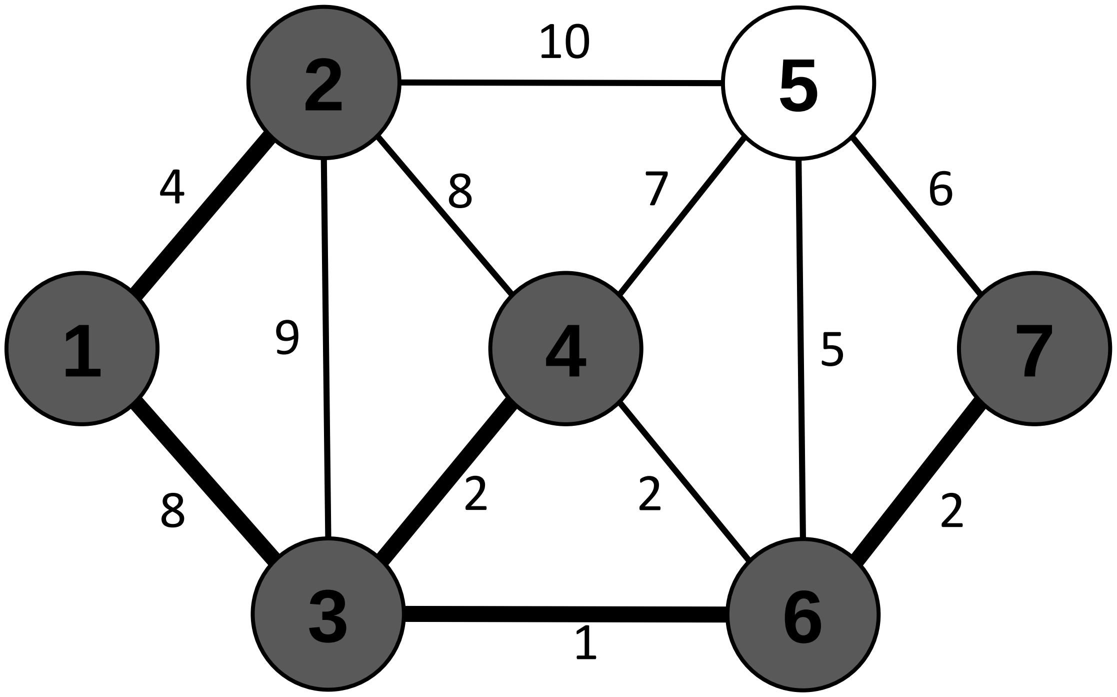
N	1	1	3	6	3	6

key

0	4	8	2	5	1	2

Q

5,5



Prim's Example

color

B	B	B	B	B	B	B

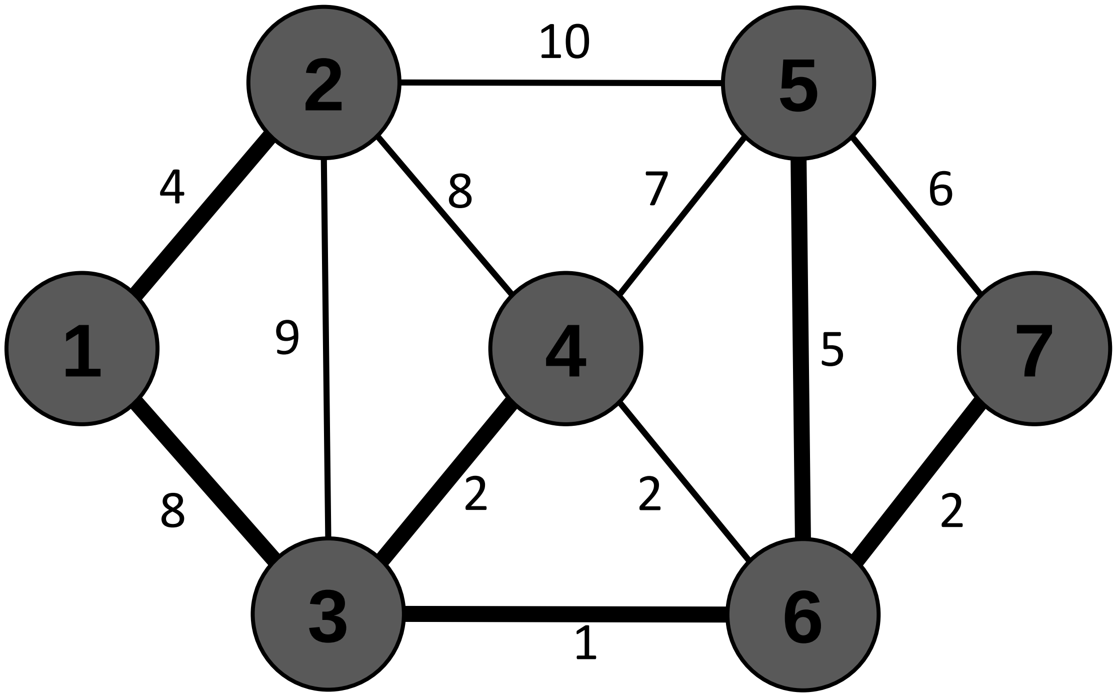
pred

N	1	1	3	6	3	6

key

0	4	8	2	5	1	2

Q



Prim's Example

color

B	B	B	B	B	B	B

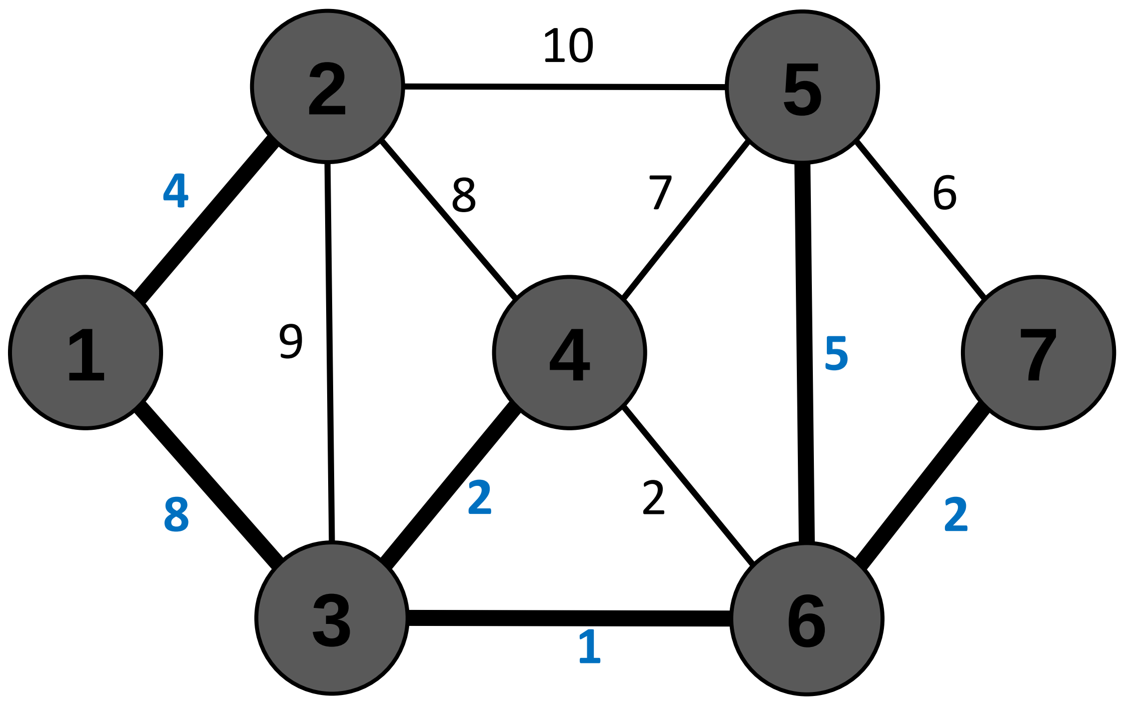
key

0	4	8	2	5	1	2

pred

N	1	1	3	6	3	6

Weight of MST = **22**



Generic Algorithm for MST problem

Definition

Let A be a set of edges such that $A \subseteq T$, where T is a MST. An edge (u, v) is a safe edge for A, if $A \cup \{(u, v)\}$ is also a subset of some MST

- If at each step, we can find a safe edge (u, v), we can grow a MST

Generic-MST(G)

```
Input: A graph $G$
Output: $A$ is the MST of $G$
$A \leftarrow$ EMPTY;
while $A$ does not form a spanning tree do
    find an edge$(u, v)$ that is safe for $A$;
    add $(u, v)$ to $A$;
end
return $A$;
```

Optimal Substructure Property

- Start with an empty graph.
- Try to add edges one at a time, always making sure that what is built remains acyclic.
- If we are sure at each step that the resulting graph is a subset of some minimum spanning tree, we are done.

Lemma

- *Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E*
- *Let A be a subset of E that is included in some minimum spanning tree for G.*

Let

- *$(S, V - S)$ be any cut of G that respects A*
- *(u, v) be a light edge crossing the cut $(S, V - S)$*

Then, edge (u, v) is safe for A.

Introduction to Greedy Algorithm

- A greedy algorithm for an optimization problem always makes the choice that looks best at the moment and adds it to the current subsolution.
- Examples already seen
  - Dijkstra’s shortest path algorithm: Select the node, among all "candidate" nodes, that is closest to the source according to estimation d[u].
  - Prim/Kruskal’s MST algorithms: Select the edge, among all "candidate" edges, that is the lightest.
- Greedy algorithms don’t always yield optimal solutions but,

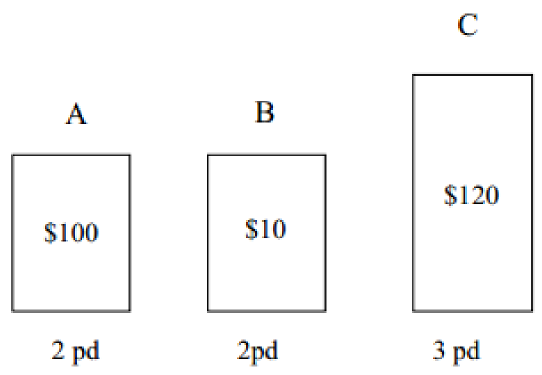
Introduction to Greedy Algorithm

- A greedy algorithm for an optimization problem always makes the choice that looks best at the moment and adds it to the current subsolution.
- Examples already seen
  - Dijkstra’s shortest path algorithm: Select the node, among all "candidate" nodes, that is closest to the source according to estimation d[u].
  - Prim/Kruskal’s MST algorithms: Select the edge, among all "candidate" edges, that is the lightest.
- Greedy algorithms don’t always yield optimal solutions but, when they do, they’re usually the simplest and most efficient algorithms available.

Outline

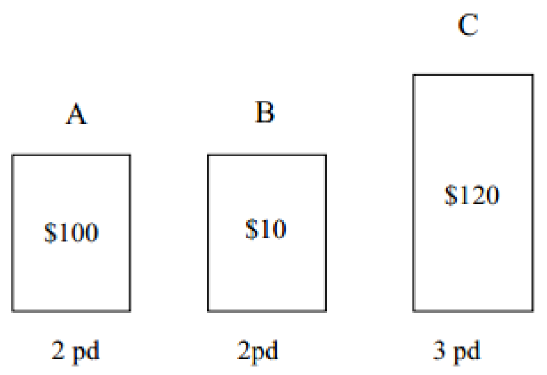
- Introduction to Part V
- The Fraction Knapsack Problem
  - Problem Definition
  - A Greedy Algorithm
  - Correctness
- Interval Scheduling and Interval Partitioning
  - Interval Scheduling
  - Interval Partitioning

The Knapsack Problem...



Capacity of knapsack: $K = 4$

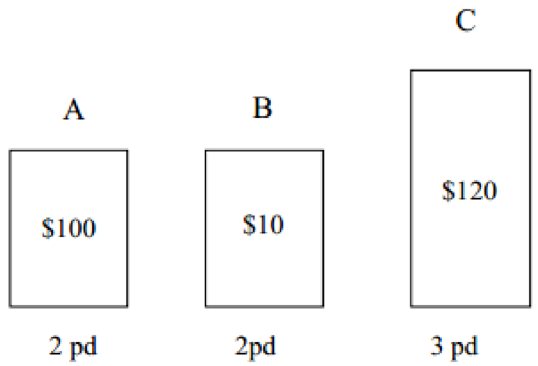
The Knapsack Problem...



Capacity of knapsack: $K = 4$

Fractional Knapsack Problem:

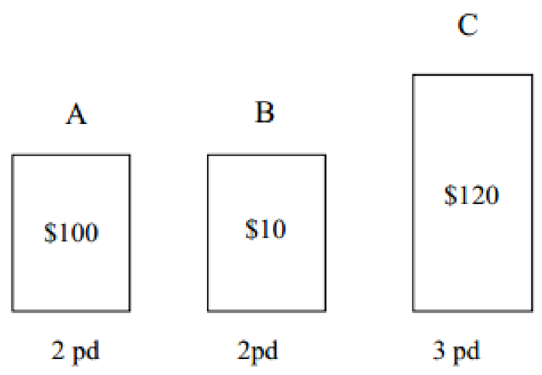
The Knapsack Problem...



Capacity of knapsack: $K = 4$

Fractional Knapsack Problem:
Can take a fraction of an item.

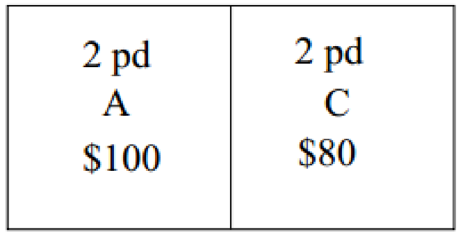
The Knapsack Problem…



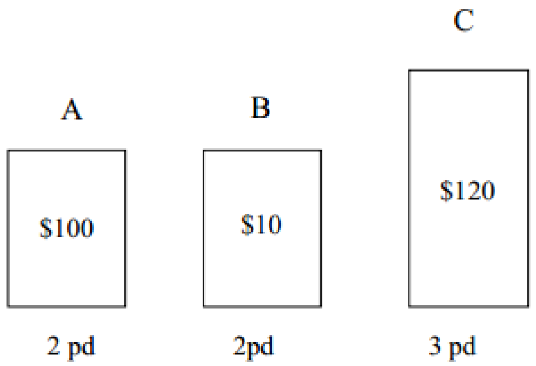
Capacity of knapsack: $K = 4$

Fractional Knapsack Problem:
Can take a fraction of an item.

Solution:



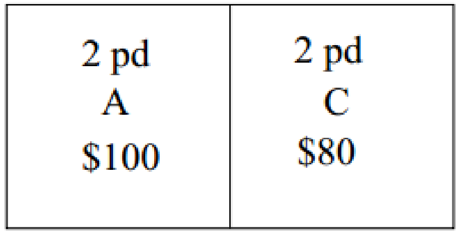
The Knapsack Problem…



Capacity of knapsack: $K = 4$

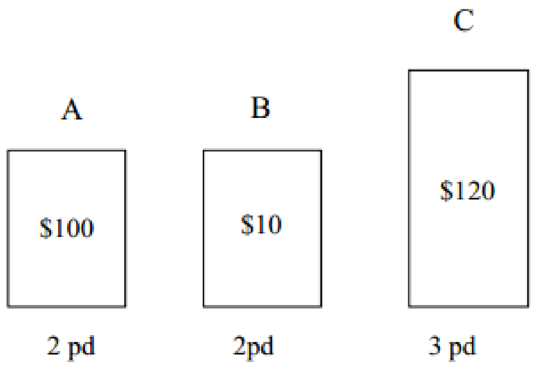
Fractional Knapsack Problem:
Can take a fraction of an item.

Solution:



0-1 Knapsack Problem:
Can only take or leave item. You can't take a fraction.

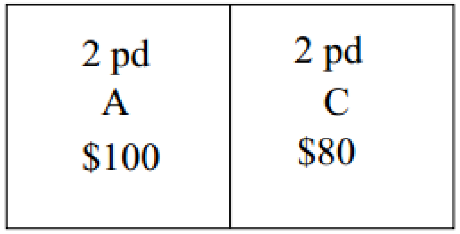
The Knapsack Problem...



Capacity of knapsack: $K = 4$

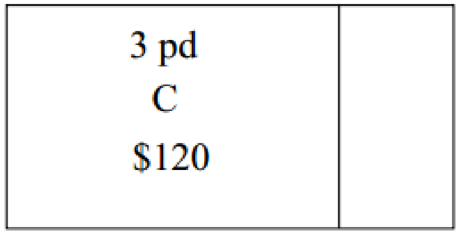
Fractional Knapsack Problem:
Can take a fraction of an item.

Solution:



0-1 Knapsack Problem:
Can only take or leave item. You can't take a fraction.

Solution:



The Fractional Knapsack Problem: Formal Definition

- Given K and a set of n items:

weight	w_1	w_2	$\dots$	w_n
value	v_1	v_2	$\dots$	v_n

- Find: $0 \le x_i \le 1$, i = 1,2,…, n such that

$$\sum_{i=1}^{n} x_i w_i \le K$$

and the following is maximized:

$$\sum_{i=1}^{n} x_i v_i$$

Outline

- Introduction to Part V
- The Fraction Knapsack Problem
  - Problem Definition
  - A Greedy Algorithm
  - Correctness
- Interval Scheduling and Interval Partitioning
  - Interval Scheduling
  - Interval Partitioning

Greedy Solution for Fractional Knapsack

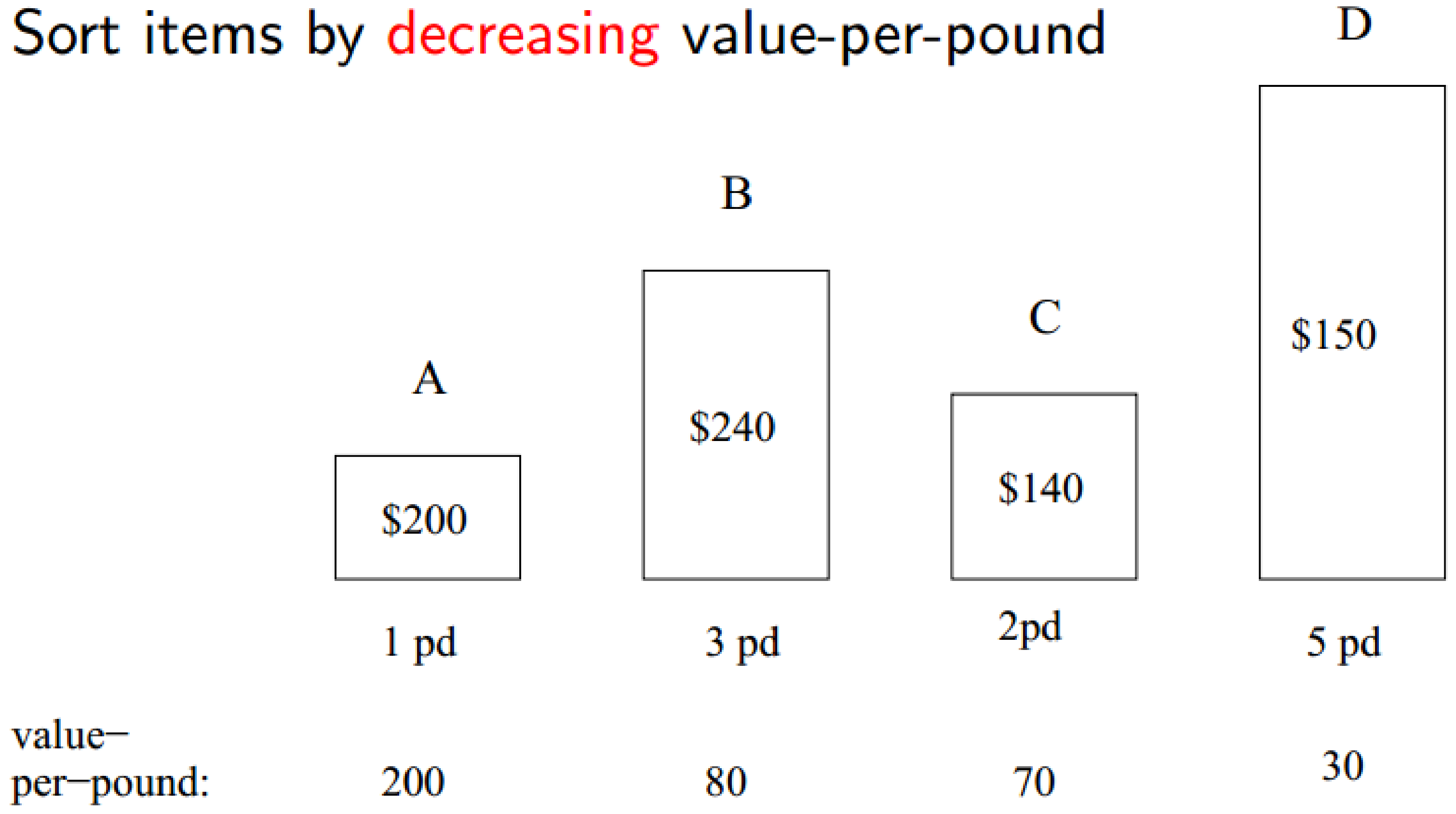
Sort items by value-per-pound

Greedy Solution for Fractional Knapsack

Sort items by decreasing value-per-pound

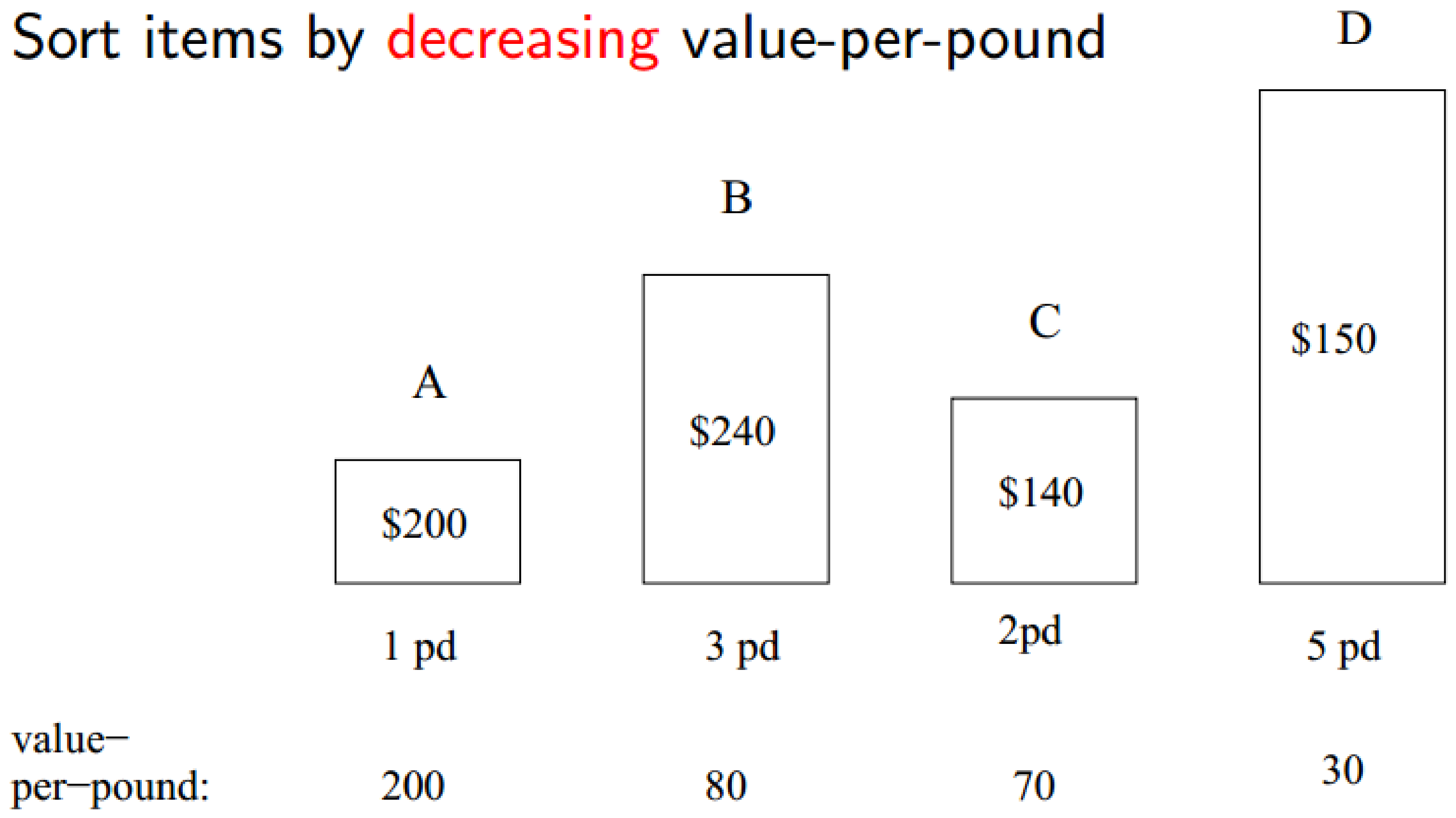
Greedy Solution for Fractional Knapsack

Sort items by decreasing value-per-pound



Greedy Solution for Fractional Knapsack

Sort items by decreasing value-per-pound



If knapsack holds $K = 5$ pd, solution is:

1	pd	A
3	pd	B
1	pd	C

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.
- Sort the items by ρ_i.

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.
- Sort the items by decreasing ρ_i.

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.
- Sort the items by decreasing ρ_i. Let the sorted item sequence be 1, 2, …, i, …, n,

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.
- Sort the items by decreasing ρ_i. Let the sorted item sequence be 1, 2, …, i, …, n, and the corresponding value-per-pound and weight be ρ_i and w_i respectively.

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.
- Sort the items by decreasing ρ_i. Let the sorted item sequence be 1, 2, …, i, …, n, and the corresponding value-per-pound and weight be ρ_i and w_i respectively.
- Let k be the current weight limit (Initially,

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.
- Sort the items by decreasing ρ_i. Let the sorted item sequence be 1, 2, …, i, …, n, and the corresponding value-per-pound and weight be ρ_i and w_i respectively.
- Let k be the current weight limit (Initially, k = K).

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.
- Sort the items by decreasing ρ_i. Let the sorted item sequence be 1, 2, …, i, …, n, and the corresponding value-per-pound and weight be ρ_i and w_i respectively.
- Let k be the current weight limit (Initially, k = K). In each iteration, we choose item i from the of the unselected list.

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.
- Sort the items by decreasing ρ_i. Let the sorted item sequence be 1, 2, …, i, …, n, and the corresponding value-per-pound and weight be ρ_i and w_i respectively.
- Let k be the current weight limit (Initially, k = K). In each iteration, we choose item i from the head of the unselected list.

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.
- Sort the items by decreasing ρ_i. Let the sorted item sequence be 1, 2, …, i, …, n, and the corresponding value-per-pound and weight be ρ_i and w_i respectively.
- Let k be the current weight limit (Initially, k = K). In each iteration, we choose item i from the head of the unselected list.
    - If $k \geq w_i$,

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.
- Sort the items by decreasing ρ_i. Let the sorted item sequence be 1, 2, …, i, …, n, and the corresponding value-per-pound and weight be ρ_i and w_i respectively.
- Let k be the current weight limit (Initially, k = K). In each iteration, we choose item i from the head of the unselected list.
  - If $k \geq w_i$, set $x_i = 1$ (we take item i), and reduce k =

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.
- Sort the items by decreasing ρ_i. Let the sorted item sequence be 1, 2, …, i, …, n, and the corresponding value-per-pound and weight be ρ_i and w_i respectively.
- Let k be the current weight limit (Initially, k = K). In each iteration, we choose item i from the head of the unselected list.
  - If $k \geq w_i$, set $x_i = 1$ (we take item i), and reduce $k = k - w_i$, then consider the next unselected item.

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.
- Sort the items by decreasing ρ_i. Let the sorted item sequence be 1, 2, …, i, …, n, and the corresponding value-per-pound and weight be ρ_i and w_i respectively.
- Let k be the current weight limit (Initially, k = K). In each iteration, we choose item i from the head of the unselected list.
    - If $k \geq w_i$, set $x_i = 1$ (we take item i), and reduce $k = k - w_i$, then consider the next unselected item.
    - If $k < w_i$,

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.
- Sort the items by decreasing ρ_i. Let the sorted item sequence be 1, 2, …, i, …, n, and the corresponding value-per-pound and weight be ρ_i and w_i respectively.
- Let k be the current weight limit (Initially, k = K). In each iteration, we choose item i from the head of the unselected list.
  - If $k \geq w_i$, set $x_i = 1$ (we take item i), and reduce $k = k - w_i$, then consider the next unselected item.
  - If $k < w_i$, set $x_i = k/w_i$ (we take a fraction k/w_i of item i),

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.
- Sort the items by decreasing ρ_i. Let the sorted item sequence be 1, 2, …, i, …, n, and the corresponding value-per-pound and weight be ρ_i and w_i respectively.
- Let k be the current weight limit (Initially, k = K). In each iteration, we choose item i from the head of the unselected list.
  - If $k \geq w_i$, set $x_i = 1$ (we take item i), and reduce $k = k - w_i$, then consider the next unselected item.
  - If $k < w_i$, set $x_i = k/w_i$ (we take a fraction k/w_i of item i), Then the algorithm terminates.

Greedy Solution for Fractional Knapsack

- Calculate the value-per-pound $\rho_i = \frac{v_i}{w_i}$ for i = 1, 2, …, n.
- Sort the items by decreasing ρ_i. Let the sorted item sequence be 1, 2, …, i, …, n, and the corresponding value-per-pound and weight be ρ_i and w_i respectively.
- Let k be the current weight limit (Initially, k = K). In each iteration, we choose item i from the head of the unselected list.
    - If $k \geq w_i$, set $x_i = 1$ (we take item i), and reduce $k = k - w_i$, then consider the next unselected item.
    - If $k < w_i$, set $x_i = k/w_i$ (we take a fraction k/w_i of item i), Then the algorithm terminates.

Running time: O(n log n).

Pseudocode

Fraction-Knapsack(n,v,w,K)

```
Input: Value array $\boldsymbol{v}$ and weight array $\boldsymbol{w}$ of $\boldsymbol{n}$ items, capacity of
       knapsack $\boldsymbol{K}$.
Output: Solution of maximum value.
Let r[1..n],x[1..n] be two new arrays;
for $i \leftarrow 1$ to $n$ do
    $r[i] \leftarrow v[i]/w[i]$;
    $x[i] \leftarrow 0$;
end
Sort the items in decreasing order of their ratios $r$, rename the items if
  necessary so that the sorted order of items is $\langle 1, 2, ..., n \rangle$;
$j \leftarrow 0$;
while $K > 0$ and $j \leq n$ do
    $j \leftarrow j + 1$;
    if $K > w[j]$ then
        $x[j] \leftarrow 1$;
        $K \leftarrow K - w[j]$;
    end
    else
        $x[j] \leftarrow k/w[j]$;
        break;
    end
end
return $x$;
```

Example of Optimal Solution Construction

	1	**2**	**3**	**4**
$v_{\boldsymbol{i}}$	60	75	100	120
$\boldsymbol{w_i}$	10	25	20	30

K = 50

Example of Optimal Solution Construction

	1	2	3	4
$\mathrm{v}_{\boldsymbol{i}}$	60	75	100	120
$\boldsymbol{w_i}$	10	25	20	30
$\boldsymbol{r_i}$	6	3	5	4

K = 50

Example of Optimal Solution Construction

	1	**3**	**4**	**2**
v_i	60	100	120	75
w_i	10	20	30	25
r_i	6	5	4	3

K = 50

Example of Optimal Solution Construction

	1	**3**	**4**	**2**
v_i	60	100	120	75
w_i	10	20	30	25
r_i	6			3
x_i	0	0	0	0

K = 50

$w_i < K$

Example of Optimal Solution Construction

	1	**3**	**4**	**2**
v_i	60	100	120	75
w_i	10	20	30	25
r_i	6	5	4	3
x_i	1	0	0	0

K = 40

Example of Optimal Solution Construction

	1	**3**	**4**	**2**
v_i	60	100	120	75
w_i	10	20	30	25
r_i	6	5	4	3
x_i	1	1	0	0

K = 20

Example of Optimal Solution Construction

	1	**3**	**4**	**2**
v_i	60	100	120	75
w_i	10	20	30	25
r_i	6	5	4	
x_i	1	1	0	0

K = 20

$w_i \geq K$

Example of Optimal Solution Construction

	1	**3**	**4**	**2**
v_i	60	100	120	75
w_i	10	20	30	25
r_i	6	5	4	3
x_i	1	1	2/3	0

K = 20

Example of Optimal Solution Construction

	1	**3**	**4**	**2**
v_i	60	100	120	75
w_i	10	20	30	25
r_i	6	5	4	3
x_i	1	1	2/3	0

Result

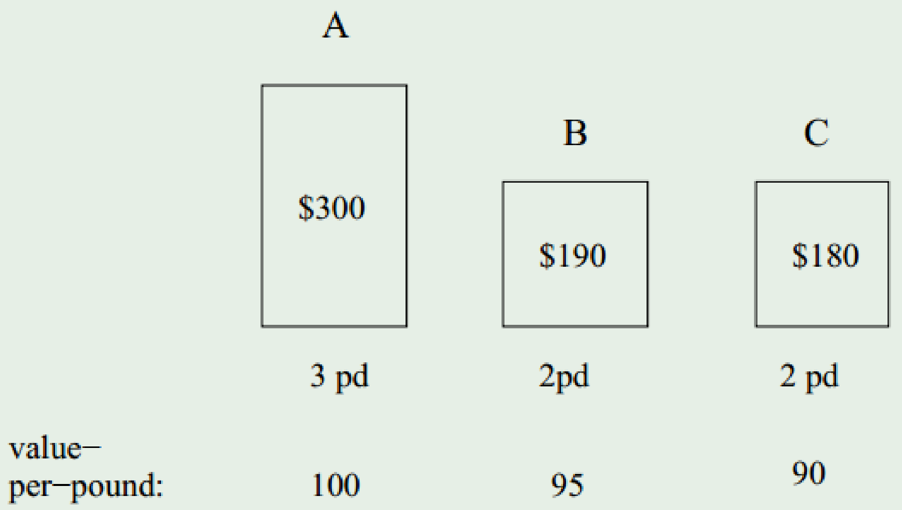
Outline

- Introduction to Part V
- The Fraction Knapsack Problem
    - Problem Definition
    - A Greedy Algorithm
    - Correctness
- Interval Scheduling and Interval Partitioning
    - Interval Scheduling
    - Interval Partitioning

Greedy solution for 0-1 Knapsack Problem?

The 0-1 Knapsack Problem does not have a greedy solution!

Example

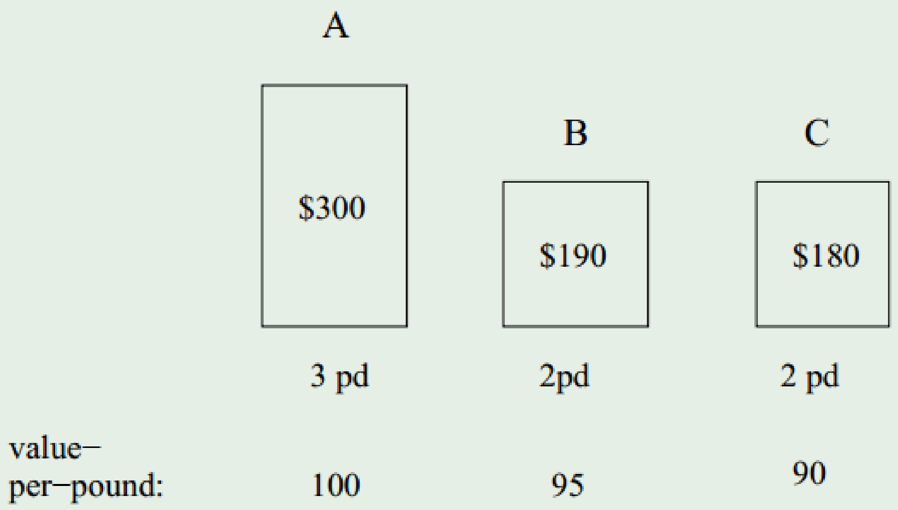


$K = 4$. Solution is

Greedy solution for 0-1 Knapsack Problem?

The 0-1 Knapsack Problem does not have a greedy solution!

Example

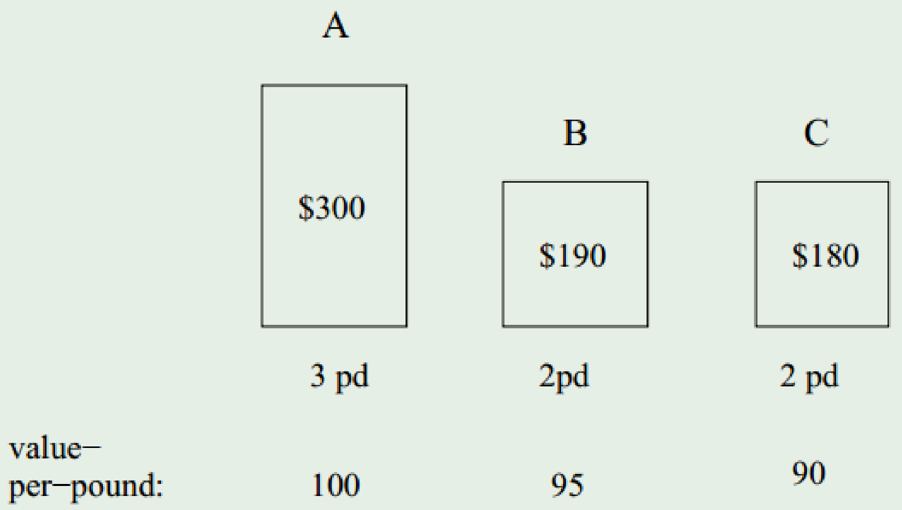


$K = 4$. Solution is item B + item C

Greedy solution for 0-1 Knapsack Problem?

The 0-1 Knapsack Problem does not have a greedy solution!

Example



$K = 4$. Solution is item B + item C

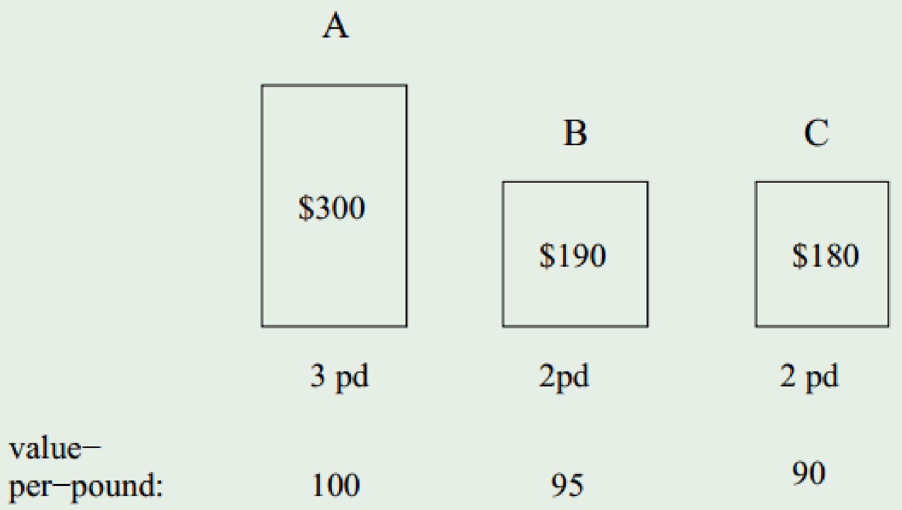
Question

Suppose we try to prove the greedy algorithm for 0-1 knapsack problem is correct.

Greedy solution for 0-1 Knapsack Problem?

The 0-1 Knapsack Problem does not have a greedy solution!

Example



$K = 4$. Solution is item B + item C

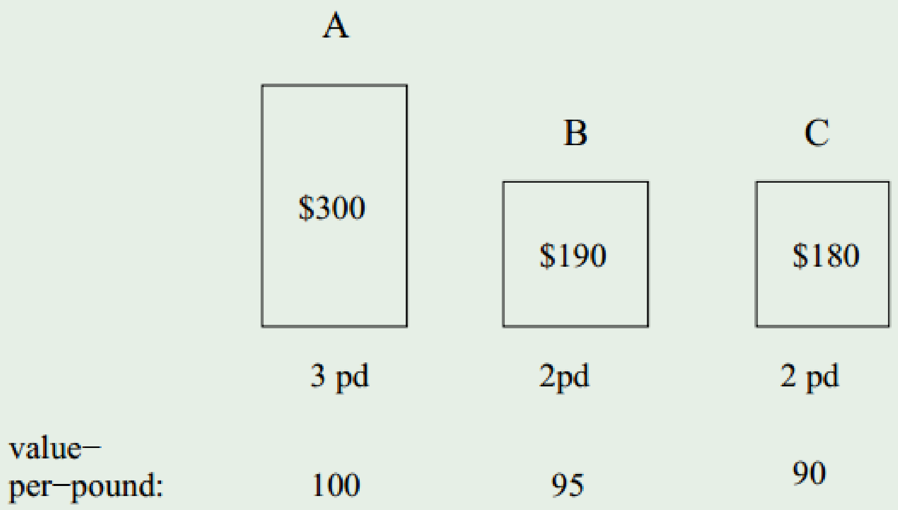
Question

Suppose we try to prove the greedy algorithm for 0-1 knapsack problem is correct. We follow exactly the same lines of arguments as fractional knapsack problem.

Greedy solution for 0-1 Knapsack Problem?

The 0-1 Knapsack Problem does not have a greedy solution!

Example



$K = 4$. Solution is item B + item C

Question

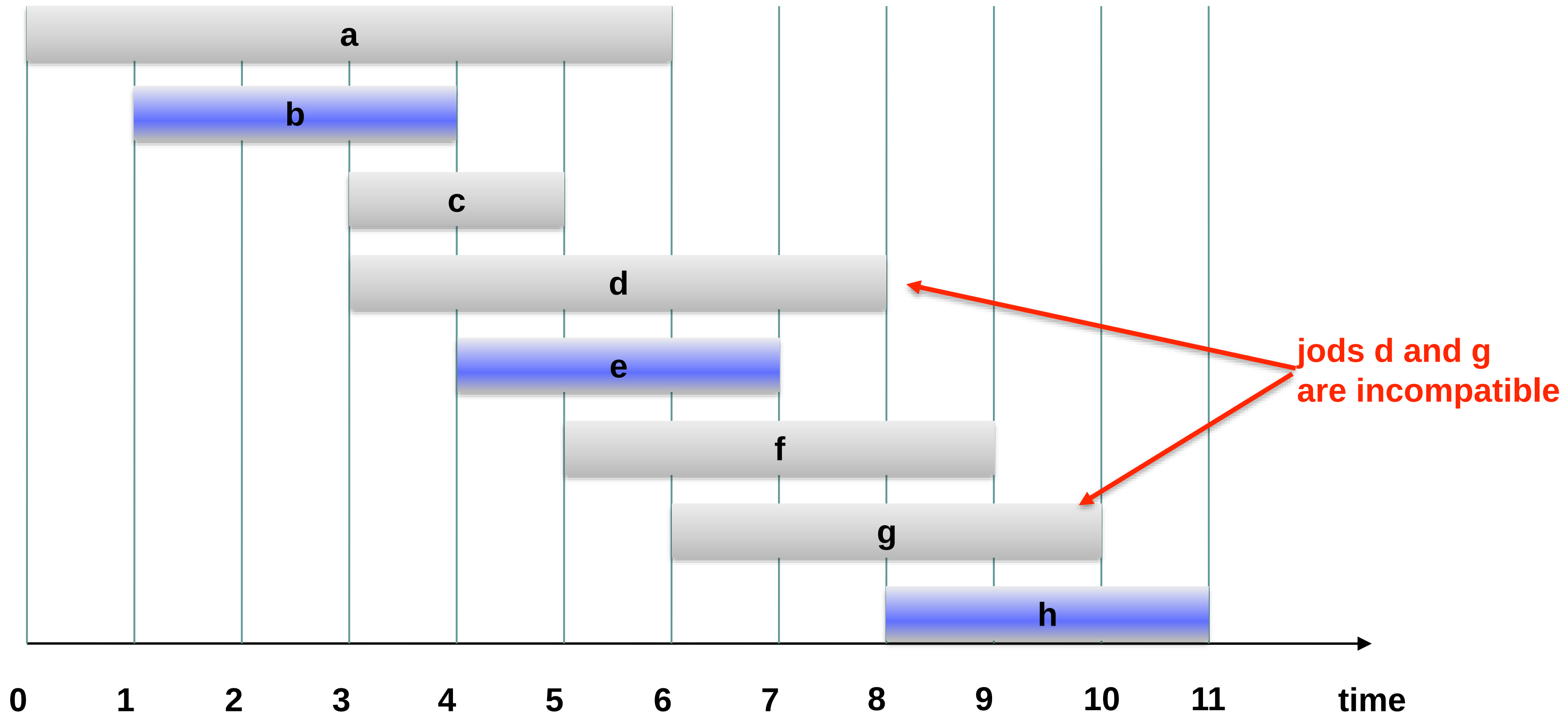
Suppose we try to prove the greedy algorithm for 0-1 knapsack problem is correct. We follow exactly the same lines of arguments as fractional knapsack problem. Of course, it must fail. Where is the problem in the proof?

Outline

- Introduction to Part V
- The Fraction Knapsack Problem
  - Problem Definition
  - A Greedy Algorithm
  - Correctness
- Interval Scheduling and Interval Partitioning
  - Interval Scheduling
  - Interval Partitioning

Interval Scheduling

- Job *j* starts at s_j and finishes at f_j .
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.



Interval Scheduling: greedy algorithms

Greedy template. Consider jobs in some natural order.

Take each job provided it’s compatible with the ones already taken.

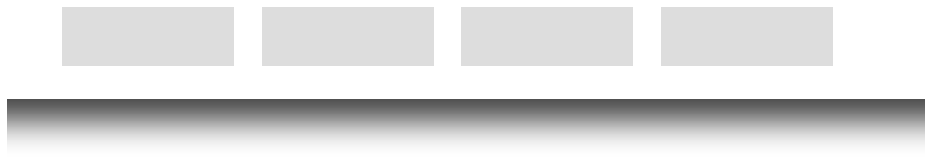
- [Earliest start time] Consider jobs in ascending order of $s_{j.}$
- [Earliest finish time] Consider jobs in ascending order of $f_{j.}$
- [Shortest interval] Consider jobs in ascending order of $f_j - s_{j.}$
- [Fewest conflicts] For each job *j,* count the number of conflicting jobs c_j. Schedule in ascending order of $c_{j.}$

Interval Scheduling: greedy algorithms

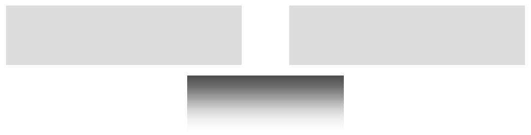
Greedy template. Consider jobs in some natural order.

Take each job provided it's compatible with the ones already taken.

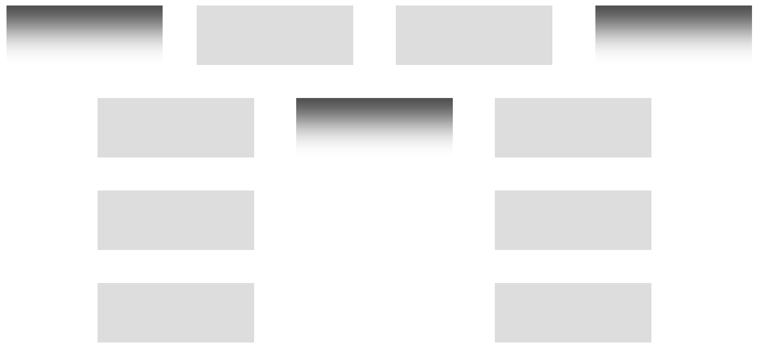
counterexample for earliest start time



counterexample for shortest interval



counterexample for fewest conflicts



Interval Scheduling: earliest-finish-time-first algorithm

$\text{Earliest-Finish-Time-First}(n, s_1, s_2, ..., s_n, f_1, f_2, ..., f_n)$

```
**Input:** $n$ jobs with start time $s_i$ and finish time $f_i$.
**Output:** Schedule with maximum compatible jobs..
Sort jobs by finish time so that $f_1 \leq f_2 \leq ... \leq f_n$.
$A \leftarrow \emptyset$; **for** $j \leftarrow 1$ *to* $n$ **do**
    **if** *job* $j$ *is compatible with* $A$ **then**
        $A \leftarrow A \cup \{j\}$;
    **end**
**end**
**return** $A$;
```

Proposition. Can implement earliest-finish-time first in O(*nlogn*) time.

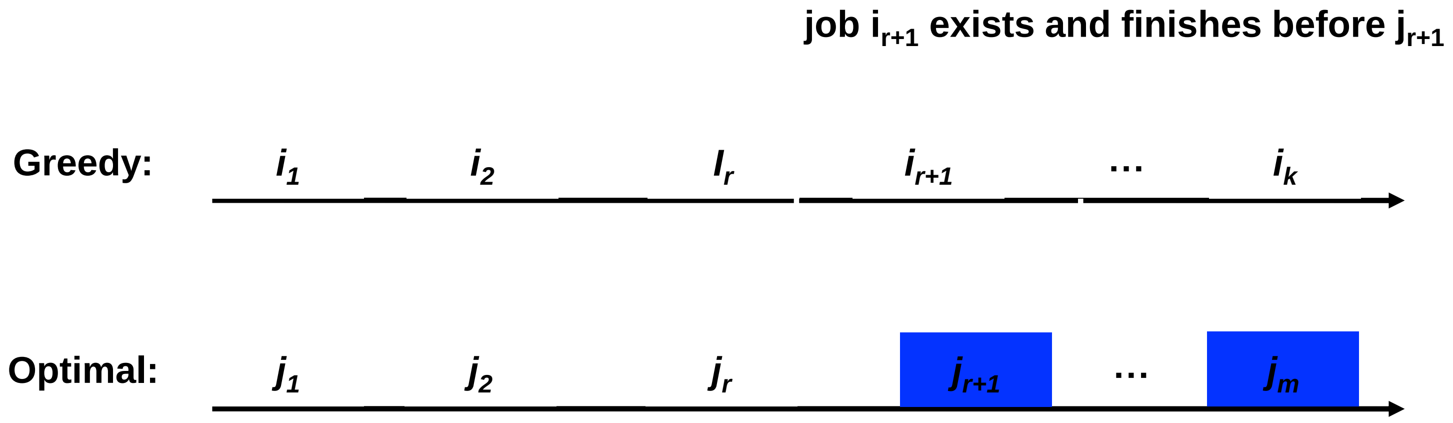
- Keep track of job *j** that was added last to A.
- Job *j* is compatible with A iff $s_j \geq f_{j*}$
- Sorting by finish time takes O(*nlogn*) time.

Interval Scheduling: earliest-finish-time-first algorithm

Theorem. The earliest-finish-time algorithm is optimal.

Pf.[by contradiction]

- Assume greedy is not optimal, and let's see what happens.
- Let's i_1, i_2, … i_k denote set of jobs selected by greedy.
- Let j_1, j_2, … j_m denote set of in an optimal solution with $i_1 = j_1$, $i_2 = j_2$, … $i_r = j_r$ for the largest possible value for *r.*



job j_{r+1} exists because m > k

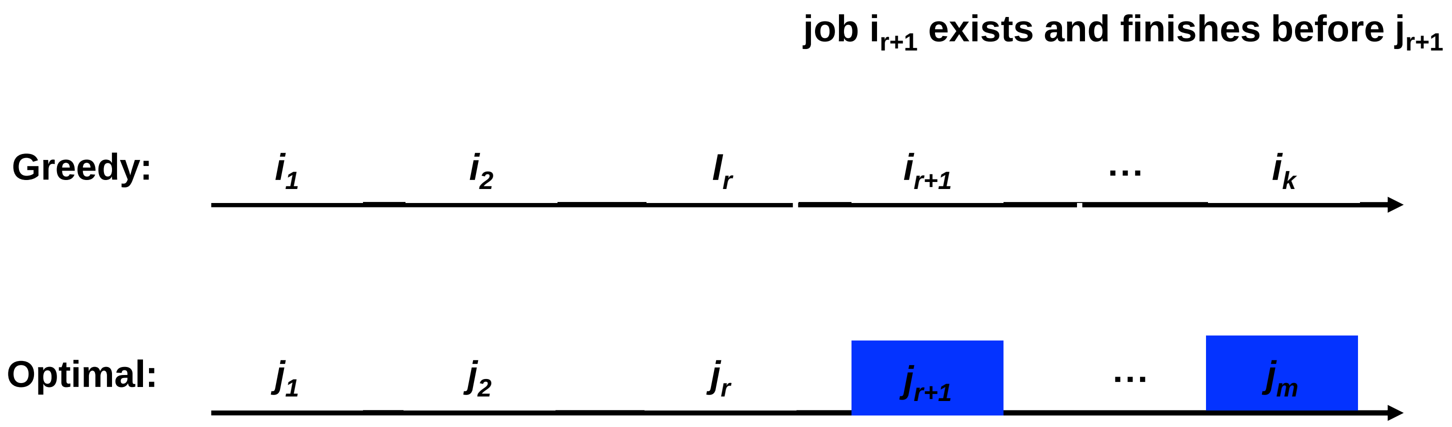
why not replace job j_{r+1} with job i_{r+1}?

Interval Scheduling: earliest-finish-time-first algorithm

Theorem. The earliest-finish-time algorithm is optimal.

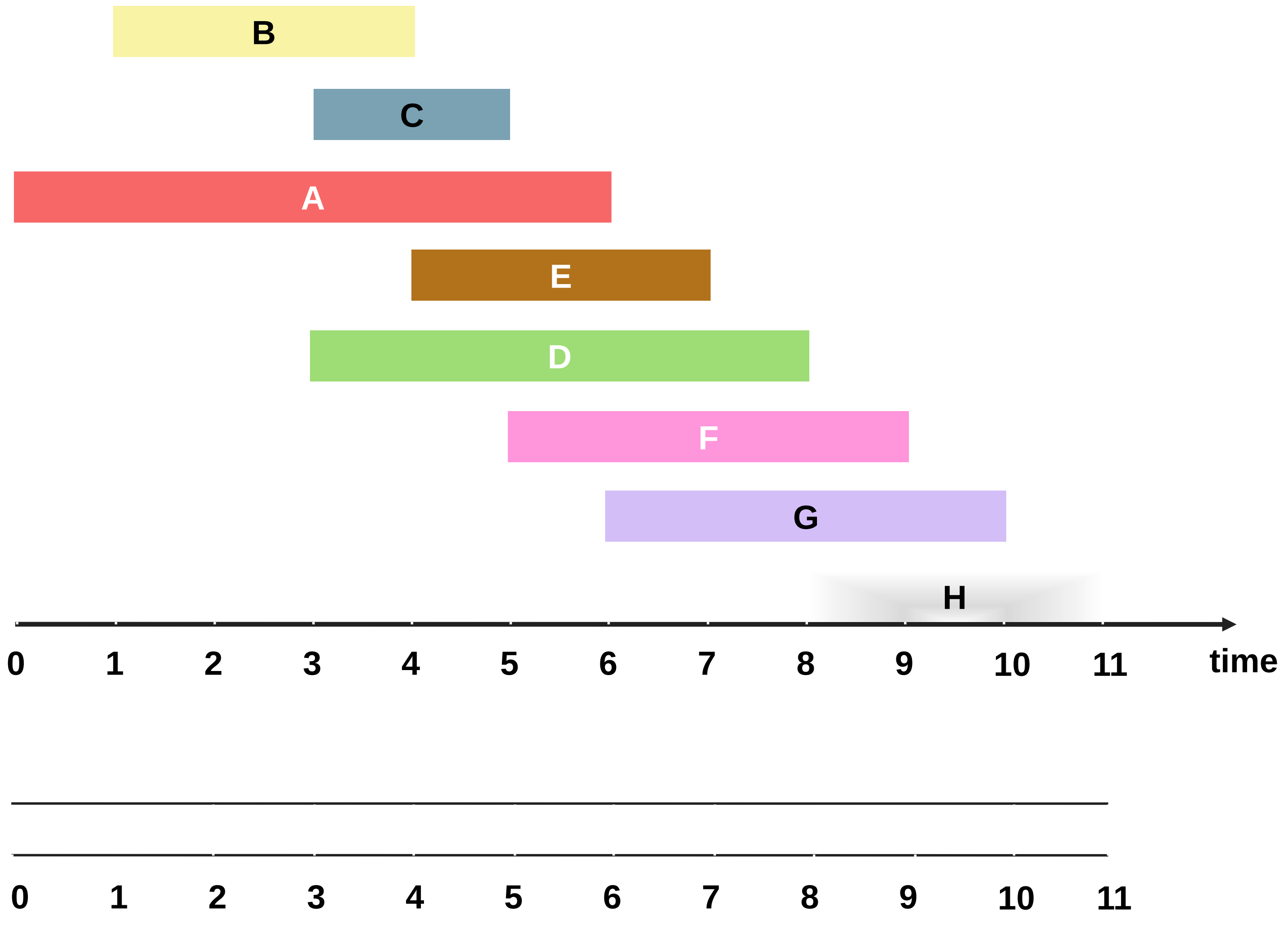
Pf.[by contradiction]

- ➓Assume greedy is not optimal, and let's see what happens.
- ➓Let's $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- ➓Let $j_1, j_2, \ldots j_m$ denote set of in an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots i_r = j_r$ for the largest possible value for r.

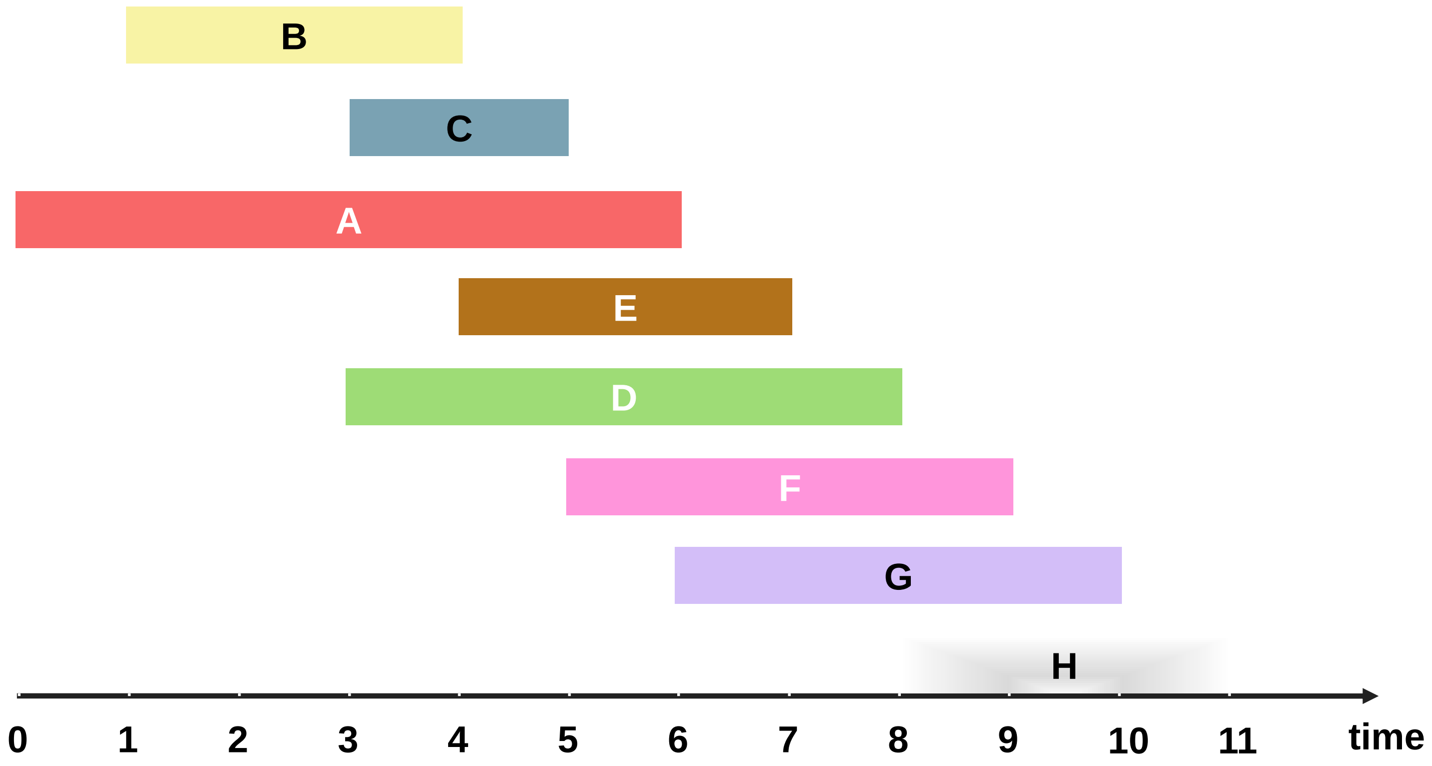


solution still feasible and optimal (but contradicts maximality of r)

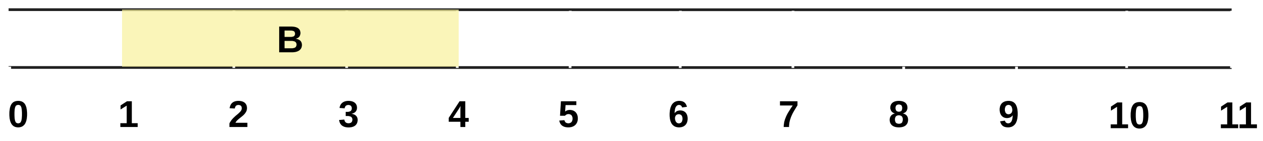
Earliest-finish-time-first algorithm demo



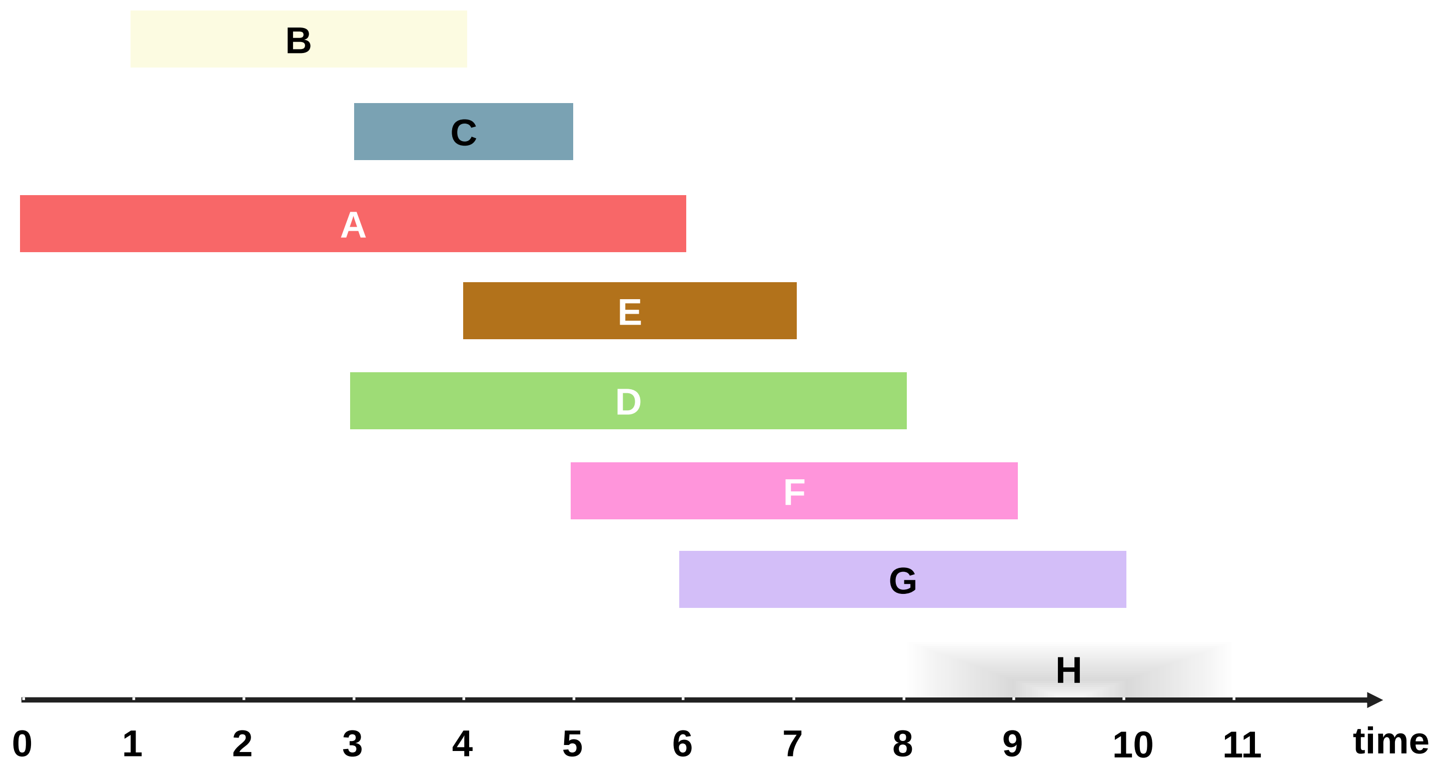
Earliest-finish-time-first algorithm demo

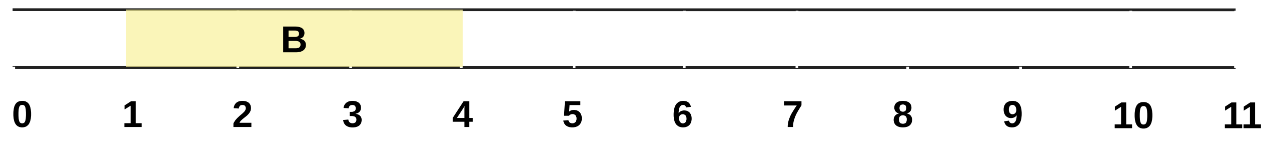


job B is compatible(add to schedule)

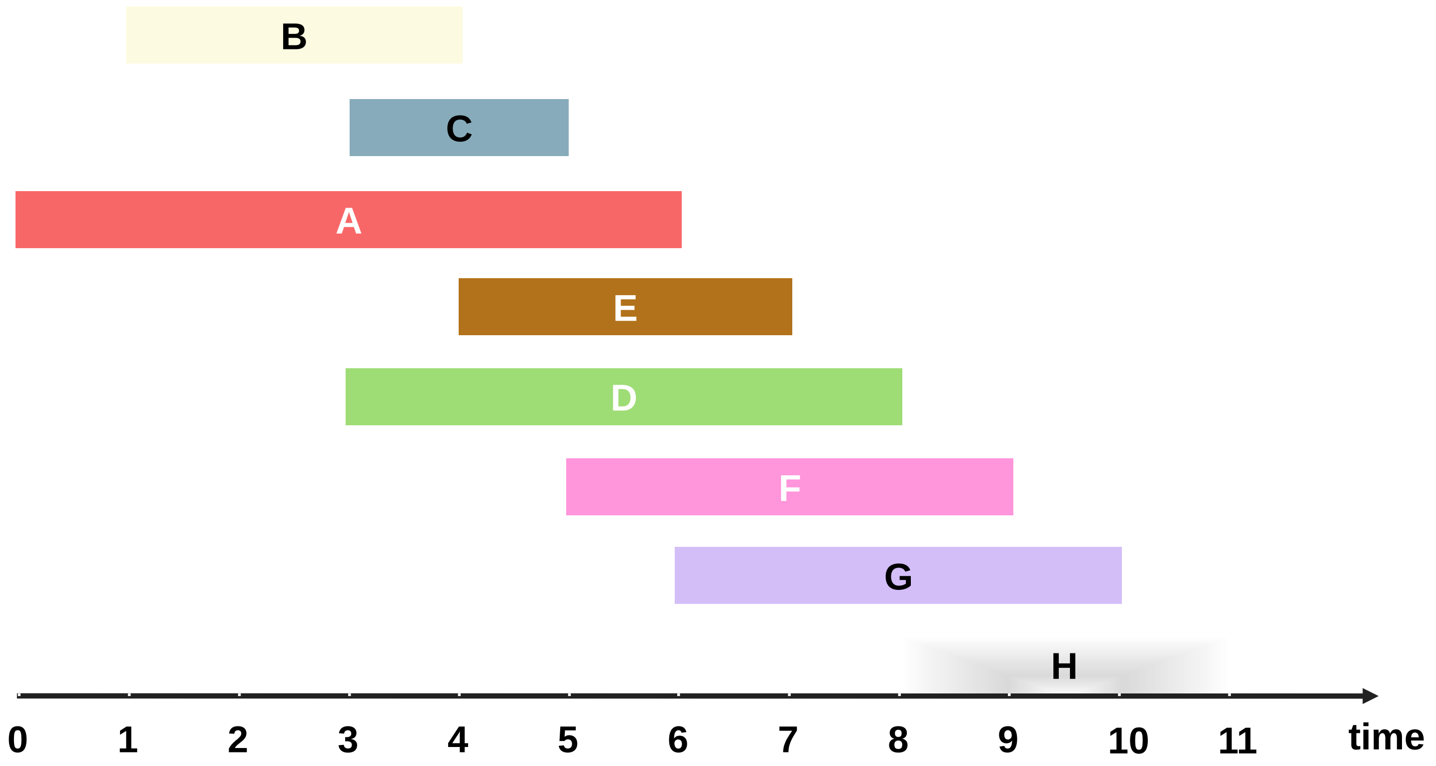


Earliest-finish-time-first algorithm demo

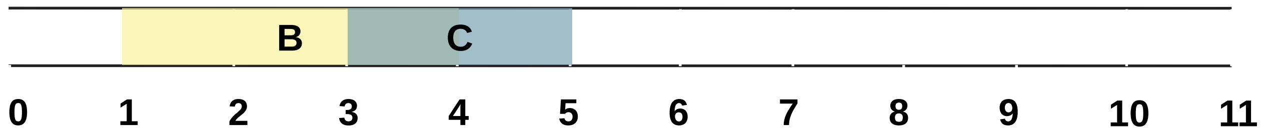




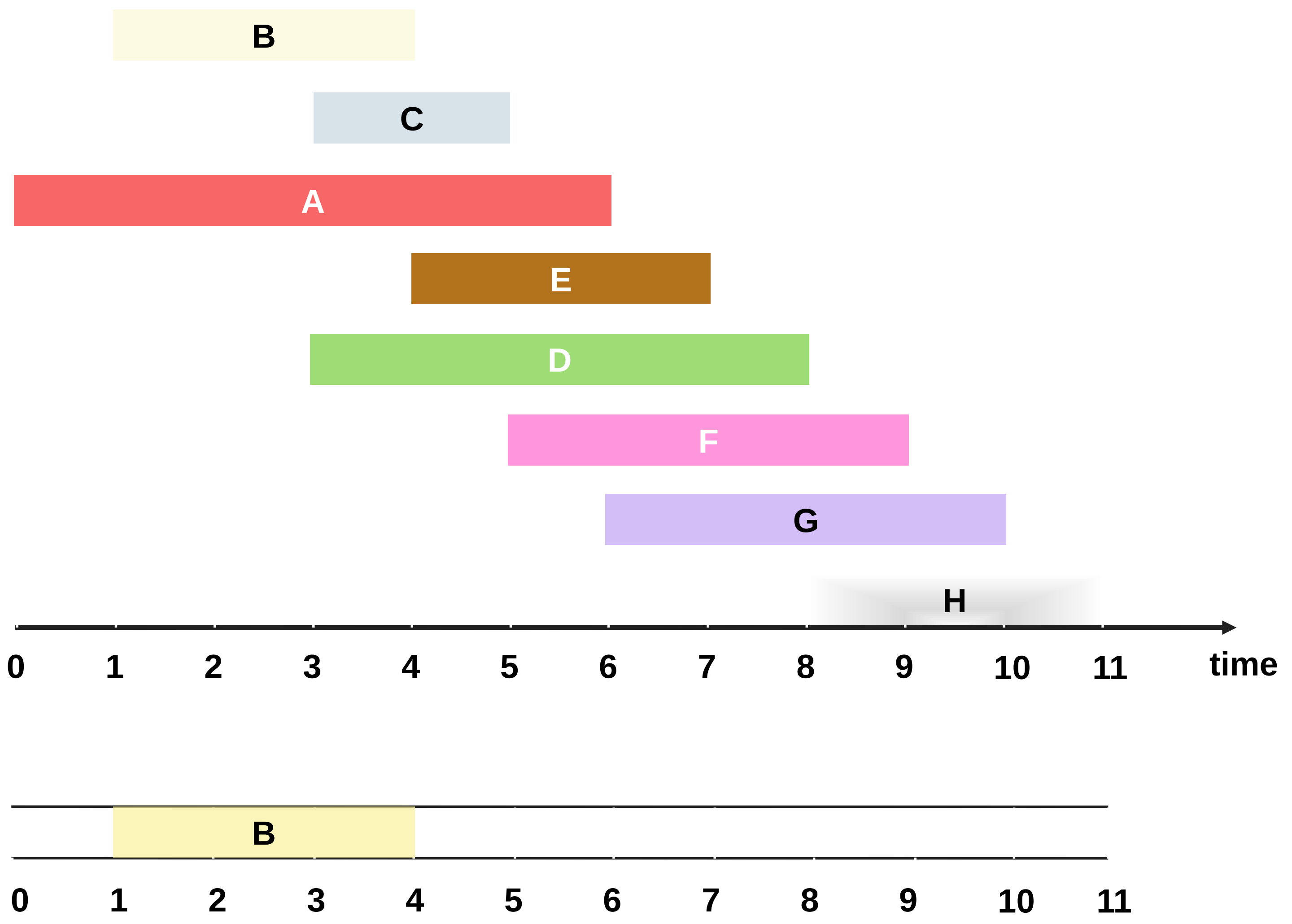
Earliest-finish-time-first algorithm demo



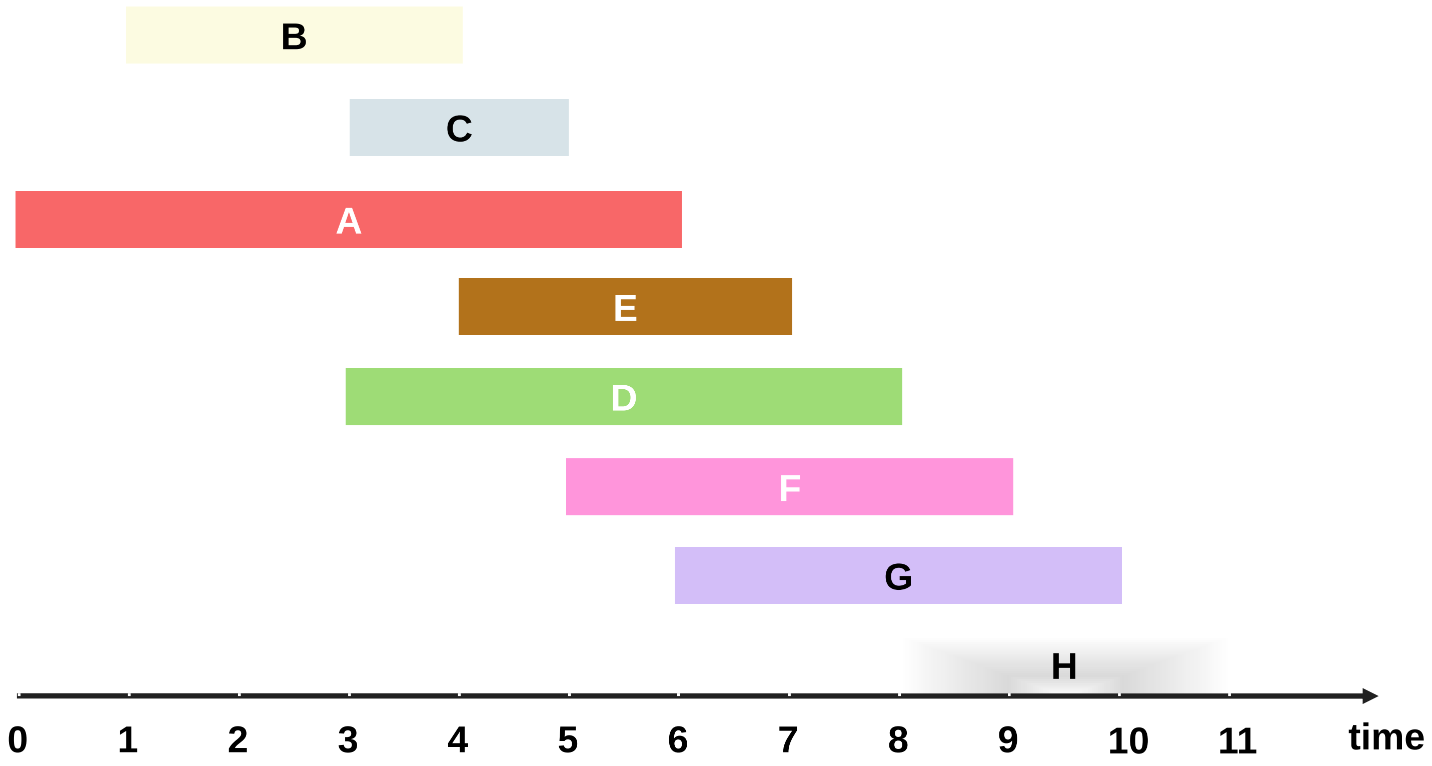
job C is incompatible(do not add to schedule)



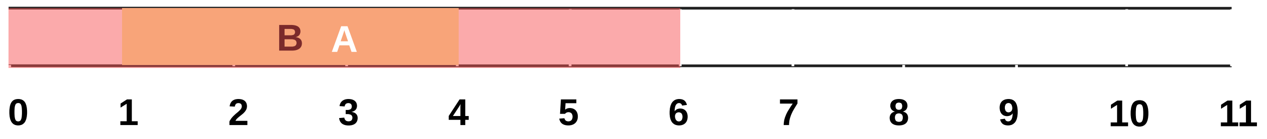
Earliest-finish-time-first algorithm demo



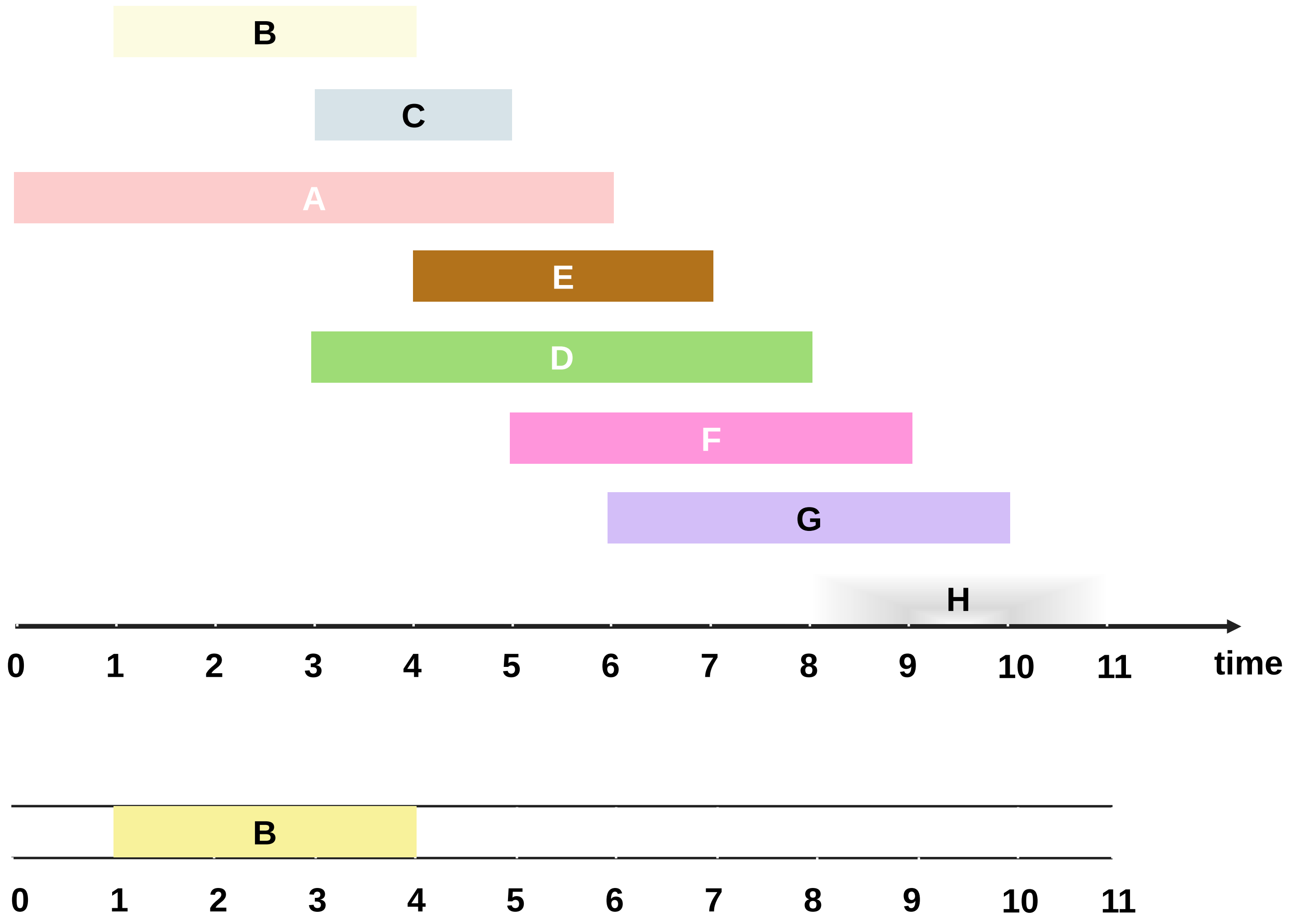
Earliest-finish-time-first algorithm demo



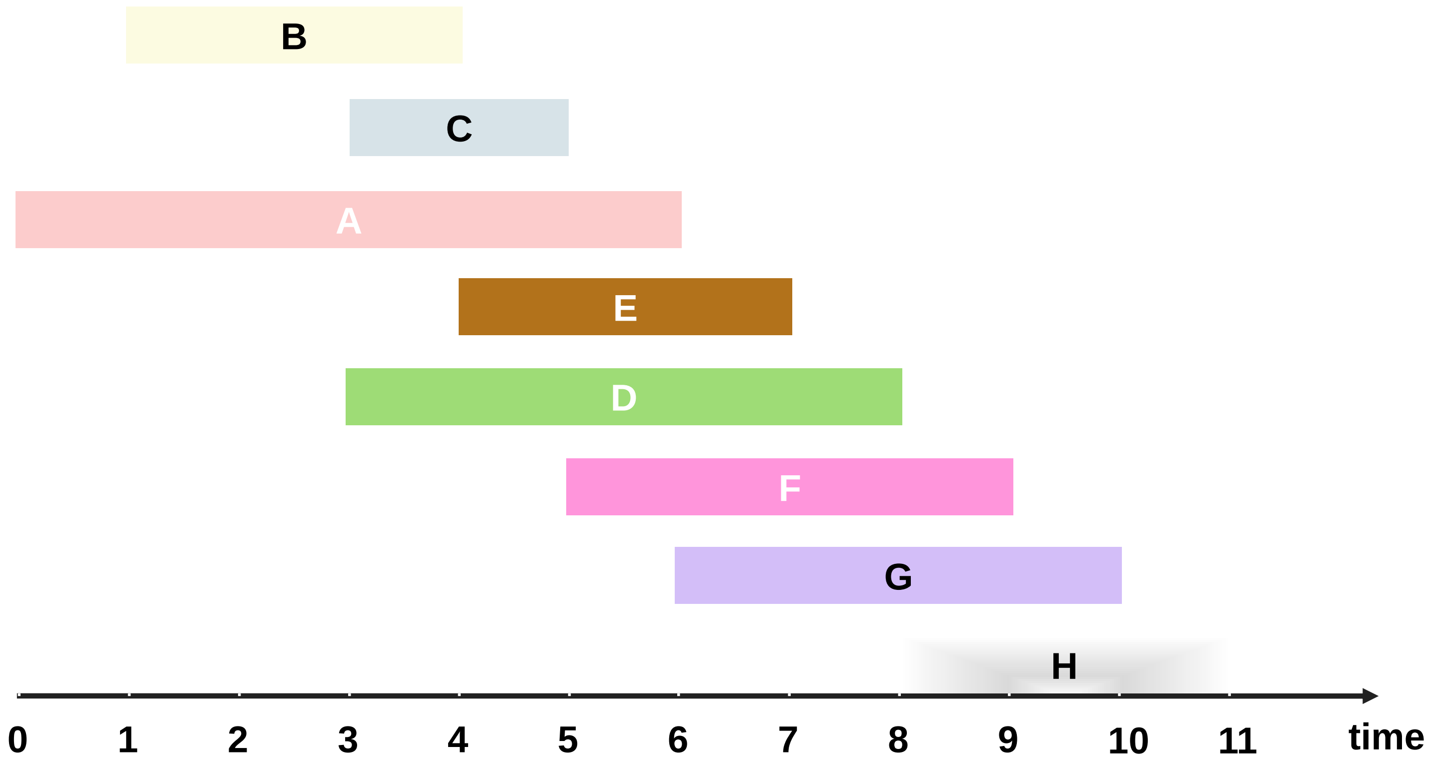
job A is incompatible(do not add to schedule)



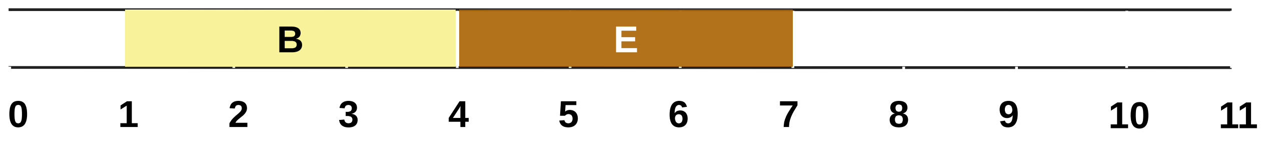
Earliest-finish-time-first algorithm demo



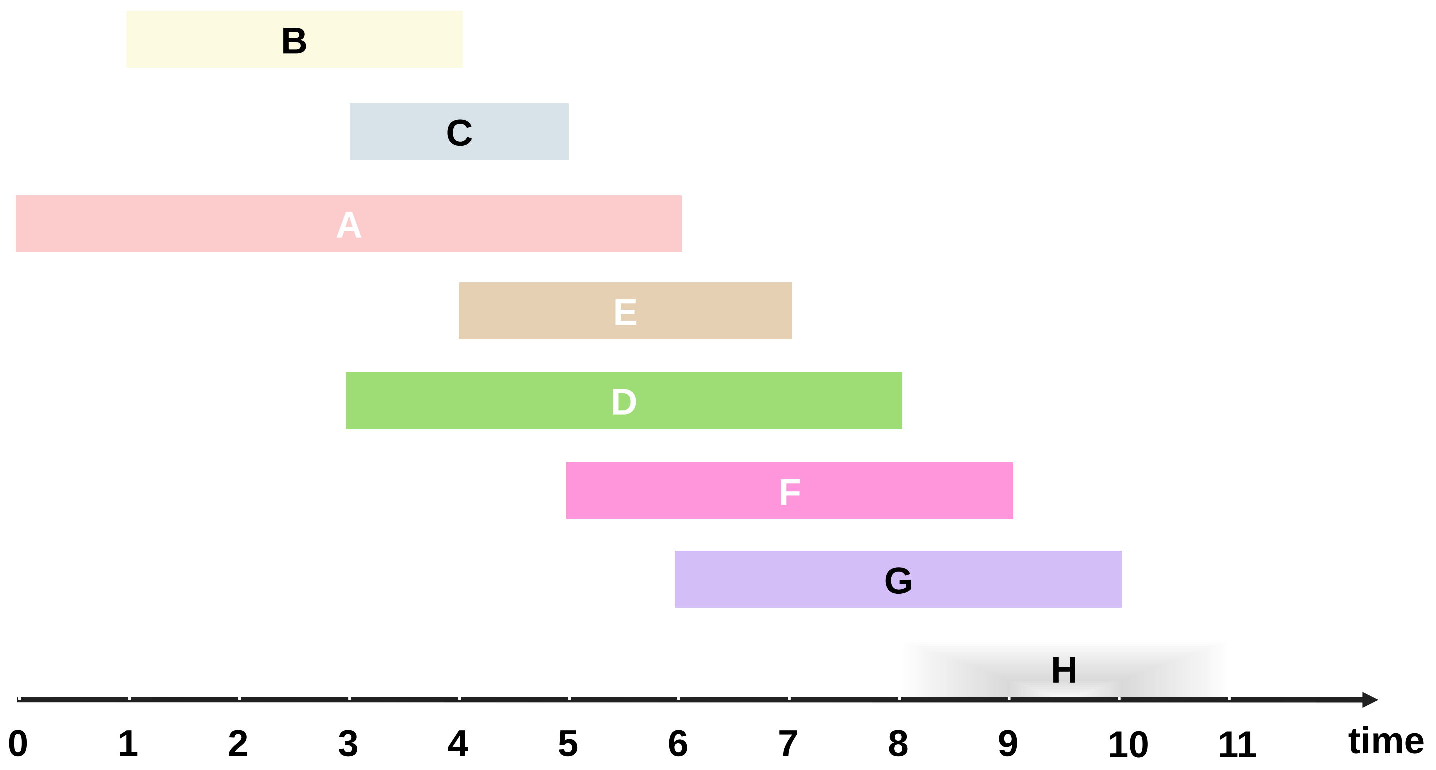
Earliest-finish-time-first algorithm demo

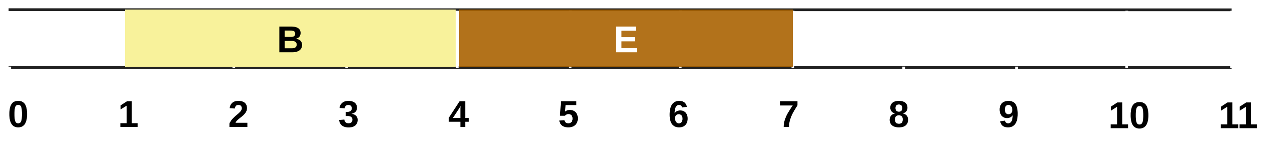


job E is compatible(add to schedule)

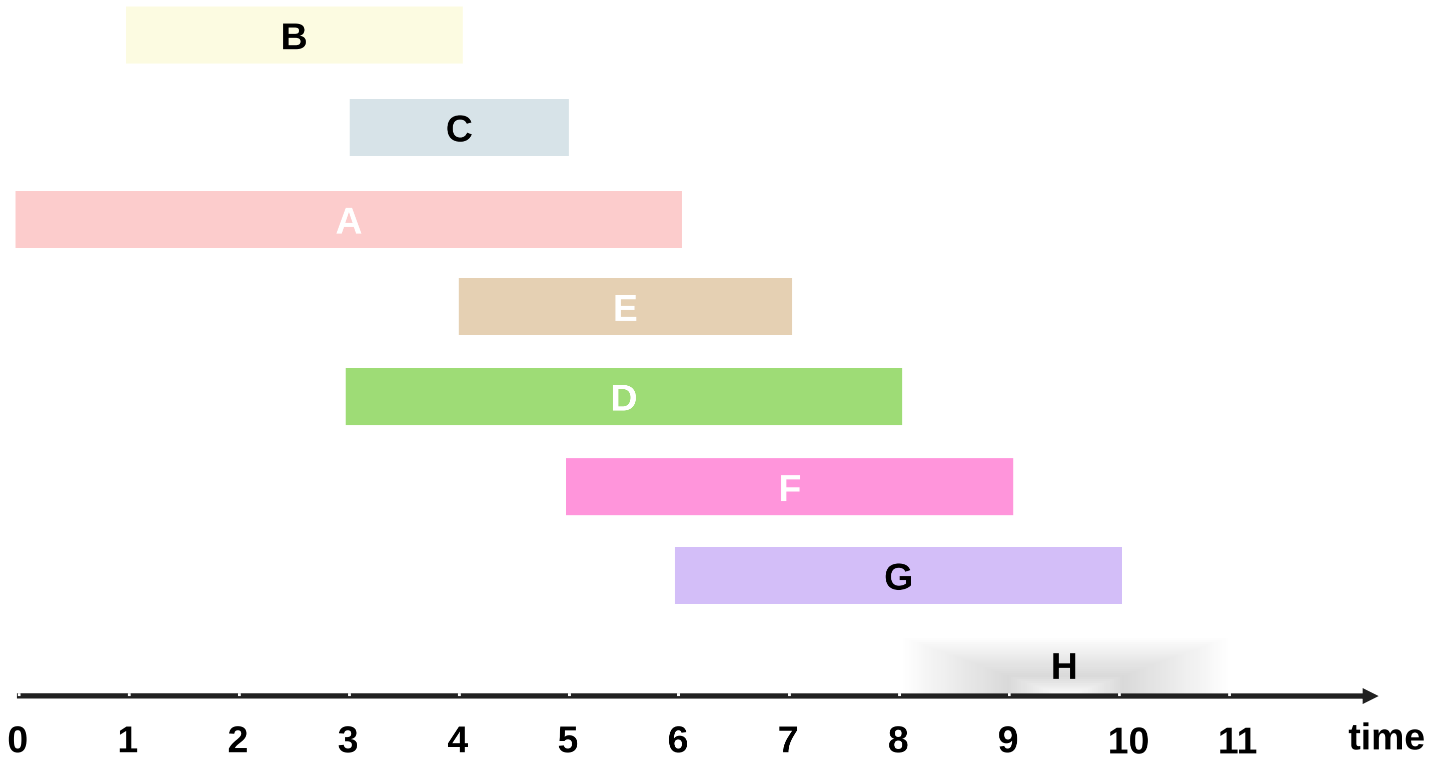


Earliest-finish-time-first algorithm demo

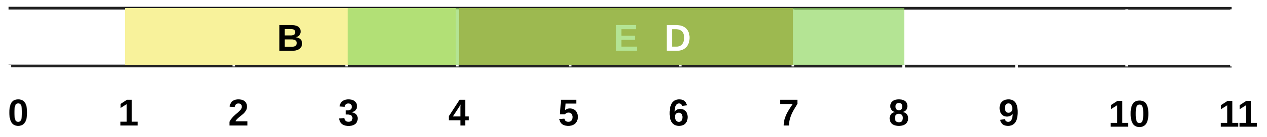




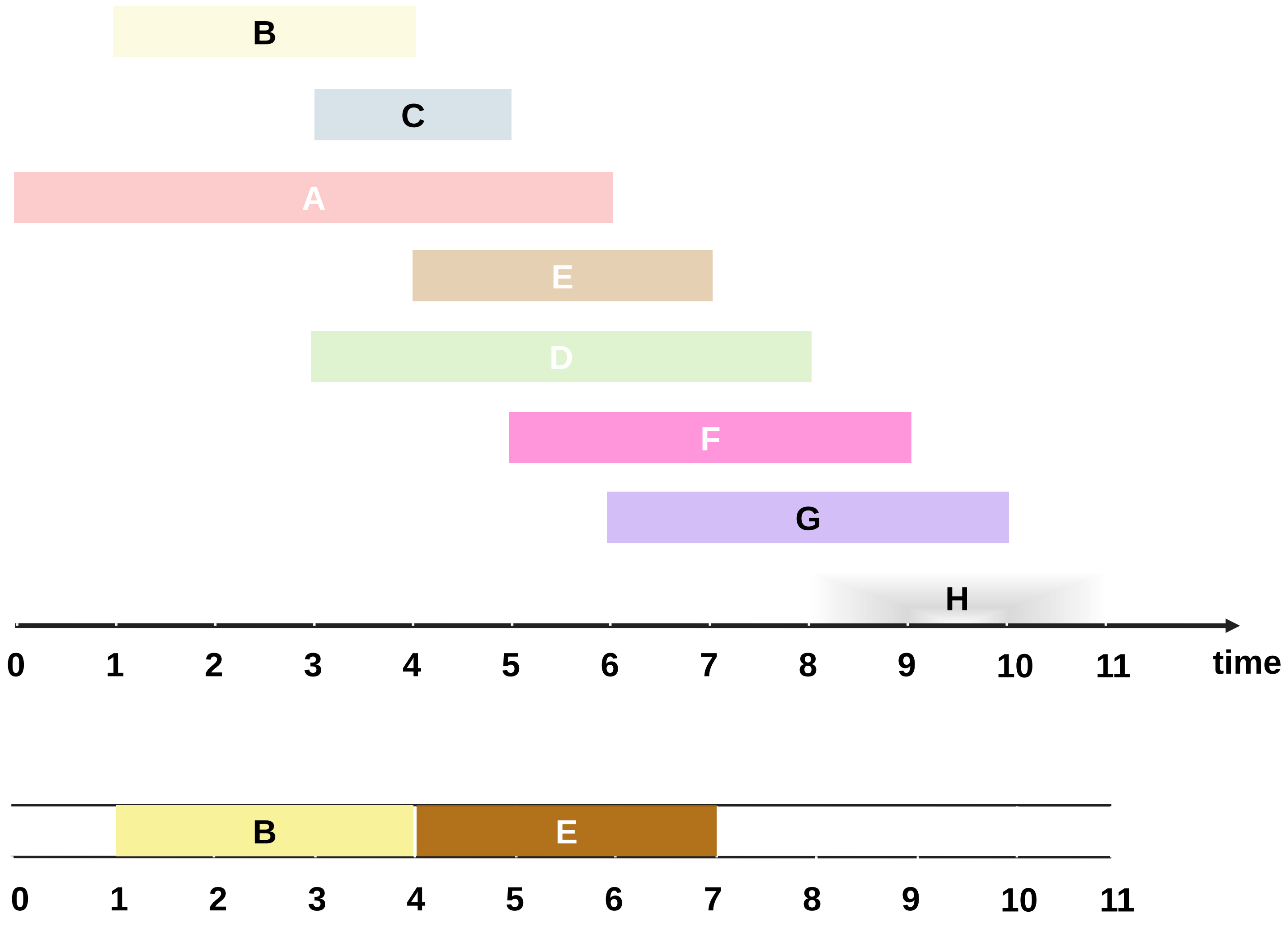
Earliest-finish-time-first algorithm demo



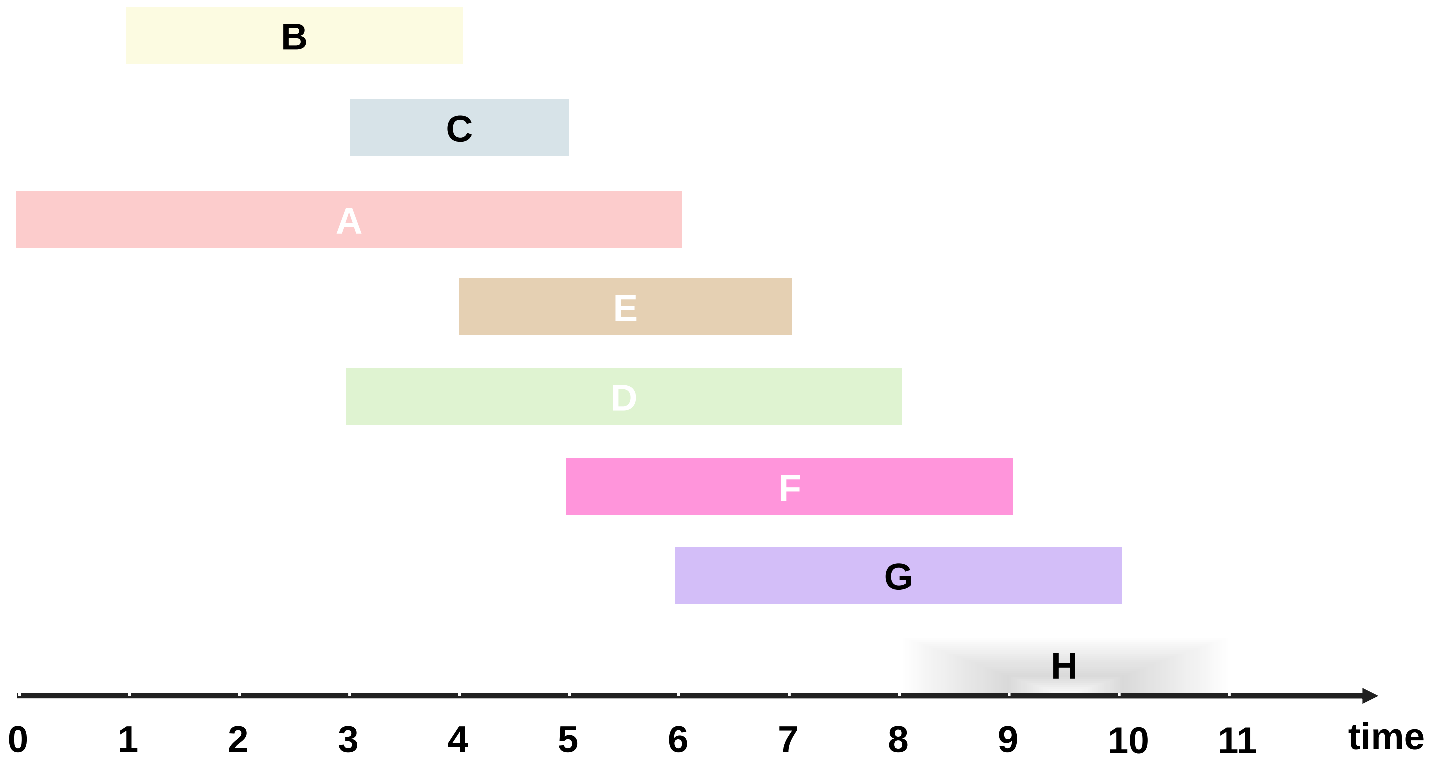
job D is incompatible(do not add to schedule)



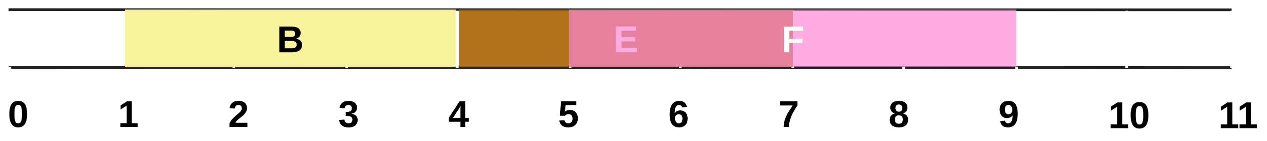
Earliest-finish-time-first algorithm demo



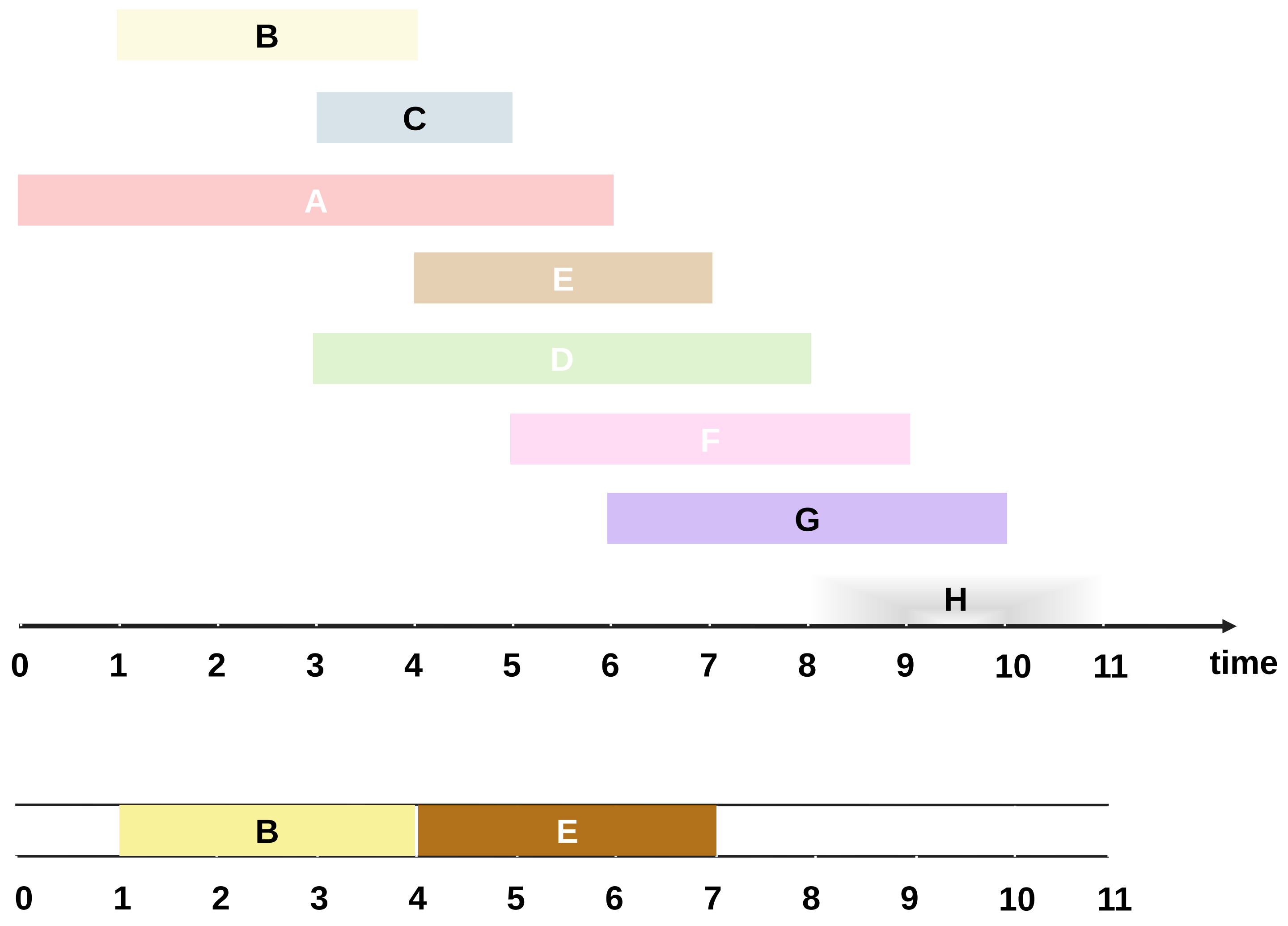
Earliest-finish-time-first algorithm demo



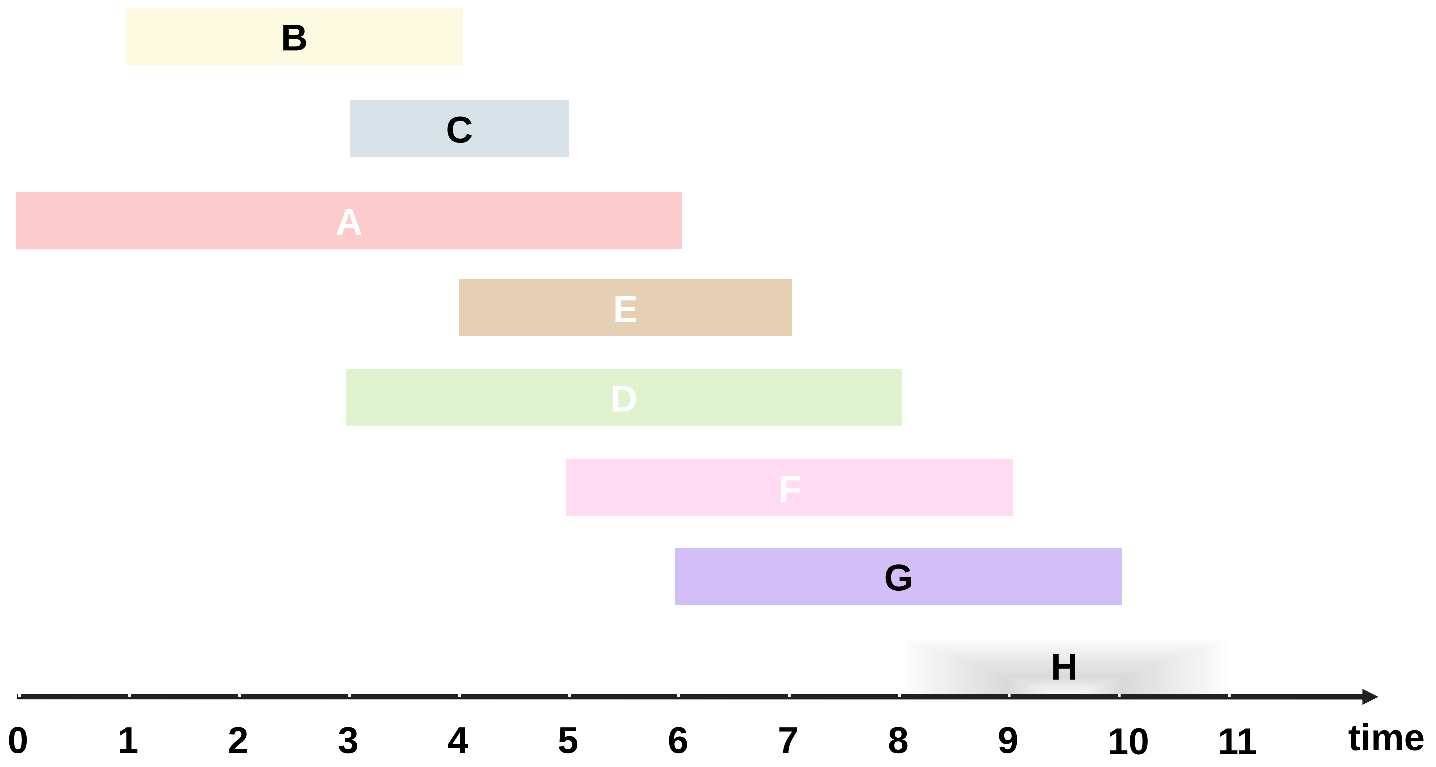
job F is incompatible(do not add to schedule)



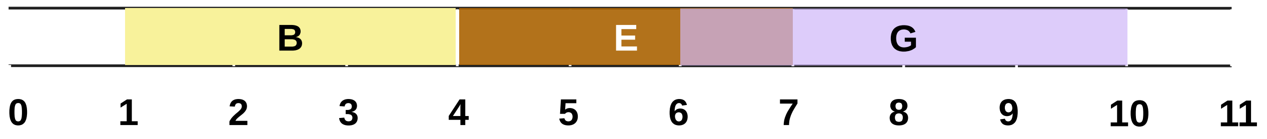
Earliest-finish-time-first algorithm demo



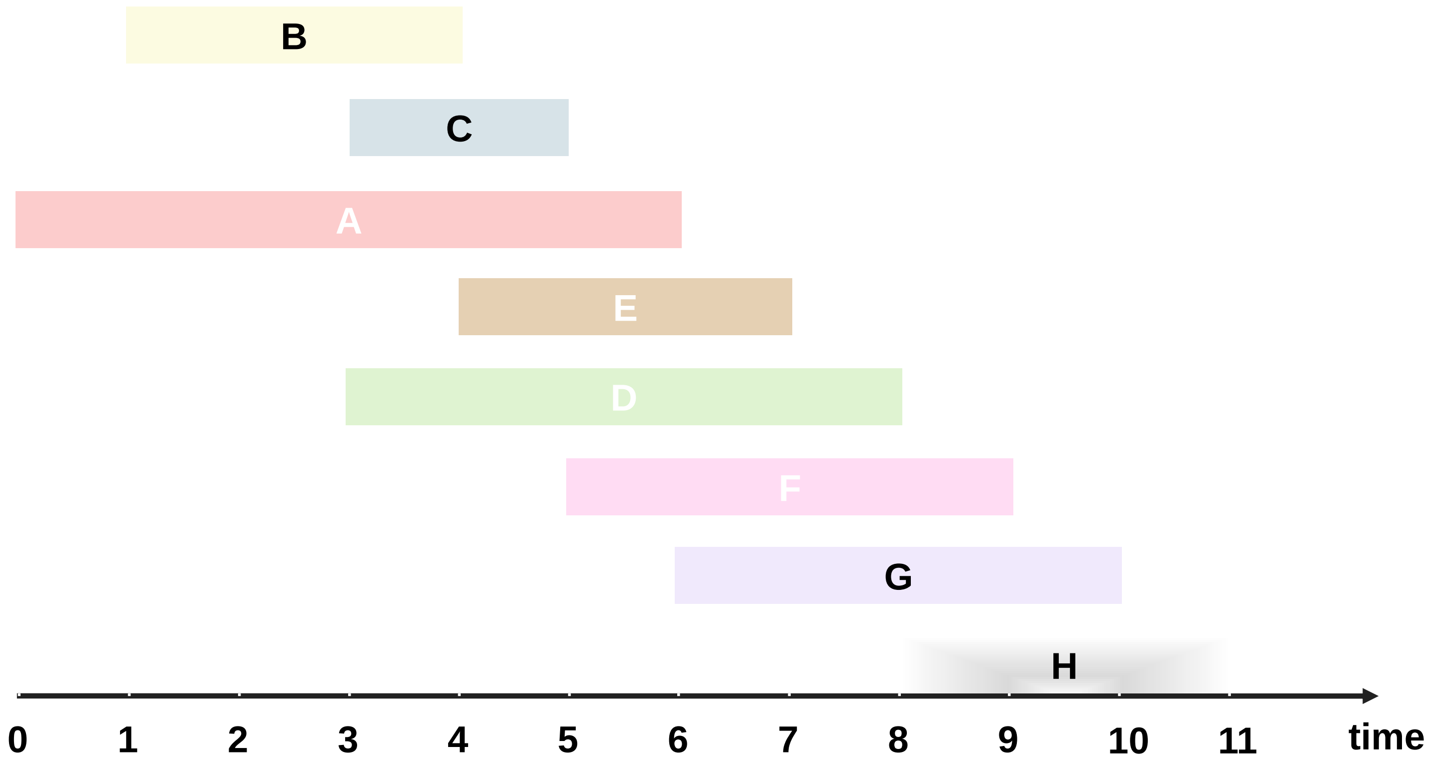
Earliest-finish-time-first algorithm demo

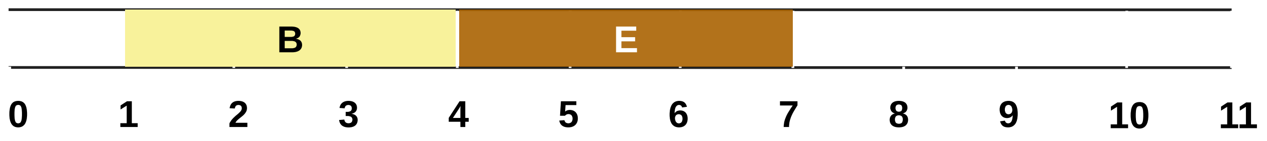


job G is incompatible(do not add to schedule)

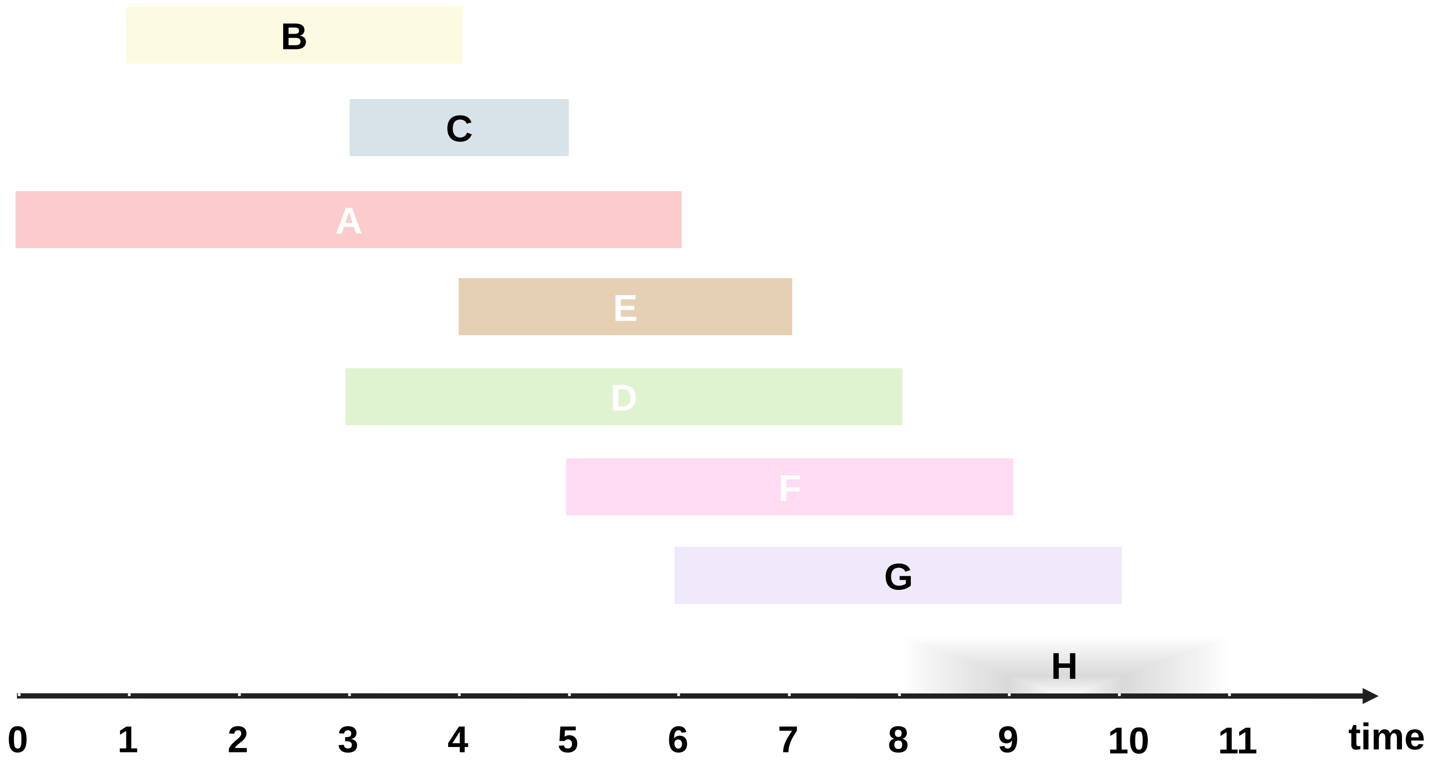


Earliest-finish-time-first algorithm demo

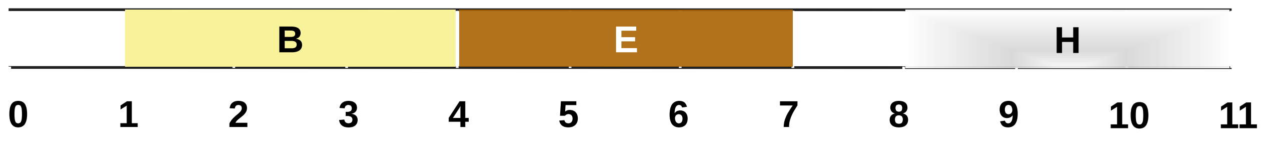




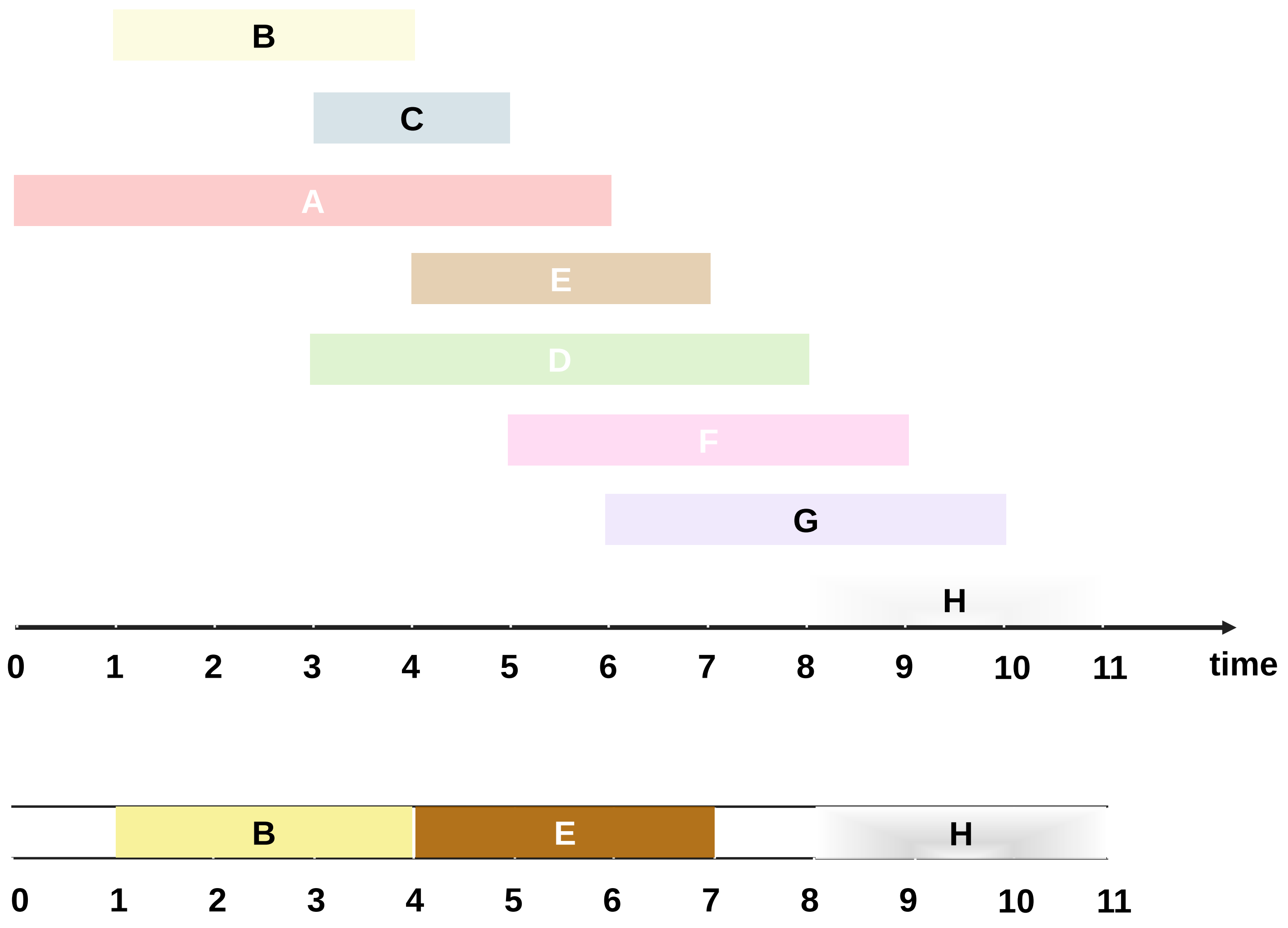
Earliest-finish-time-first algorithm demo



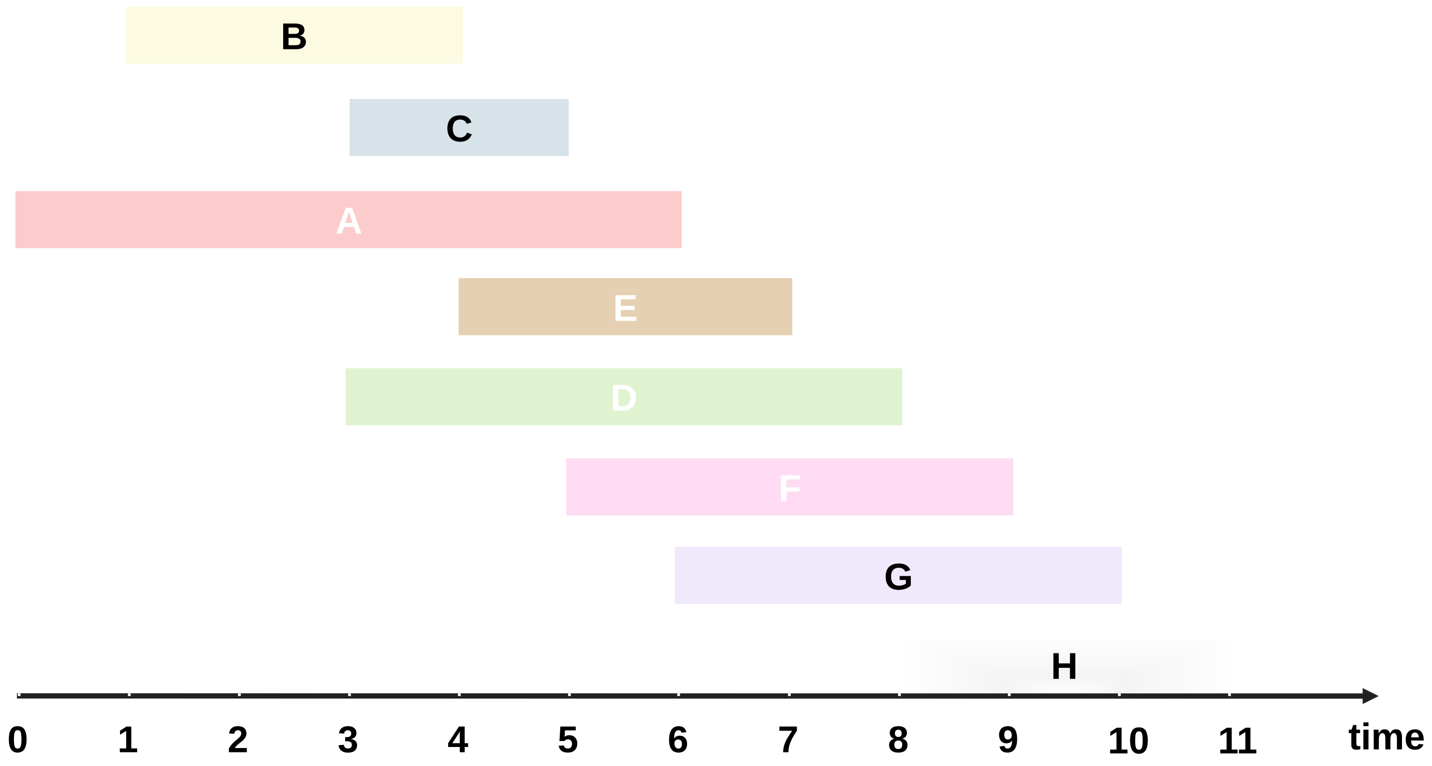
job H is compatible(add to schedule)



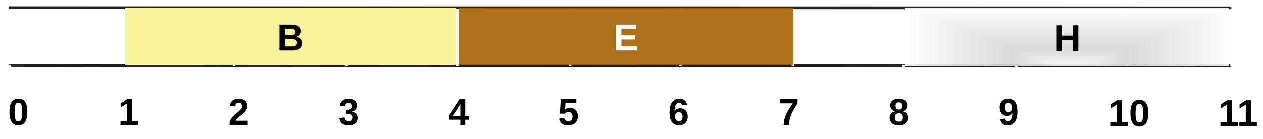
Earliest-finish-time-first algorithm demo



Earliest-finish-time-first algorithm demo



done (an optimal set of job)



Outline

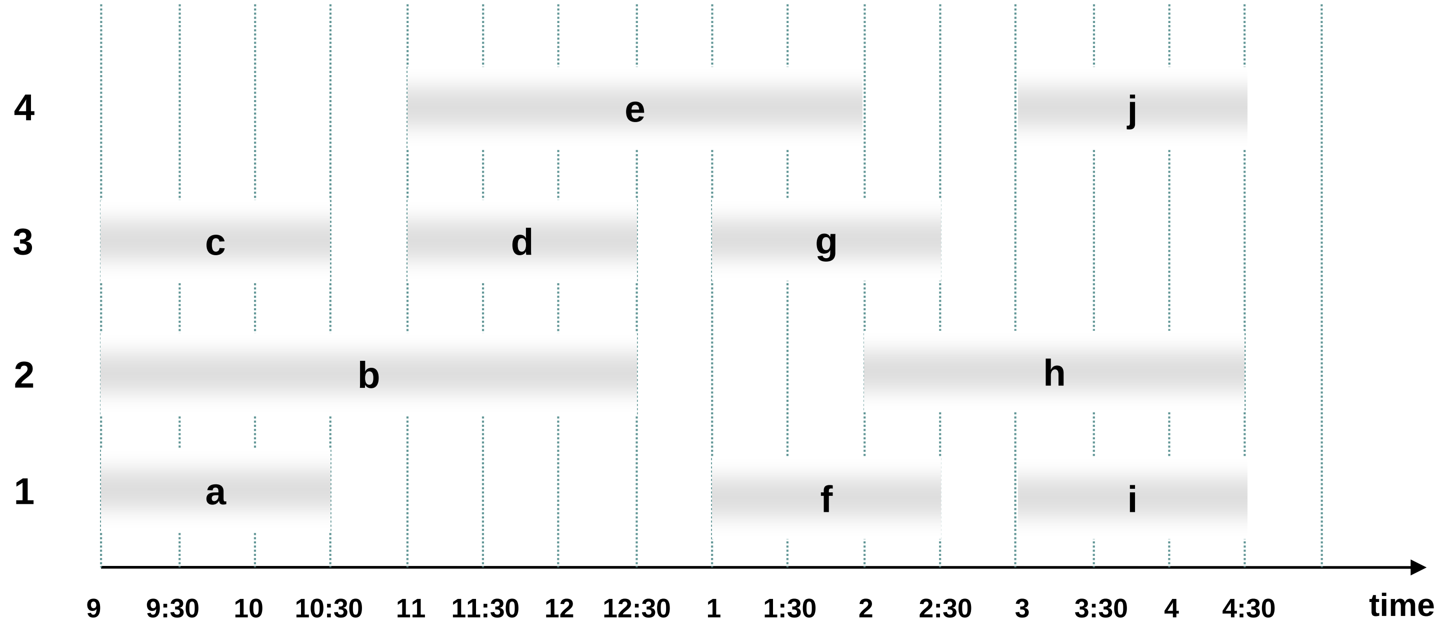
- Introduction to Part V
- The Fraction Knapsack Problem
  - Problem Definition
  - A Greedy Algorithm
  - Correctness
- Interval Scheduling and Interval Partitioning
  - Interval Scheduling
  - Interval Partitioning

Interval Partitioning

Interval partitioning

- Lecture *j* starts at s_j and finishes at f_j .
- Goal: find manimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same room.

Ex. This schedule uses 4 classrooms to schedule 10 lectures

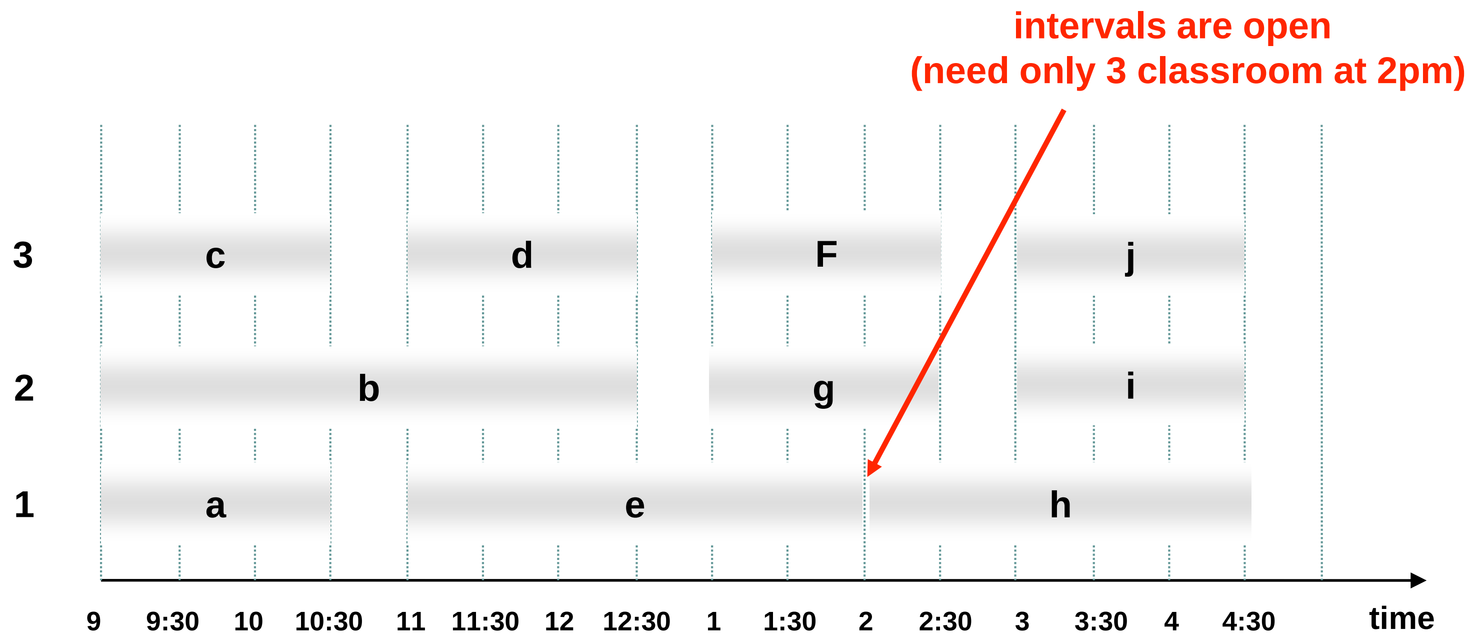


Interval Partitioning

Interval partitioning

- Lecture *j* starts at s_j and finishes at f_j .
- Goal: find manimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same room.

Ex. This schedule uses 3 classrooms to schedule 10 lectures



Interval Partitioning: greedy algorithms

Greedy template. Consider lectures in some natural order.

Assign each lecture to an available classroom(which one?);

Allocate a new classroom if none are available.

- [Earliest start time] Consider lectures in ascending order of $s_{j.}$
- [Earliest finish time] Consider lectures in ascending order of $f_{j.}$
- [Shortest interval] Consider lectures in ascending order of f_j - $s_{j.}$
- [Fewest conflicts] For each lecture *j,* count the number of conflicting lectures c_j. Schedule in ascending order of $c_{j.}$

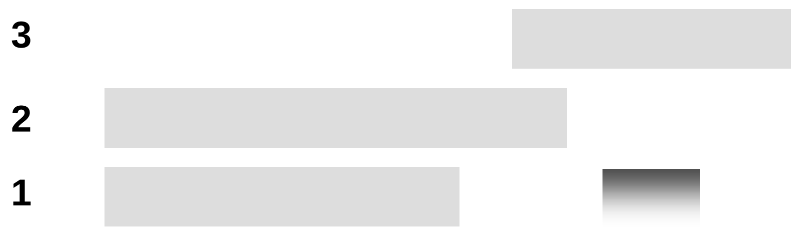
Interval Partitioning: greedy algorithms

Greedy template. Consider lectures in some natural order.

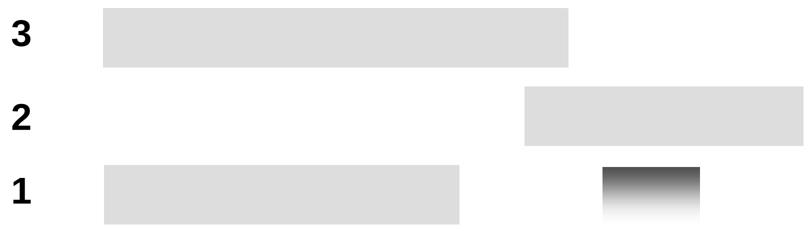
Assign each lecture to an available classroom(which one?);

Allocate a new classroom if none are available.

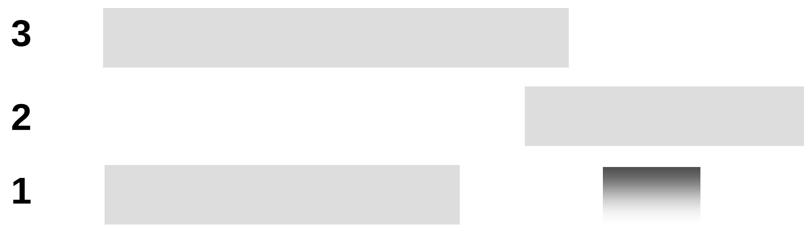
counterexample for earliest finish time



counterexample for shortest interval



counterexample for fewest conflicts



Interval Partitioning: earliest-start-time-first algorithm

Earliest-Start-Time-First$(n, s_1, s_2, ..., s_n, f_1, f_2, ..., f_n)$

```
Input: $n$ jobs with start time $s_i$ and finish time $f_i$.
Output: Schedule with minimum number of classrooms.
Sort lectures by finish time so that $s_1 \leq s_2 \leq ... \leq s_n$.
$d \leftarrow 0$;
for $j = 1$ to $n$ do
    if Lecture $j$ is compatible with some classroom then
        Schedule lecture $j$ in any such class room $k$;
    end
    else
        Allocate a new classroom $d + l$;
        Schedule lecture $j$ in classroom $d + l$;
    end
end
return schedule;
```

Interval Partitioning: earliest-start-time-first algorithm

Proposition. The earliest-time-first algorithm can be implemented in *O(n log n)* time.

Pf. Store classroom in a priority queue (key = finish time of its last lecture).

- To determine whether lecture *j* is compatible with some classroom, compare s_j to key min classroom *k* in priority queue.
- To add lecture *j* to classroom *k*, increase key of classroom *k* to f_j.
- Total number of priority queue operation is *O(n)*.
- Sorting by start time takes *O(n log n)* time.

Remark. This implement chooses a classroom *k* whose finish time of its last lecture is the earliest.

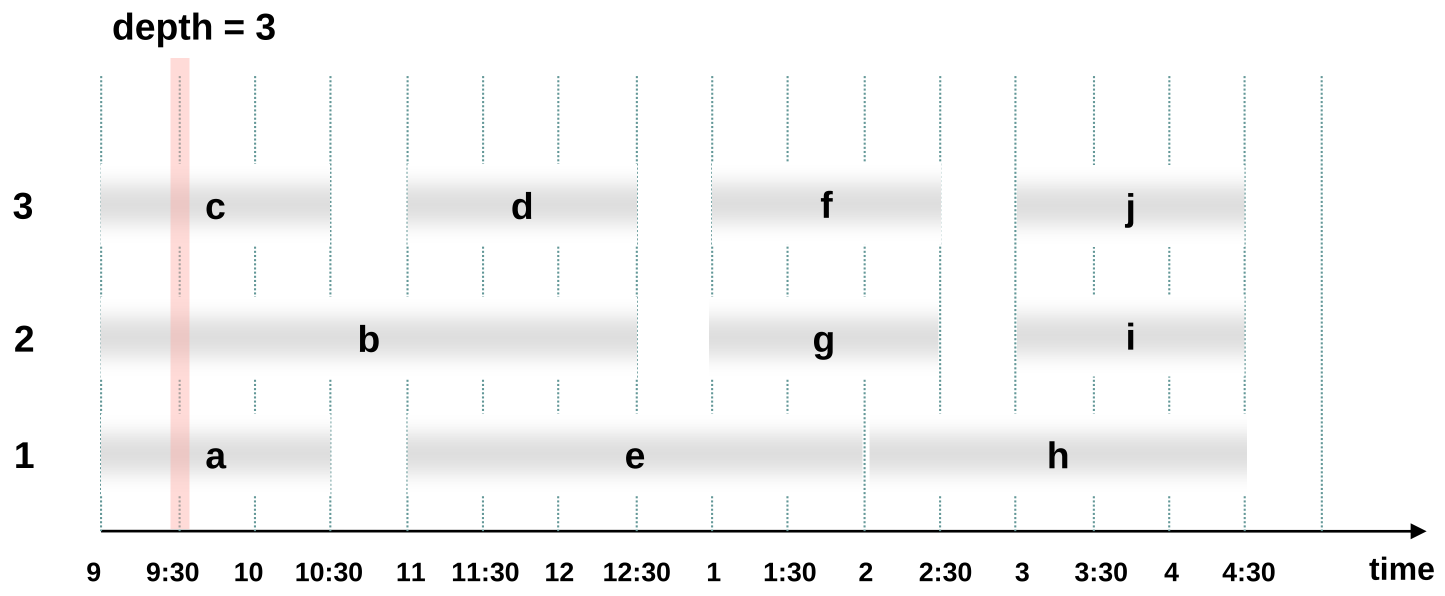
Interval Partitioning:lower bound on optimal solution

Def. The depth of a set of open interval is the maximum number of intervals that contain any given time.

Key observation. Number of classrooms needed ≥ depth.

Q. Does minimum number of classrooms needed always equal depth?

A. Yes! Moreover, earliest-start-time-first algorithm finds a schedule whose number of classrooms equals the depth.



Interval Partitioning:analysis of earliest-start-time-first algorithm

Observation. The earliest-start-time-first algorithm never schedules two incompatible lectures in the same classroom.

Theorem. Earliest-start-time-first algorithm is optimal.

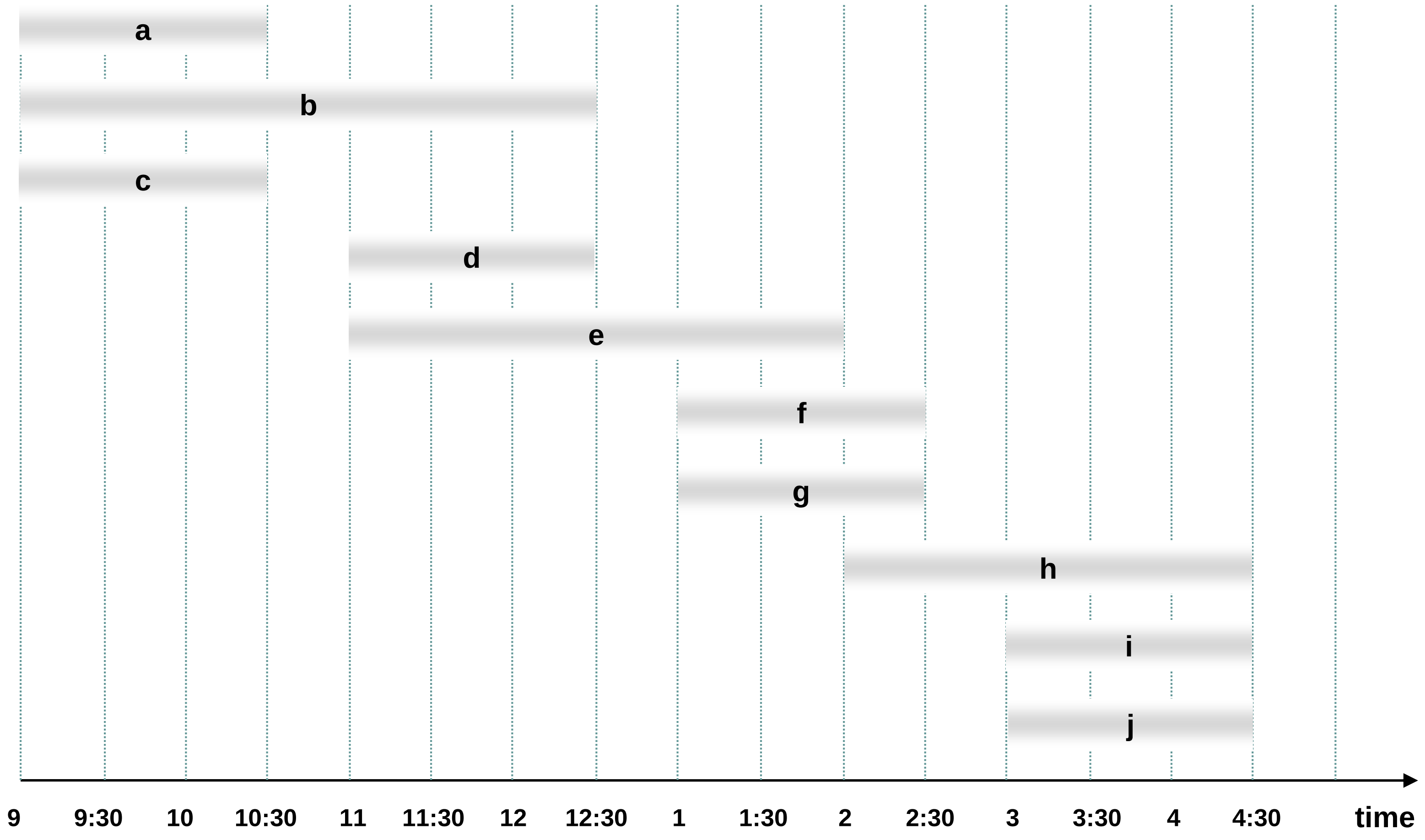
Pf.

- ➓Let d = number of classrooms that the algorithm allocates.
- ➓Classroom d is opened because we needed to schedule a lecture, say j, that is incompatible with all $d-1$ other classrooms.
- ➓These d lectures each end after s_j.
- ➓Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_j.
- ➓Thus, we have d lectures overlapping at time $s_j + \varepsilon$.
- ➓Key observation => all schedules use $\geq d$ classrooms.

Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- ❿Assign next lecture to any compatible classroom (if ones exists).
- ❿Otherwise, open up a new classroom.

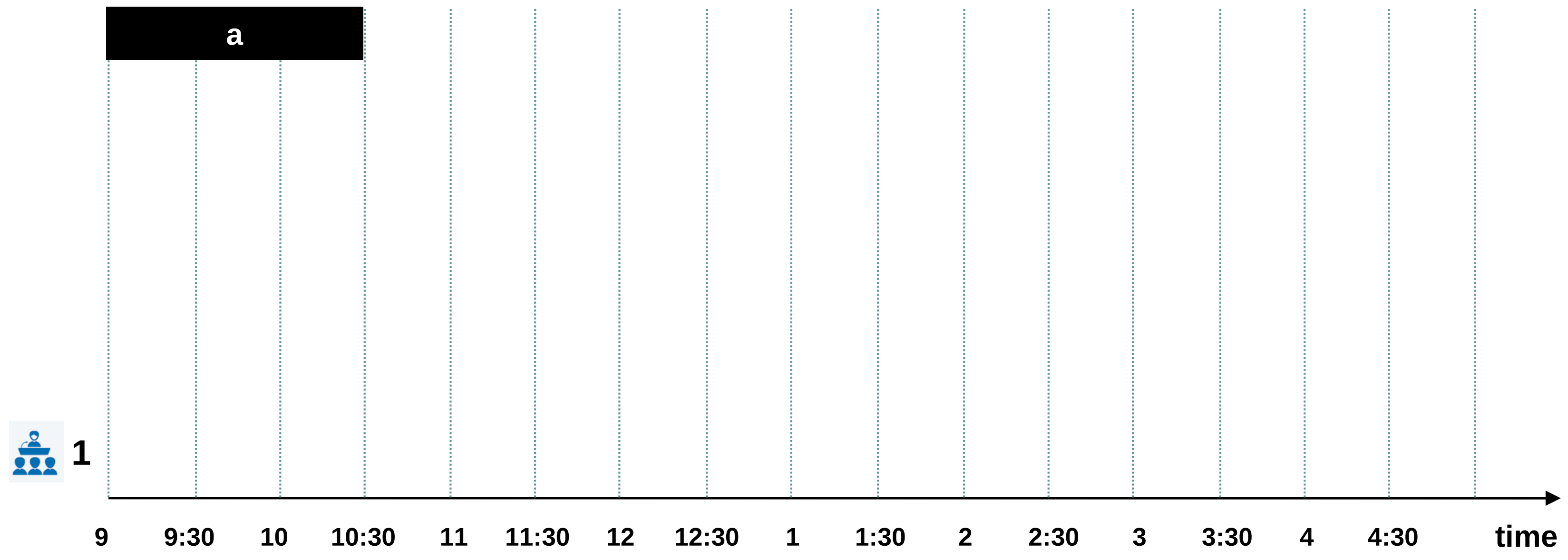


Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- ➉Assign next lecture to any compatible classroom (if ones exists).
- ➉Otherwise, open up a new classroom.

No compatible classroom: open up a new classroom and assign lecture to it.

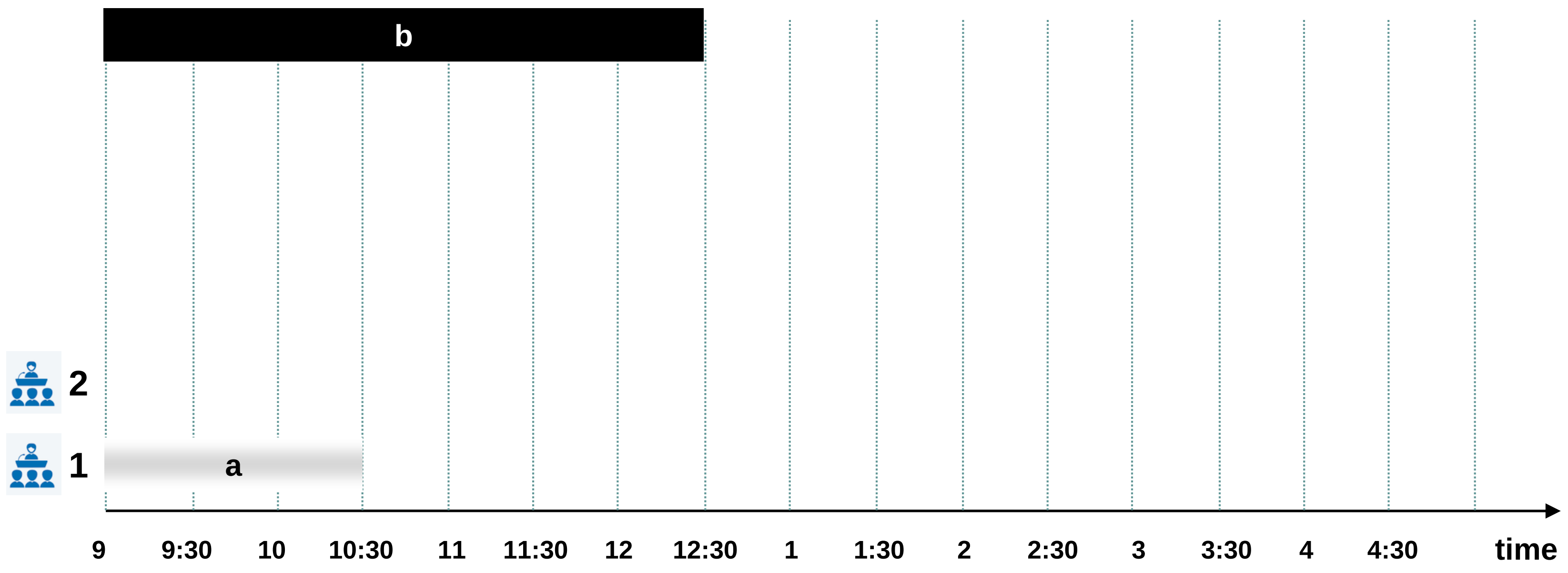


Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- ❿Assign next lecture to any compatible classroom (if ones exists).
- ❿Otherwise, open up a new classroom.

No compatible classroom: open up a new classroom and assign lecture to it.

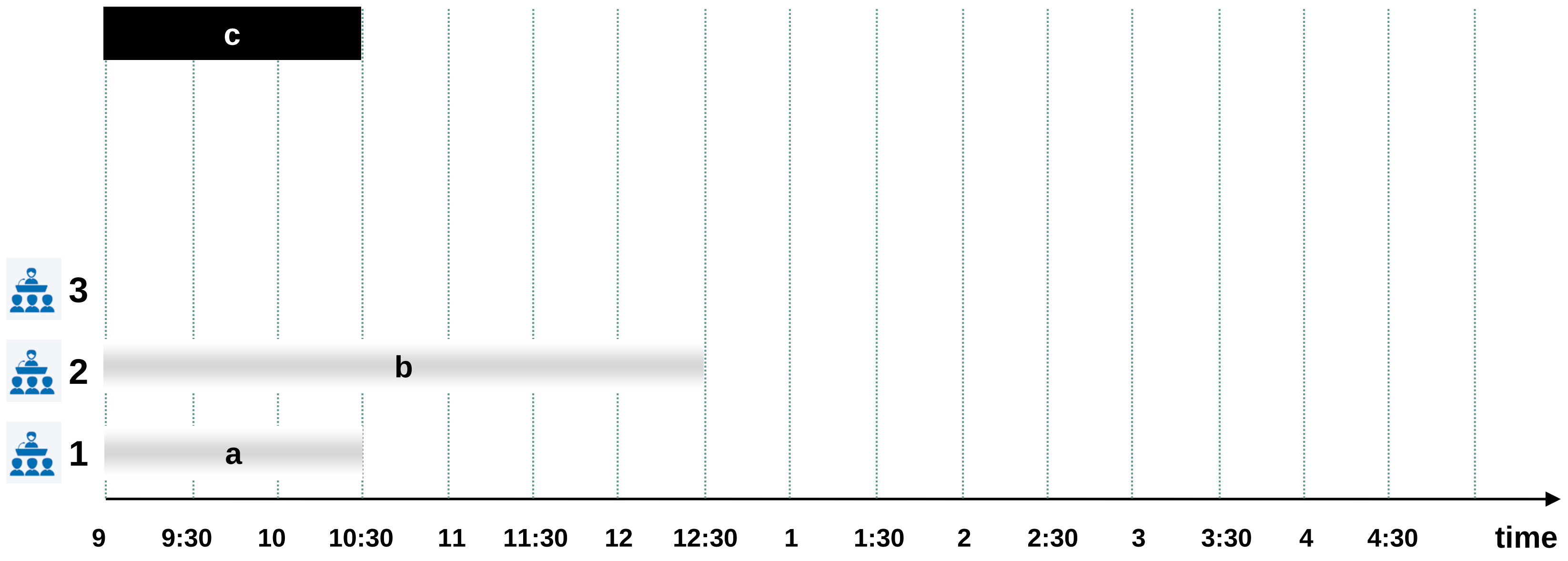


Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- Assign next lecture to any compatible classroom (if ones exists).
- Otherwise, open up a new classroom.

No compatible classroom: open up a new classroom and assign lecture to it.

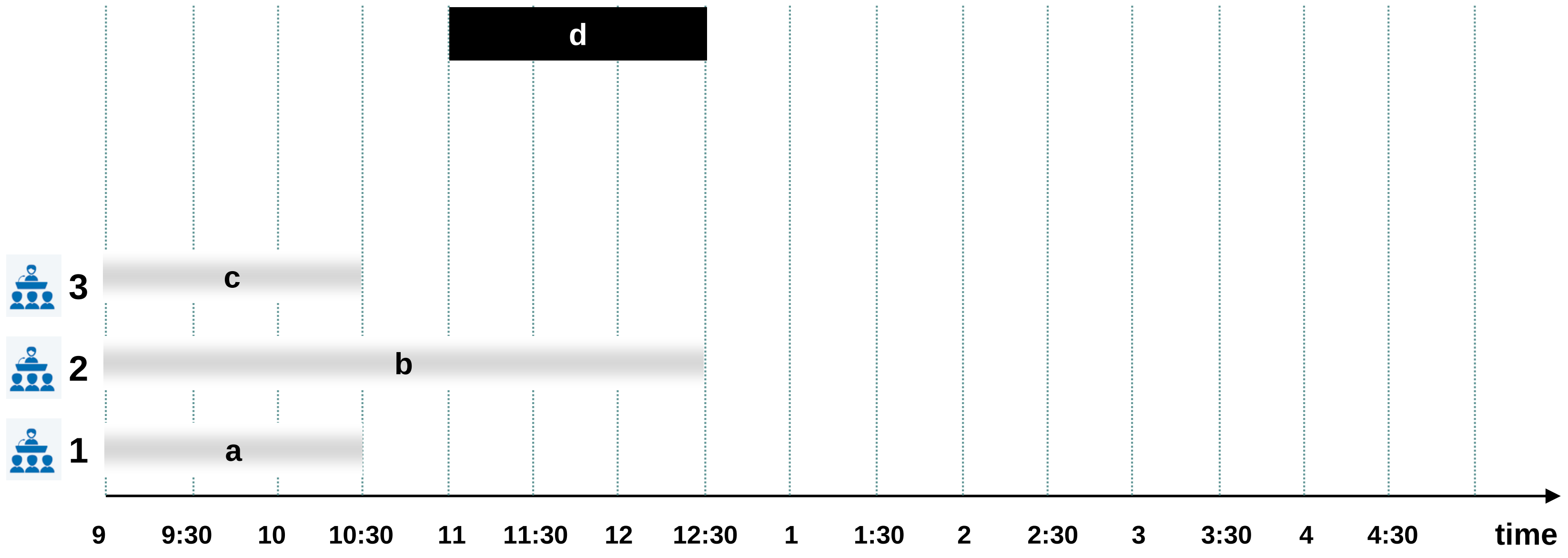


Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- ➓Assign next lecture to any compatible classroom (if ones exists).
- ➓Otherwise, open up a new classroom.

lecture d is compatible with classroom 1 and 3

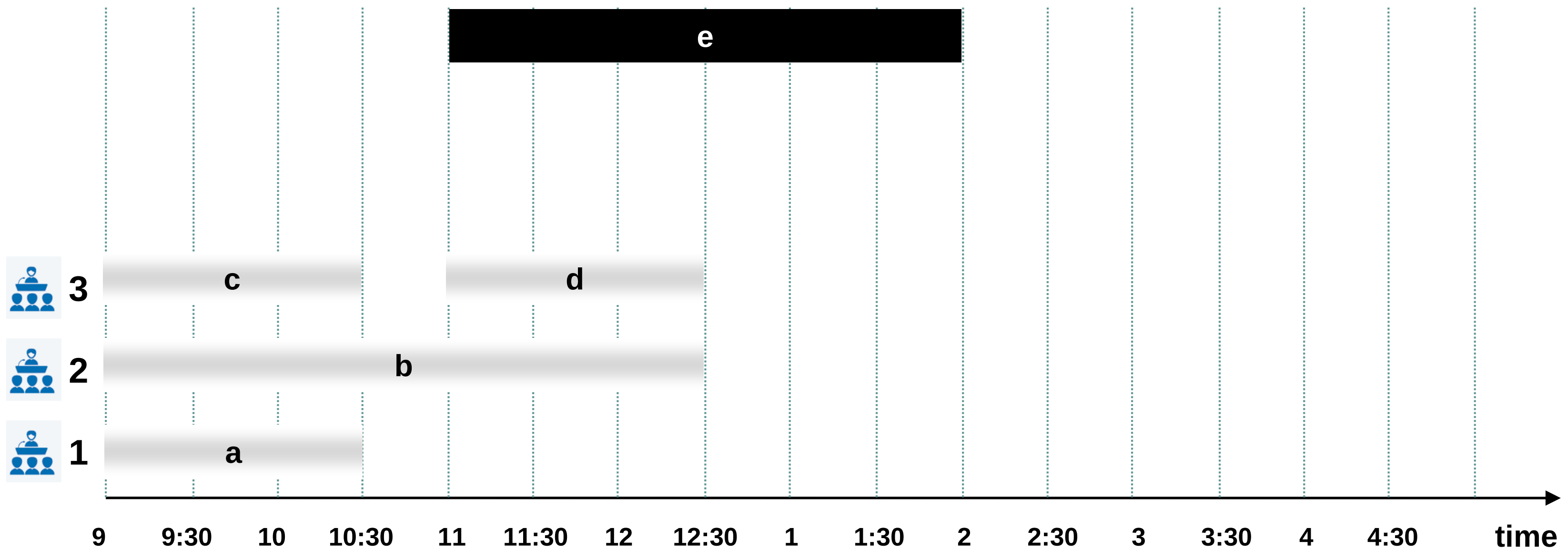


Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- ❿Assign next lecture to any compatible classroom (if ones exists).
- ❿Otherwise, open up a new classroom.

lecture e is compatible with classroom 1

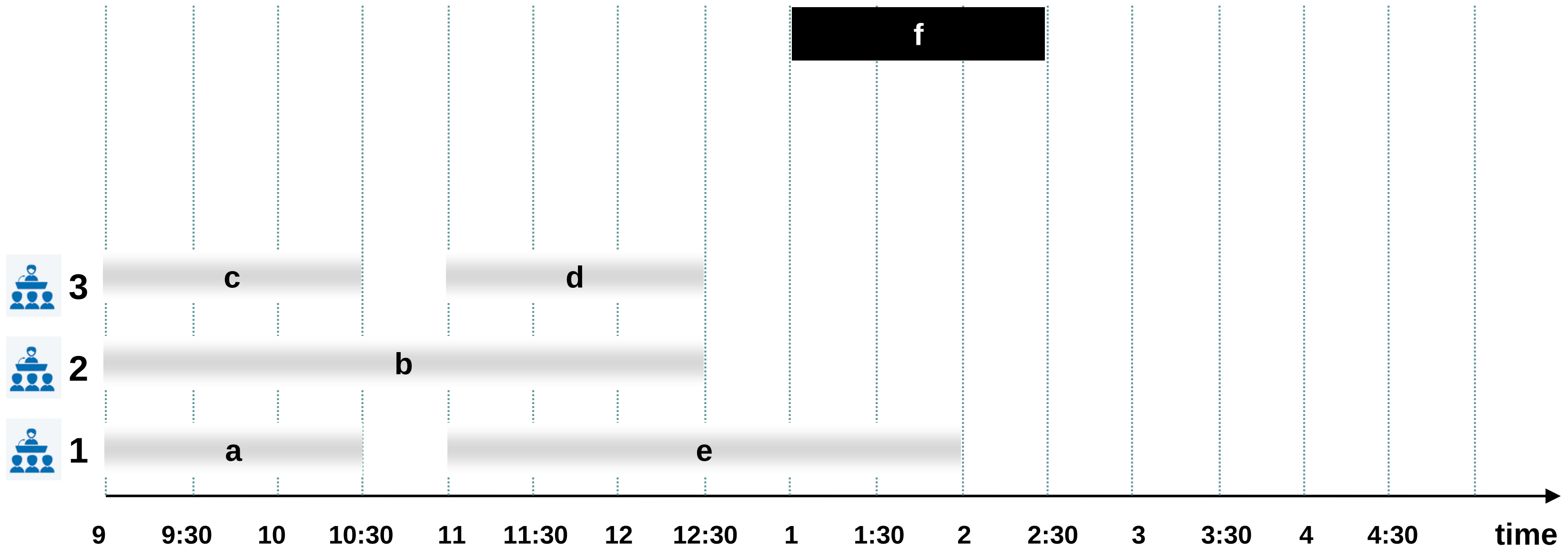


Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- Assign next lecture to any compatible classroom (if ones exists).
- Otherwise, open up a new classroom.

lecture f is compatible with classroom 2 and 3

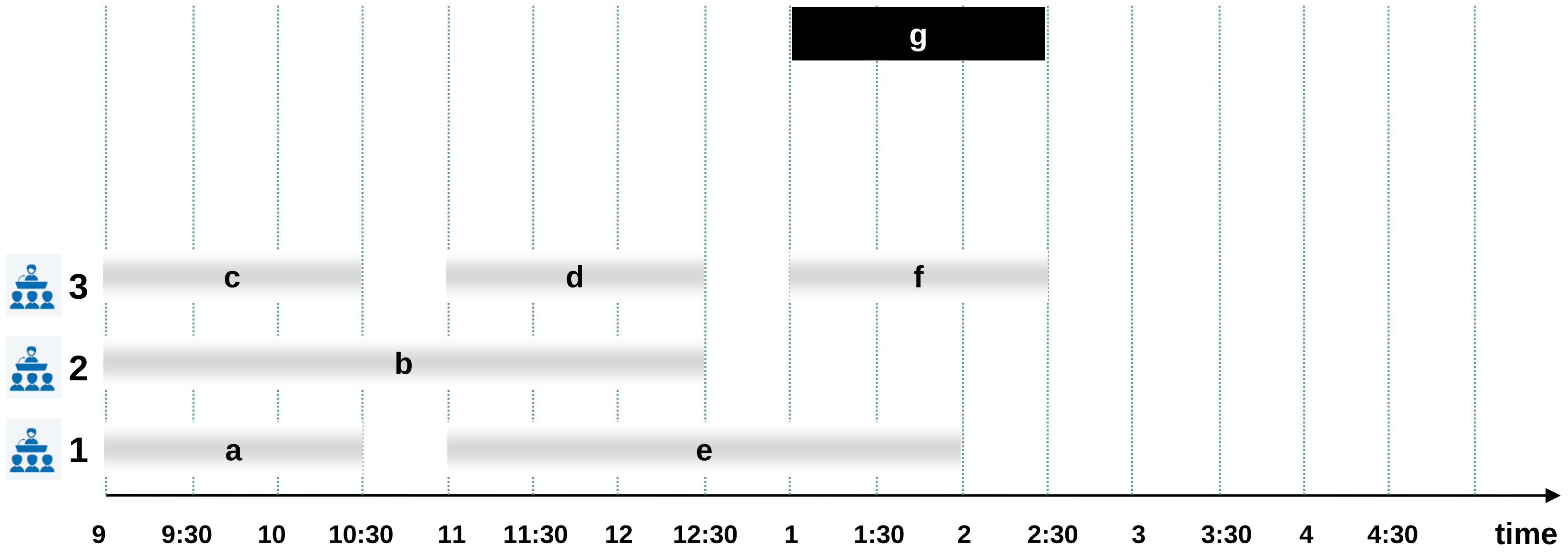


Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- Assign next lecture to any compatible classroom (if ones exists).
- Otherwise, open up a new classroom.

lecture g is compatible with classroom 2

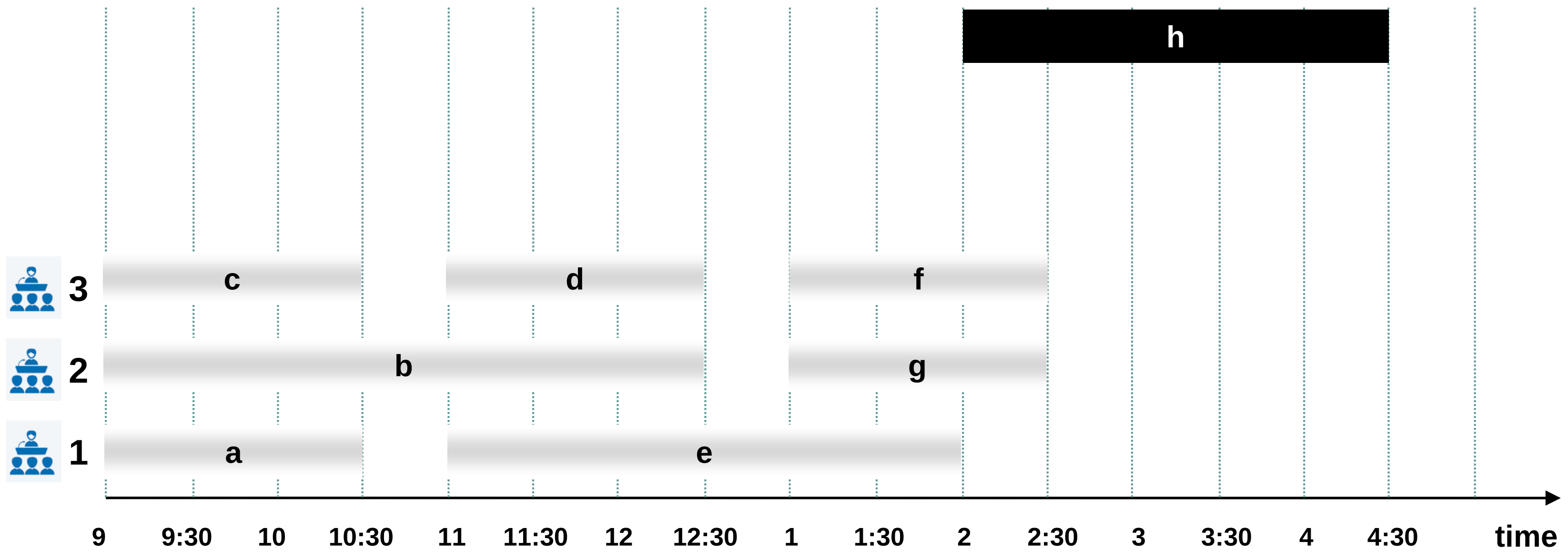


Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- Assign next lecture to any compatible classroom (if ones exists).
- Otherwise, open up a new classroom.

lecture h is compatible with classroom 1

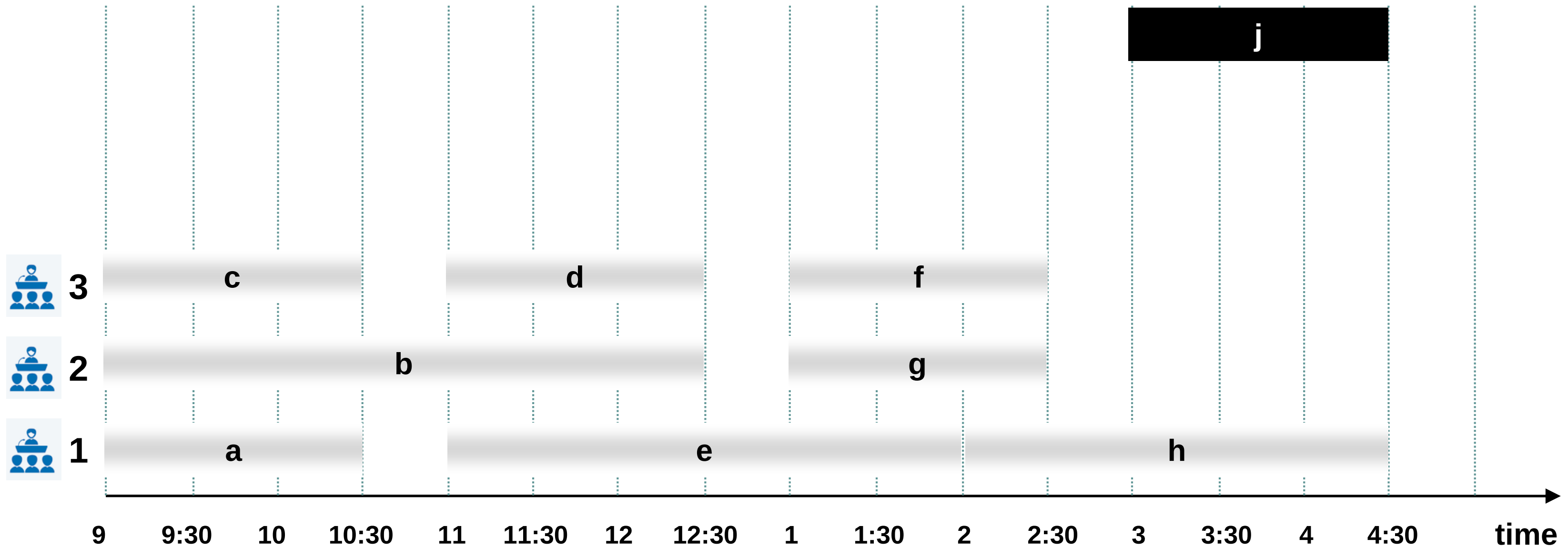


Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- ➓Assign next lecture to any compatible classroom (if ones exists).
- ➓Otherwise, open up a new classroom.

lecture j is compatible with classroom 2 and 3

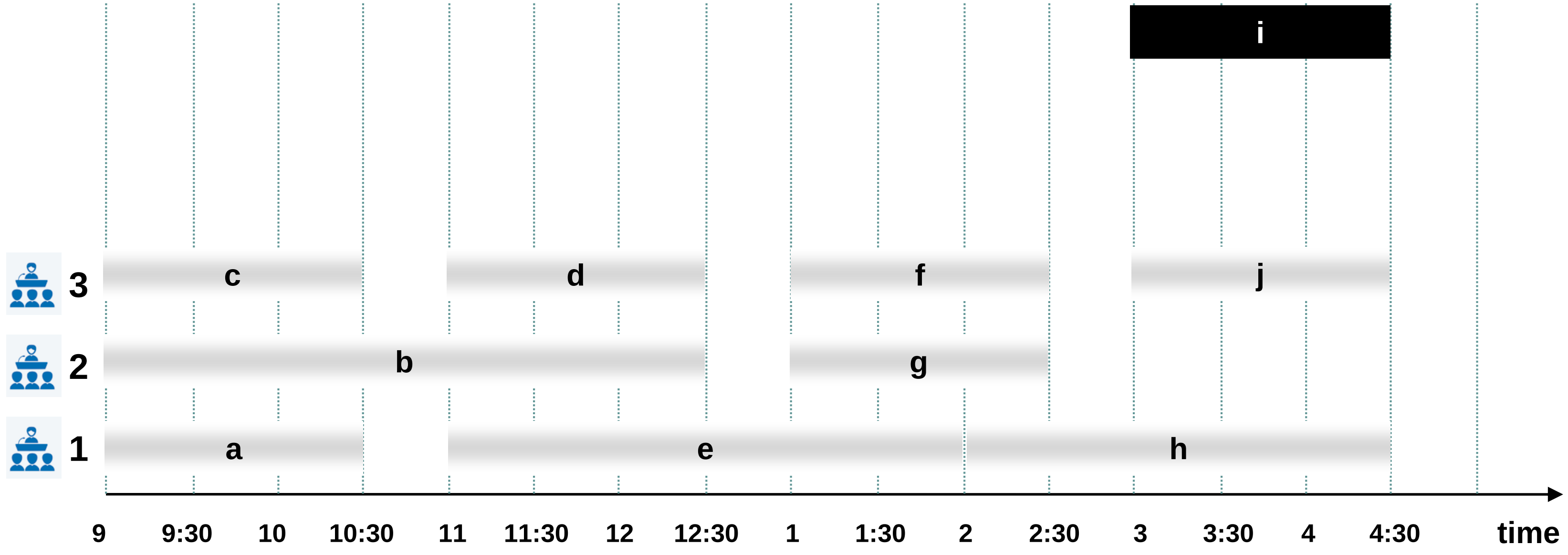


Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- Assign next lecture to any compatible classroom (if ones exists).
- Otherwise, open up a new classroom.

lecture i is compatible with classroom 2

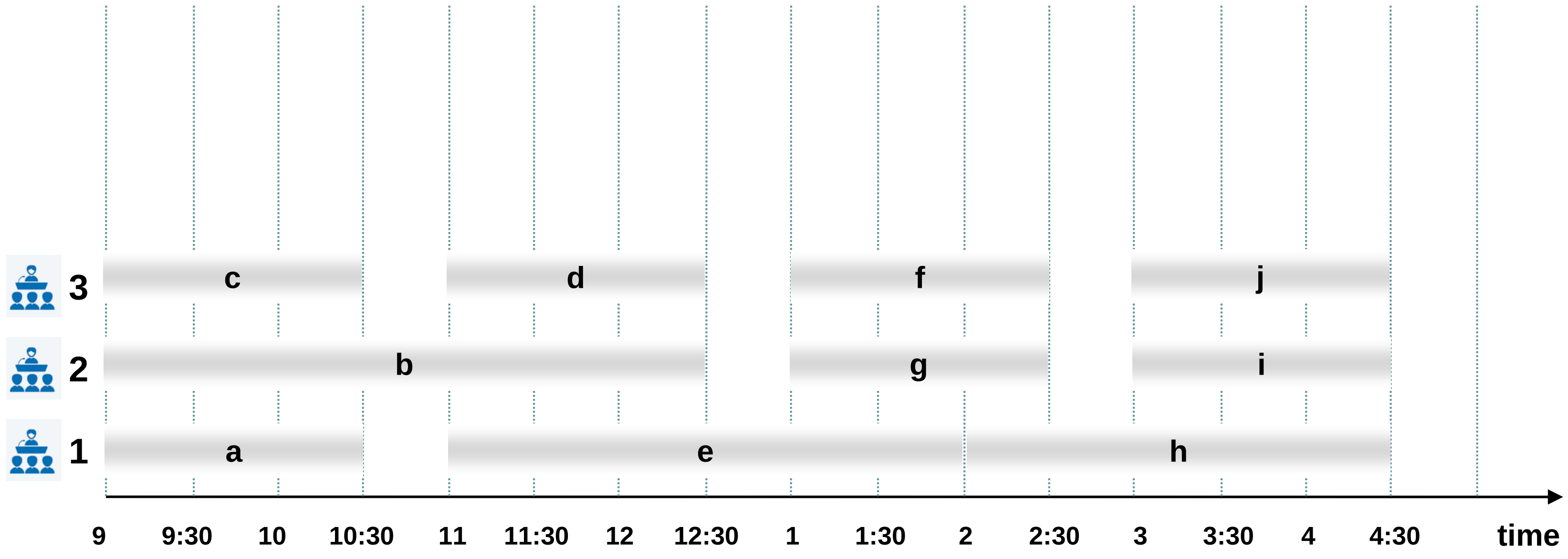


Earliest-start-time-first algorithm demo

Consider lectures in order of start time:

- ❿Assign next lecture to any compatible classroom (if ones exists).
- ❿Otherwise, open up a new classroom.

done



dank u
ju faleminderit
Tack
Asante
谢谢
Tak
mulţumesc
kiitos
Gracias
Salamat!
Terima kasih
Aliquam
Merci
Dankie
Obrigado
ありがとう
köszönöm
grazie
Aliquam
Go raibh maith agat
děkuji
Thank you
gam