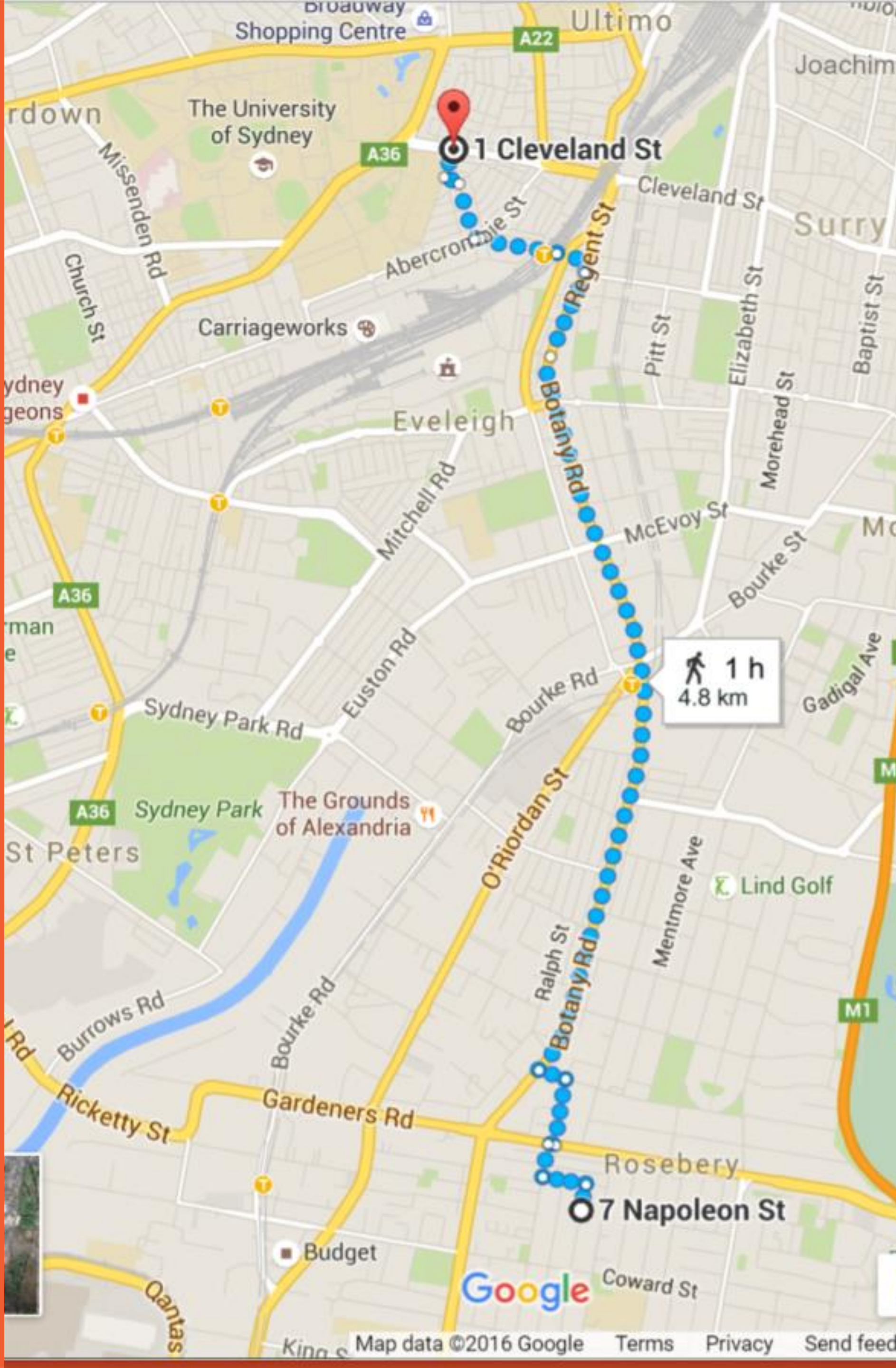
Lecture 3:
Greedy algorithms



THE UNIVERSITY OF
SYDNEY

General techniques in this course

- Greedy algorithms [today]
- Divide & Conquer algorithms [21 Aug]
- Sweepline algorithms [28 Aug]
- Dynamic programming algorithms [4 and 11 Sep]
- Network flow algorithms [18 Sep and 9 Oct]

Greedy algorithms

A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

Greedy algorithms

Greedy algorithms can be some of the simplest algorithms to implement, but they're often among the hardest algorithms to design and analyse.

Greedy: Overview

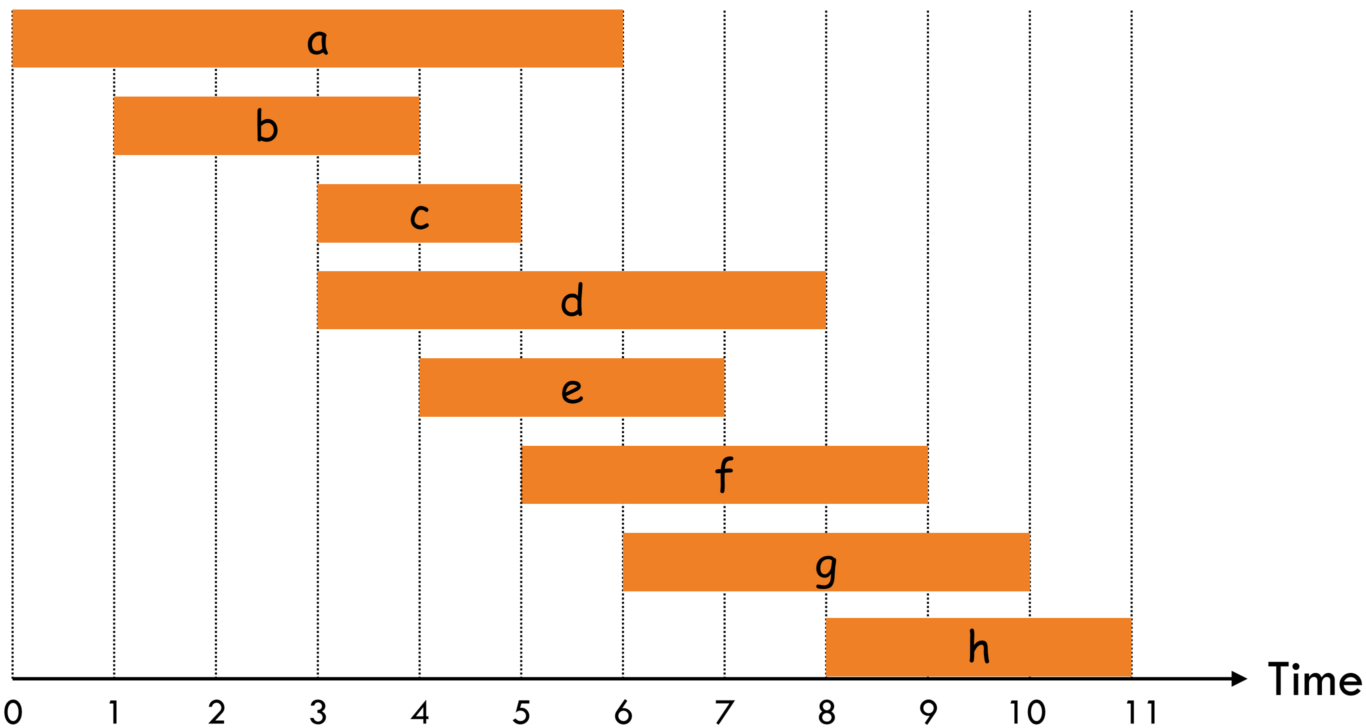
Consider problems that can be solved using a greedy algorithm.

– Interval scheduling/partitioning
– Scheduling to minimize lateness
– Shortest path
– Minimum spanning trees

Interval Scheduling

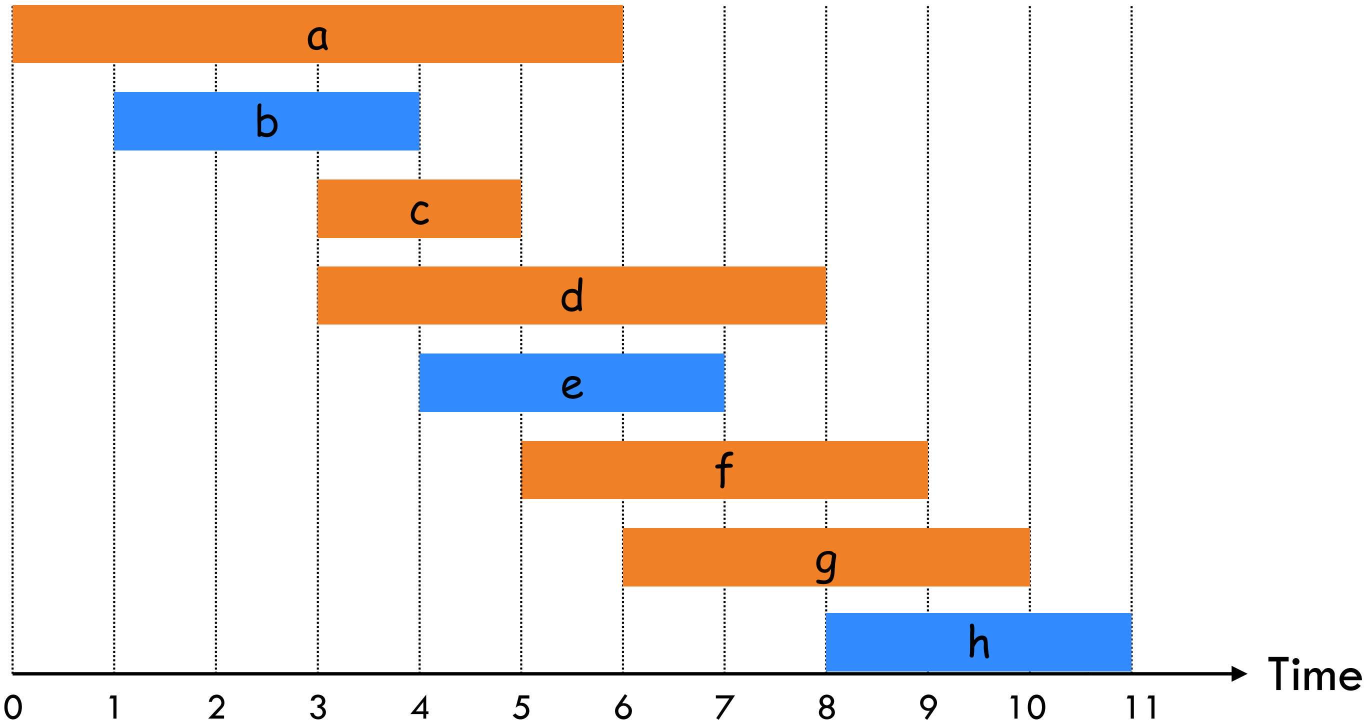
Interval Scheduling

- Interval scheduling.
  - **Input:** Set of n jobs. Each job i starts at time s_i and finishes at time f_i.
  - Two jobs are compatible if they don't overlap in time.
  - **Goal:** find maximum subset of mutually compatible jobs.



Interval Scheduling

- Interval scheduling.
  - **Input:** Set of n jobs. Each job i starts at time s_i and finishes at time f_i.
  - Two jobs are compatible if they don't overlap in time.
  - **Goal:** find maximum subset of mutually compatible jobs.

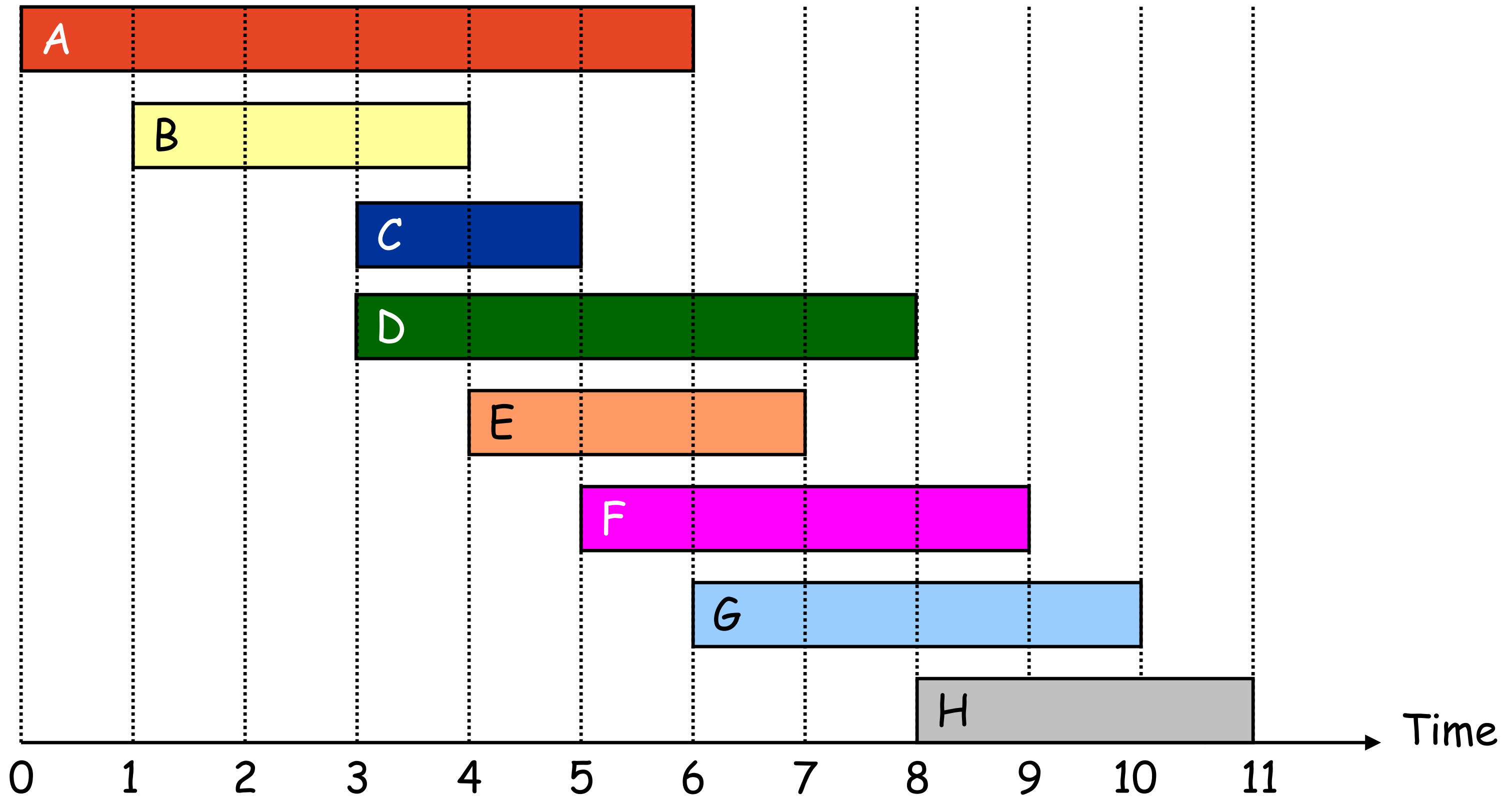


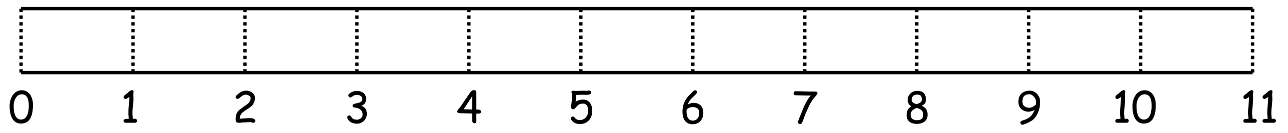
Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it is compatible with the ones already taken.

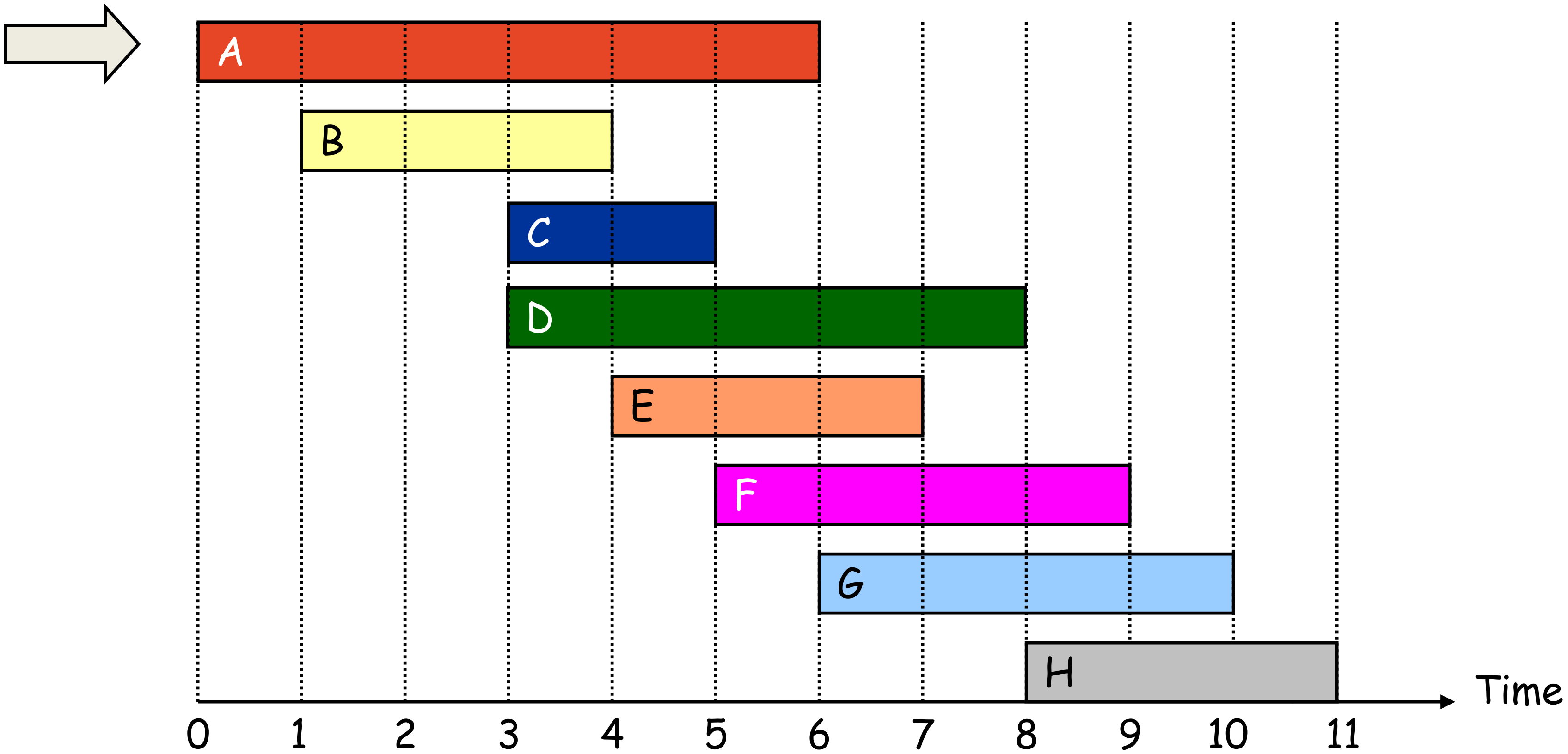
– [Earliest start time] Consider jobs in ascending order of start time s_j.

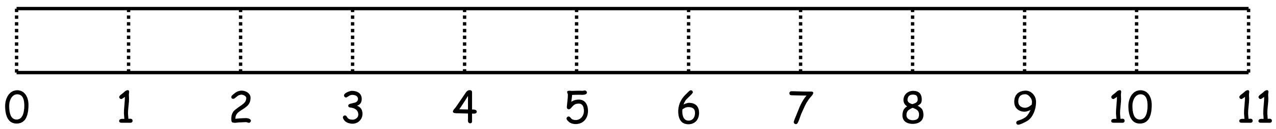
Interval Scheduling - [Earliest start time]



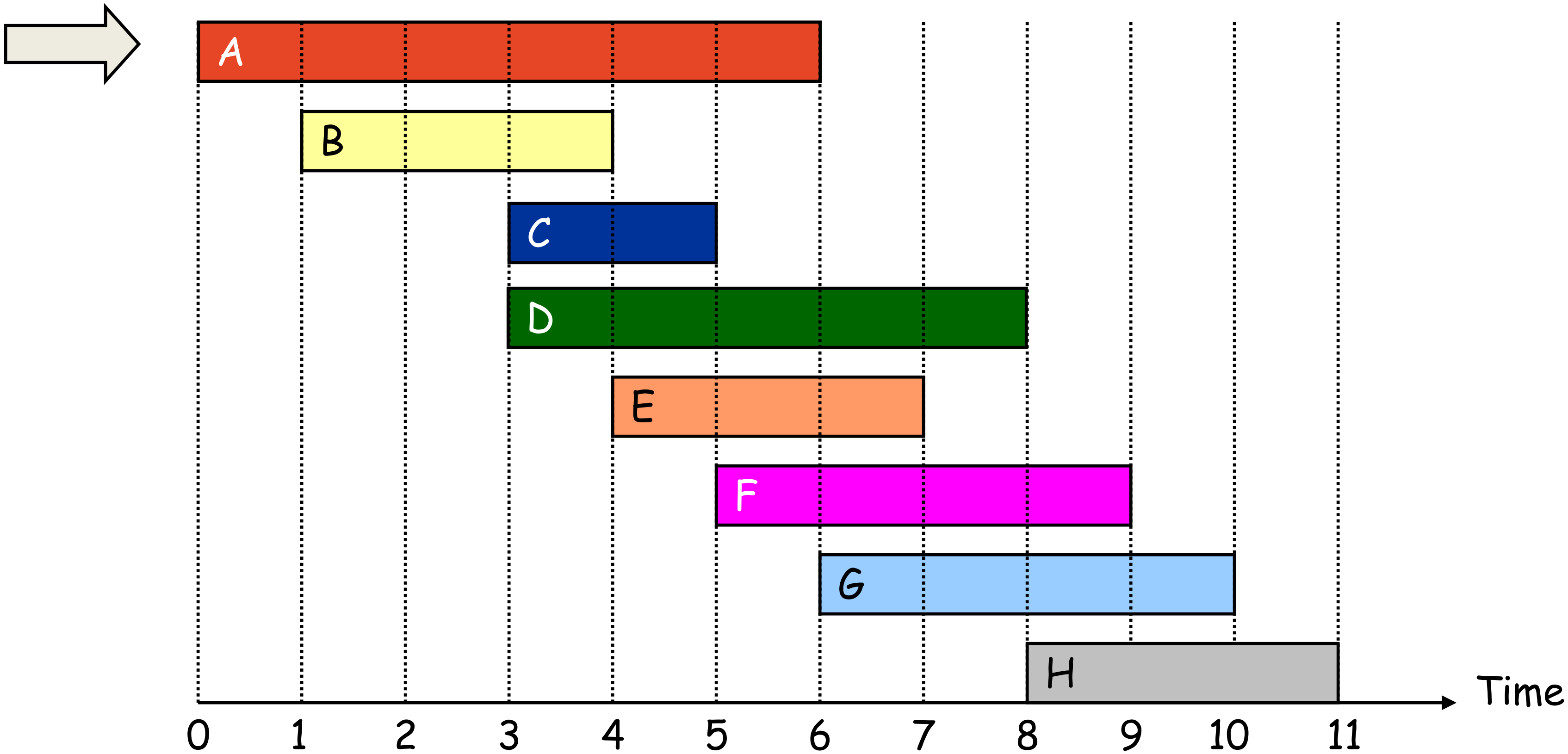


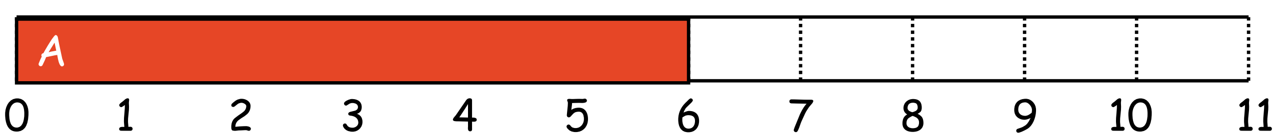
Interval Scheduling - [Earliest start time]



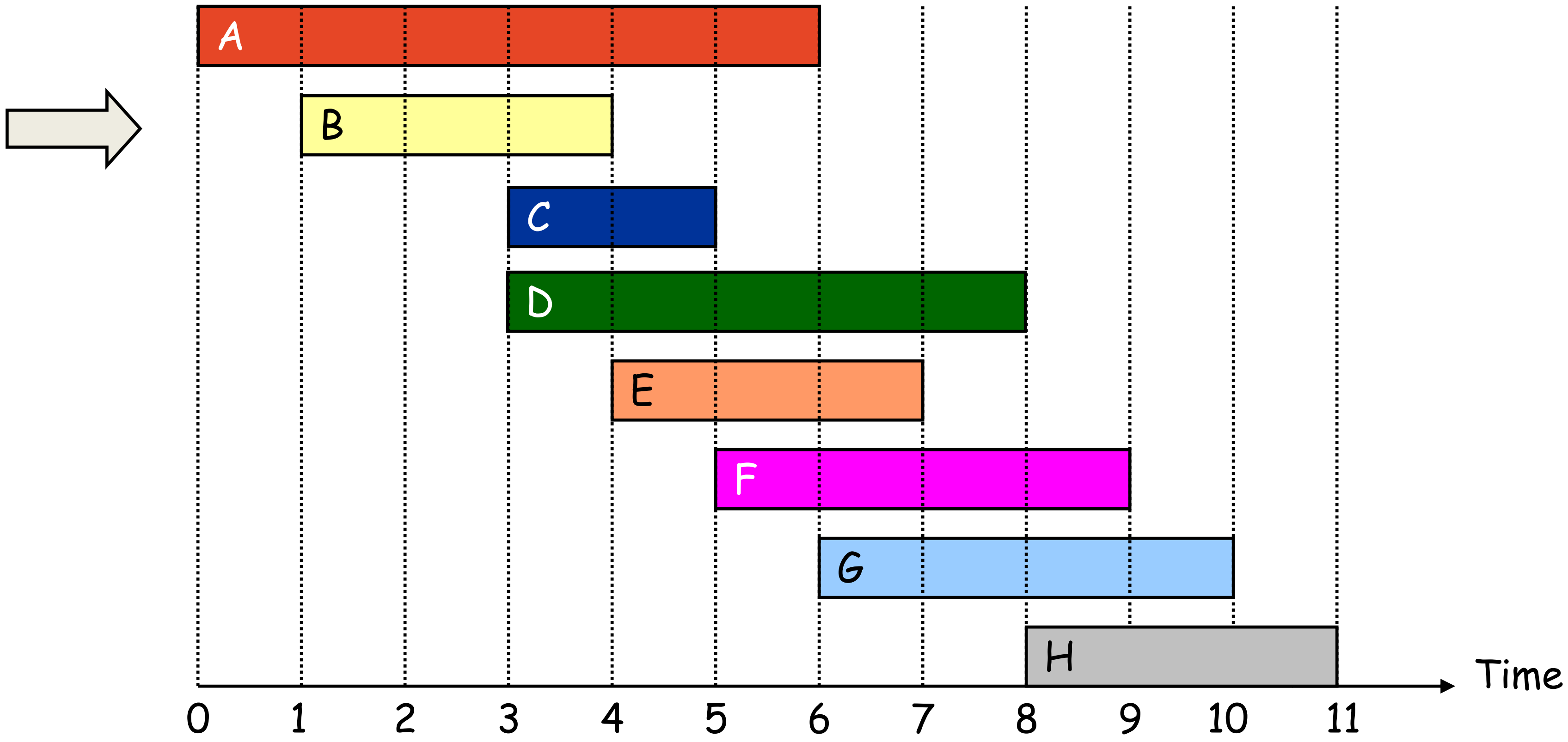


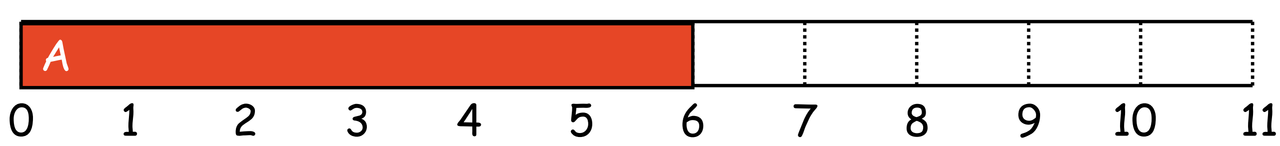
Interval Scheduling - [Earliest start time]



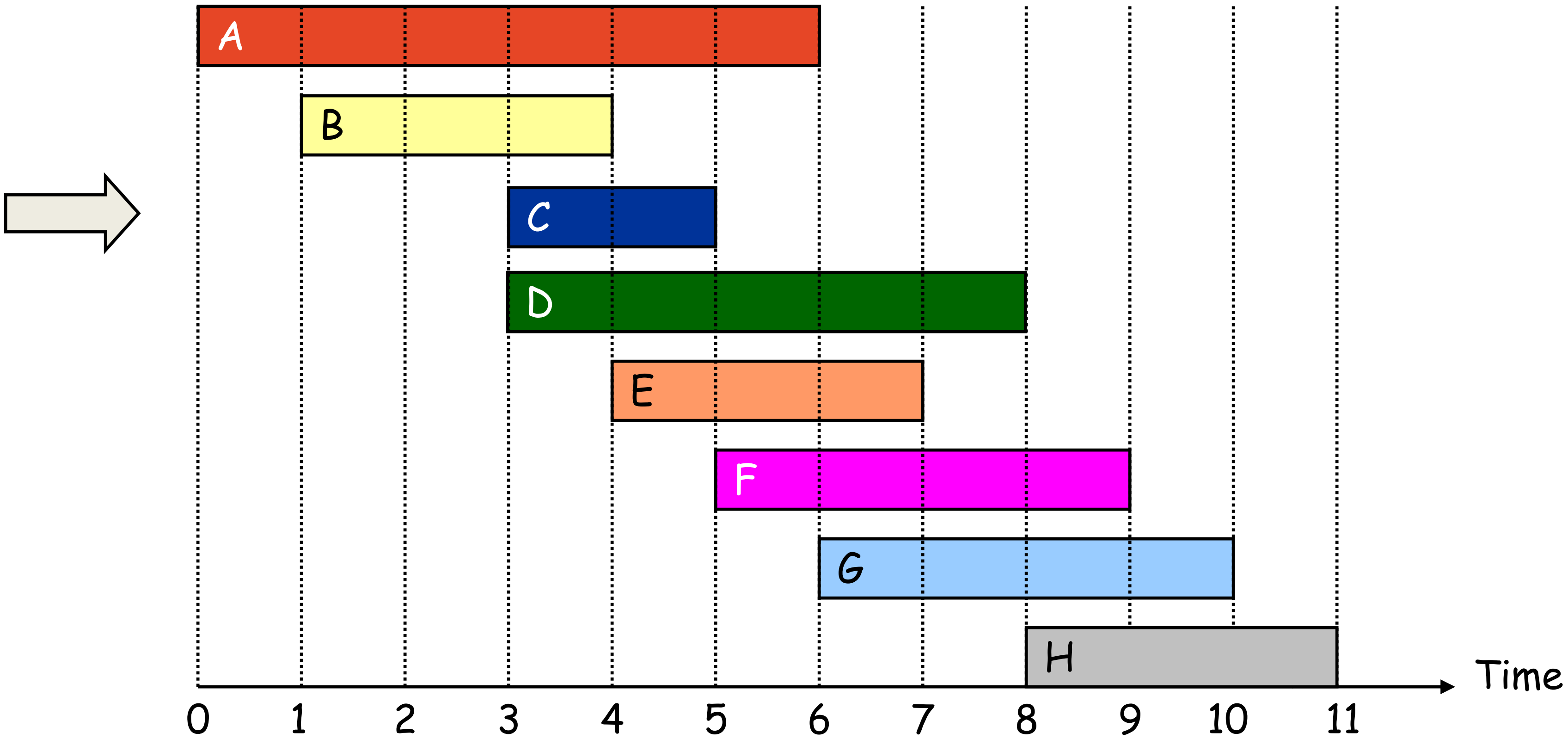


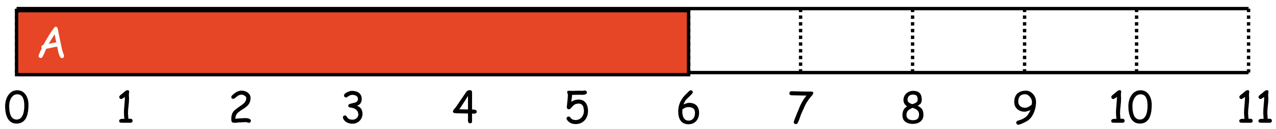
Interval Scheduling - [Earliest start time]



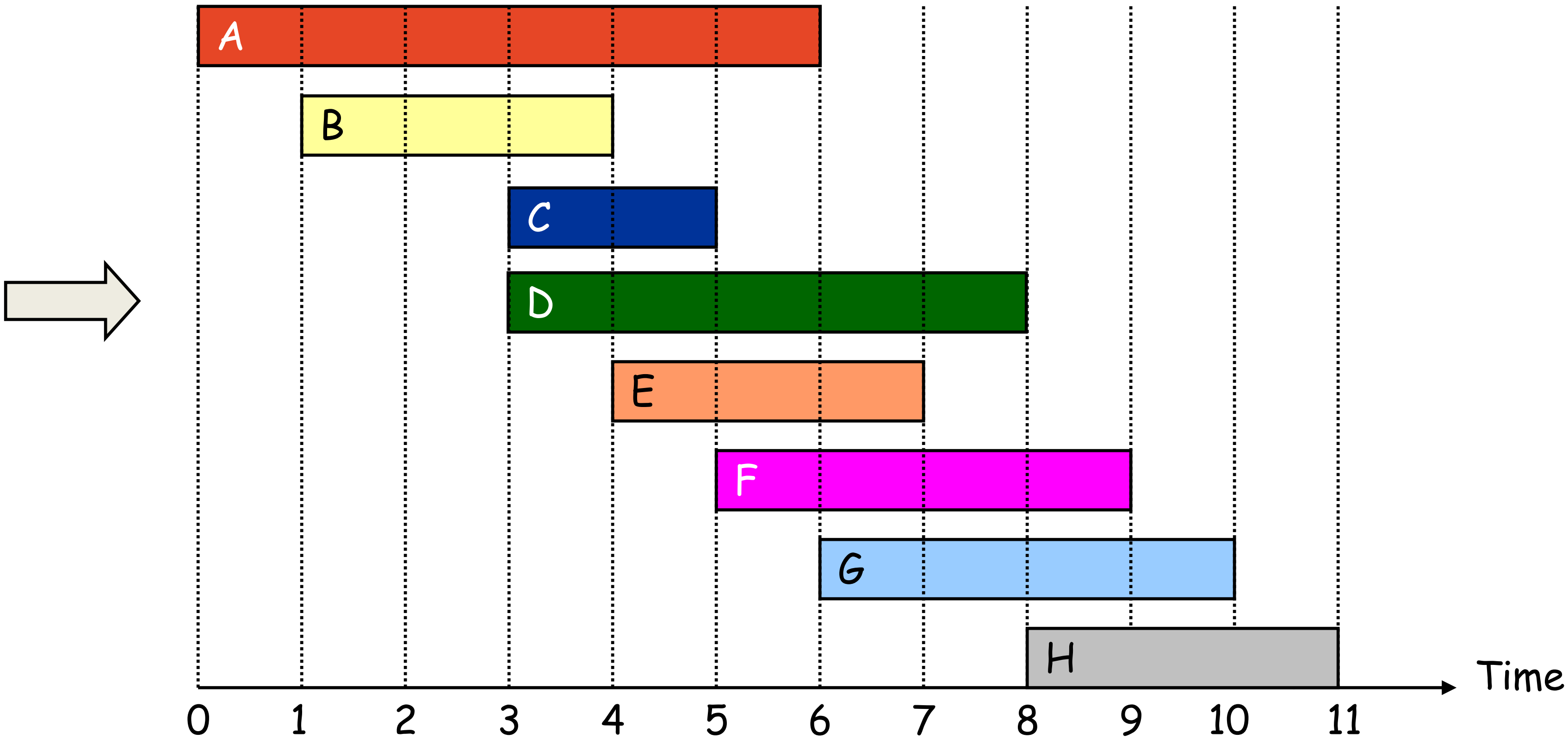


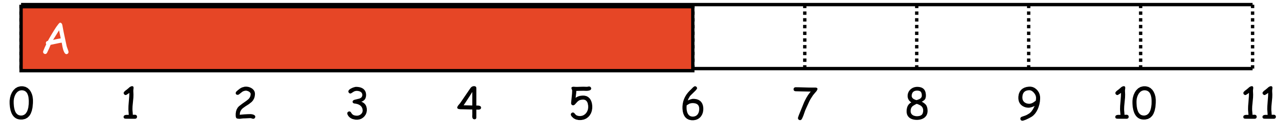
Interval Scheduling - [Earliest start time]



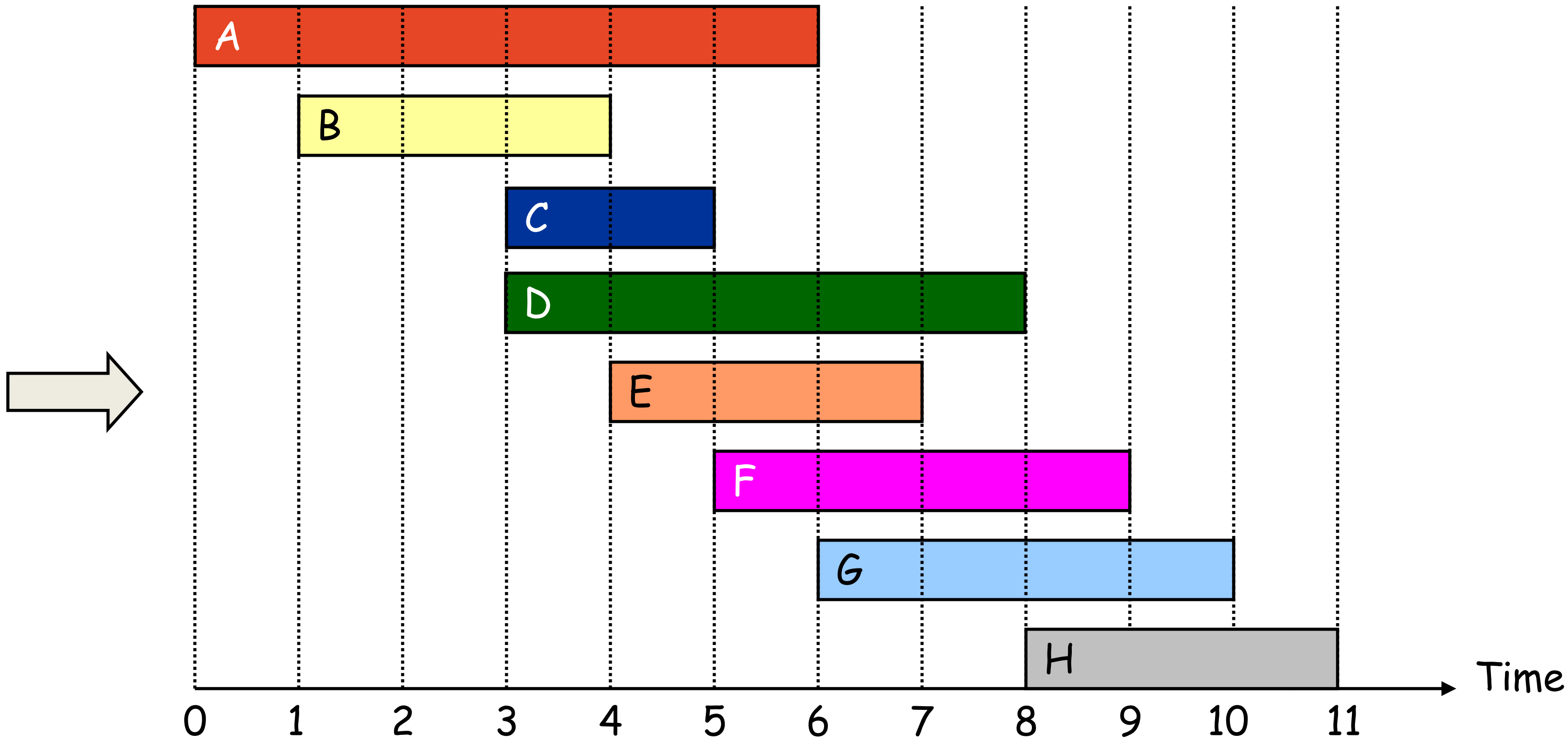


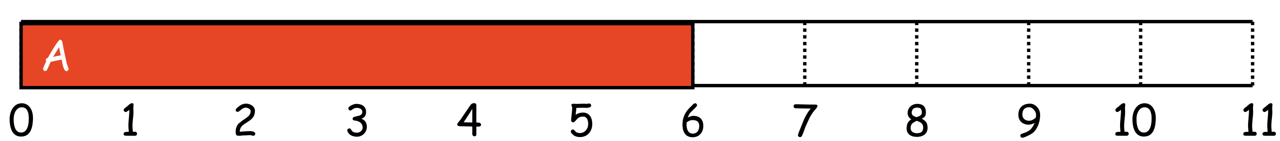
Interval Scheduling - [Earliest start time]



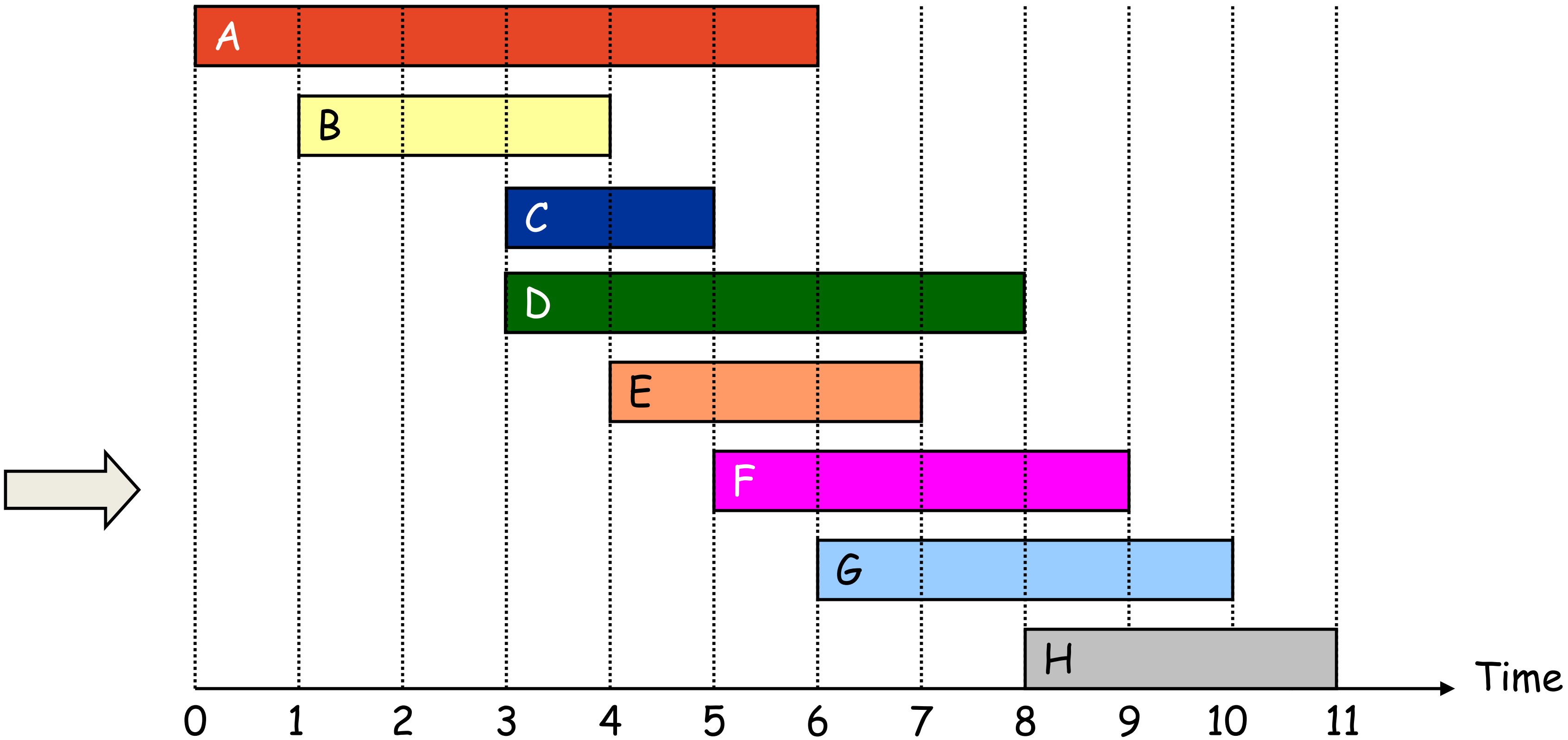


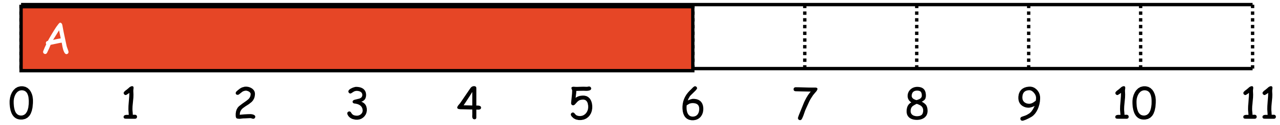
Interval Scheduling - [Earliest start time]



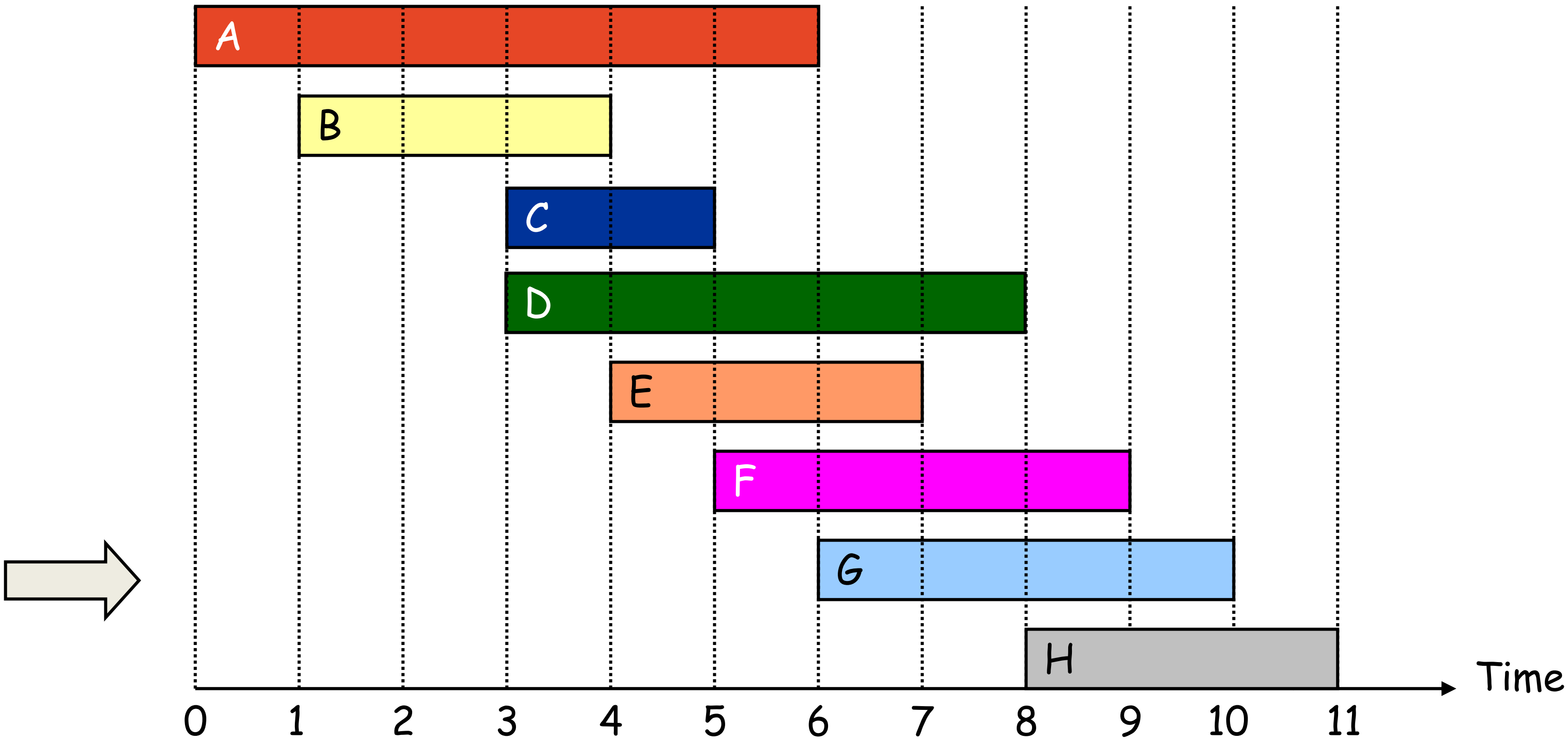


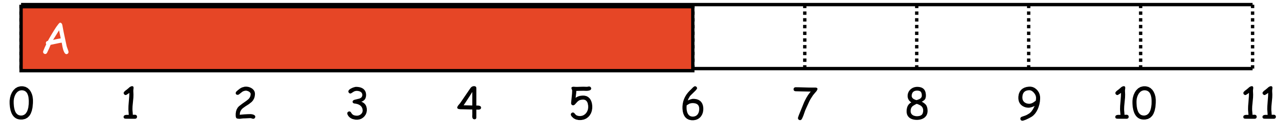
Interval Scheduling - [Earliest start time]



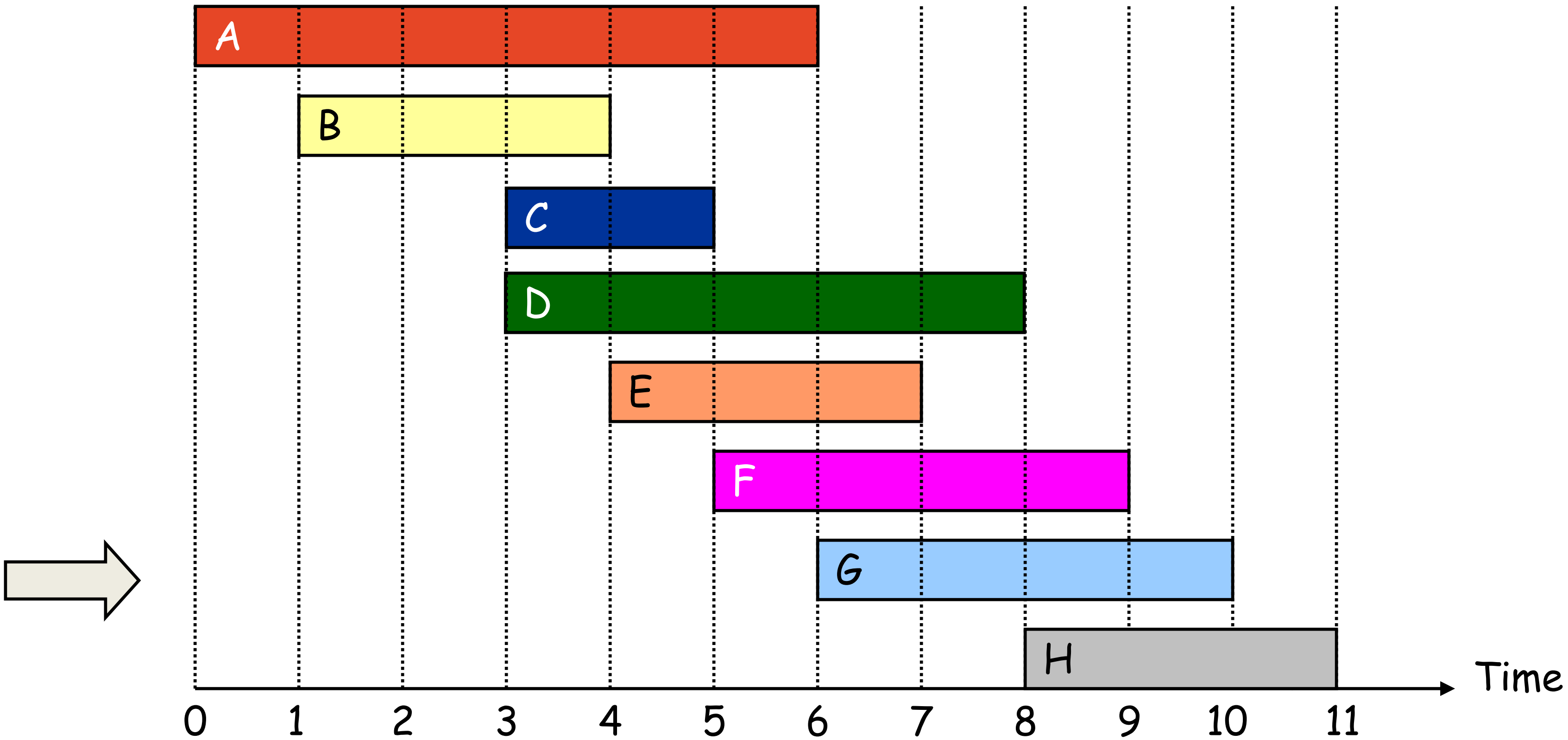


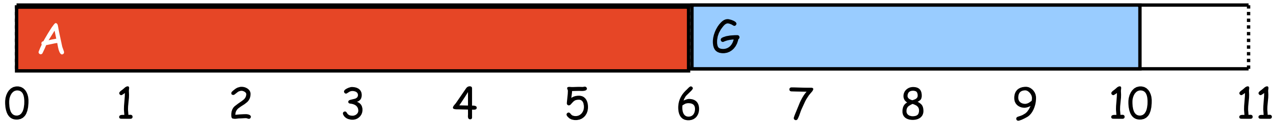
Interval Scheduling - [Earliest start time]



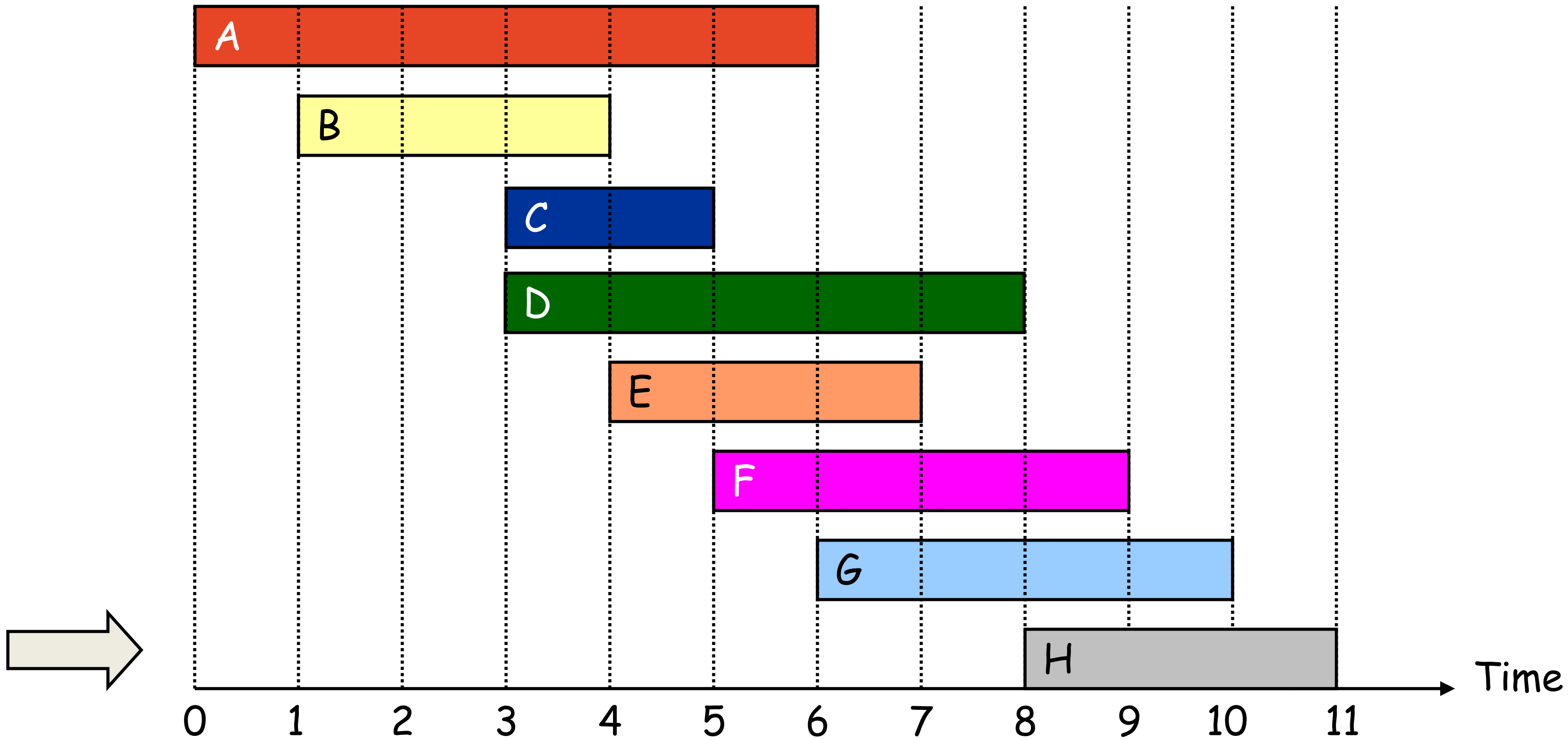


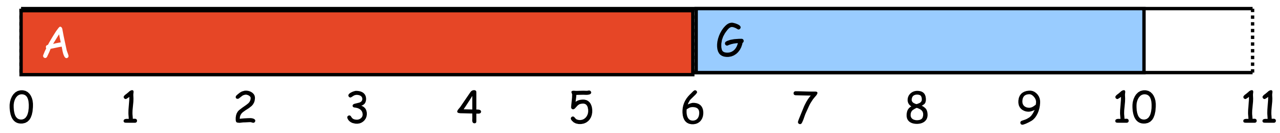
Interval Scheduling - [Earliest start time]





Interval Scheduling - [Earliest start time]





Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it is compatible with the ones already taken.

breaks [Earliest start time]

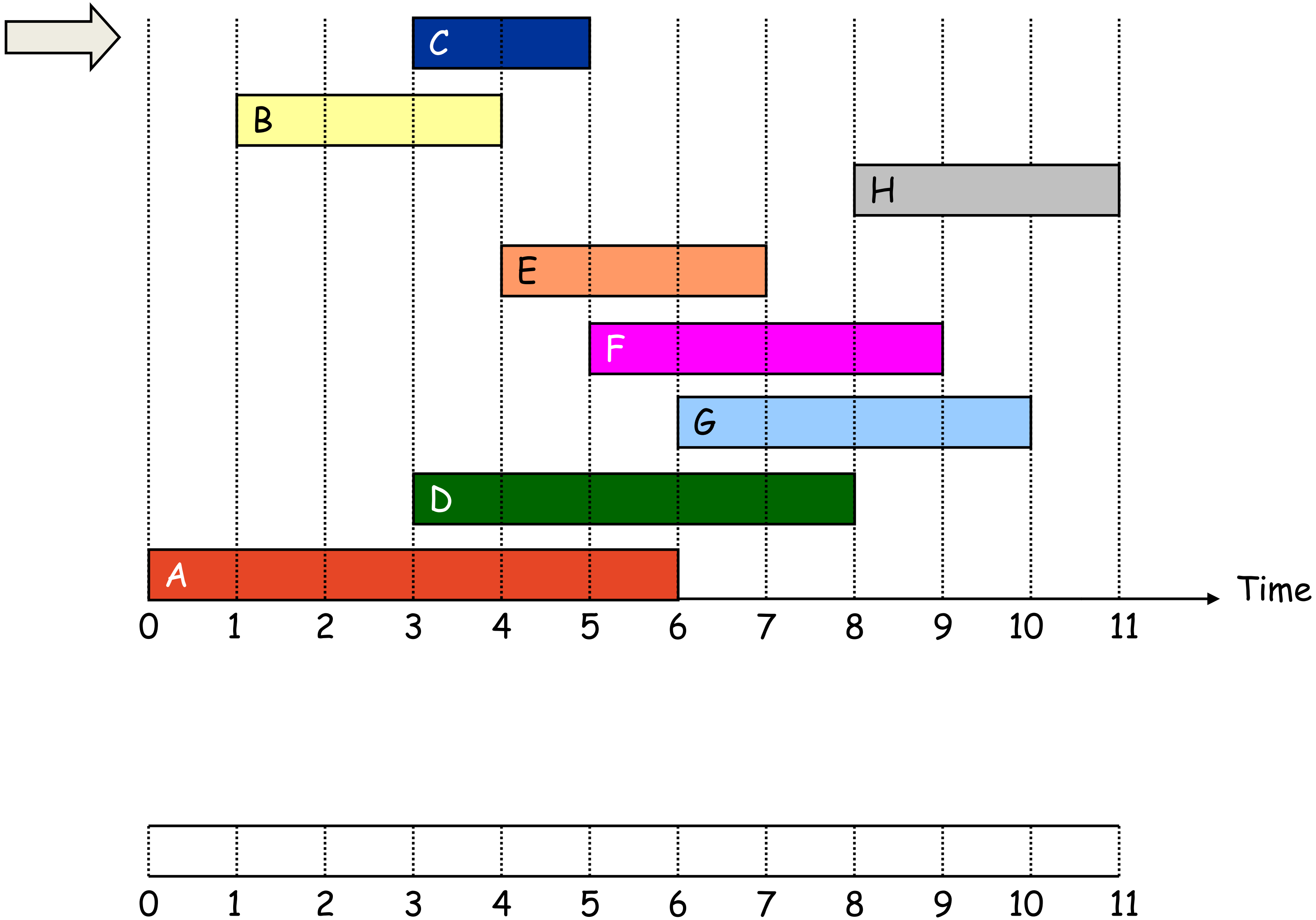
Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it is compatible with the ones already taken.

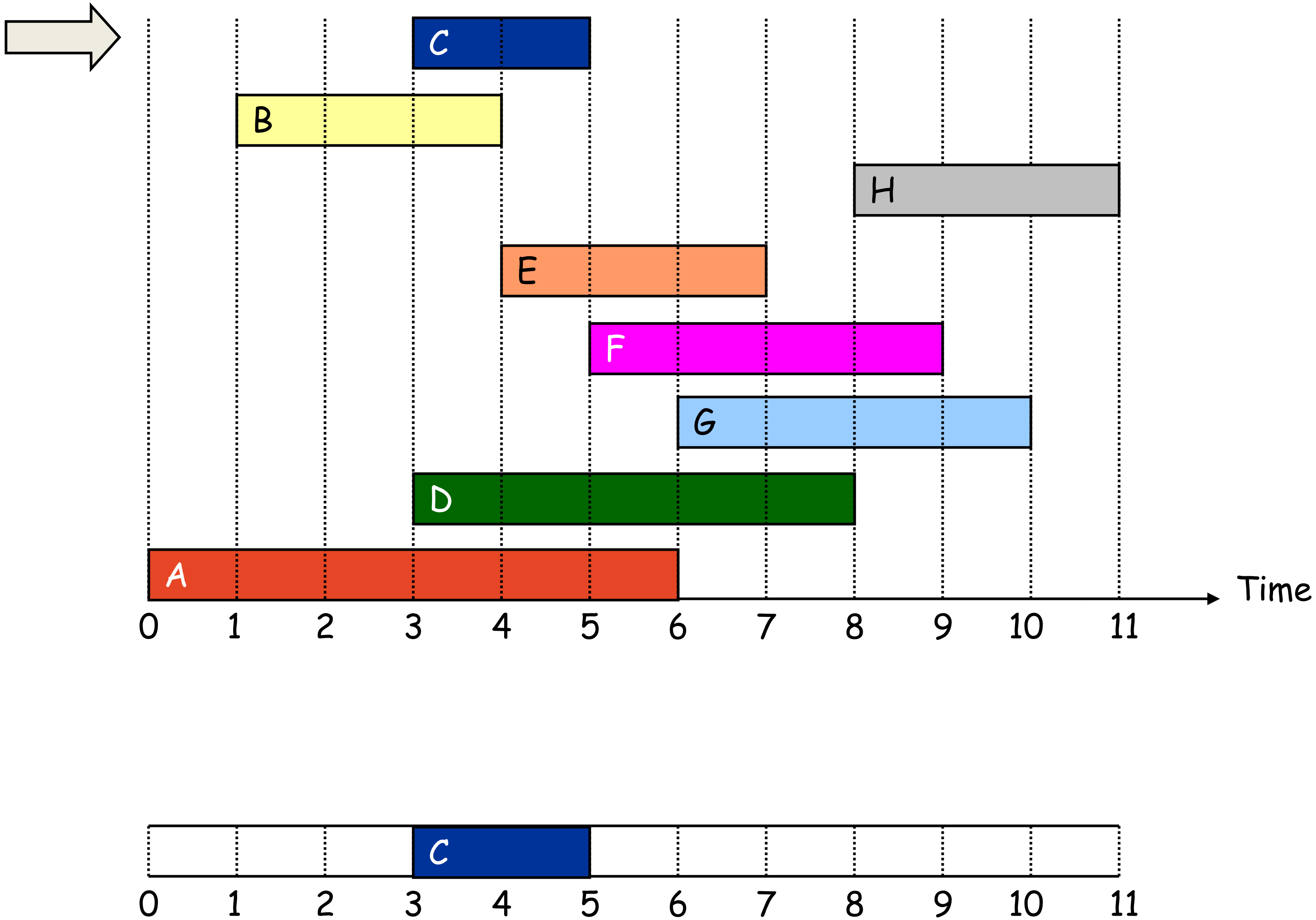
– [Earliest start time] Consider jobs in ascending order of start time s_j.

– [Shortest interval] Consider jobs in ascending order of interval length $f_j - s_j$.

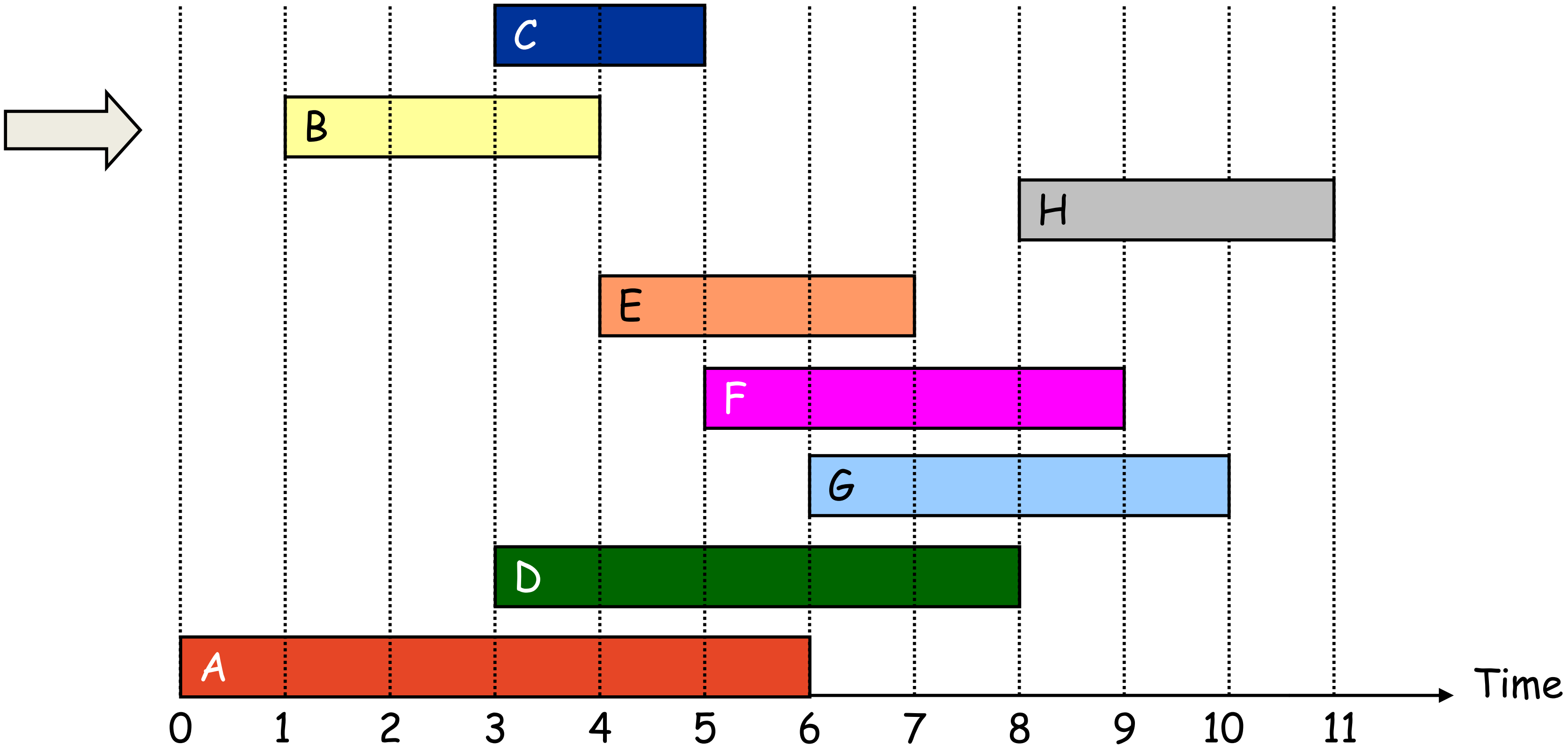
Interval Scheduling - [Shortest interval]

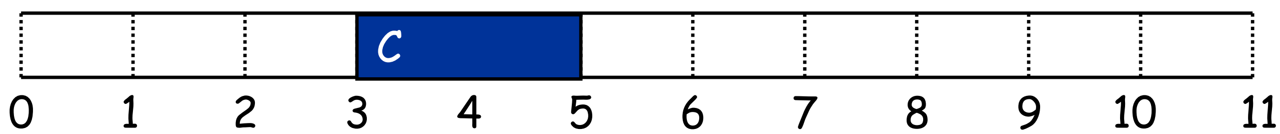


Interval Scheduling - [Shortest interval]

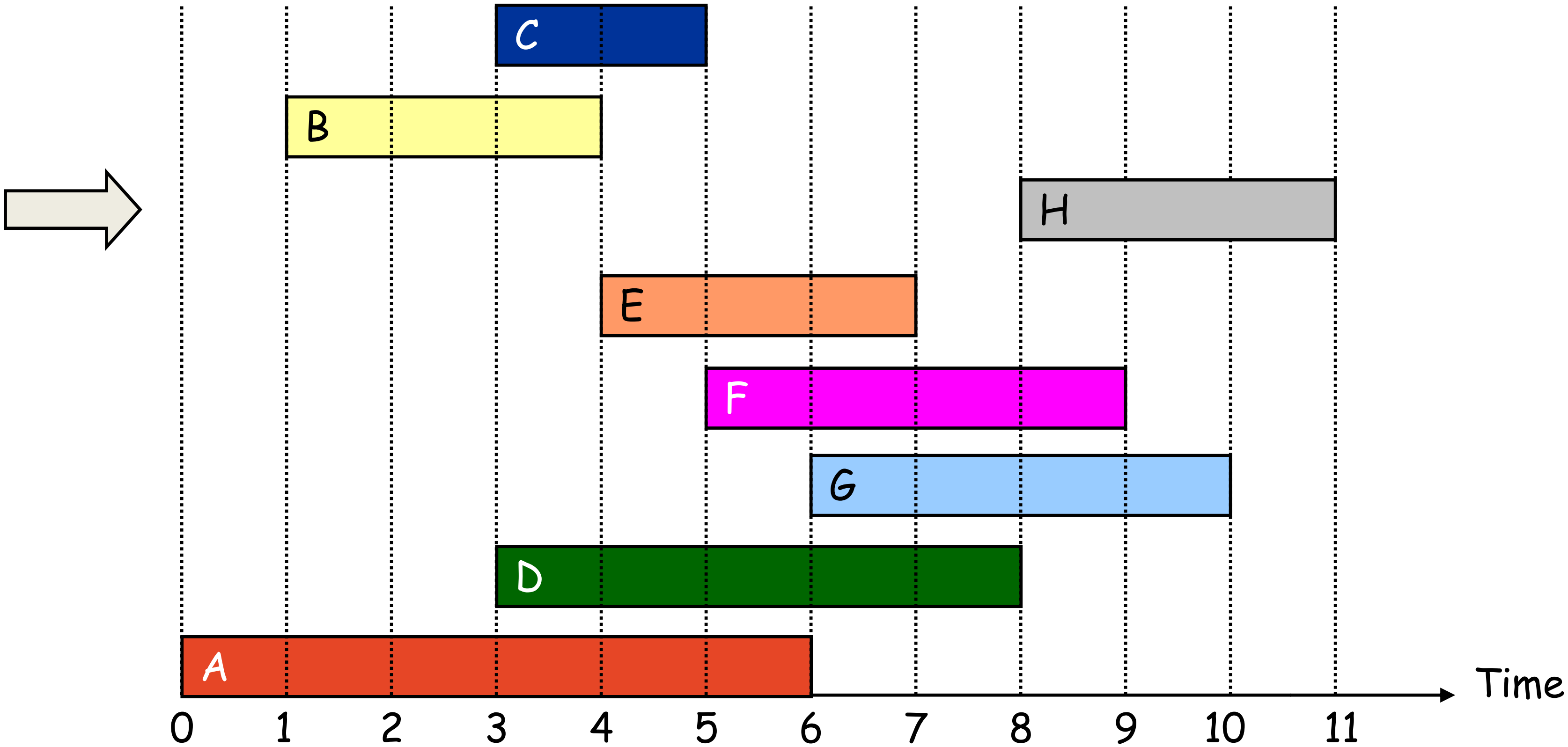


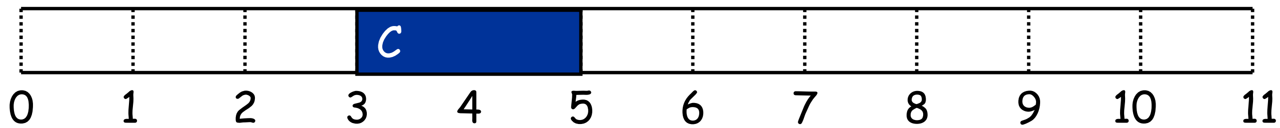
Interval Scheduling - [Shortest interval]



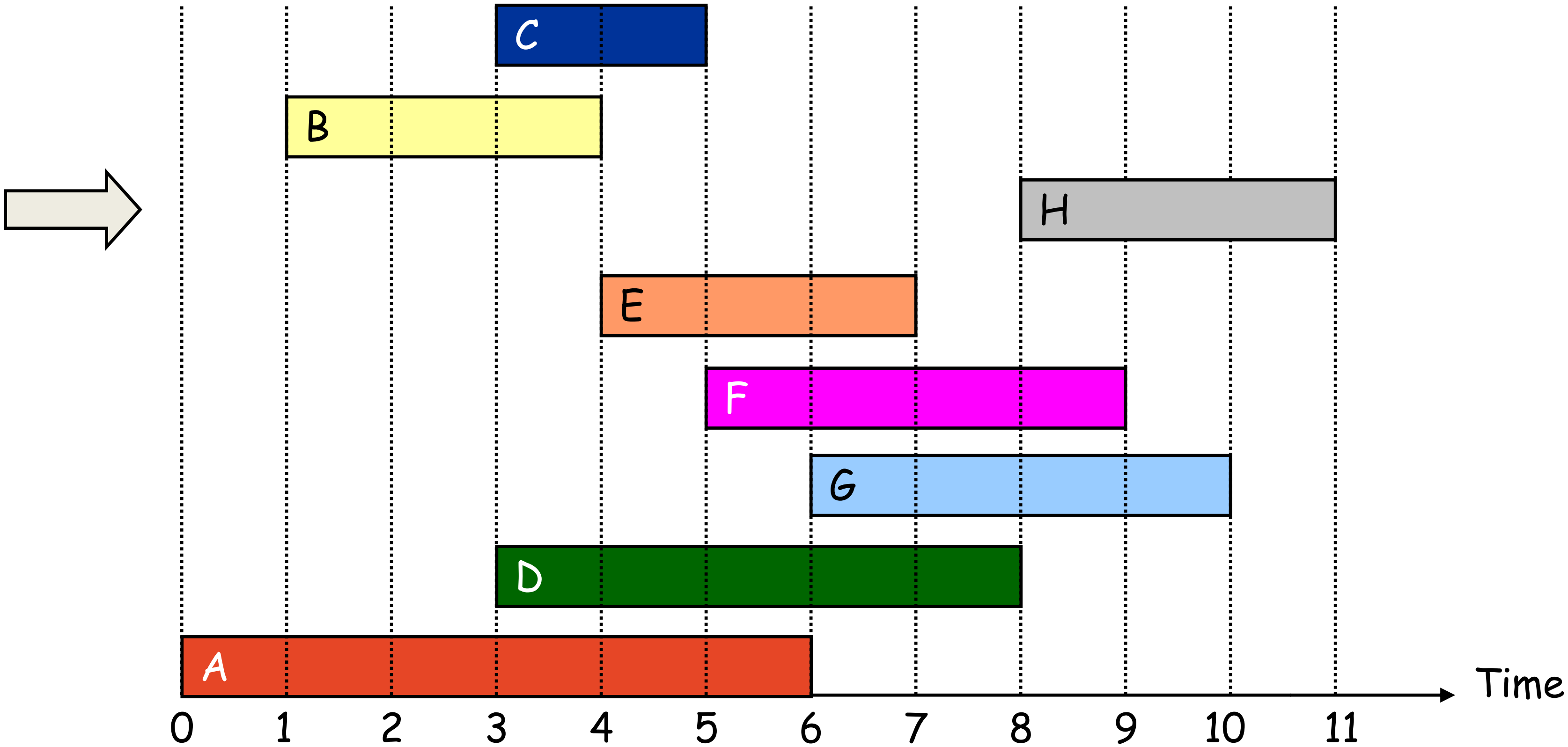


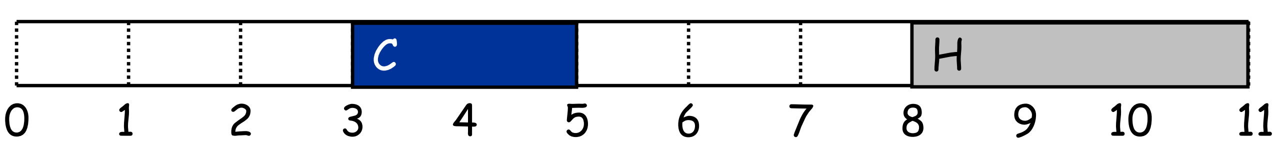
Interval Scheduling - [Shortest interval]



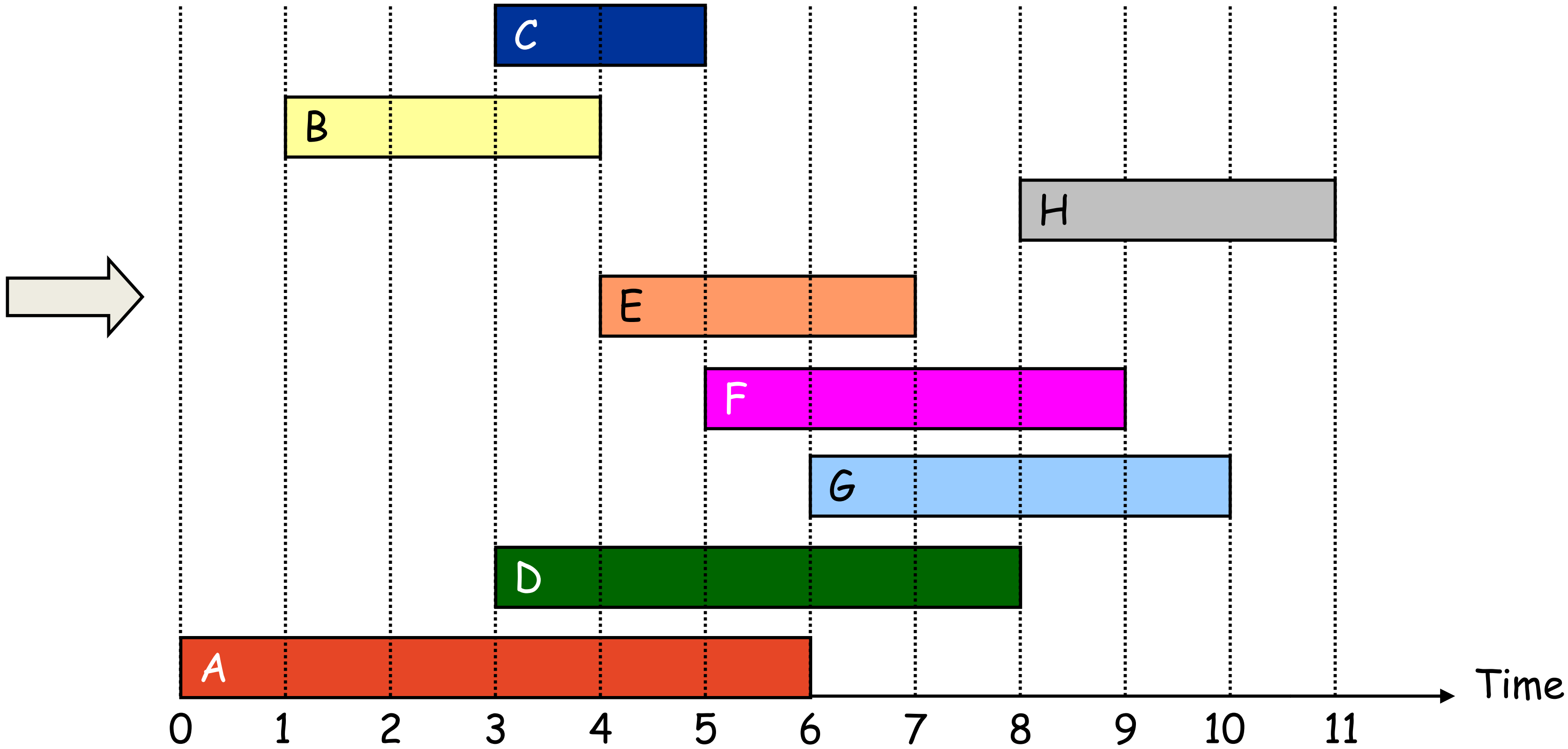


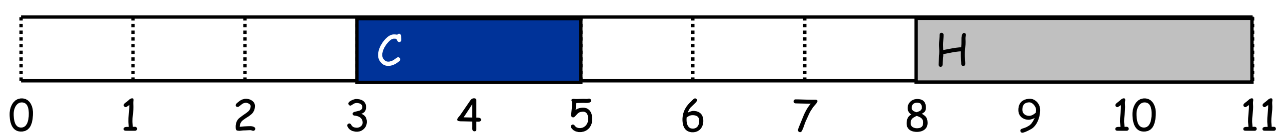
Interval Scheduling - [Shortest interval]



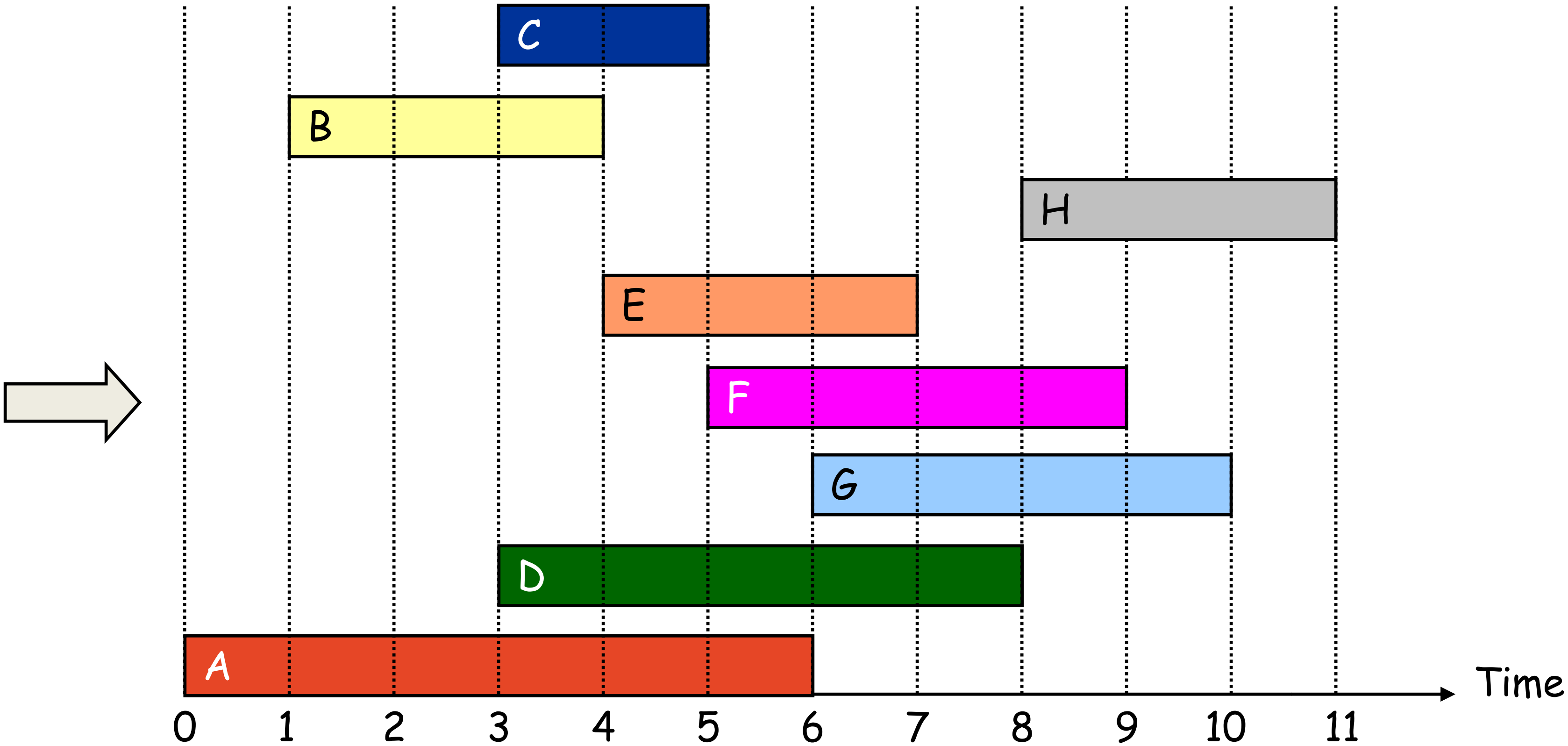


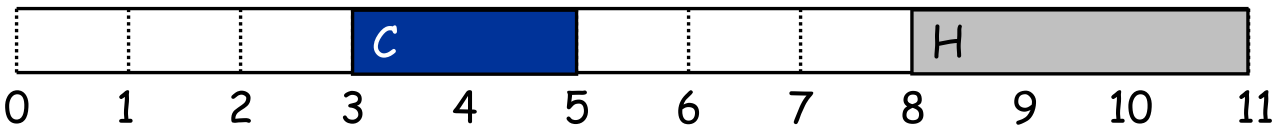
Interval Scheduling - [Shortest interval]



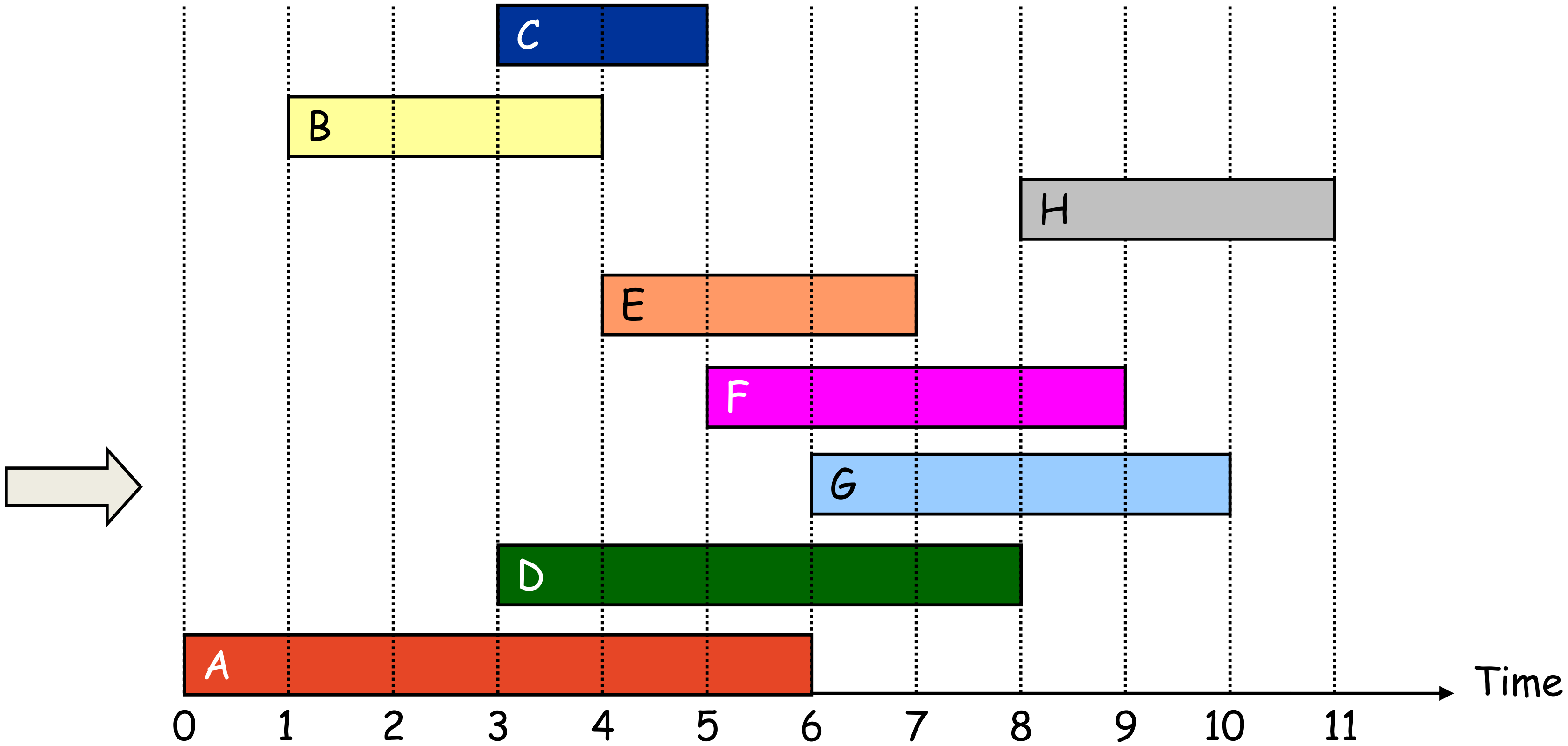


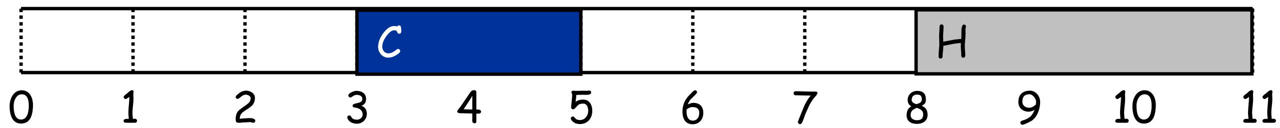
Interval Scheduling - [Shortest interval]



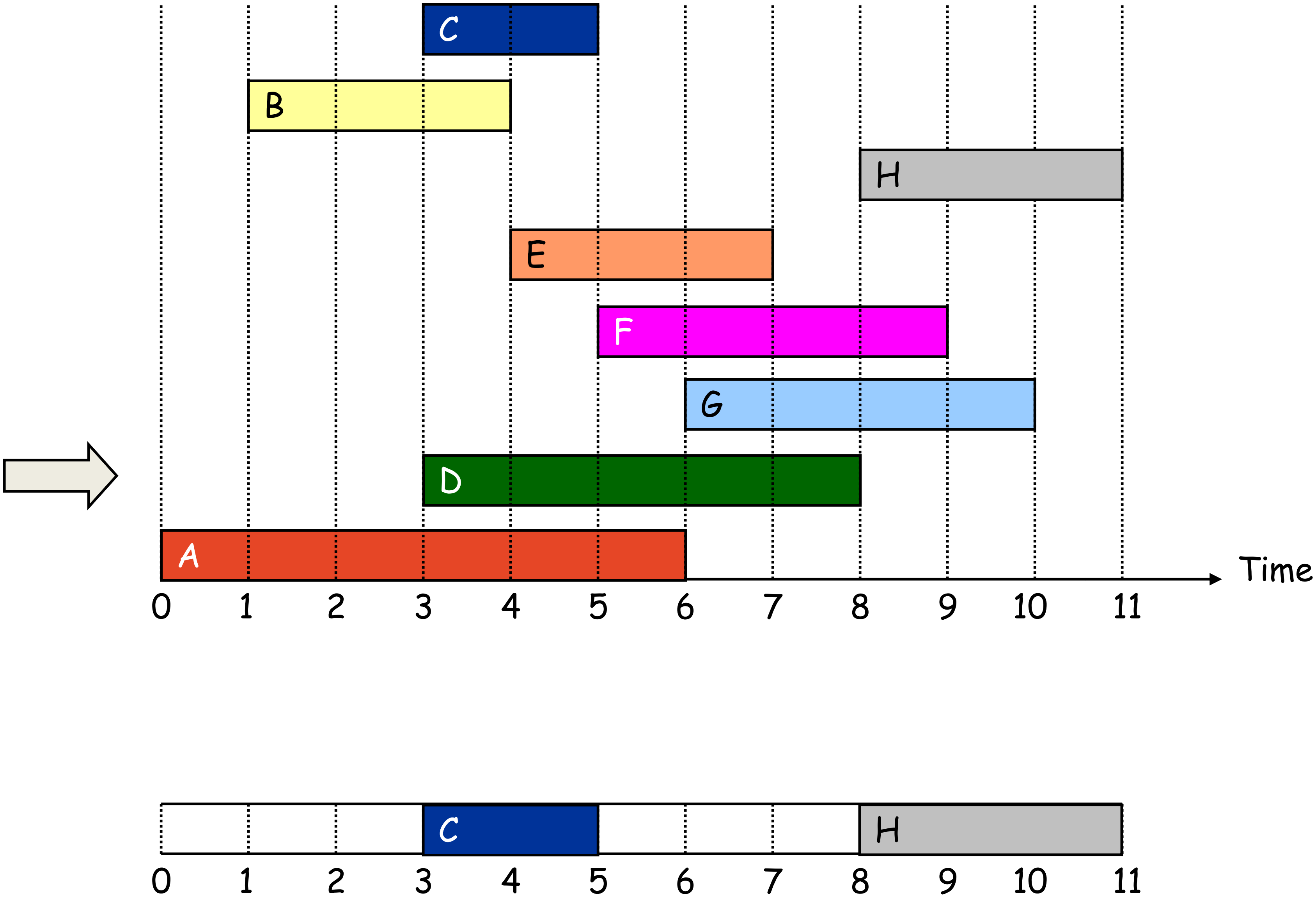


Interval Scheduling - [Shortest interval]

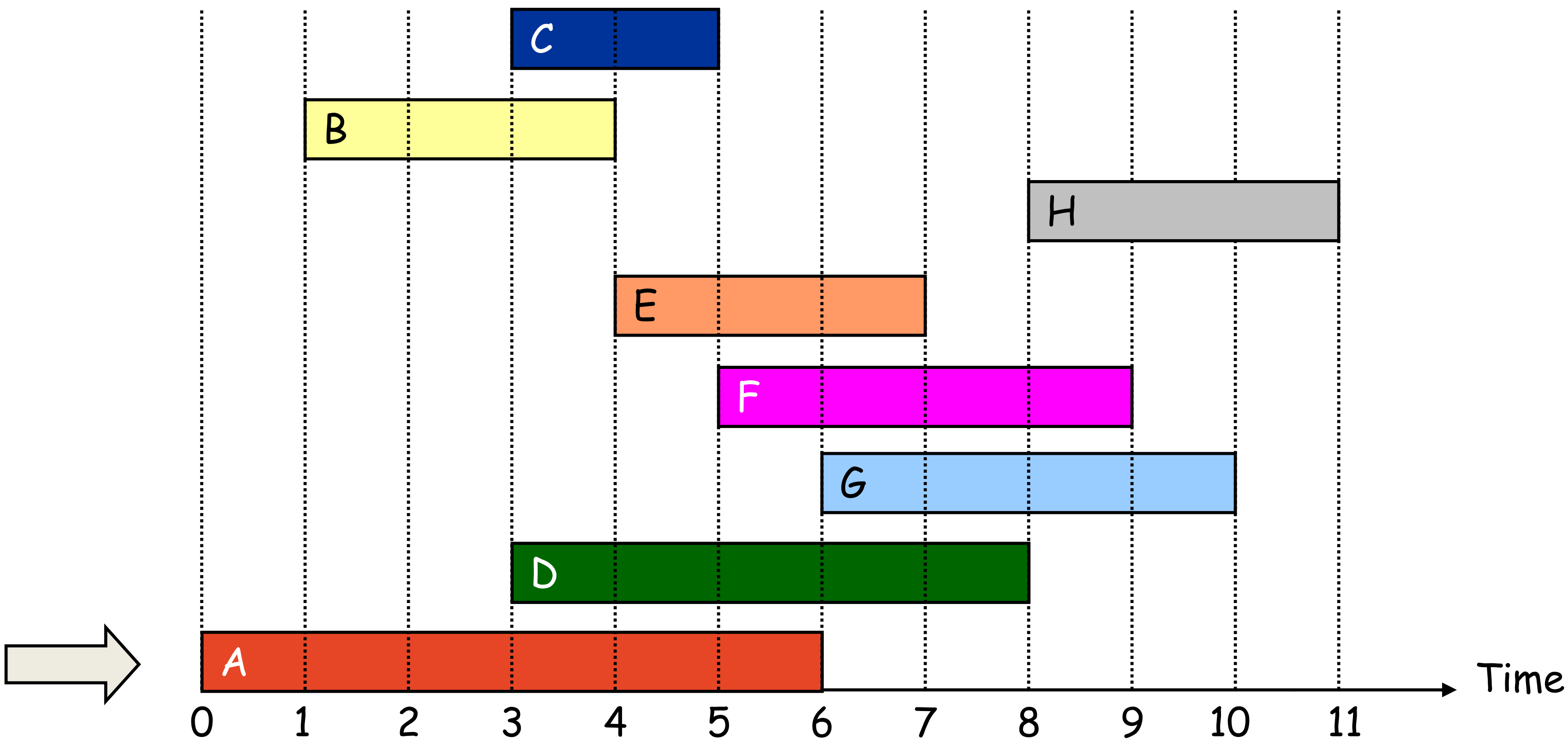


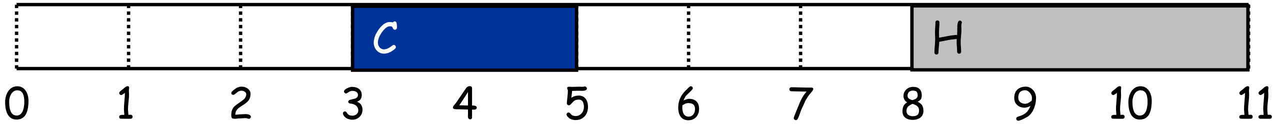


Interval Scheduling - [Shortest interval]



Interval Scheduling - [Shortest interval]





Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it is compatible with the ones already taken.

breaks [Earliest start time]

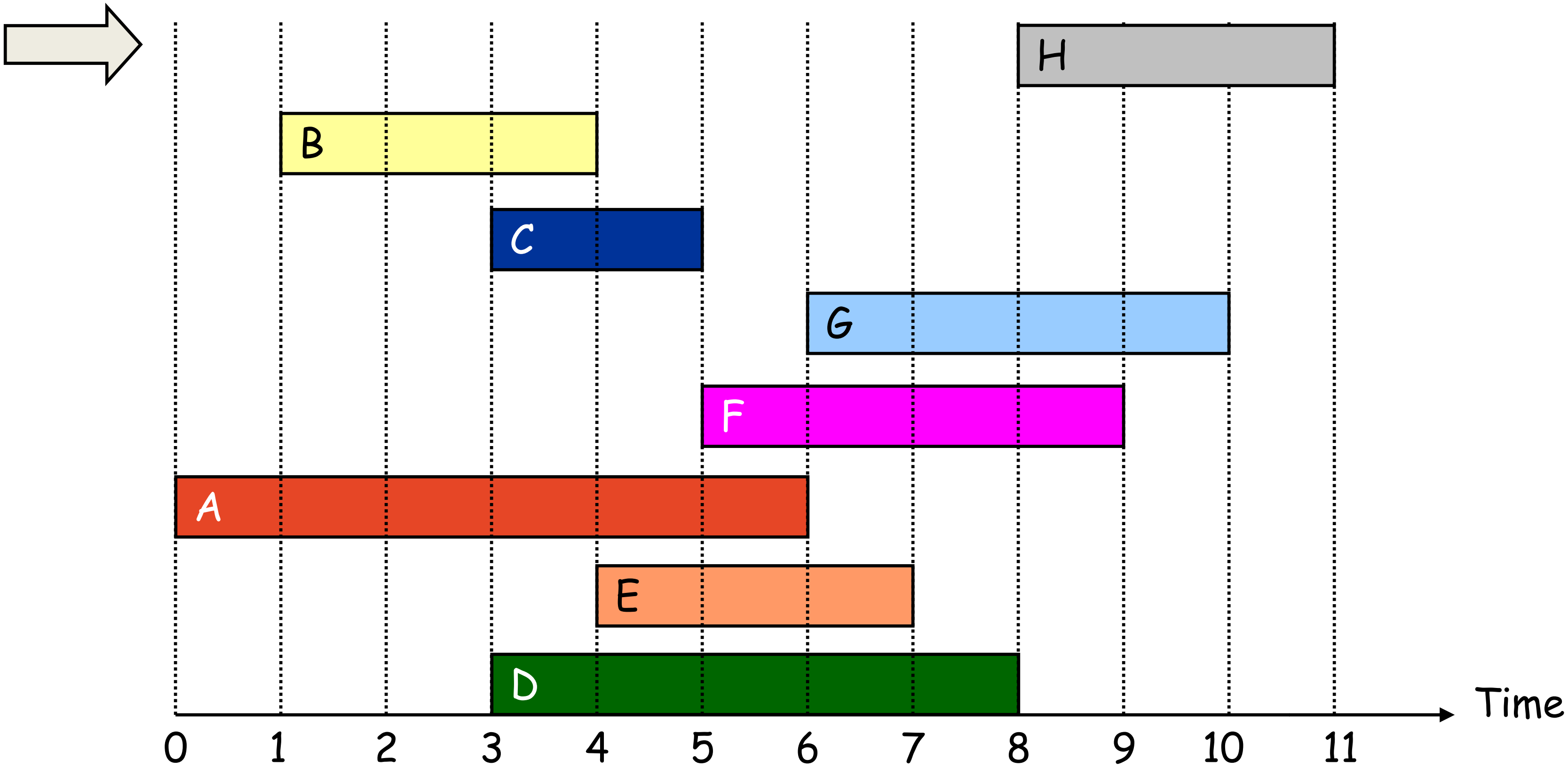
breaks [Shortest interval]

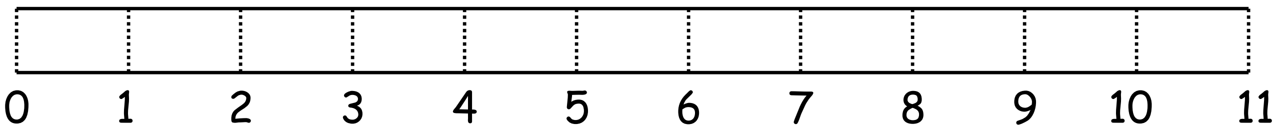
Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it is compatible with the ones already taken.

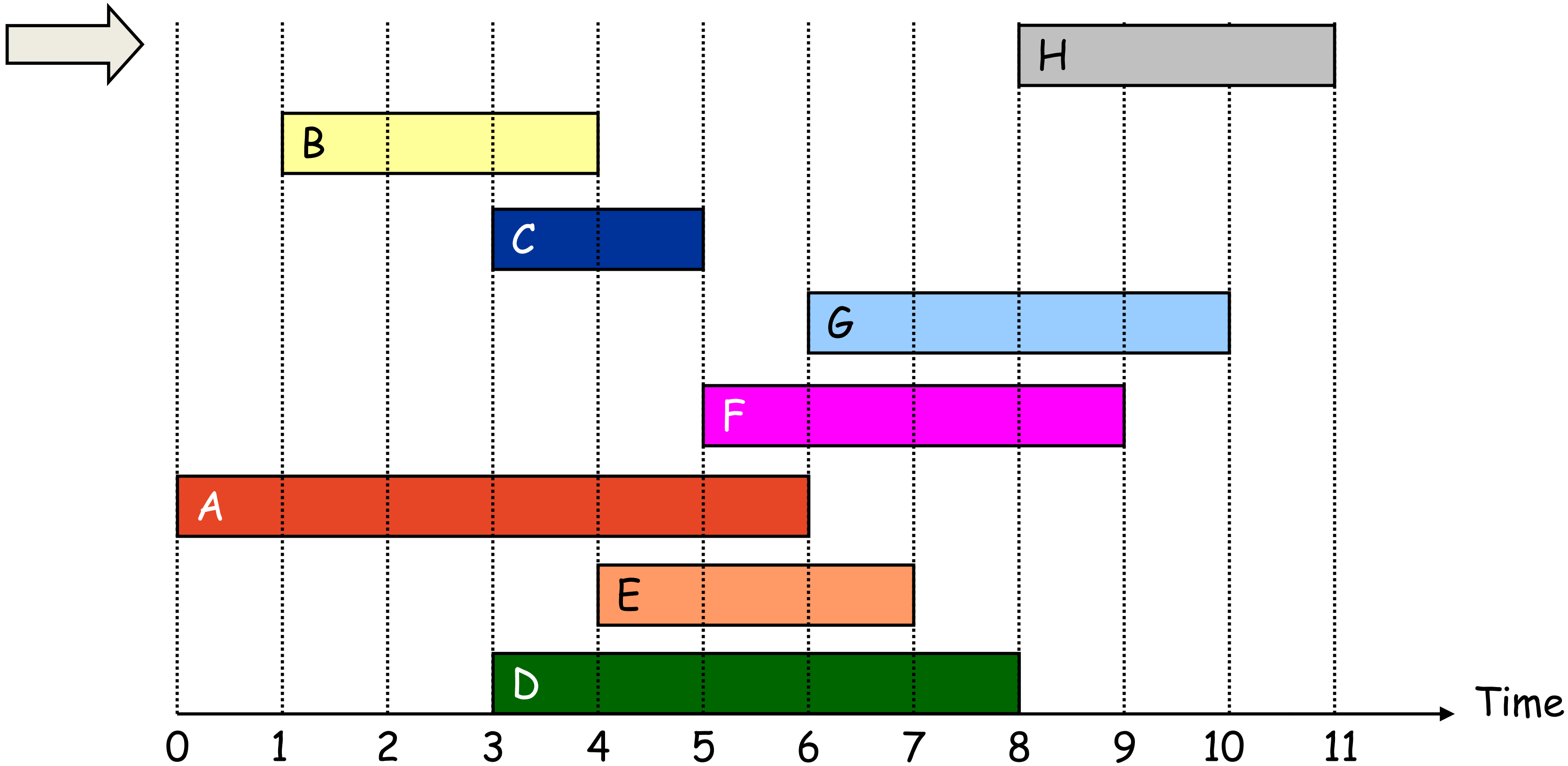
- [Earliest start time] Consider jobs in ascending order of start time s_i.
- [Shortest interval] Consider jobs in ascending order of interval length $f_i - s_i$.
- [Fewest conflicts] For each job, count the number of conflicting jobs c_i. Schedule in ascending order of conflicts c_i.

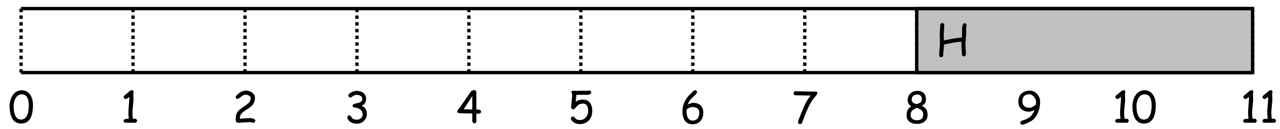
Interval Scheduling - [Fewest Conflicts]



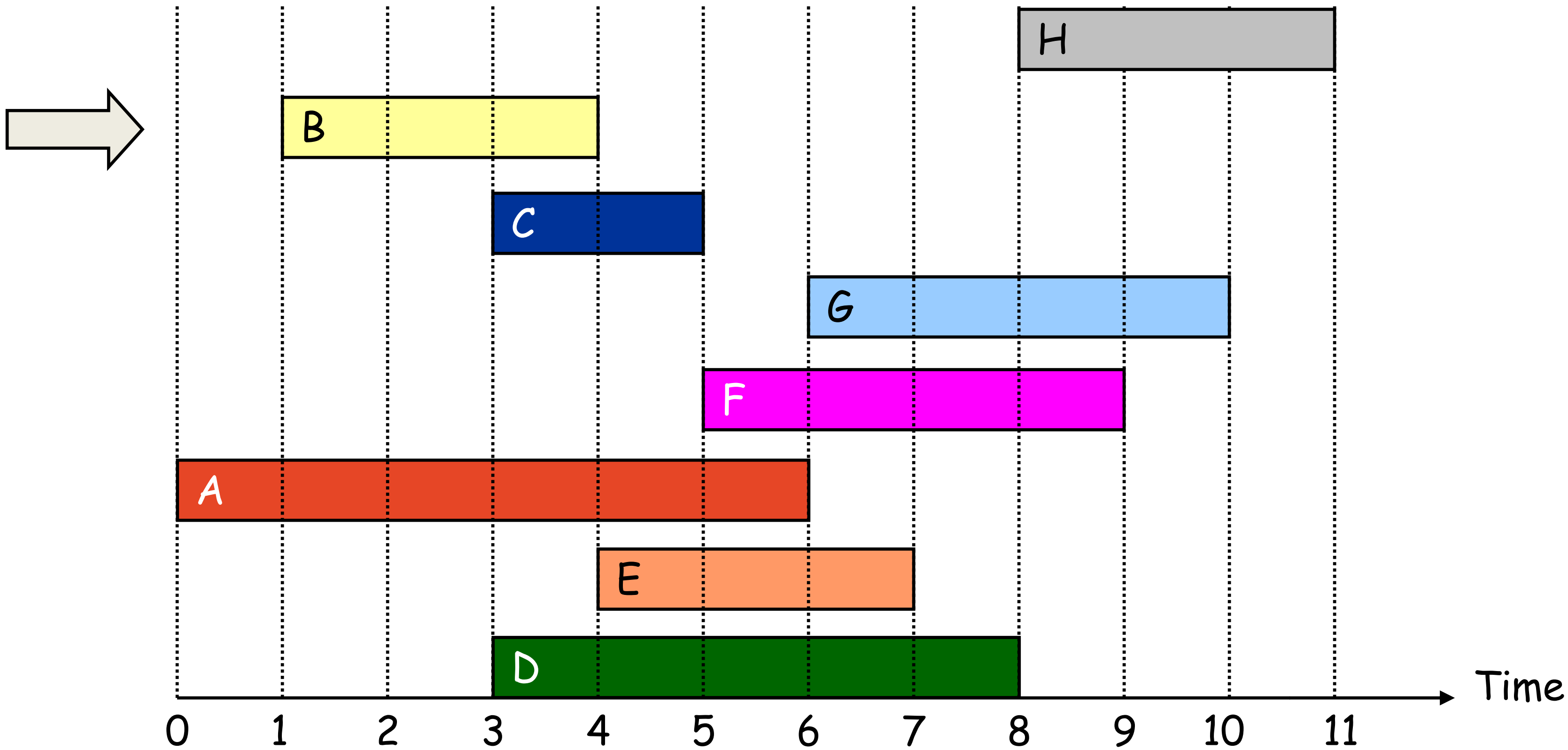


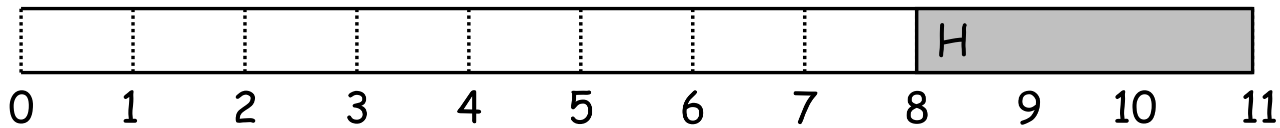
Interval Scheduling - [Fewest Conflicts]



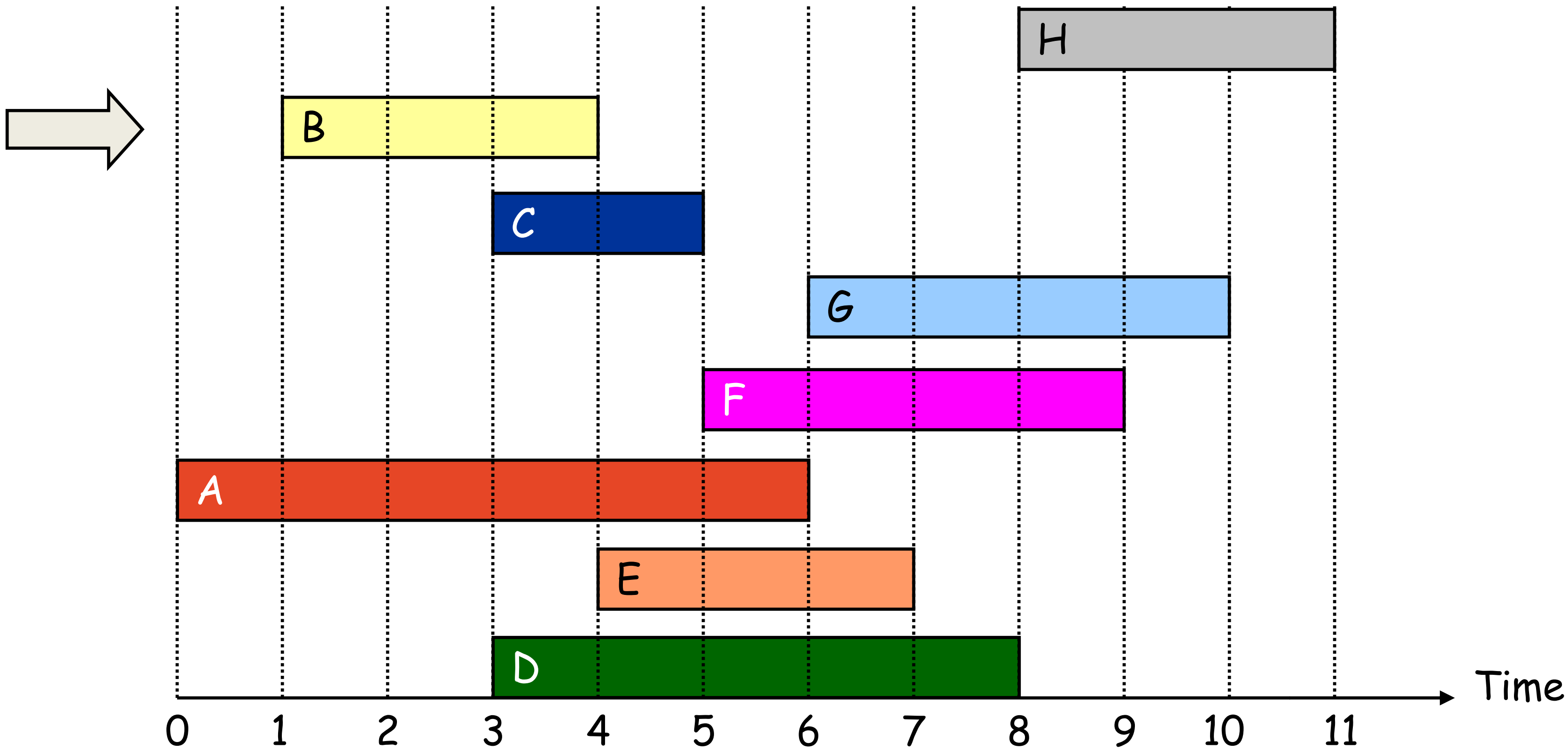


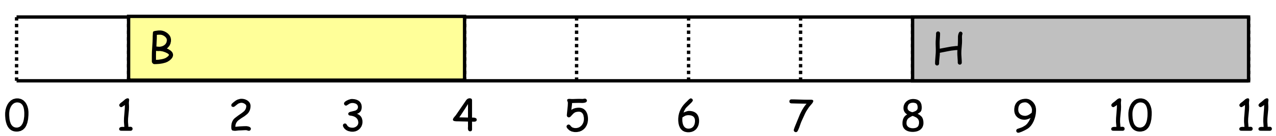
Interval Scheduling - [Fewest Conflicts]



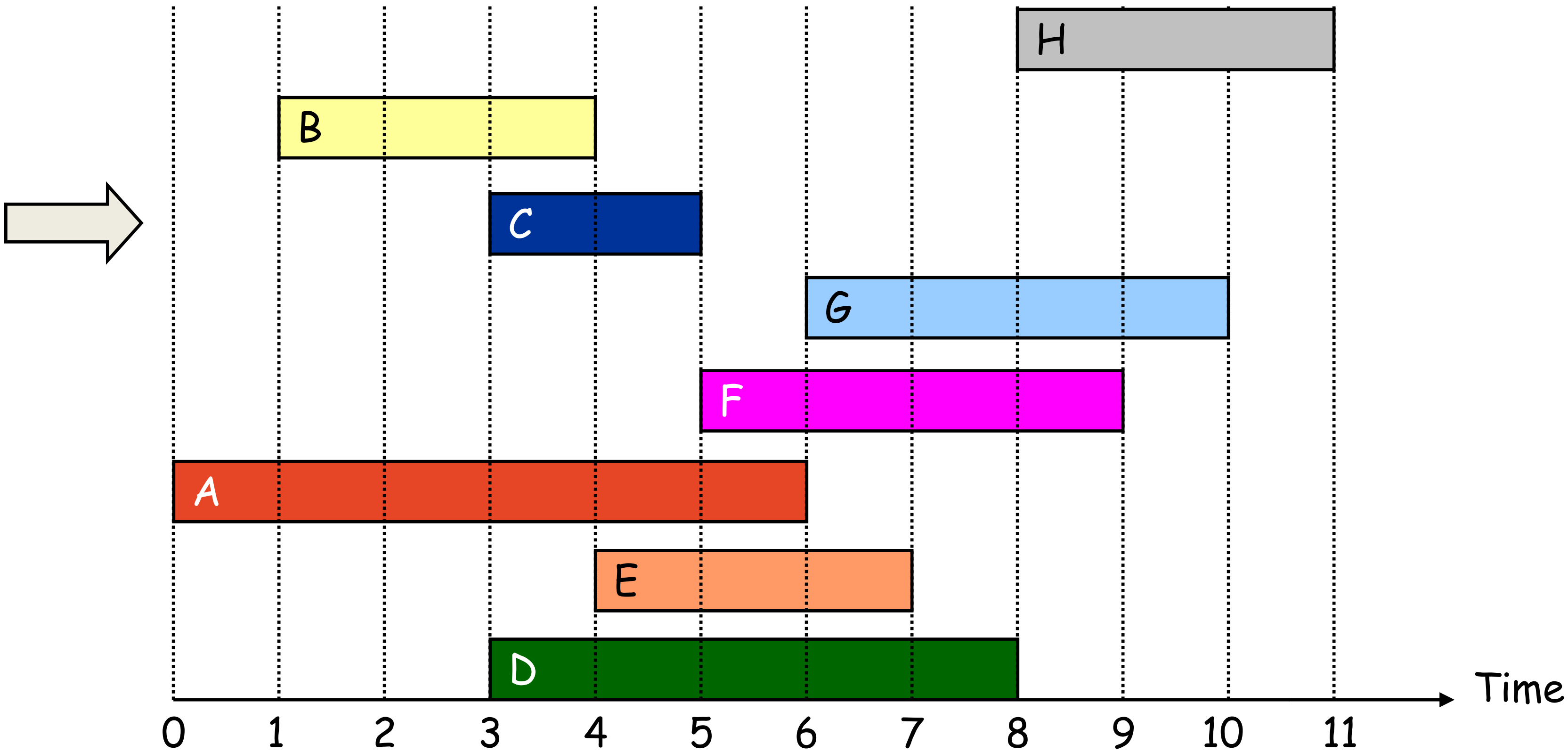


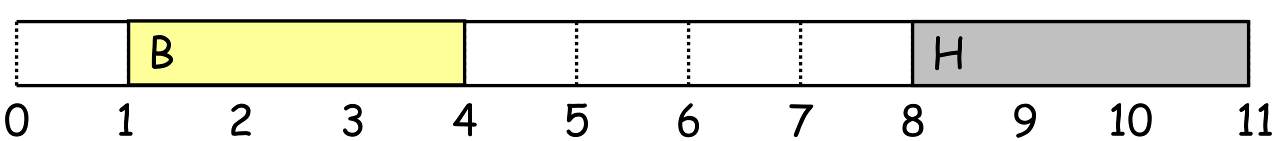
Interval Scheduling - [Fewest Conflicts]



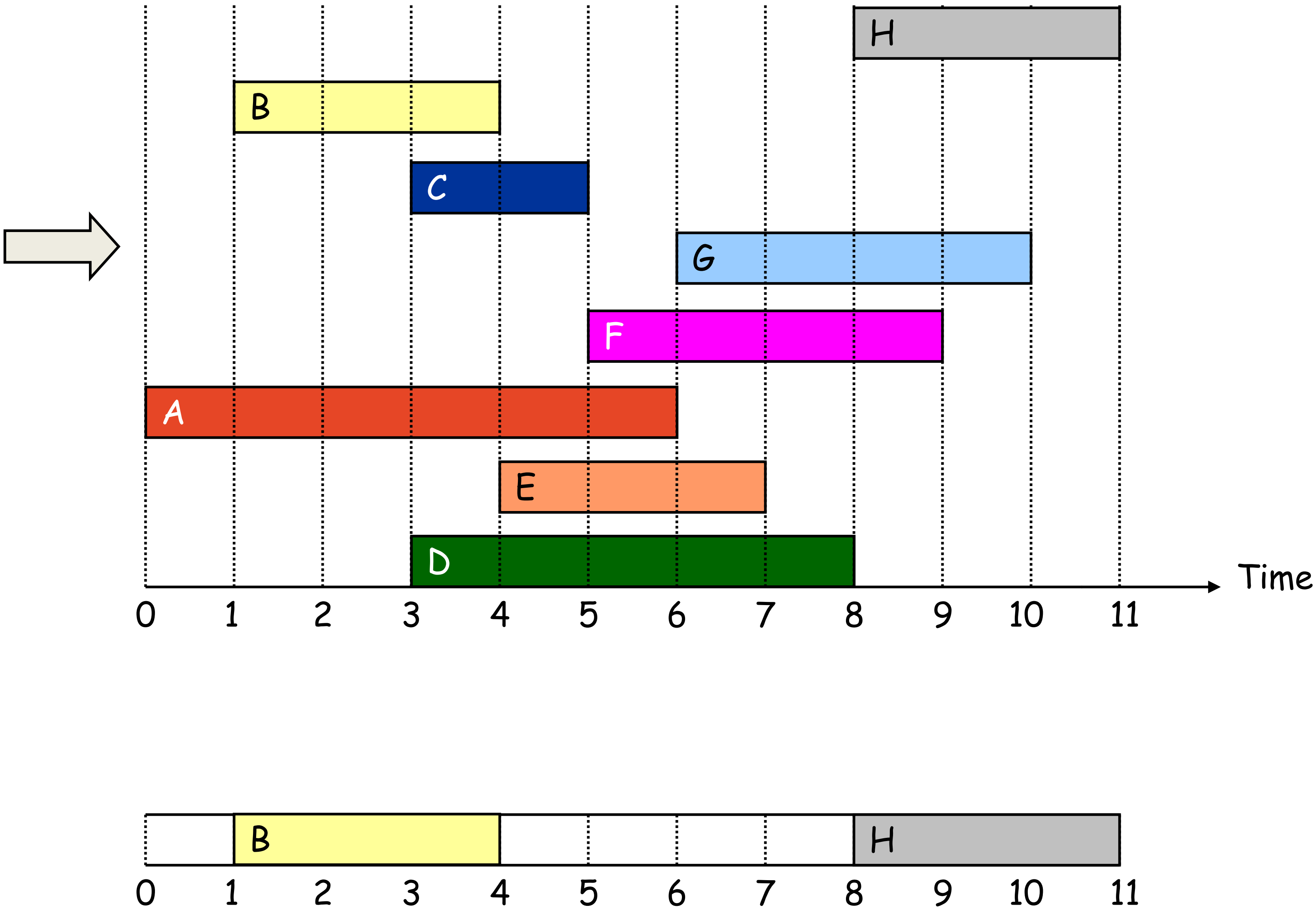


Interval Scheduling - [Fewest Conflicts]

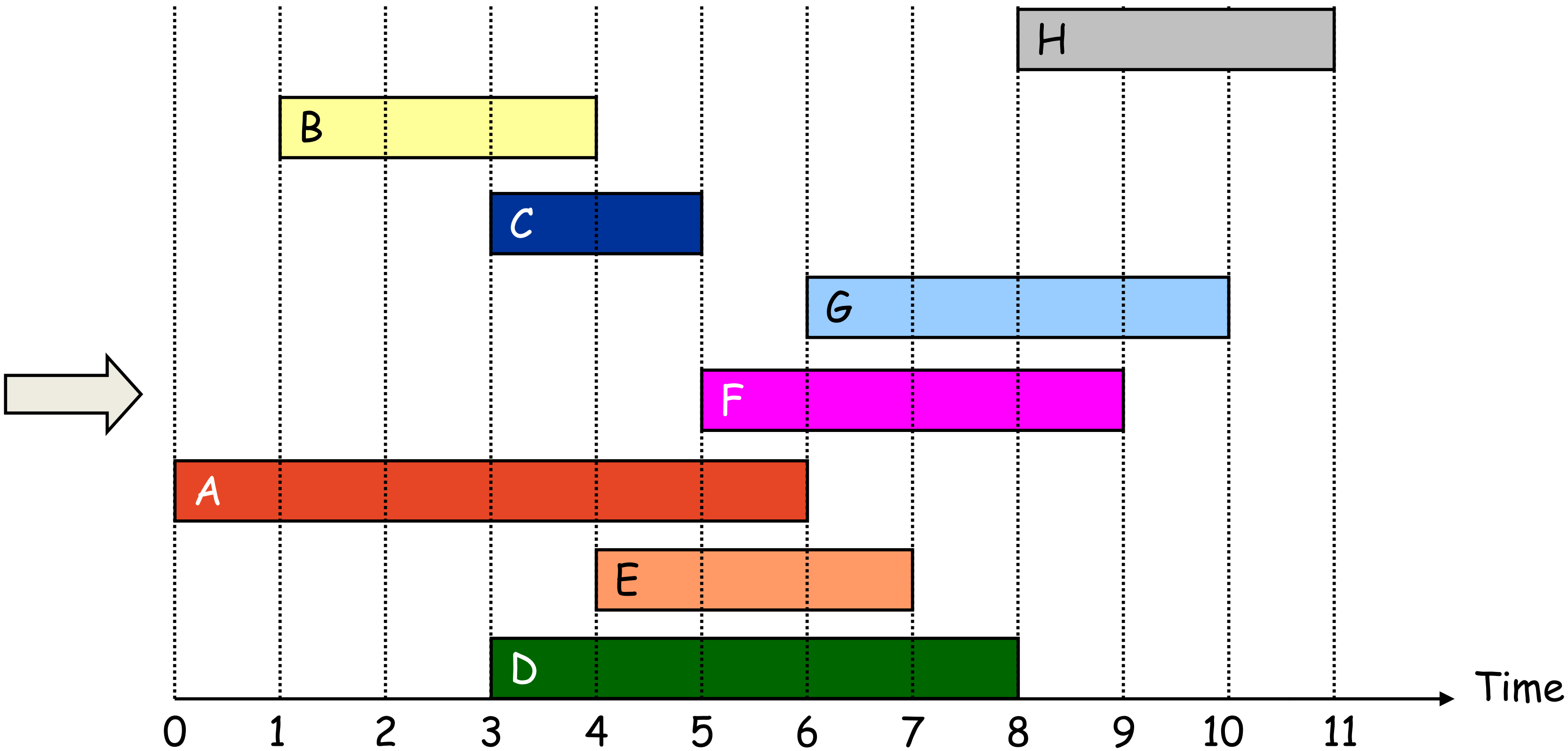


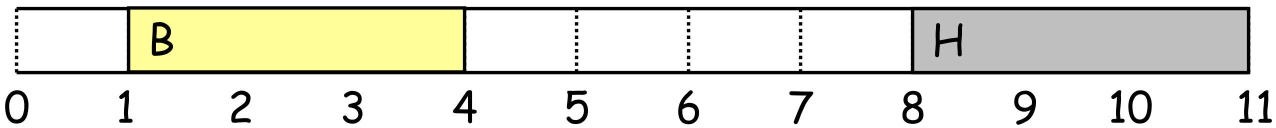


Interval Scheduling - [Fewest Conflicts]

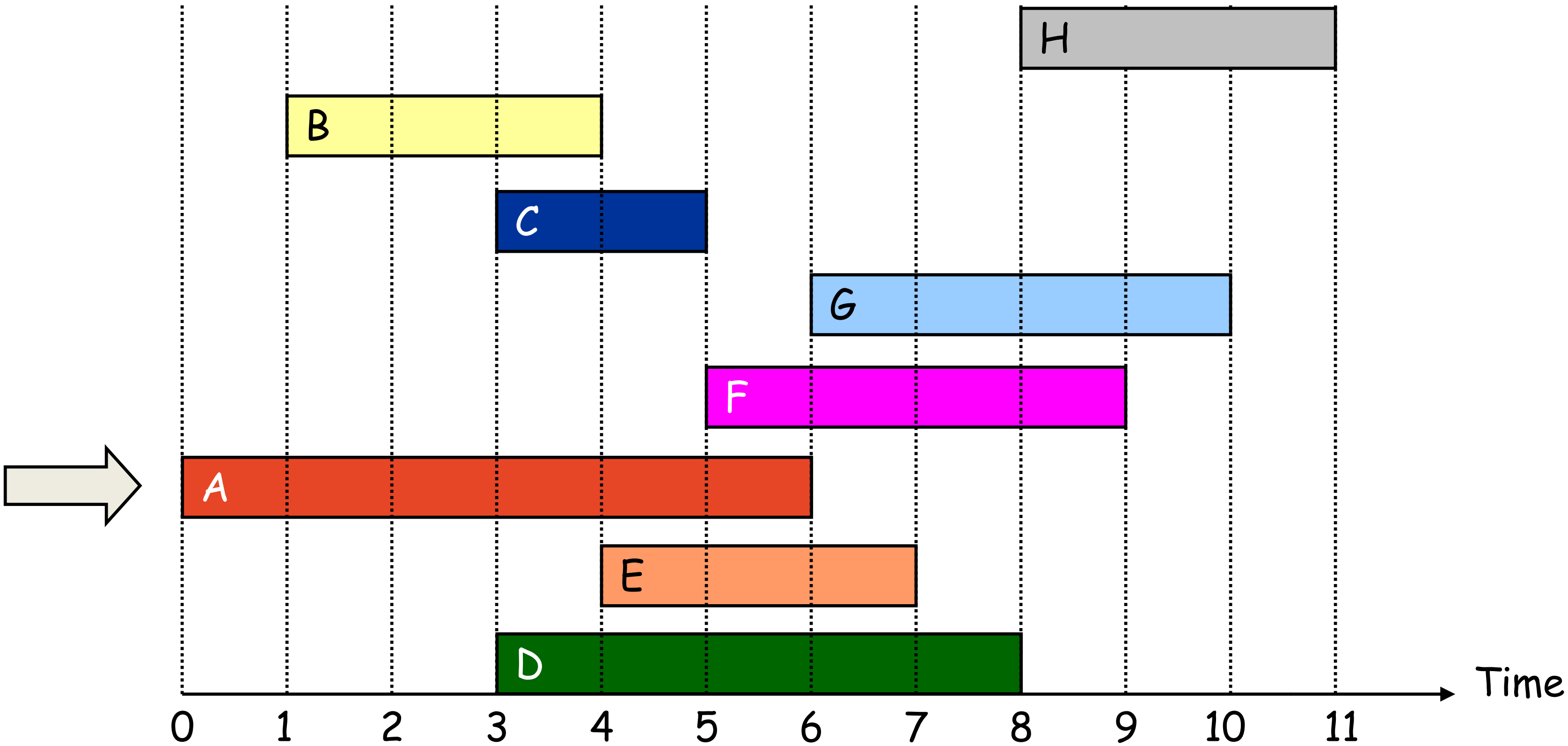


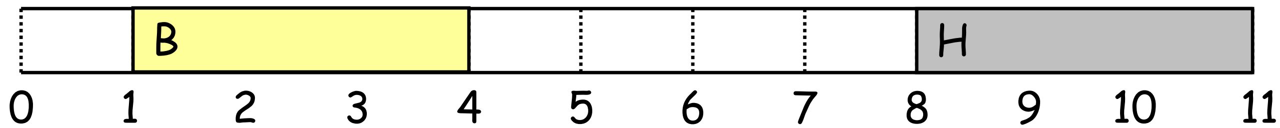
Interval Scheduling - [Fewest Conflicts]



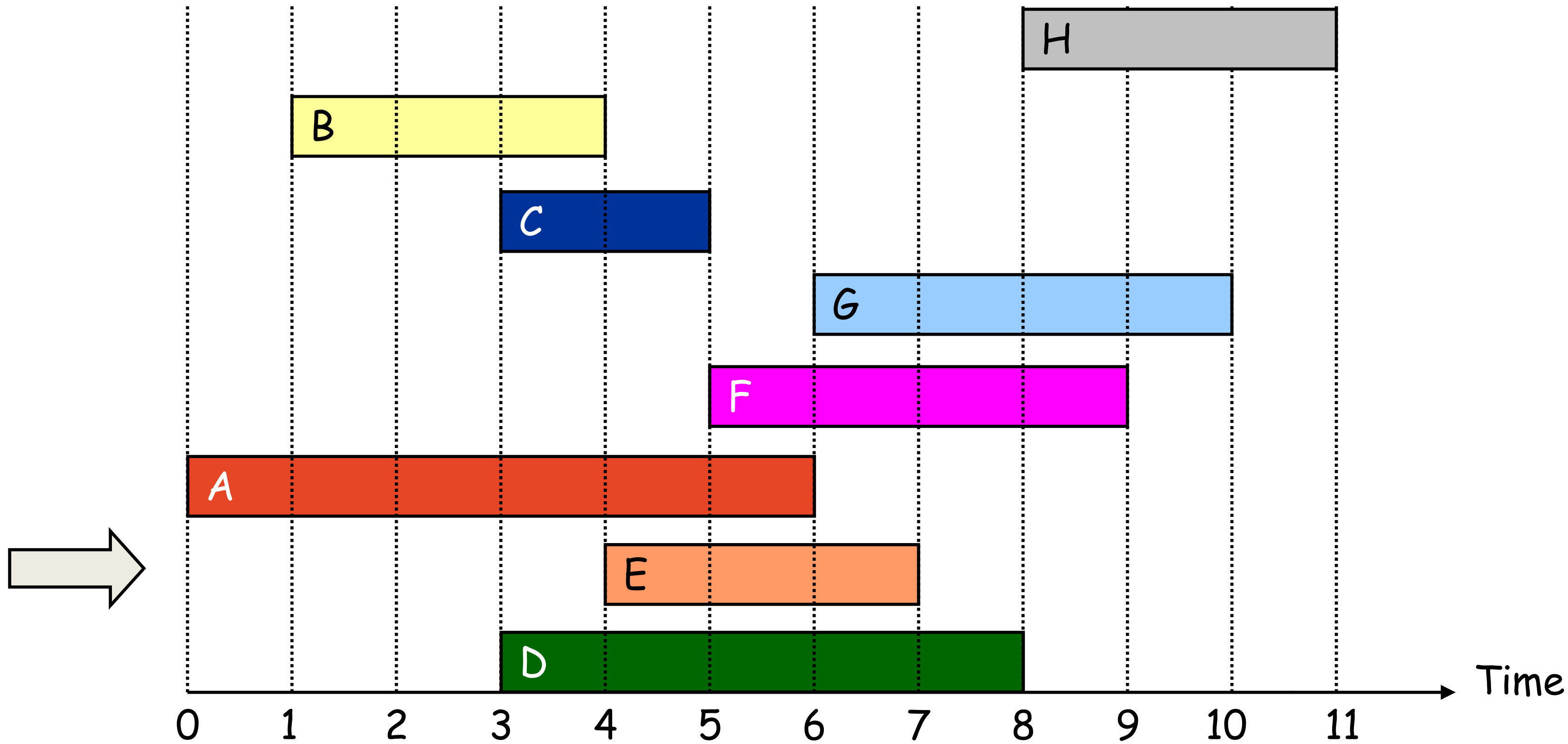


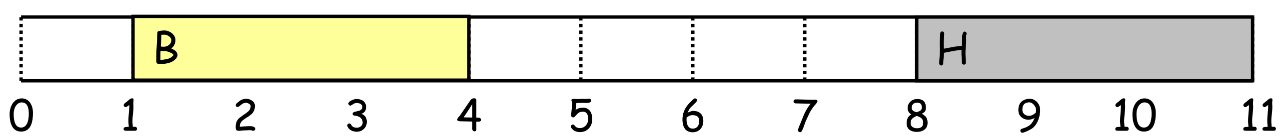
Interval Scheduling - [Fewest Conflicts]



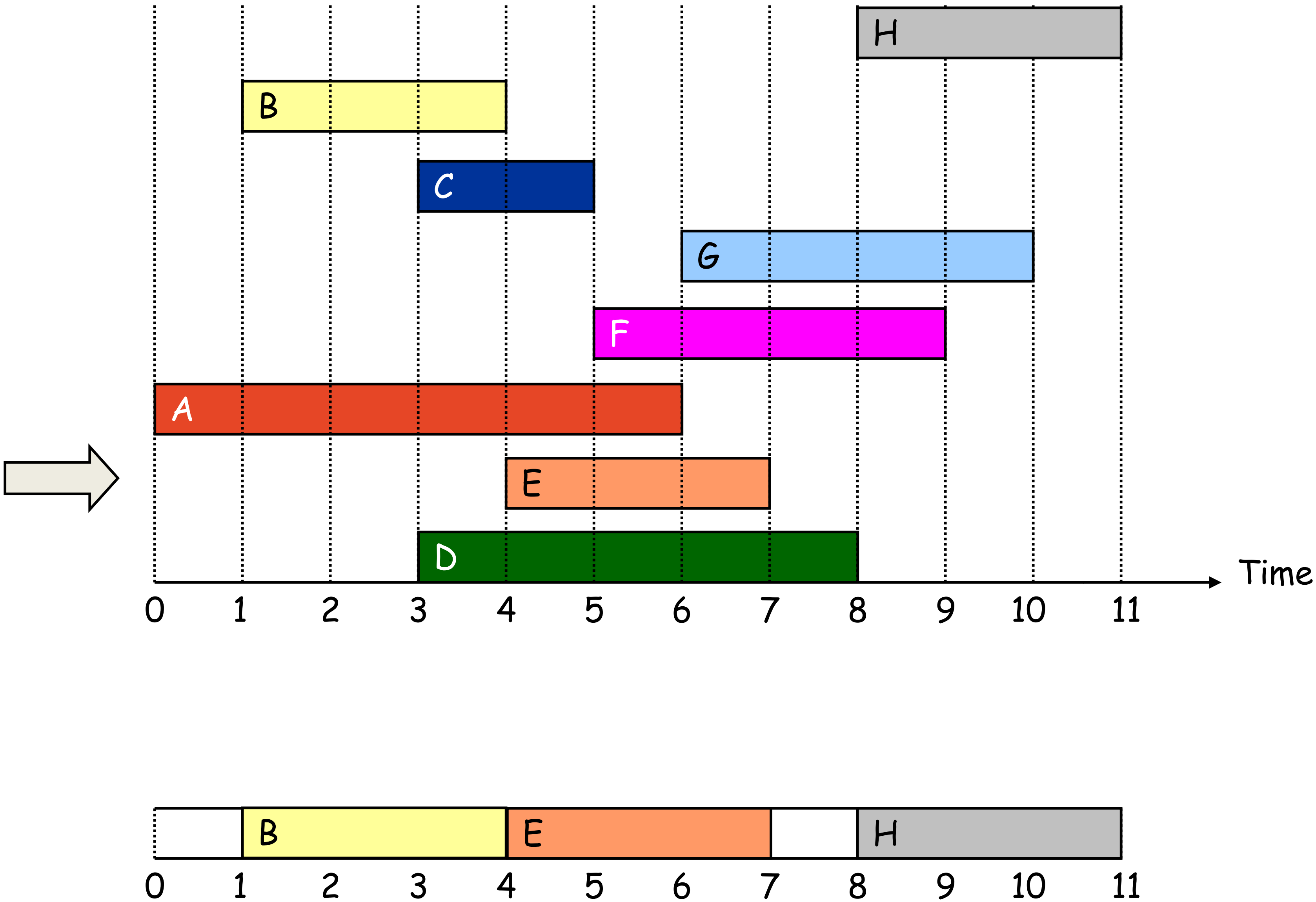


Interval Scheduling - [Fewest Conflicts]

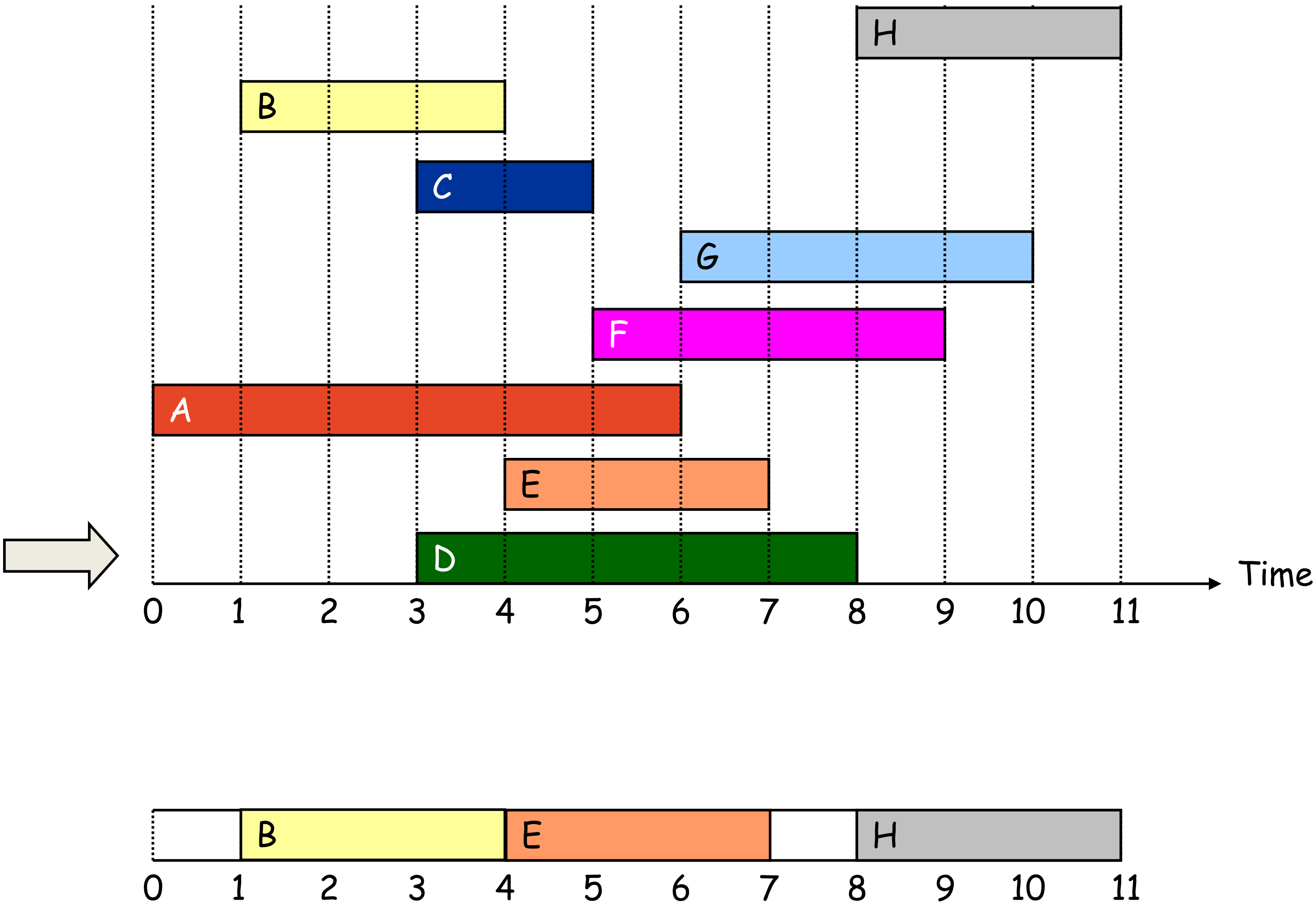




Interval Scheduling - [Fewest Conflicts]



Interval Scheduling - [Fewest Conflicts]



Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it is compatible with the ones already taken.

breaks [Earliest start time]

breaks [Shortest interval]

breaks [Fewest conflicts]

Interval Scheduling: Greedy Algorithm

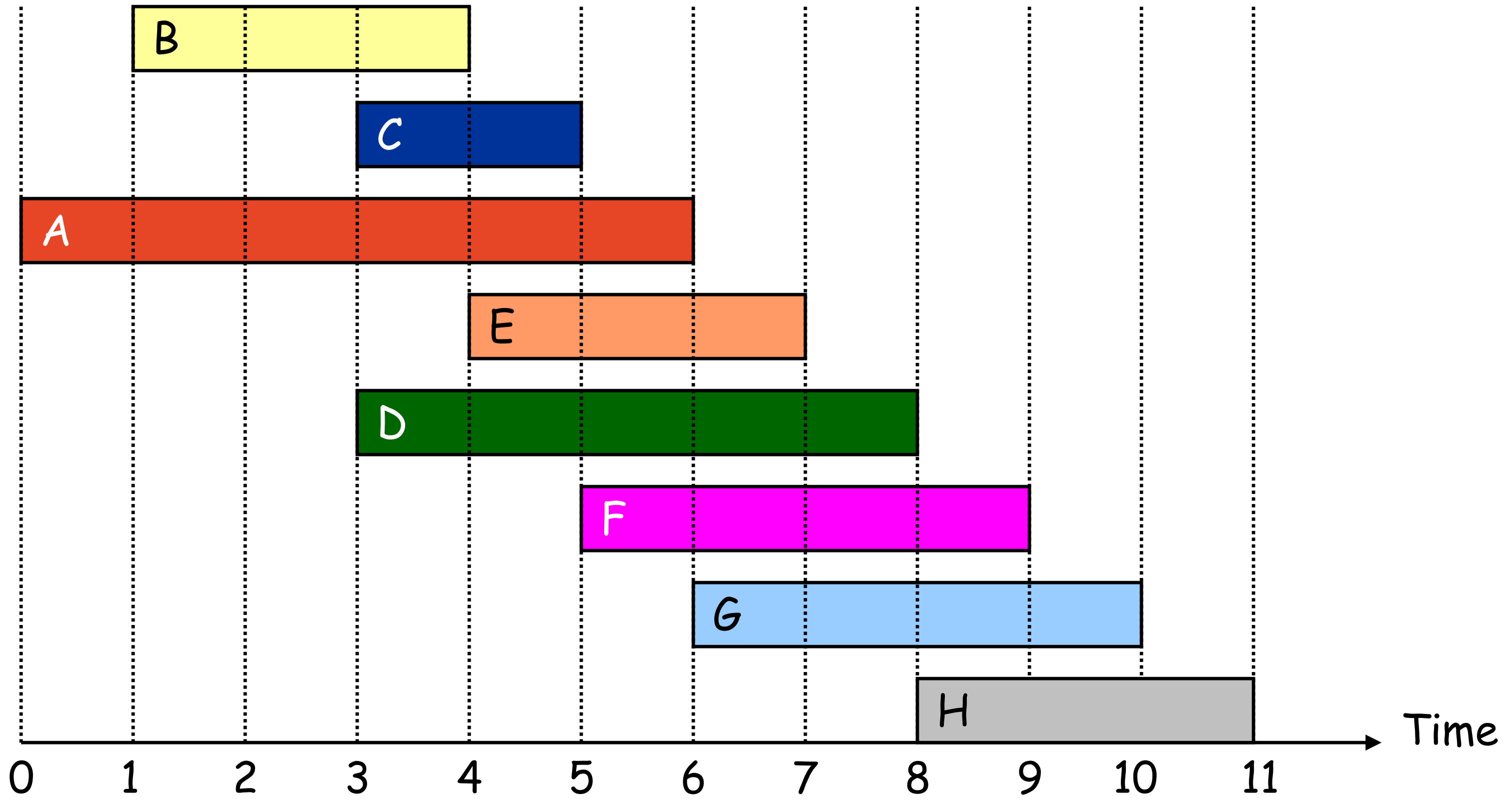
Only [Earliest finish time] remains to be tested.

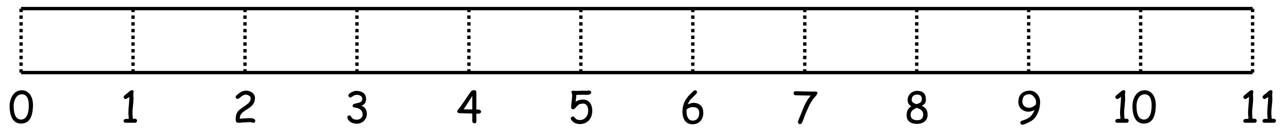
– Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it is compatible with the ones already taken.

```
Sort jobs by finish times so that f₁ ≤ f₂ ≤ ... ≤ fₙ.

 jobs selected
A ← ∅
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

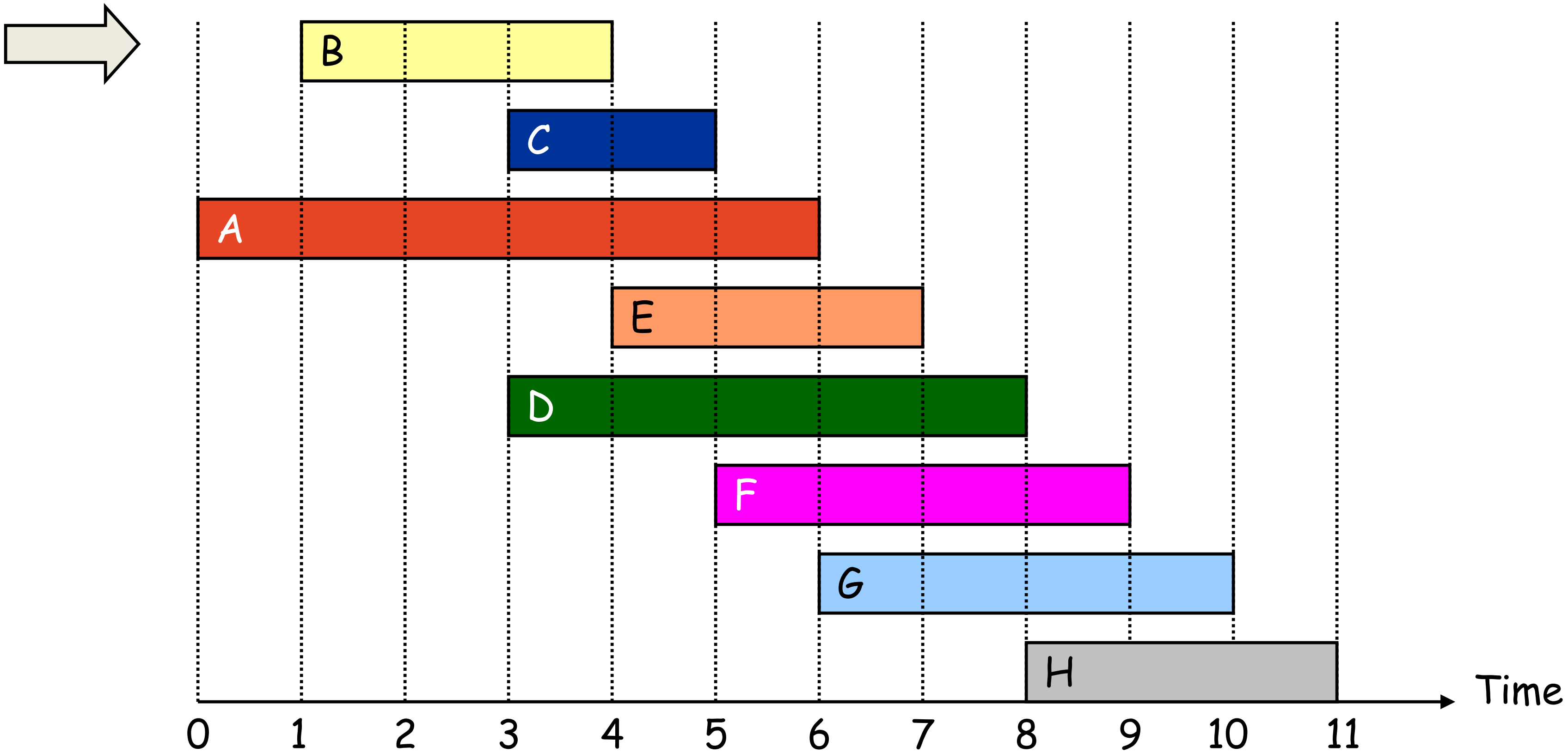
– Implementation. O(n log n).
  – Remember job j* that was added last to A.
  – Job j is compatible with A if $s_j \geq f_{j^*}$.

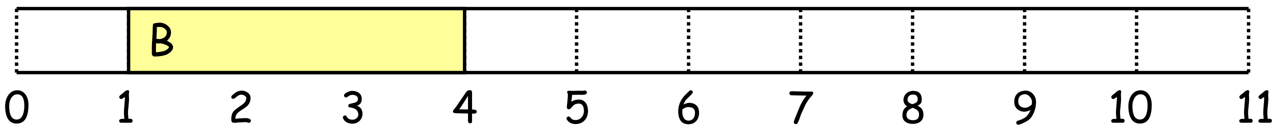
Interval Scheduling



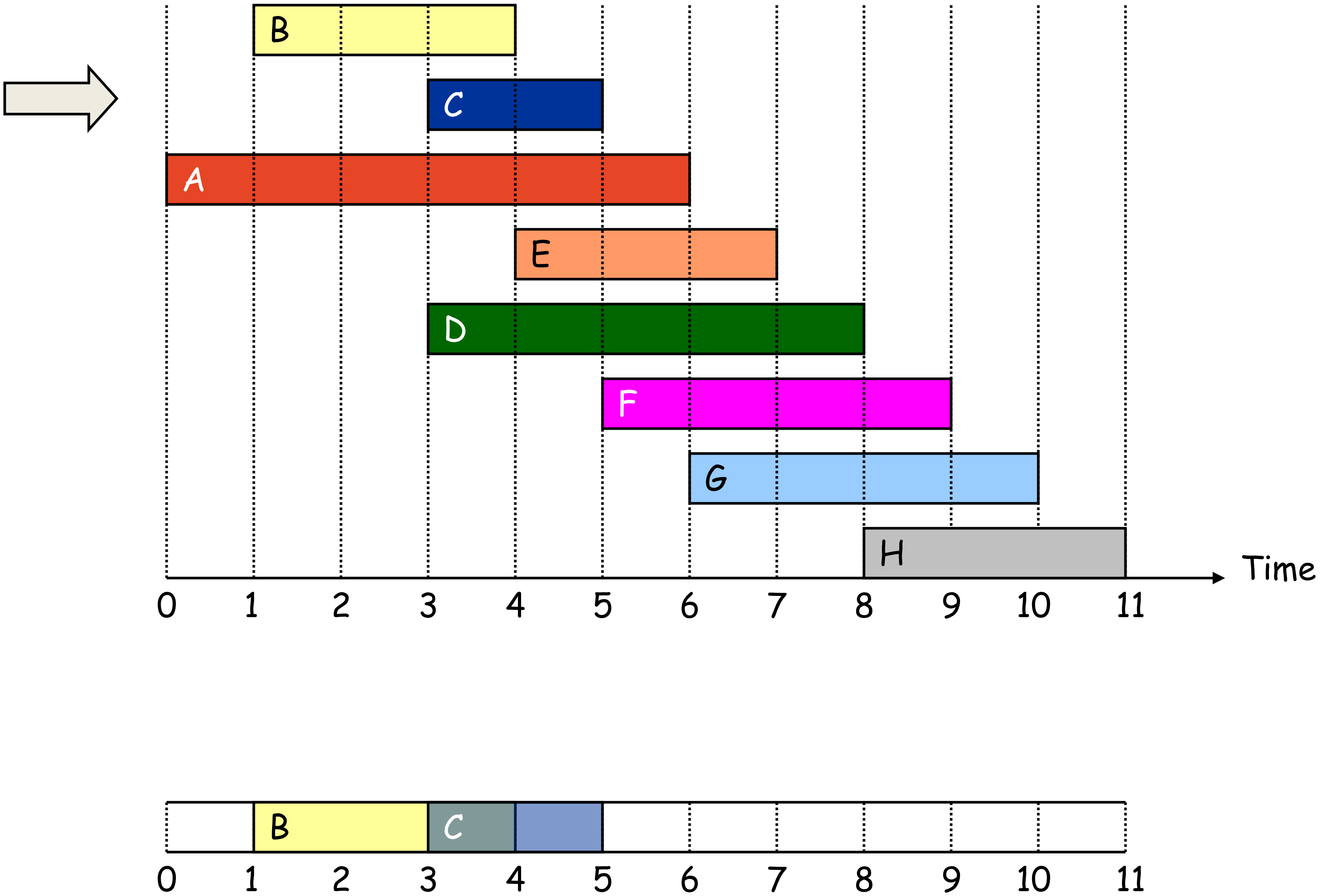


Interval Scheduling

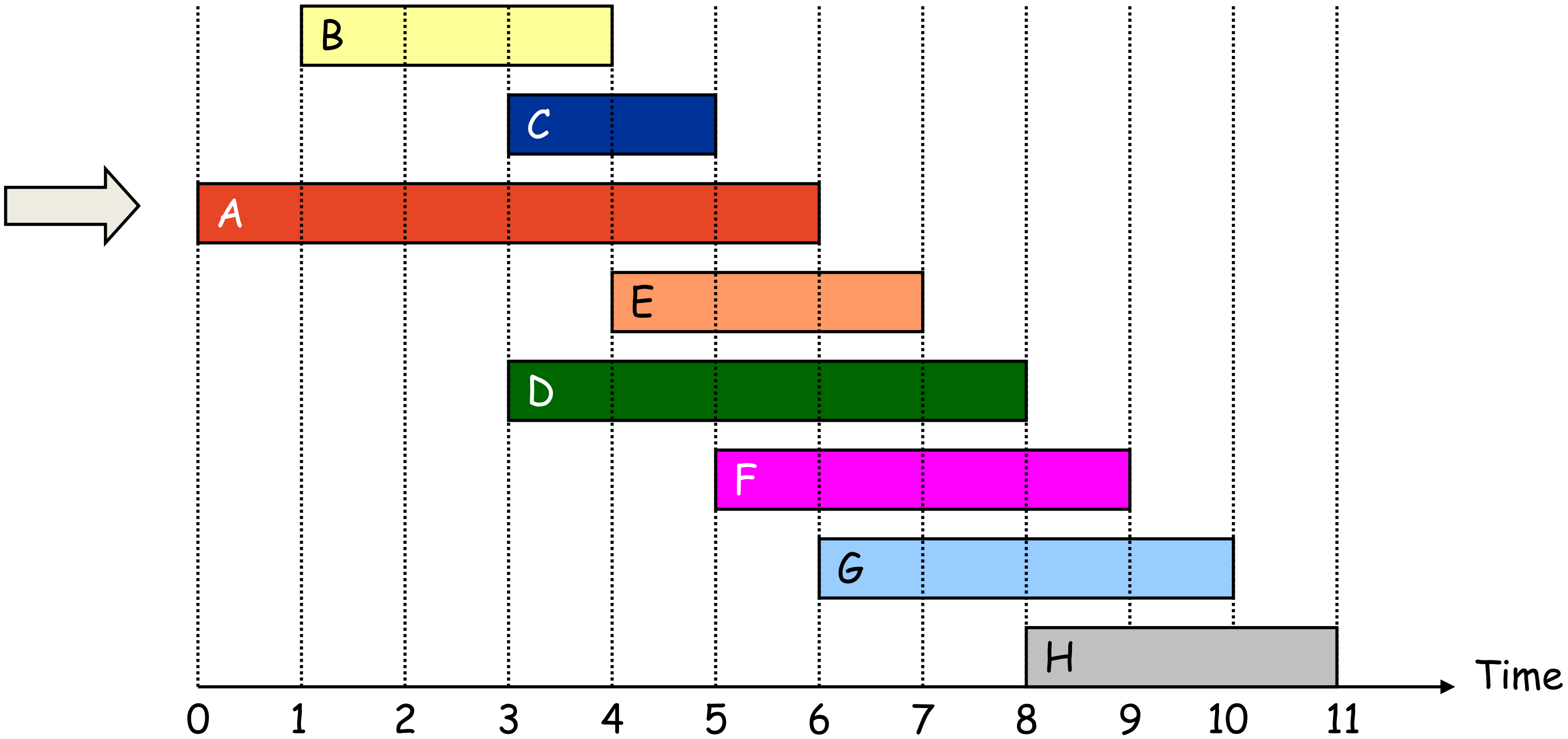


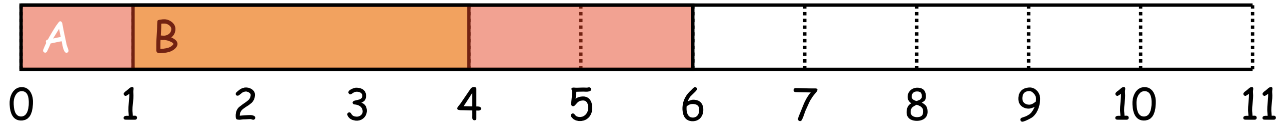


Interval Scheduling

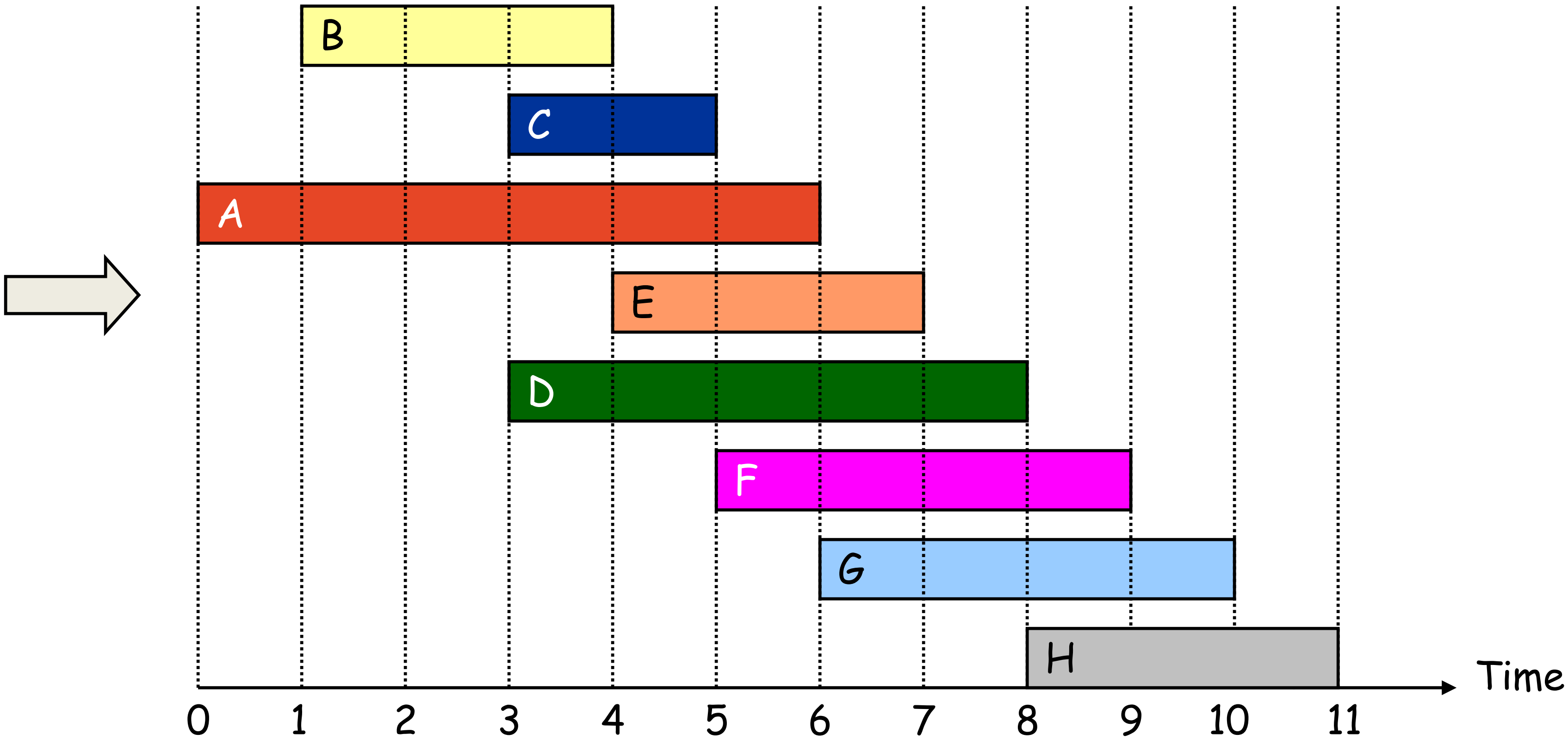


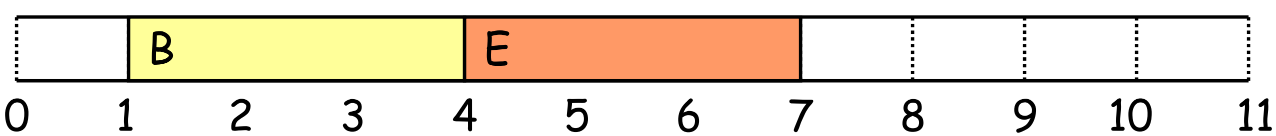
Interval Scheduling



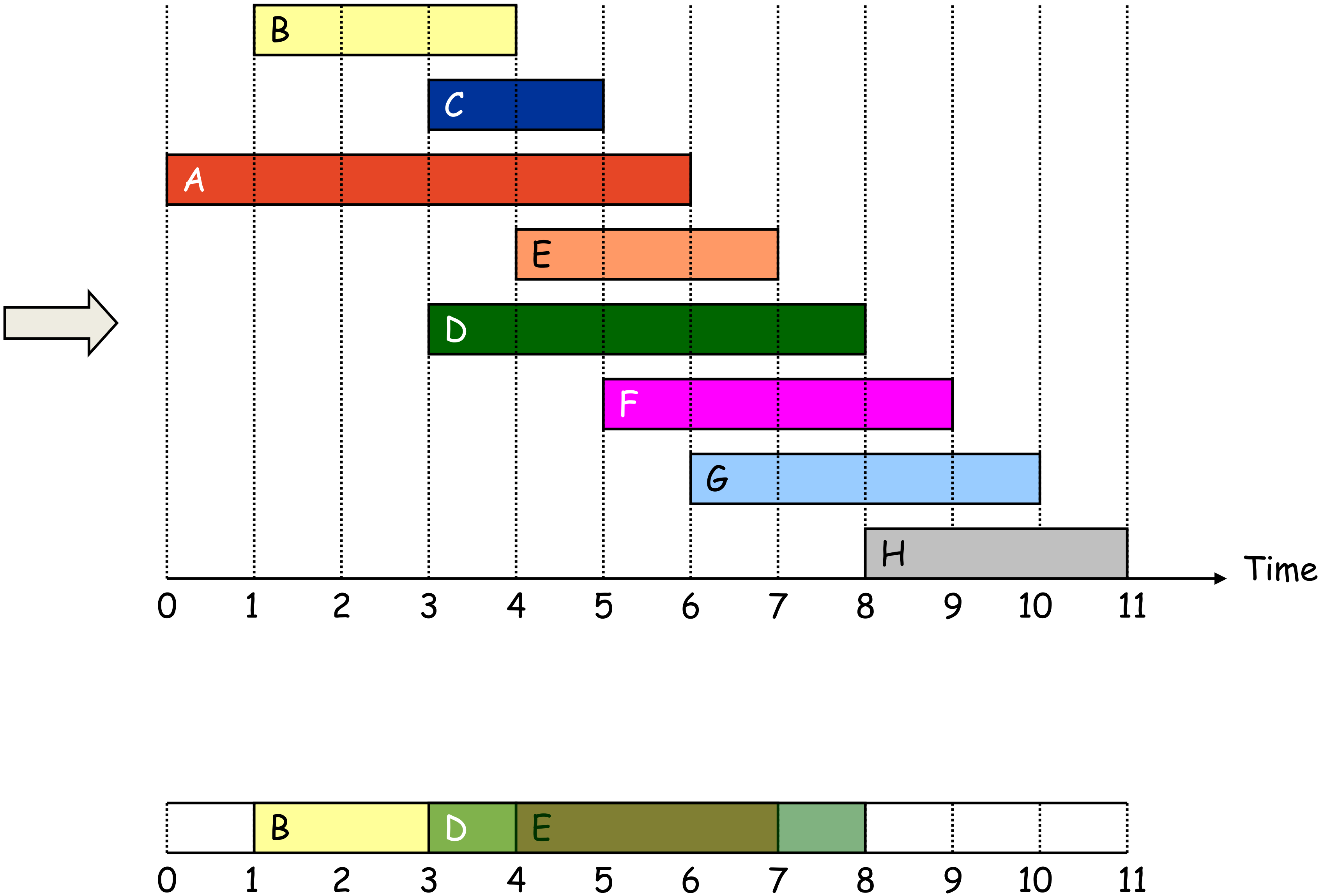


Interval Scheduling

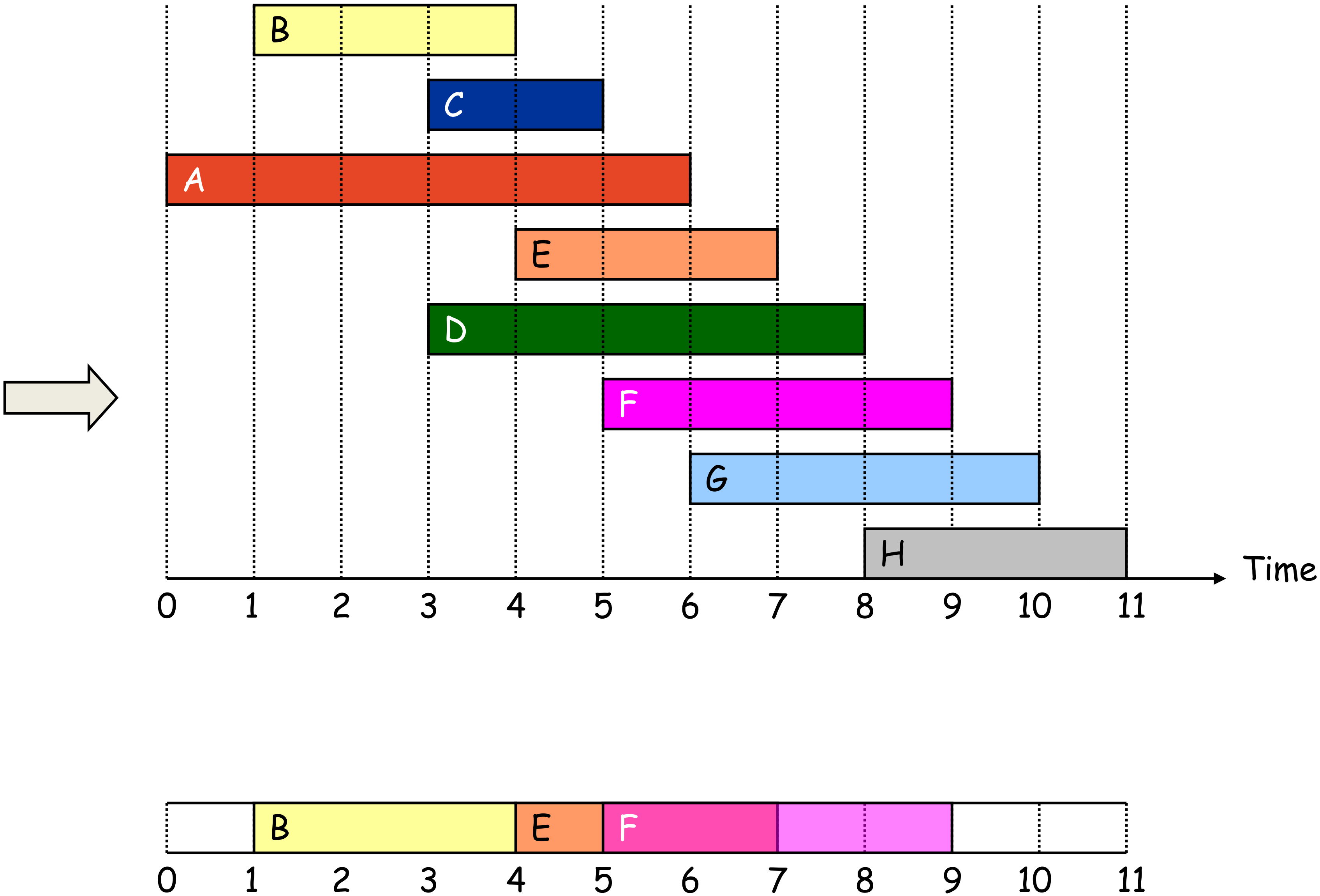




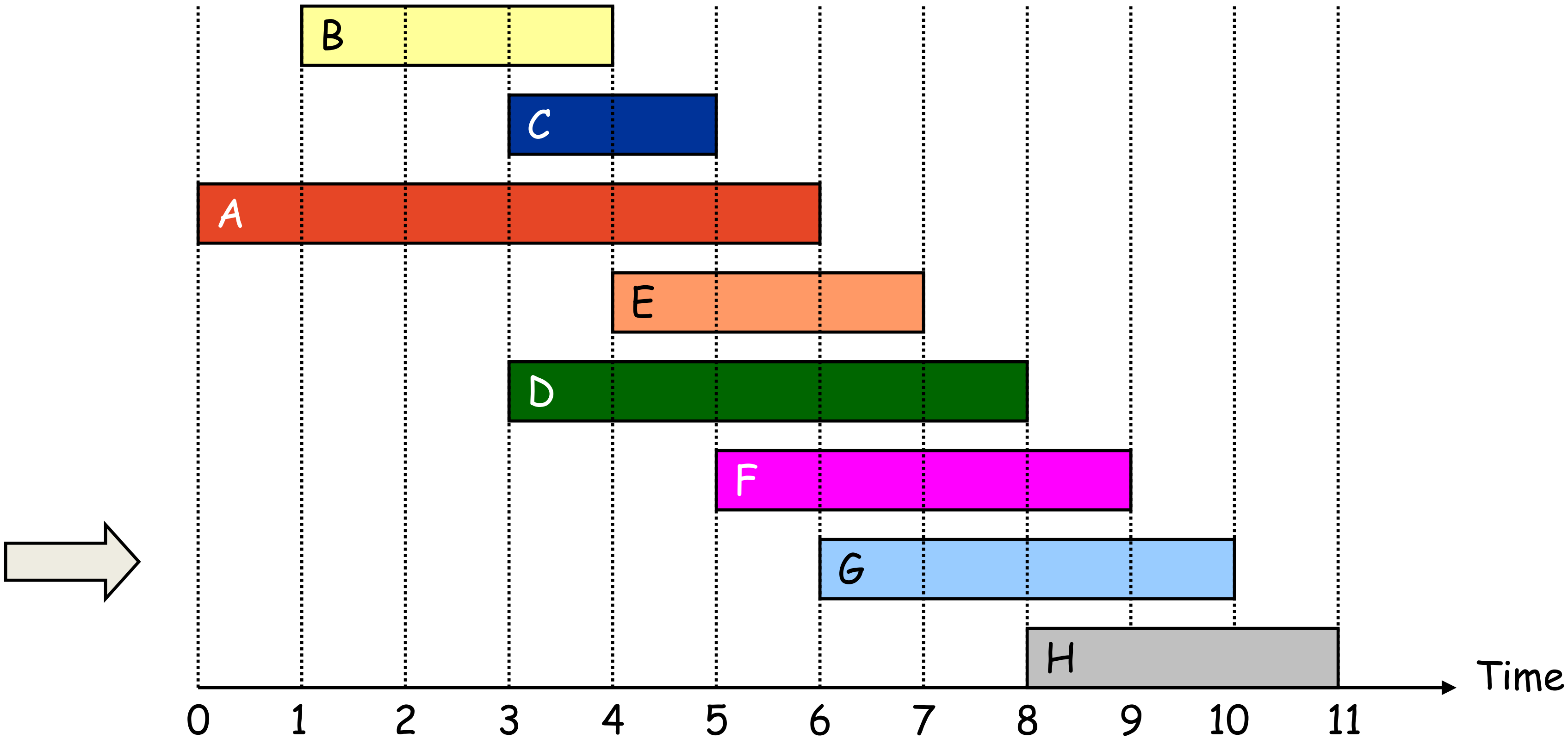
Interval Scheduling

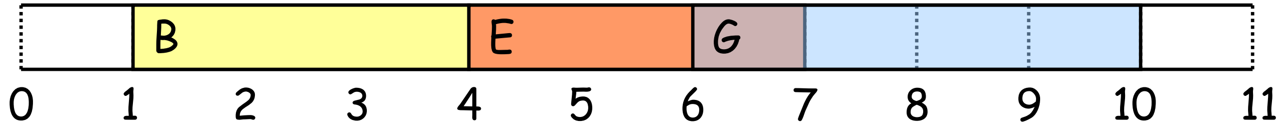


Interval Scheduling

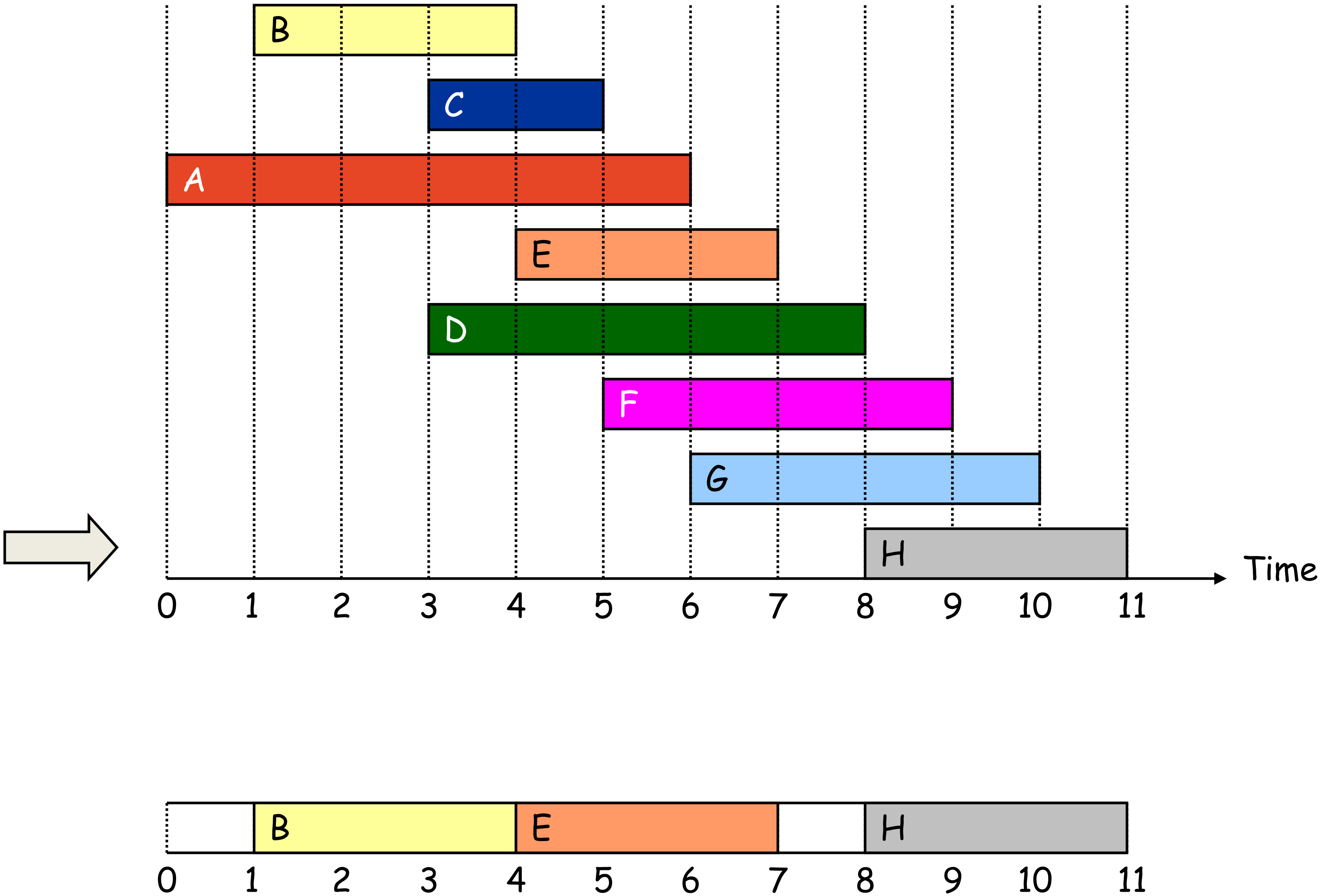


Interval Scheduling





Interval Scheduling



Interval Scheduling: Analysis

The standard way of proving the correctness of a greedy algorithm is by using an exchange argument.

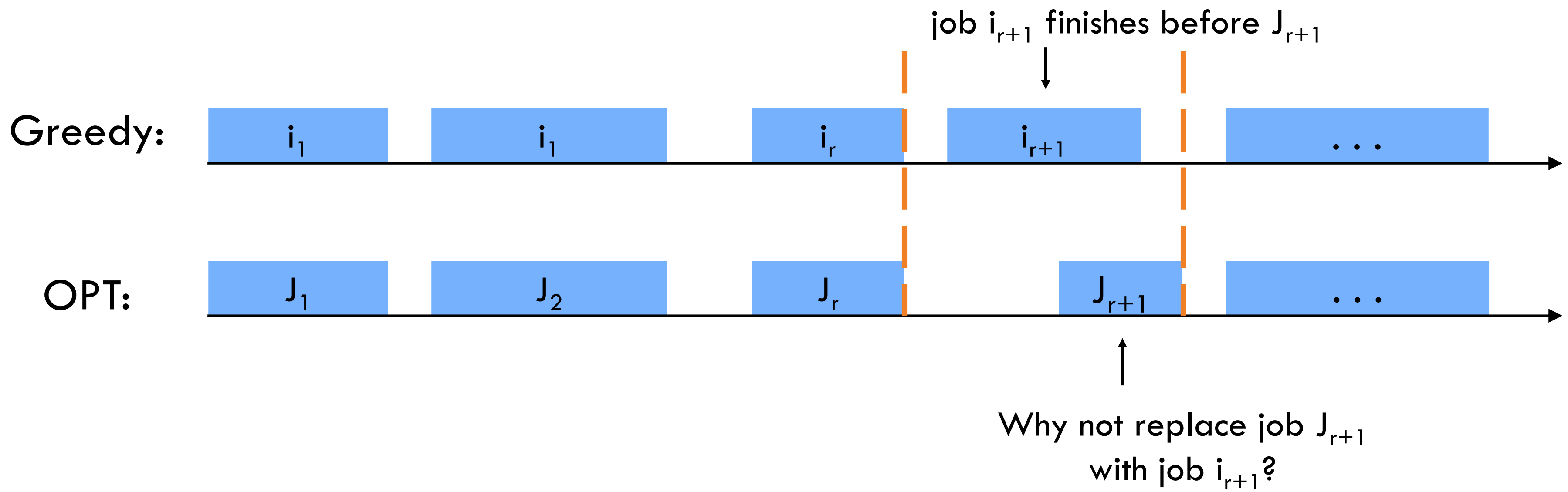
1. Define your greedy solution.
2. Compare solutions. If $X_{greedy} \neq X_{opt}$, then they must differ in some specific way.
3. Exchange Pieces. Transform X_{opt} to a solution that is "closer" to X_{greedy} and prove cost doesn't increase.
4. Iterate. By iteratively exchanging pieces one can turn X_{opt} into X_{greedy} without impacting the quality of the solution.

Interval Scheduling: Analysis

– **Theorem:** Greedy algorithm [Earliest finish time] is optimal.

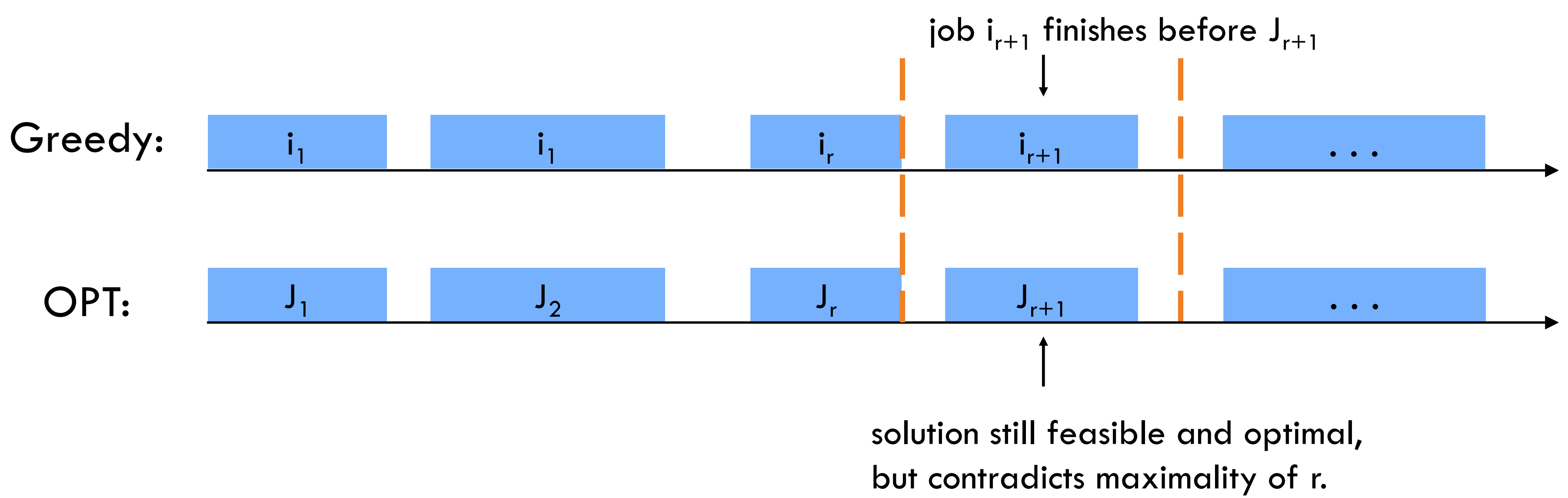
– **Proof:** (by contradiction)

  – Assume greedy is not optimal, and let's see what happens.
  – Let $i_1, i_2, \ldots i_k$ denote the set of jobs selected by greedy.
  – Let $J_1, J_2, \ldots J_m$ denote the set of jobs in an optimal solution with $i_1 = J_1, i_2 = J_2, \ldots, i_r = J_r$ for the largest possible value of r.



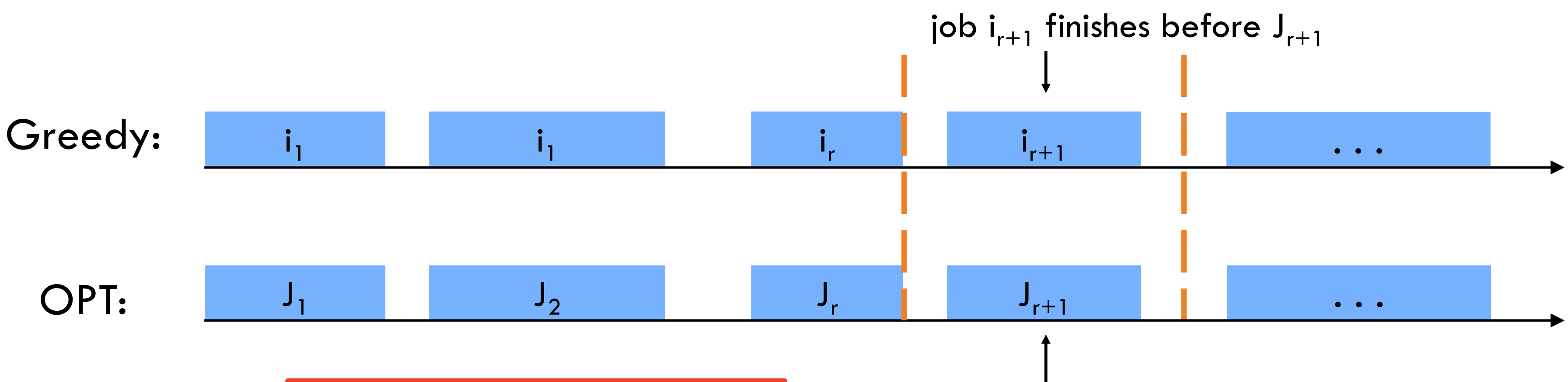
Interval Scheduling: Analysis

- **Theorem:** Greedy algorithm [Earliest finish time] is optimal.
- **Proof:** (by contradiction)
  - Assume greedy is not optimal, and let's see what happens.
  - Let $i_1, i_2, \ldots i_k$ denote the set of jobs selected by greedy.
  - Let $J_1, J_2, \ldots J_m$ denote the set of jobs in an optimal solution with $i_1 = J_1, i_2 = J_2, \ldots, i_r = J_r$ for the largest possible value of r.



Interval Scheduling: Analysis

- **Theorem:** Greedy algorithm [Earliest finish time] is optimal.
- **Proof:** (by contradiction)
  - Assume greedy is not optimal, and let's see what happens.
  - Let $i_1, i_2, \ldots i_k$ denote the set of jobs selected by greedy.
  - Let $J_1, J_2, \ldots J_m$ denote the set of jobs in an optimal solution with $i_1 = J_1, i_2 = J_2, \ldots, i_r = J_r$ for the largest possible value of r.



solution still feasible and optimal, but contradicts maximality of r.

Exchange argument!

Interval Scheduling

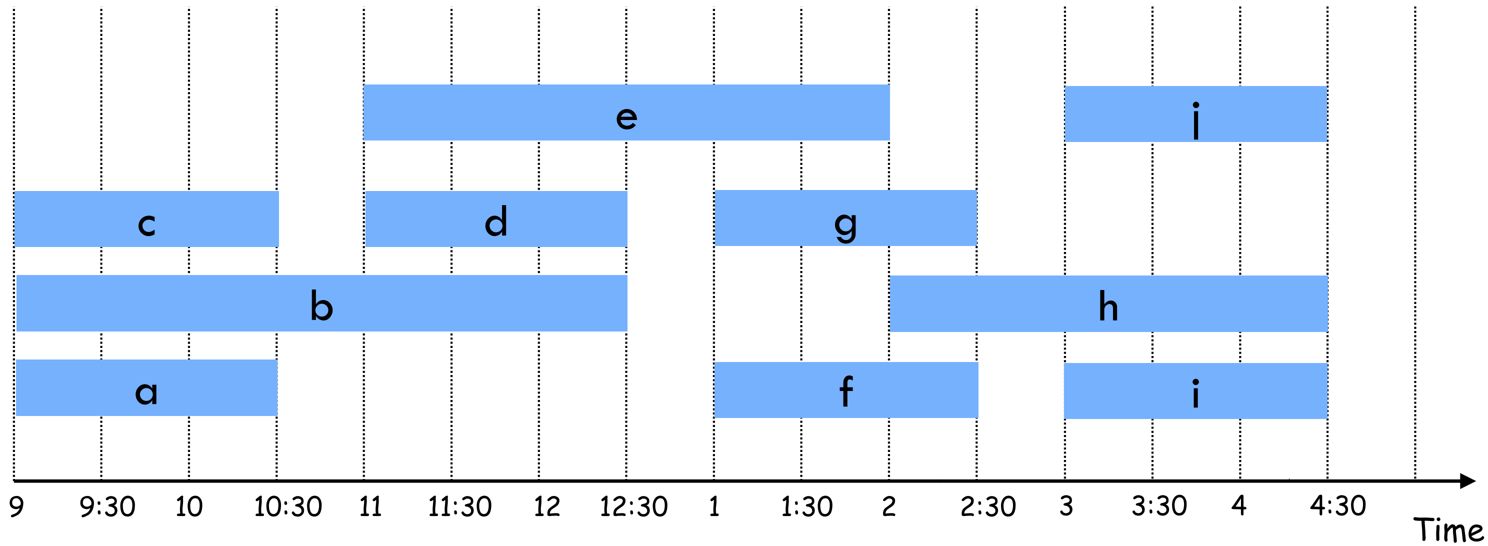
There exists a greedy algorithm [Earliest finish time] that computes the optimal solution in O(n log n) time.

Interval Partitioning

Interval Partitioning

- Interval partitioning.
  - Lecture i starts at s_i and finishes at f_i.
  - Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

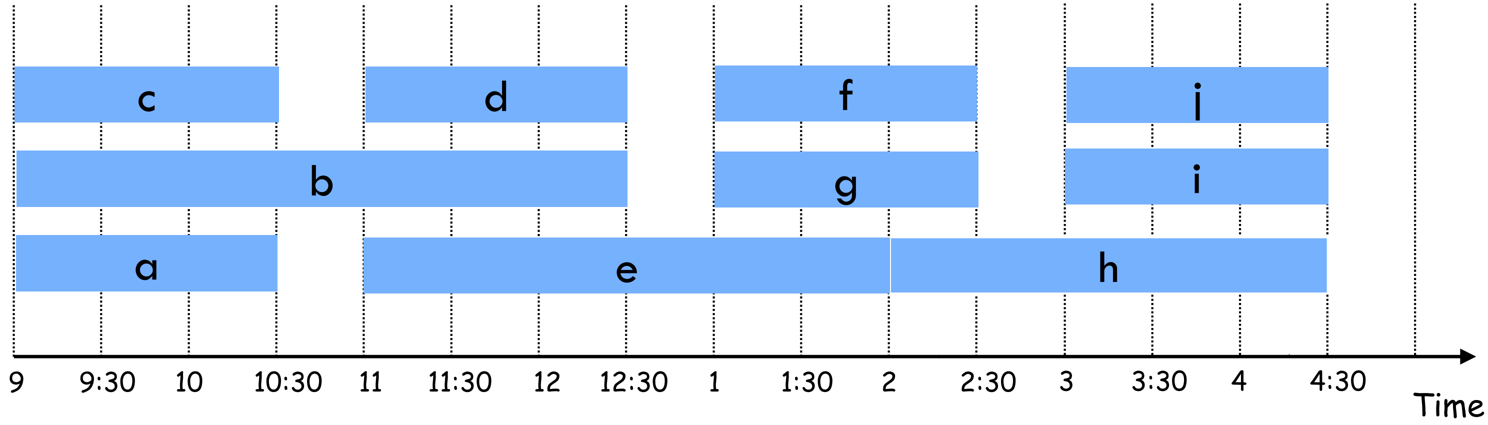
- Ex: This schedule uses 4 classrooms to schedule 10 lectures.



Interval Partitioning

- Interval partitioning.
  - Lecture i starts at s_i and finishes at f_i.
  - Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

- Ex: This schedule uses only 3.

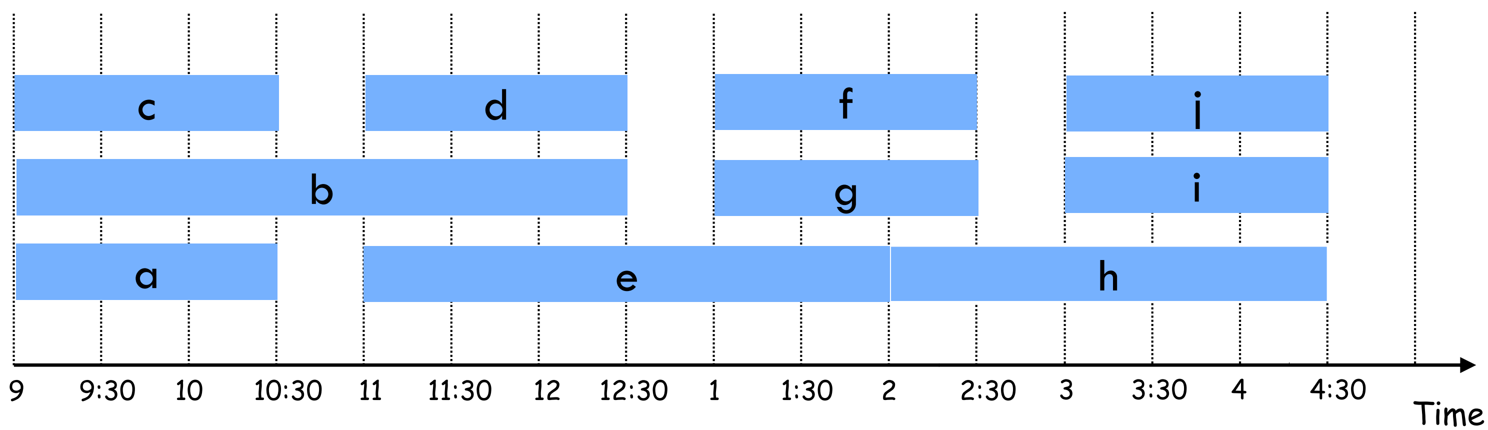


Interval Partitioning: Lower bound

- **Definition:** The depth of a set of open intervals is the maximum number that contain any given time.

- **Observation:** Number of classrooms needed $\geq$ depth.

- Example: Depth of schedule below is 3 (a, b, c all contain 9:30)
  $\Rightarrow$ schedule below is optimal.
- **Question:** Does there always exist a schedule equal to depth of intervals?



Interval Partitioning: Greedy Algorithm

– **Greedy algorithm.** Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
Sort intervals by starting time so that s1 ≤ s2 ≤ ... ≤ sn.
d ← 0    ← number of allocated classrooms

for i = 1 to n {
   if (lecture i is compatible with some classroom k)
      schedule lecture i in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture i in classroom d + 1
      d ← d + 1
}
```

Interval Partitioning: Greedy Algorithm

- **Greedy algorithm.** Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
Sort intervals by starting time so that s1 ≤ s2 ≤ ... ≤ sn.
d ← 0   ← number of allocated classrooms

for i = 1 to n {
   if (lecture i is compatible with some classroom k)
      schedule lecture i in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture i in classroom d + 1
      d ← d + 1
}
```

- **Implementation.** O(n log n).
  - For each classroom k, maintain the finish time of the last job added.
  - Keep the classrooms in a priority queue.

Interval Partitioning: Greedy Analysis

- **Observation:** Greedy algorithm never schedules two incompatible lectures in the same classroom.

- **Theorem:** Greedy algorithm is optimal.
- **Proof:**
  - d = number of classrooms that the greedy algorithm allocates.
  - Classroom d is opened because we needed to schedule a job, say i, that is incompatible with all d-1 other classrooms.
  - Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_i.
  - Thus, we have d lectures overlapping at time $s_i + \varepsilon$.
  - Key observation $\Rightarrow$ all schedules use $\geq$ d classrooms.

just after time s_i

■

Interval Partitioning

There exists a greedy algorithm [Earliest starting time] that computes the optimal solution in O(n log n) time.

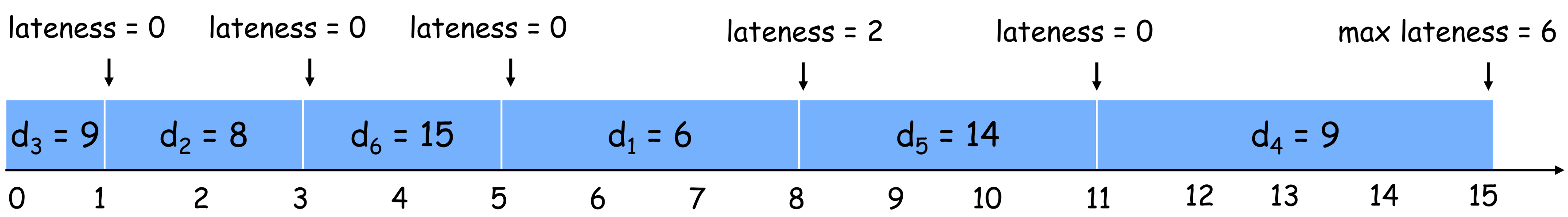
Scheduling to Minimize Lateness

Scheduling to Minimizing Lateness

- Minimizing lateness problem. [No fix start time]
  - Single resource processes one job at a time.
  - Job i requires t_i units of processing time and is due at time d_i.
  - If i starts at time s_i, it finishes at time $f_i = s_i + t_i$.
  - Lateness: $\ell_i = \max \{ 0, \ f_i - d_i \}$.
  - **Goal:** schedule all jobs to minimize maximum lateness $L = \max \ell_i$.

- Ex:

	1	2	3	4	5	6	jobs
t_i	3	2	1	4	3	2	processing time
d_i	6	8	9	9	14	15	due time



Minimizing Lateness: Greedy Algorithms

- Greedy template. Consider jobs in some order.

  - [Shortest processing time first] Consider jobs in ascending order of processing time t_i.

  - [Earliest deadline first] Consider jobs in ascending order of deadline d_i.

  - [Smallest slack] Consider jobs in ascending order of slack $d_i - t_i$.

Minimizing Lateness: Greedy Algorithms

- Greedy template. Consider jobs in some order.

  - [Shortest processing time first] Consider jobs in ascending order of processing time t_i.

	1	2
t_i	1	10
d_i	100	10

counterexample

  - [Smallest slack] Consider jobs in ascending order of slack $d_i - t_i$.

	1	2
t_i	1	10
d_i	2	10

counterexample

Minimizing Lateness: Greedy Algorithm

– Greedy algorithm. [Earliest deadline first]

```
Sort n jobs by deadline so that d₁ ≤ d₂ ≤ … ≤ dₙ

t ← 0
for j = 1 to n
   Assign job j to interval [t, t + tⱼ]
   sⱼ ← t, fⱼ ← t + tⱼ
   t ← t + tⱼ
output intervals [sⱼ, fⱼ]
```

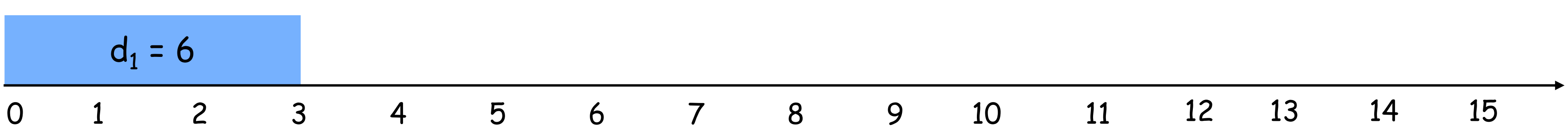
	1	2	3	4	5	6	jobs
t_i	3	2	1	4	3	2	processing time
d_i	6	8	9	9	14	15	due time

Minimizing Lateness: Greedy Algorithm

– Greedy algorithm. [Earliest deadline first]

```
Sort n jobs by deadline so that d1 ≤ d2 ≤ … ≤ dn

t ← 0
for j = 1 to n
   Assign job j to interval [t, t + tj]
   sj ← t, fj ← t + tj
   t ← t + tj
output intervals [sj, fj]
```

	1	2	3	4	5	6	jobs
t_i	3	2	1	4	3	2	processing time
d_i	6	8	9	9	14	15	due time

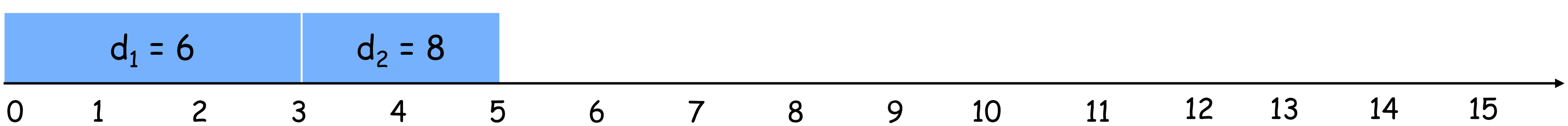


Minimizing Lateness: Greedy Algorithm

– Greedy algorithm. [Earliest deadline first]

```
Sort n jobs by deadline so that d₁ ≤ d₂ ≤ … ≤ dₙ

t ← 0
for j = 1 to n
   Assign job j to interval [t, t + tⱼ]
   sⱼ ← t, fⱼ ← t + tⱼ
   t ← t + tⱼ
output intervals [sⱼ, fⱼ]
```

	1	2	3	4	5	6	jobs
t_i	3	2	1	4	3	2	processing time
d_i	6	8	9	9	14	15	due time

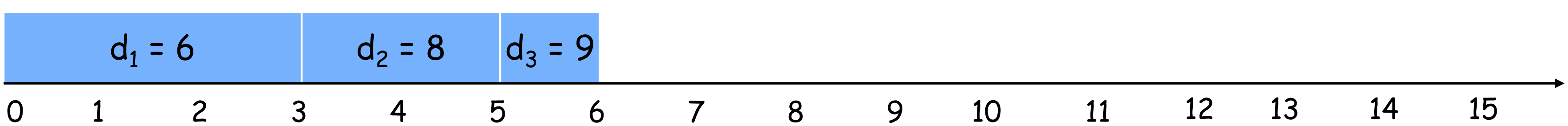


Minimizing Lateness: Greedy Algorithm

– Greedy algorithm. [Earliest deadline first]

```
Sort n jobs by deadline so that d1 ≤ d2 ≤ … ≤ dn

t ← 0
for j = 1 to n
   Assign job j to interval [t, t + tj]
   sj ← t, fj ← t + tj
   t ← t + tj
output intervals [sj, fj]
```

	1	2	3	4	5	6	jobs
t_i	3	2	1	4	3	2	processing time
d_i	6	8	9	9	14	15	due time



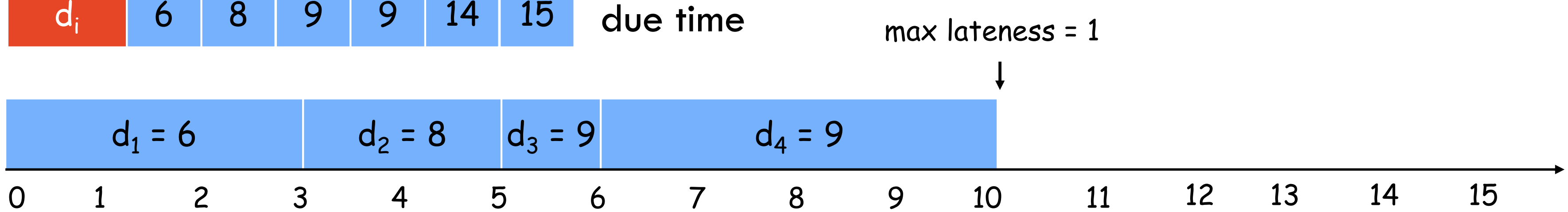
Minimizing Lateness: Greedy Algorithm

– Greedy algorithm. [Earliest deadline first]

Sort n jobs by deadline so that $d_1 \le d_2 \le \ldots \le d_n$

$t \leftarrow 0$
for j = 1 **to** n
    Assign job j to interval $[t, t + t_j]$
    $s_j \leftarrow t$, $f_j \leftarrow t + t_j$
    $t \leftarrow t + t_j$
output intervals $[s_j, f_j]$

	1	2	3	4	5	6	jobs
t_i	3	2	1	4	3	2	processing time
d_i	6	8	9	9	14	15	due time

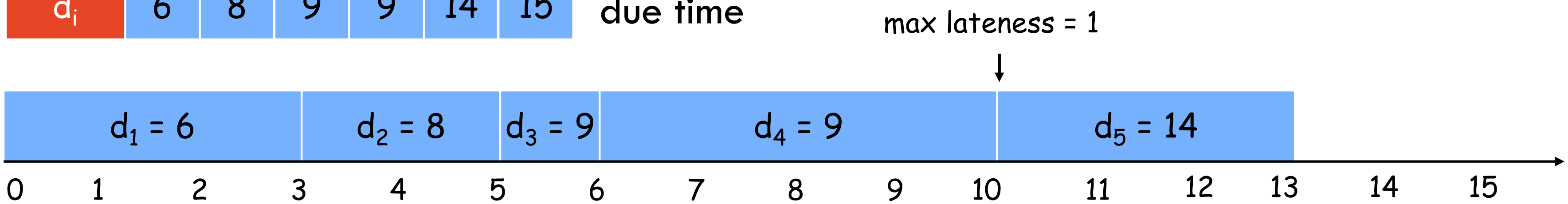


Minimizing Lateness: Greedy Algorithm

– Greedy algorithm. [Earliest deadline first]

```
Sort n jobs by deadline so that d1 ≤ d2 ≤ … ≤ dn

t ← 0
for j = 1 to n
   Assign job j to interval [t, t + tj]
   sj ← t, fj ← t + tj
   t ← t + tj
output intervals [sj, fj]
```

	1	2	3	4	5	6	jobs
t_i	3	2	1	4	3	2	processing time
d_i	6	8	9	9	14	15	due time

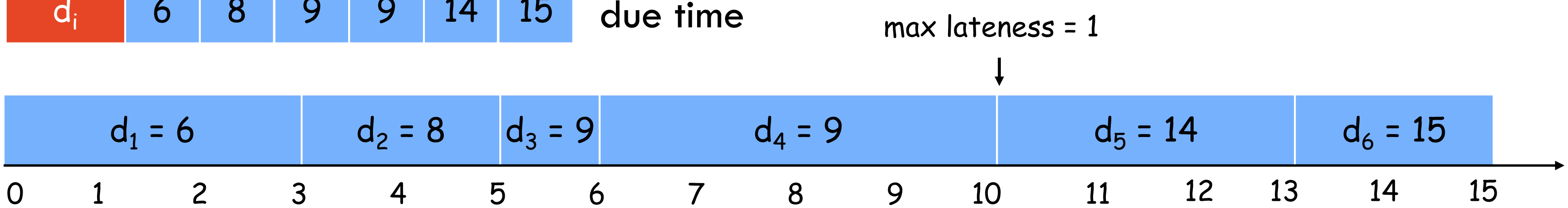


Minimizing Lateness: Greedy Algorithm

– Greedy algorithm. [Earliest deadline first]

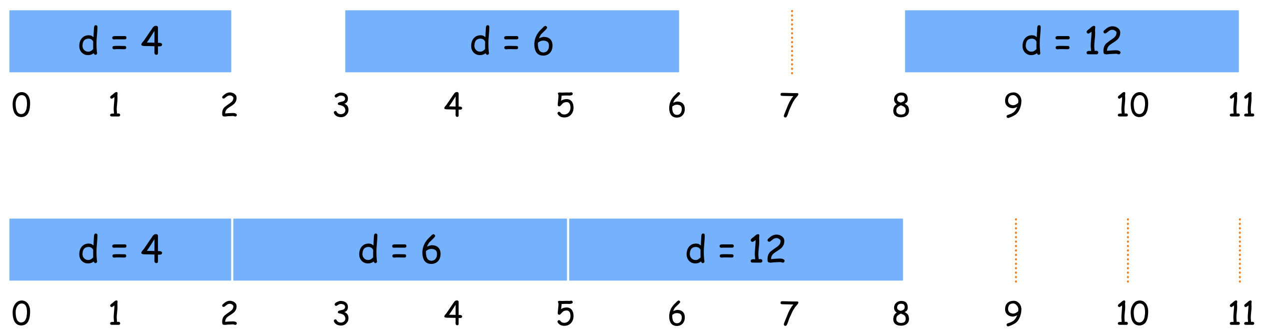
```
Sort n jobs by deadline so that d₁ ≤ d₂ ≤ … ≤ dₙ

t ← 0
for j = 1 to n
   Assign job j to interval [t, t + tⱼ]
   sⱼ ← t, fⱼ ← t + tⱼ
   t ← t + tⱼ
output intervals [sⱼ, fⱼ]
```

	1	2	3	4	5	6	jobs
t_i	3	2	1	4	3	2	processing time
d_i	6	8	9	9	14	15	due time



Minimizing Lateness: No Idle Time

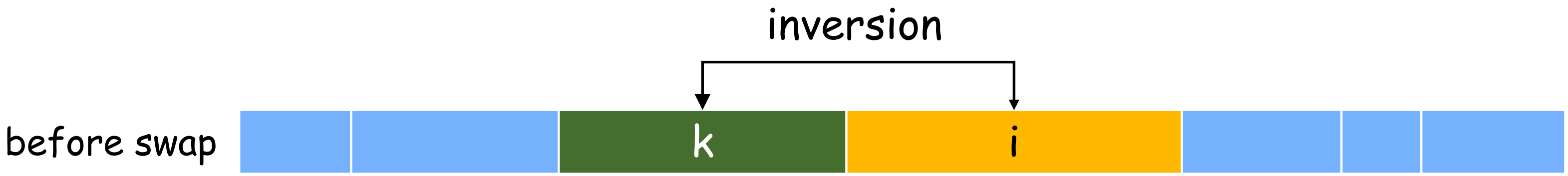
– **Observation:** There exists an optimal schedule with no idle time.



– **Observation:** The greedy schedule has no idle time.

Minimizing Lateness: Inversions

– **Definition:** An inversion in schedule S is a pair of jobs i and k such that i < k (by deadline) but k is scheduled before i.

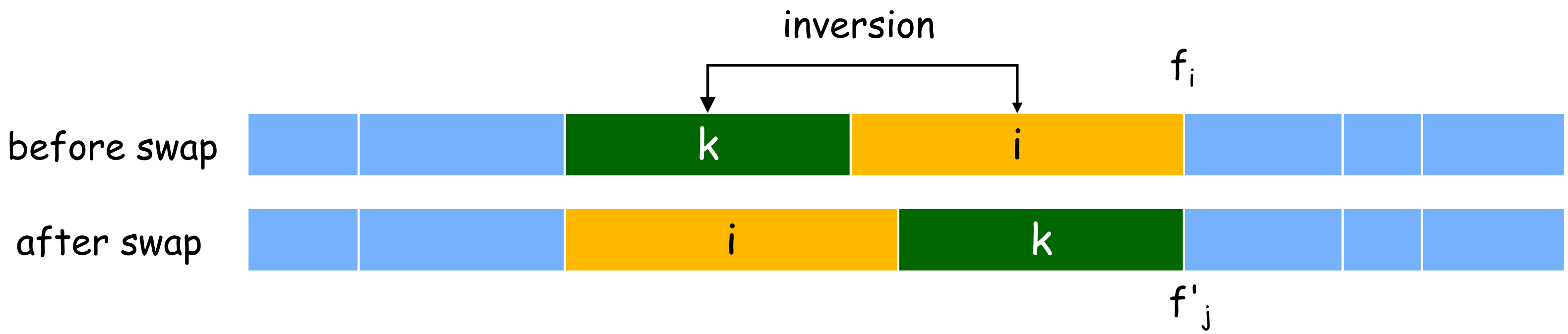


– **Observation:** Greedy schedule has no inversions.

– **Observation:** If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.

Minimizing Lateness: Inversions

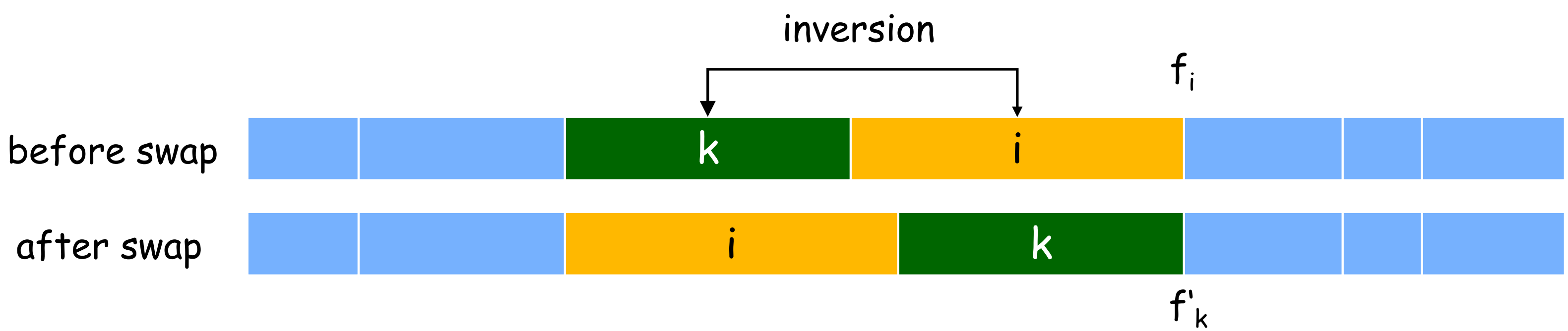
– **Definition:** An inversion in schedule S is a pair of jobs i and k such that i < k (by deadline) but k is scheduled before i.



– **Claim:** Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

Minimizing Lateness: Inversions

- **Definition:** An inversion in schedule S is a pair of jobs i and k such that i < k (by deadline) but k is scheduled before i.



- **Claim:** Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

- **Proof:** Let ℓ be the lateness before the swap, and let ℓ' be the lateness after the swap.
  - $\ell'_x = \ell_x$ for all $x \neq i, k$
  - $\ell'_i \leq \ell_i$
  - If job k is late:

$$
\begin{aligned}
\ell'_k &= f'_k - d_k && \text{(definition)} \\
&= f_i - d_k && (i \text{ finishes at time } f_i) \\
&\leq f_i - d_i && (i < k) \\
&\leq \ell_i && \text{(definition)}
\end{aligned}
$$

Minimizing Lateness: Analysis of Greedy Algorithm

- **Theorem:** Greedy schedule S is optimal.

- **Proof:** Define S* to be an optimal schedule that has the fewest number of inversions, and let's see what happens.
  - Can assume S* has no idle time.
  - If S* has no inversions, then S = S*.
  - If S* has an inversion, let i-k be an adjacent inversion.
    - swapping i and k does not increase the maximum lateness and strictly decreases the number of inversions
    - this contradicts definition of S* ■

Minimizing Lateness

There exists a greedy algorithm [Earliest deadline first] that computes the optimal solution in O(n log n) time.

Greedy Analysis Strategies

- Greedy algorithm stays ahead. Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.

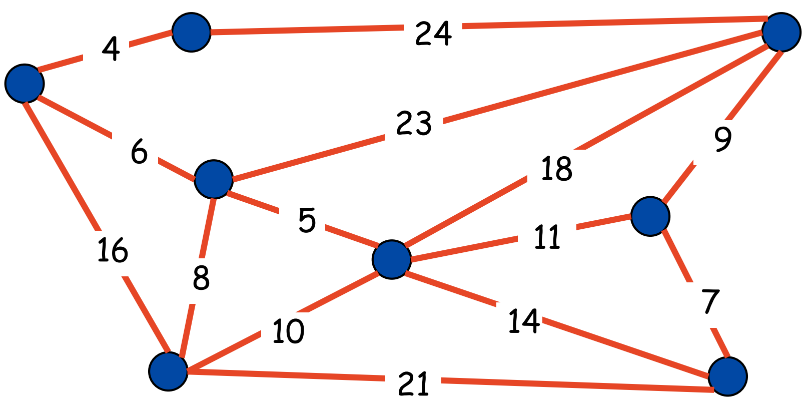
- Exchange argument. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

- Structural. Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

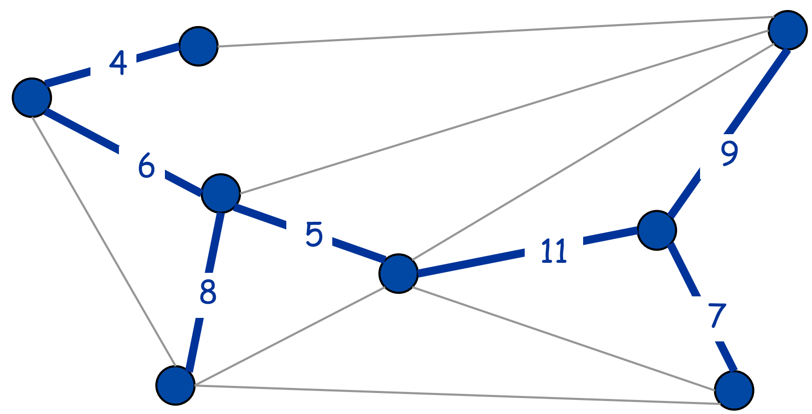
Minimum Spanning Tree

Minimum Spanning Tree

– **Minimum spanning tree (MST).** Given a connected graph G = (V, E) with real-valued edge weights c_e, an MST is a subset of the edges $T \subseteq E$ such that T is a spanning tree whose sum of edge weights is minimized.



$G = (V, E)$



$T, \; \Sigma_{e \in T} \, c_e = 50$

– **Cayley's Theorem.** There are n^{n-2} spanning trees of K_n.

can't solve by brute force

Applications

MST is fundamental problem with diverse applications.

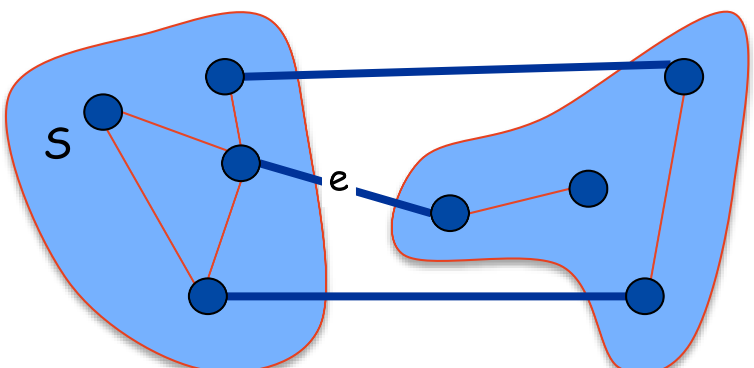
- Network design.
  - telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems.
  - traveling salesperson problem, Steiner tree
- Indirect applications.
  - max bottleneck paths
  - LDPC codes for error correction
  - image registration with Renyi entropy
  - learning salient features for real-time face verification
  - reducing data storage in sequencing amino acids in a protein
  - ...

MST properties

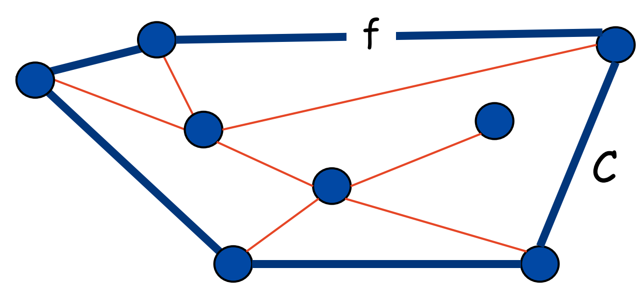
- **Simplifying assumption.** All edge costs c_e are distinct.

- **Cut property.** Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST contains e.

- **Cycle property.** Let C be any cycle, and let f be the max cost edge belonging to C. Then the MST does not contain f.



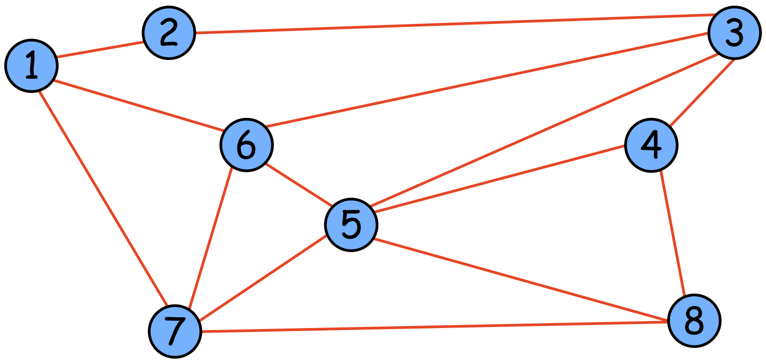
e is in the MST



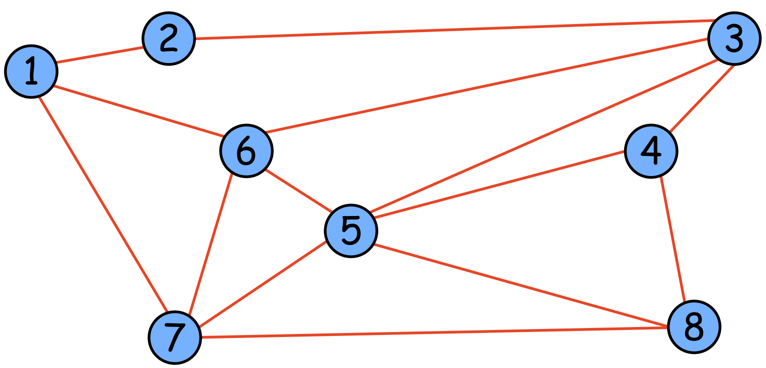
f is not in the MST

Cycles and Cuts

– **Cycle.** Set of edges of the form a-b, b-c, c-d, …, y-z, z-a.



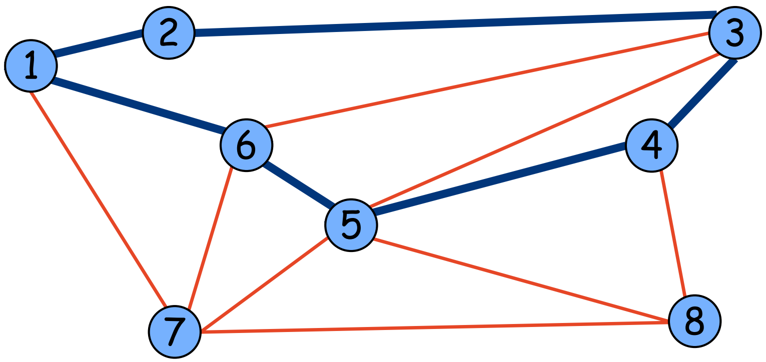
– **Cutset.** A cut is a subset of nodes S. The corresponding cutset D is the subset of edges with exactly one endpoint in S.



Cut S = { 4, 5, 8 }
Cutset D = 5-6, 5-7, 3-4, 3-5, 7-8

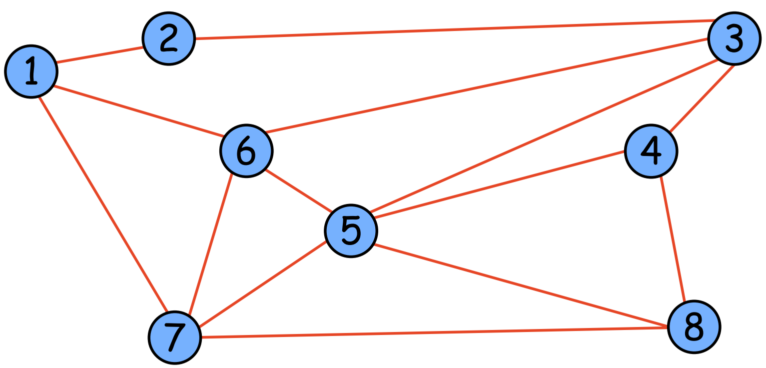
Cycles and Cuts

- **Cycle.** Set of edges of the form a-b, b-c, c-d, …, y-z, z-a.



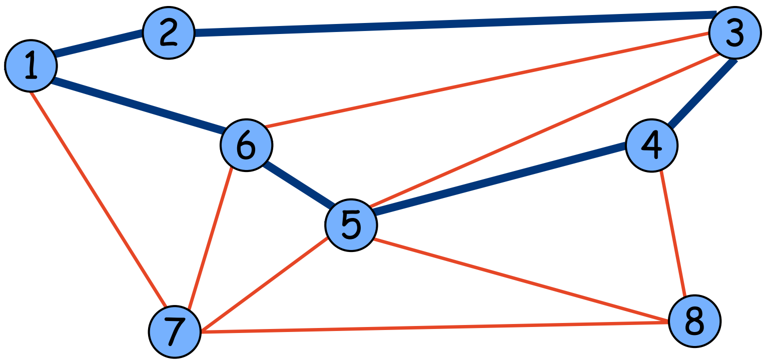
Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1

- **Cutset.** A cut is a subset of nodes S. The corresponding cutset D is the subset of edges with exactly one endpoint in S.



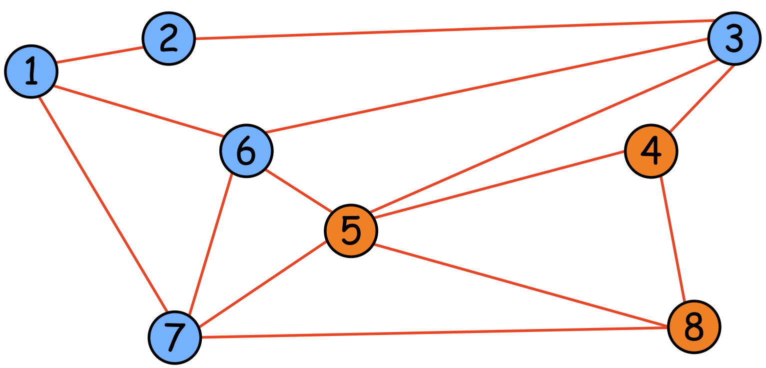
Cycles and Cuts

– **Cycle.** Set of edges of the form a-b, b-c, c-d, …, y-z, z-a.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1

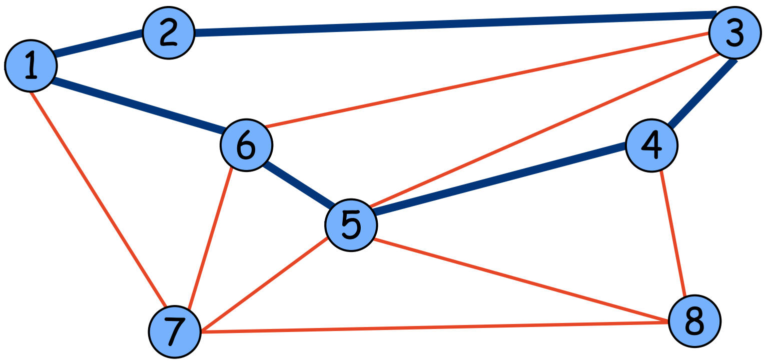
– **Cutset.** A cut is a subset of nodes S. The corresponding cutset D is the subset of edges with exactly one endpoint in S.



Cut S = { 4, 5, 8 }

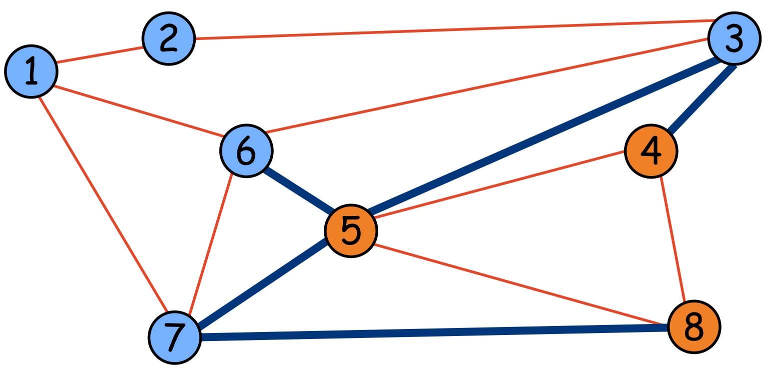
Cycles and Cuts

– **Cycle.** Set of edges of the form a-b, b-c, c-d, …, y-z, z-a.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1

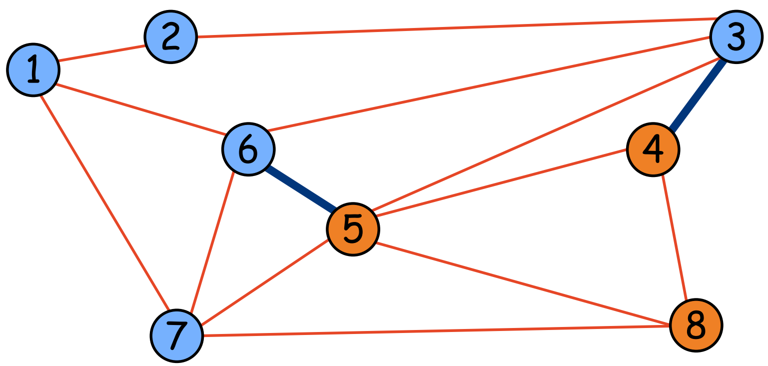
– **Cutset.** A cut is a subset of nodes S. The corresponding cutset D is the subset of edges with exactly one endpoint in S.



Cut S = { 4, 5, 8 }
Cutset D = 5-6, 5-7, 3-4, 3-5, 7-8

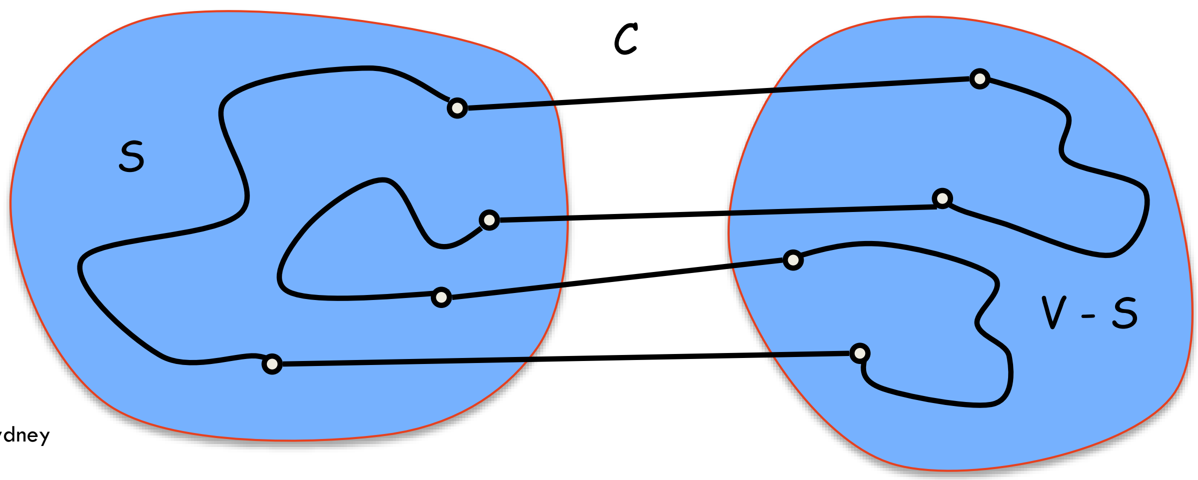
Cycle-Cut Intersection

– **Claim.** A cycle and a cutset intersect in an even number of edges.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1
Cutset D = 3-4, 3-5, 5-6, 5-7, 7-8
Intersection = 3-4, 5-6

– **Proof:** (by picture)

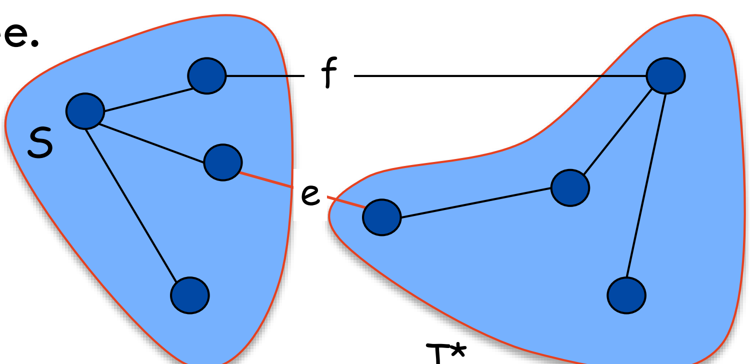


Greedy Algorithms

- **Simplifying assumption.** All edge costs c_e are distinct.

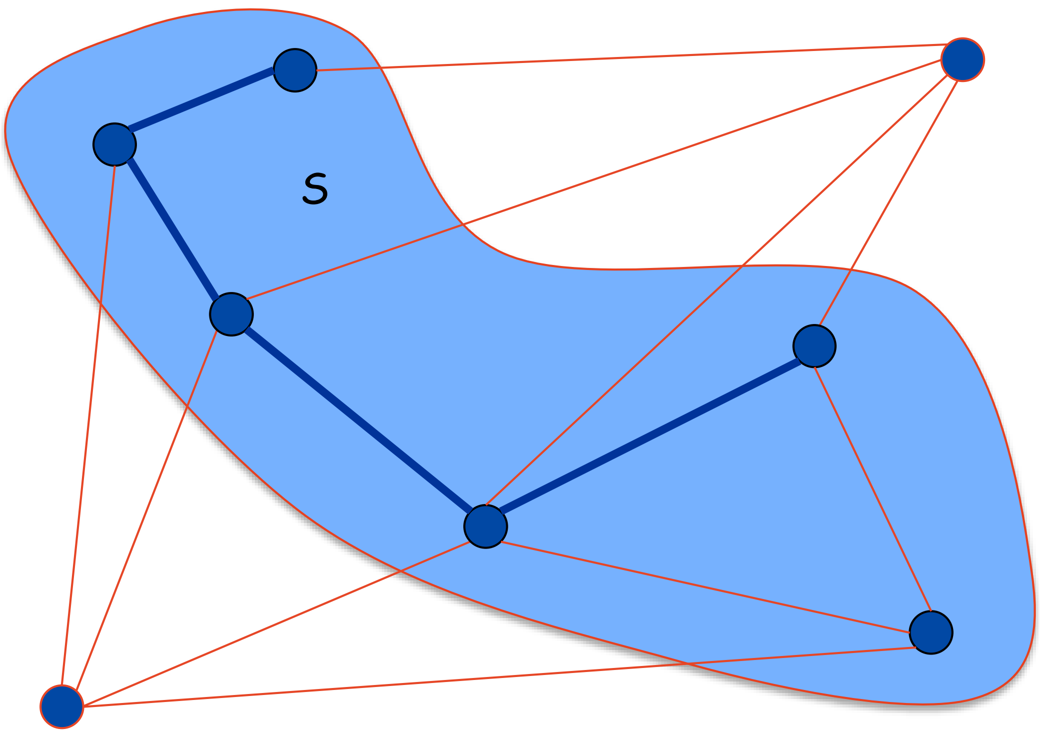
- **Cut property.** Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST T* contains e.

- **Proof:** (exchange argument)
  - Suppose e does not belong to T*, and let's see what happens.
  - Adding e to T* creates a cycle C in T*.
  - Edge e is both in the cycle C and in the cutset D corresponding to S $\Rightarrow$ there exists another edge, say f, that is in both C and D.
  - $T' = T^* \cup \{ e \} - \{ f \}$ is also a spanning tree.
  - Since $c_e < c_f$, cost(T') < cost(T*).
  - This is a contradiction. ■

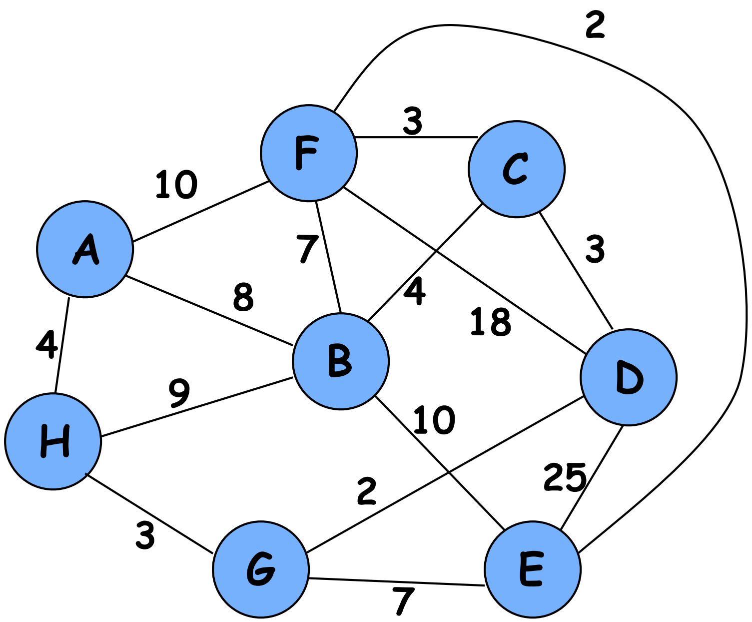


Prim's Algorithm

- Prim's algorithm. [Jarník 1930, Dijkstra 1957, Prim 1959]
  - Initialize S = any node.
  - Apply cut property to S.
  - Add min cost edge in cutset corresponding to S to T, and add one new explored node u to S.



Walk-Through



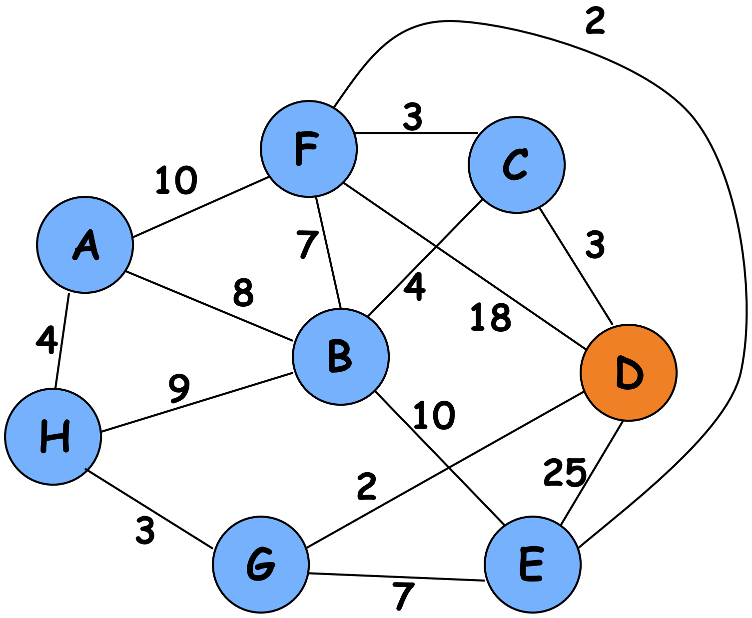
Initialize array

	S	d_v	p_v
A	F	∞	–
B	F	∞	–
C	F	∞	–
D	F	∞	–
E	F	∞	–
F	F	∞	–
G	F	∞	–
H	F	∞	–

Set S

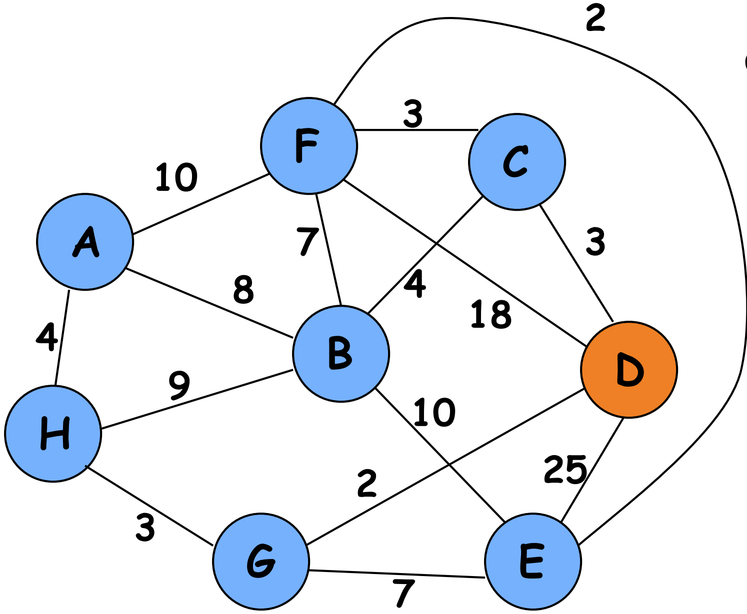
Min distance to S

Closest vertex in S



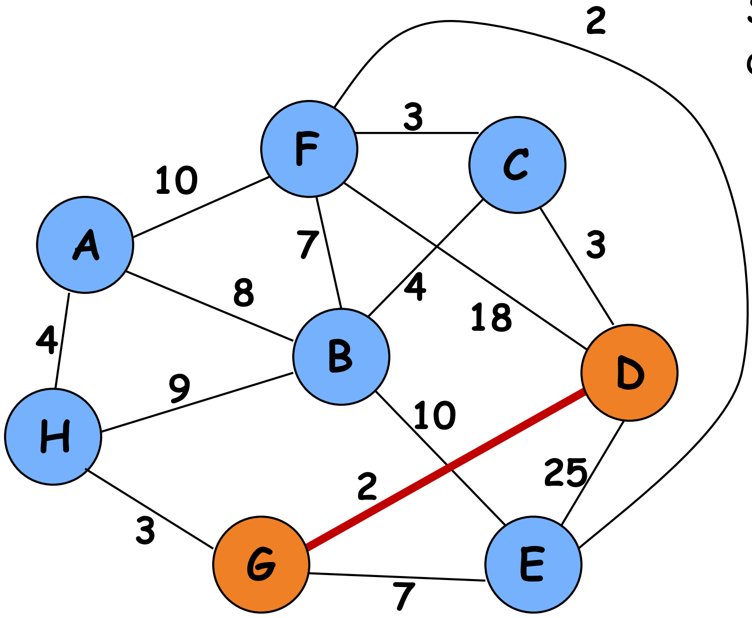
Start with any node, say D

	S	d_v	p_v
A			
B			
C			
D	T	0	–
E			
F			
G			
H			



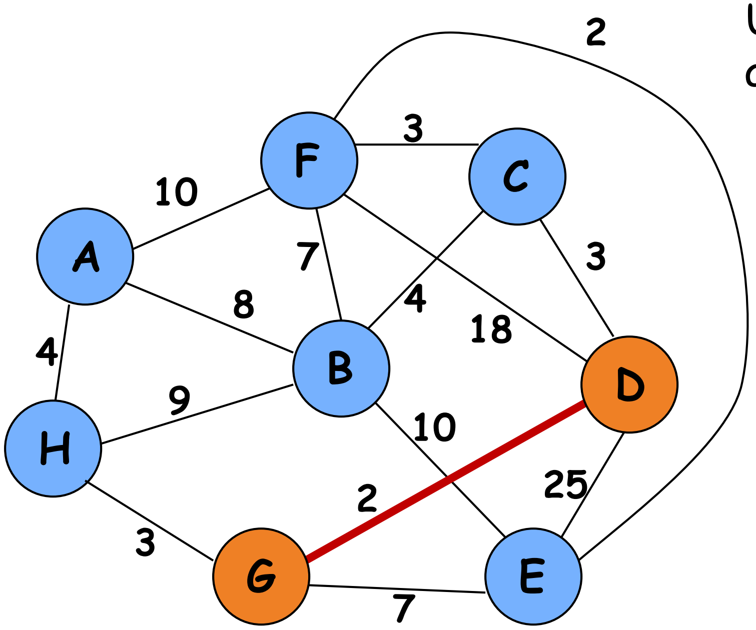
Update distances of adjacent, unselected nodes

	S	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		25	D
F		18	D
G		2	D
H			



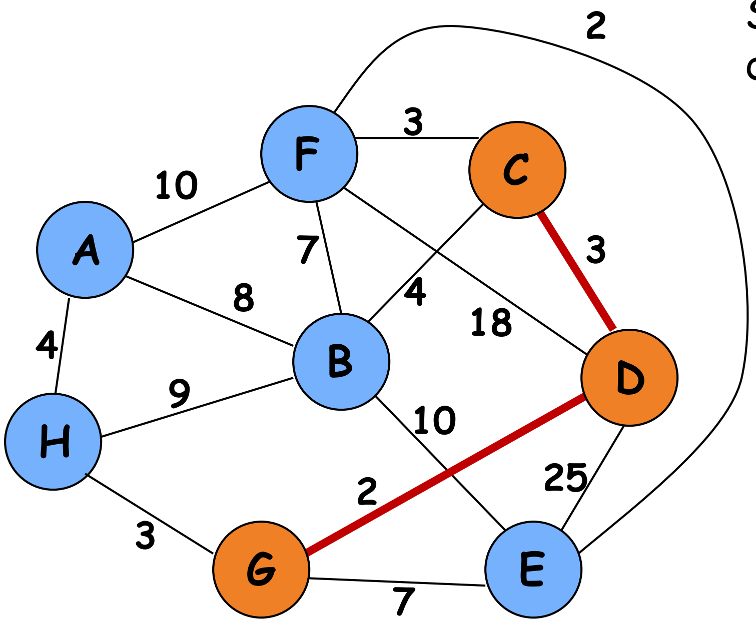
Select node with minimum distance

	S	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		25	D
F		18	D
G	T	2	D
H			



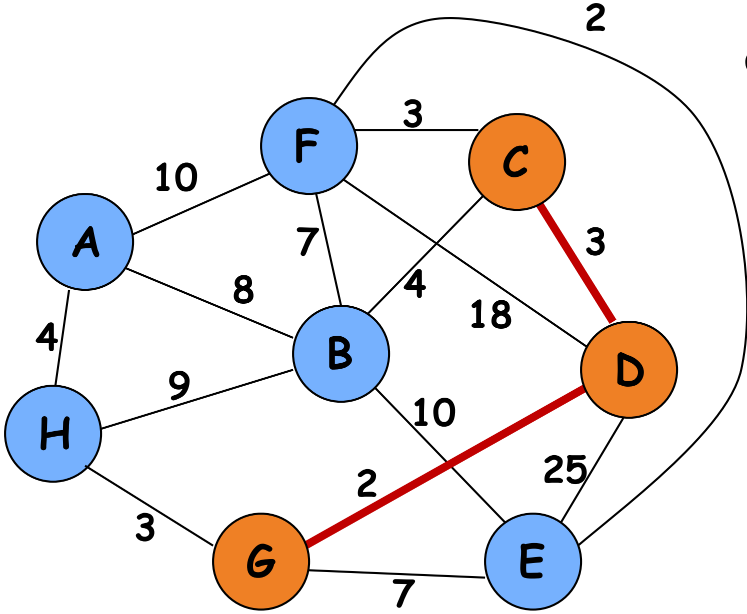
Update distances of adjacent, unselected nodes

	S	d_v	p_v
A			
B			
C		3	D
D	T	0	–
E		7	G
F		18	D
G	T	2	D
H		3	G



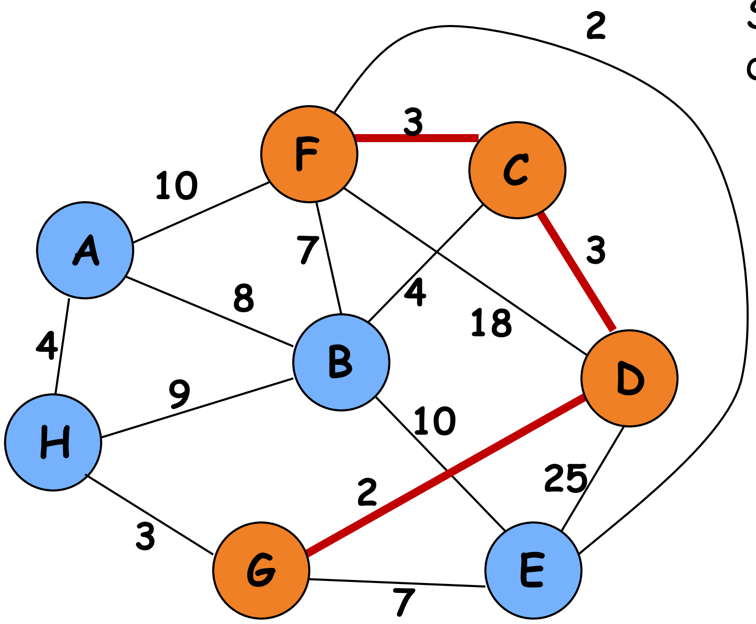
Select node with minimum distance

	S	d_v	p_v
A			
B			
C	T	3	D
D	T	0	–
E		7	G
F		18	D
G	T	2	D
H		3	G



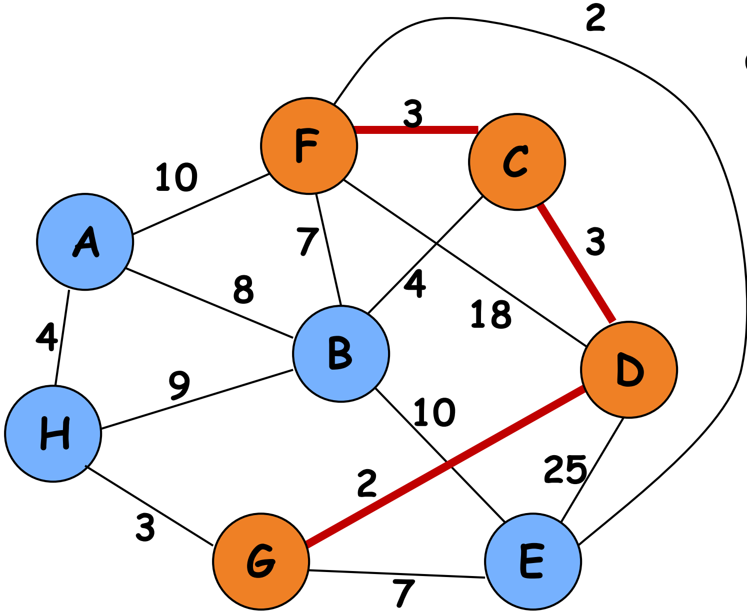
Update distances of
adjacent, unselected nodes

	S	d_v	p_v
A			
B		4	C
C	T	3	D
D	T	0	–
E		7	G
F		3	C
G	T	2	D
H		3	G



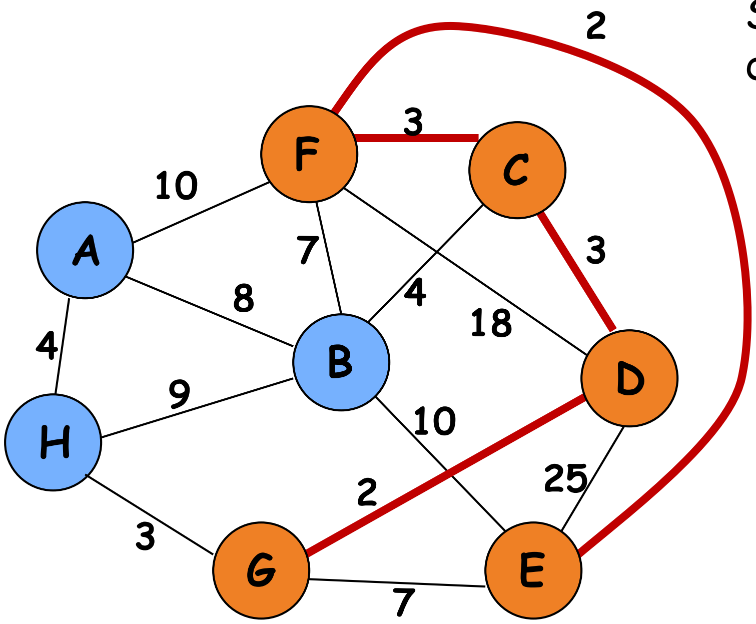
Select node with minimum distance

	S	d_v	p_v
A			
B		4	C
C	T	3	D
D	T	0	–
E		7	G
F	T	3	C
G	T	2	D
H		3	G



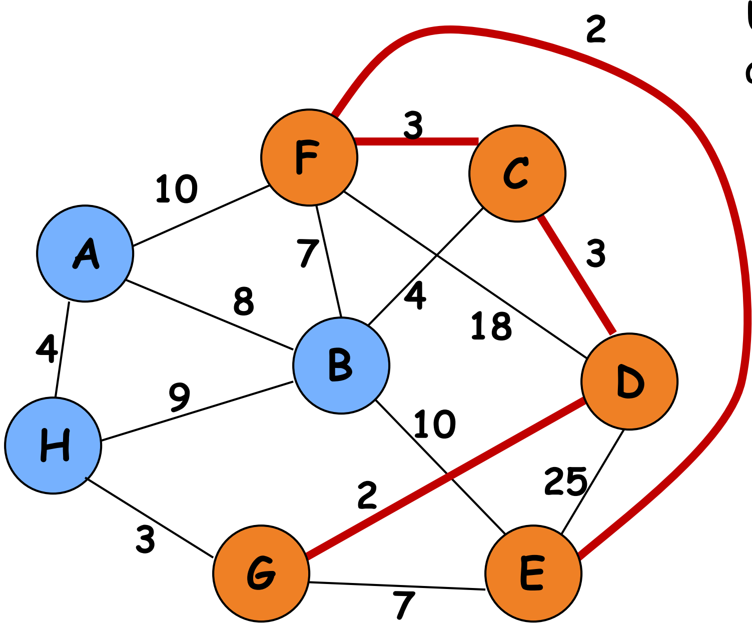
Update distances of adjacent, unselected nodes

	S	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E		2	F
F	T	3	C
G	T	2	D
H		3	G



Select node with minimum distance

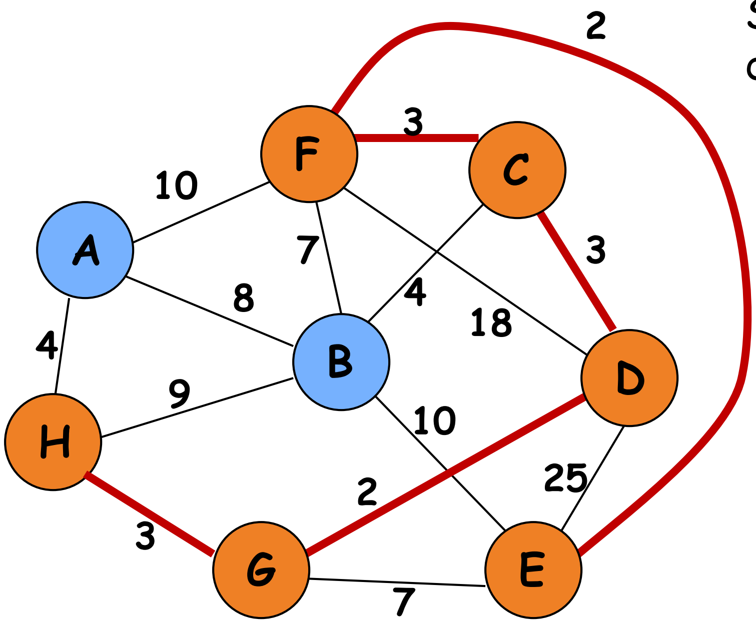
	S	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G



Update distances of adjacent, unselected nodes

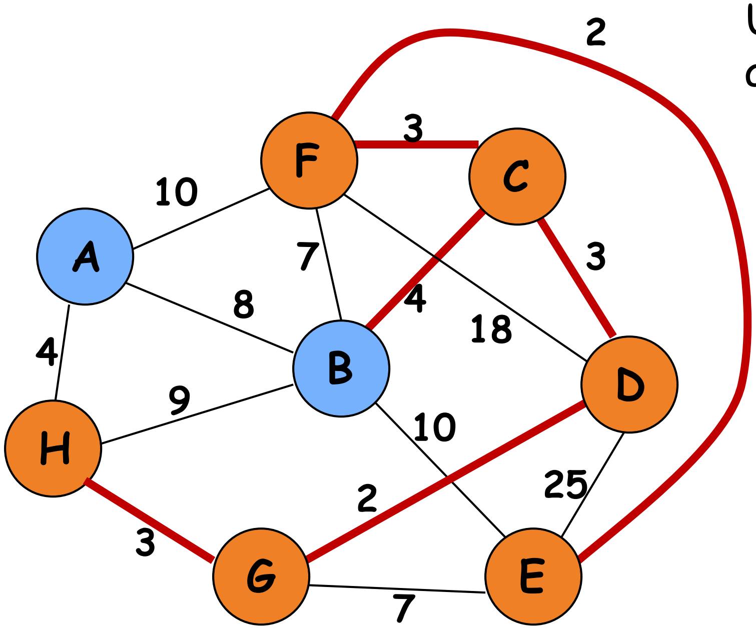
	S	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G

Table entries unchanged



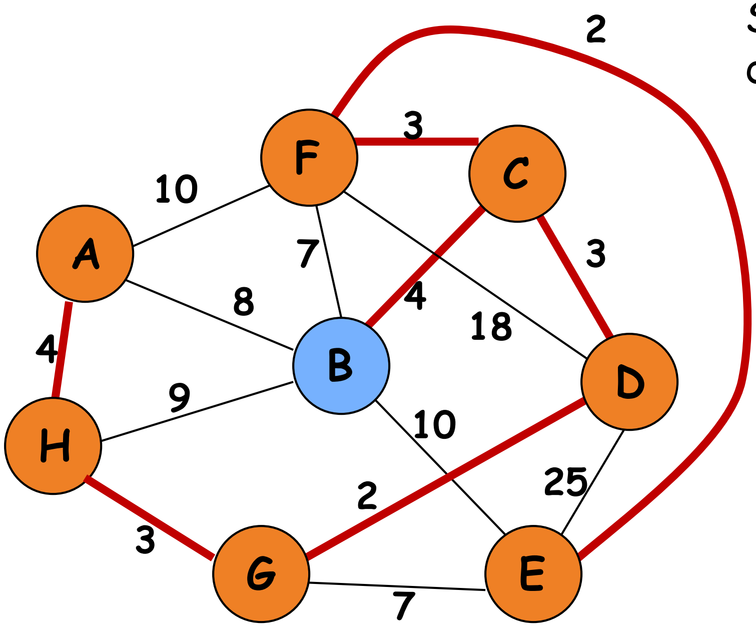
Select node with minimum distance

	S	d_v	p_v
A		10	F
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



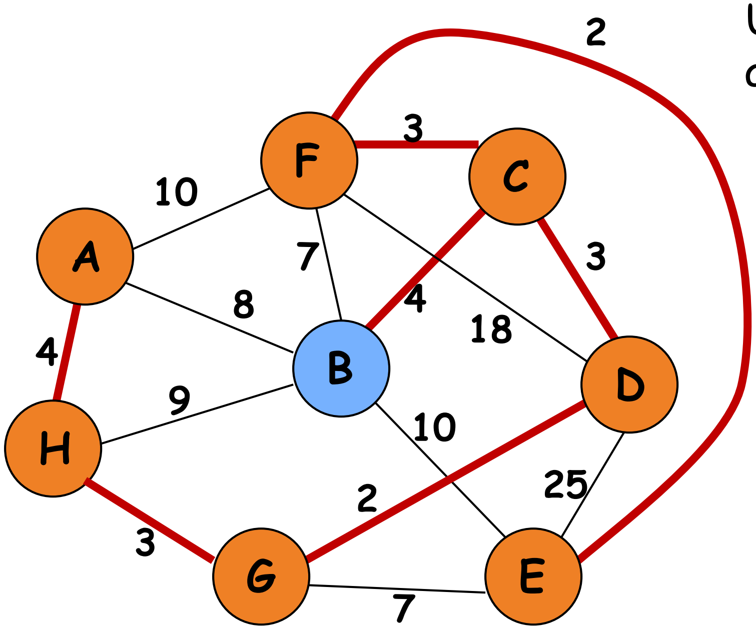
Update distances of adjacent, unselected nodes

	S	d_v	p_v
A		4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Select node with minimum distance

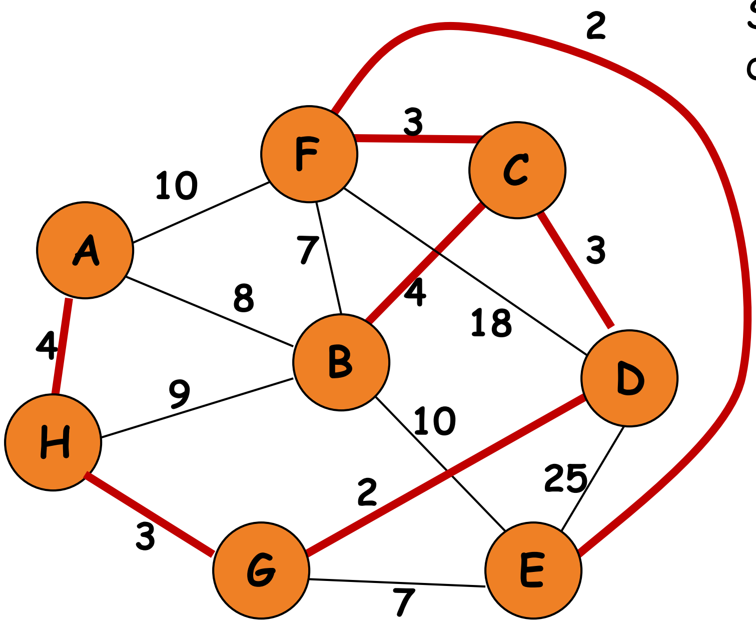
	S	d_v	p_v
A	T	4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Update distances of adjacent, unselected nodes

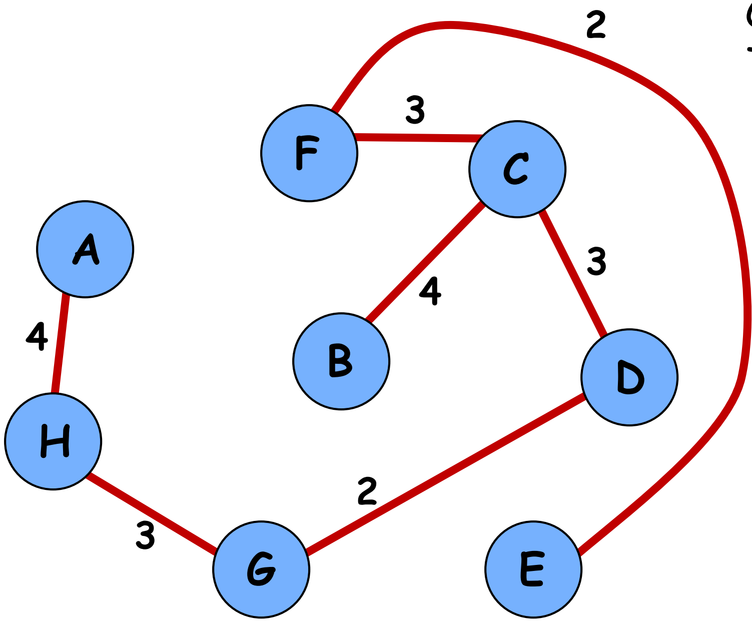
	S	d_v	p_v
A	S	4	H
B		4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Table entries unchanged



Select node with minimum distance

	S	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G



Cost of Minimum Spanning Tree = $\Sigma\, d_v$ = 21

	S	d_v	p_v
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	–
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Done!

Implementation: Prim's Algorithm

- Implementation. Use a priority queue as in Dijkstra's algorithm.
  - Maintain set of explored nodes S.
  - For each unexplored node v, maintain attachment cost a[v] = cost of cheapest edge v to a node in S.
  - O(n^2) with an array; O(m log n) with a binary heap.

```
Prim(G, c) {
   foreach (v ∈ V) d[v] ← ∞
   Initialize an empty priority queue Q
   foreach (v ∈ V) insert v onto Q
   Initialize set of explored nodes S ← φ

   while (Q is not empty) {
      u ← delete min element from Q
      S ← S ∪ { u }
      foreach (edge e = (u, v) incident to u)
          if ((v ∉ S) and (c_e < d[v]))
             decrease priority d[v] to c_e
}
```

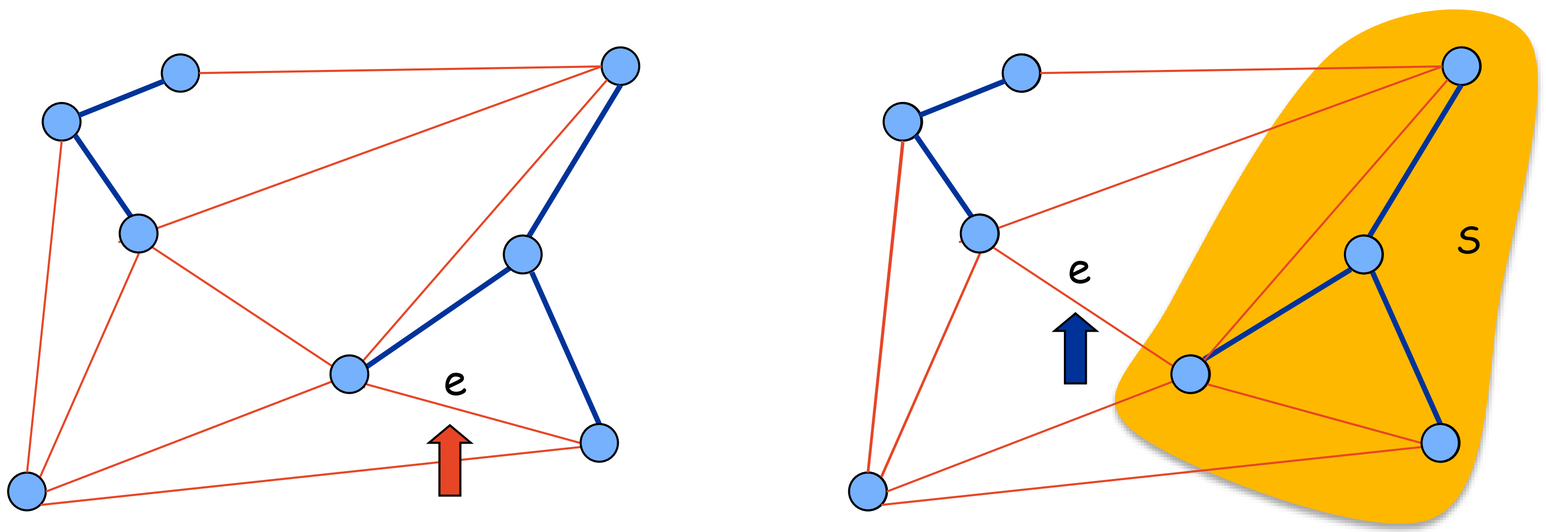
Kruskal's Algorithm

Kruskal's algorithm. [Kruskal, 1956]

Consider edges in ascending order of weight.

Case 1: If adding e to T creates a cycle, discard e according to cycle property.

Case 2: Otherwise, insert e = (u, v) into T according to cut property where S = set of nodes in u's connected component.



Case 1

Case 2

Lexicographic Tiebreaking

To remove the assumption that all edge costs are distinct: perturb all edge costs by tiny amounts to break any ties.

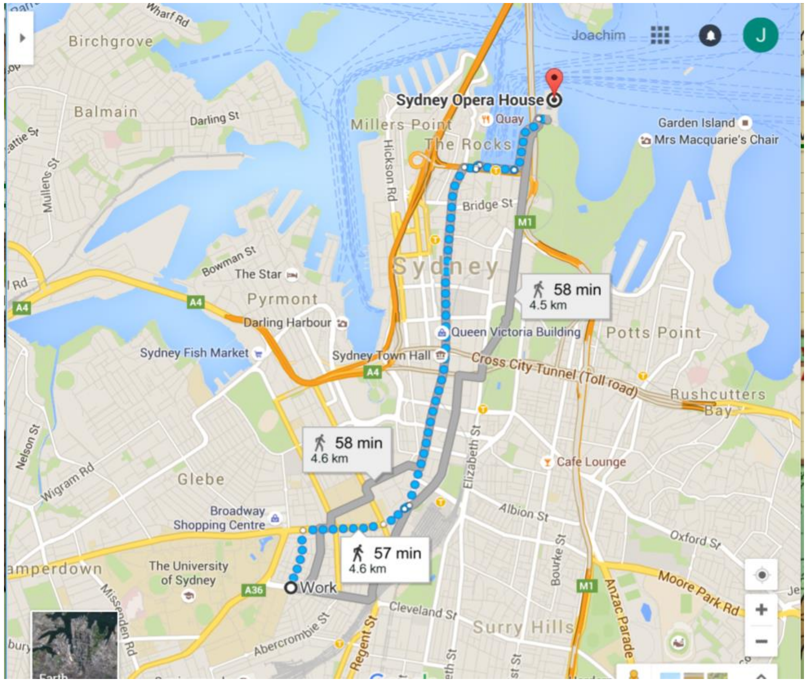
Impact. Kruskal and Prim only interact with costs via pairwise comparisons. If perturbations are sufficiently small, MST with perturbed costs is MST with original costs.

e.g., if all edge costs are integers, perturbing cost of edge e_i by i / n^2

```
boolean less(i, j) {
   if       (cost(e_i) < cost(e_j)) return true
   else if  (cost(e_i) > cost(e_j)) return false
   else if  (i < j)                 return true
   else                             return false
}
```

Implementation. Can handle arbitrarily small perturbations implicitly by breaking ties lexicographically, according to index.

Shortest Paths in a Graph



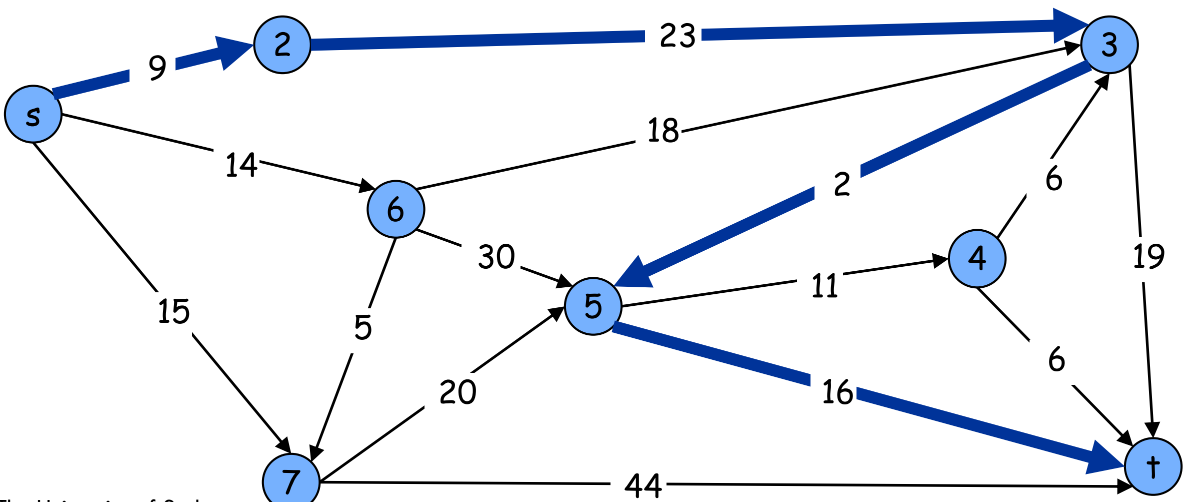
Shortest path from SIT to the Sydney Opera house

Shortest Path Problem

– Shortest path network.
  – Directed graph $G = (V, E)$.
  – Source s, destination t.
  – Length ℓ_e = length of edge e.

– Shortest path problem: find shortest directed path from s to t.

cost of path = sum of edge costs in path



Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 50

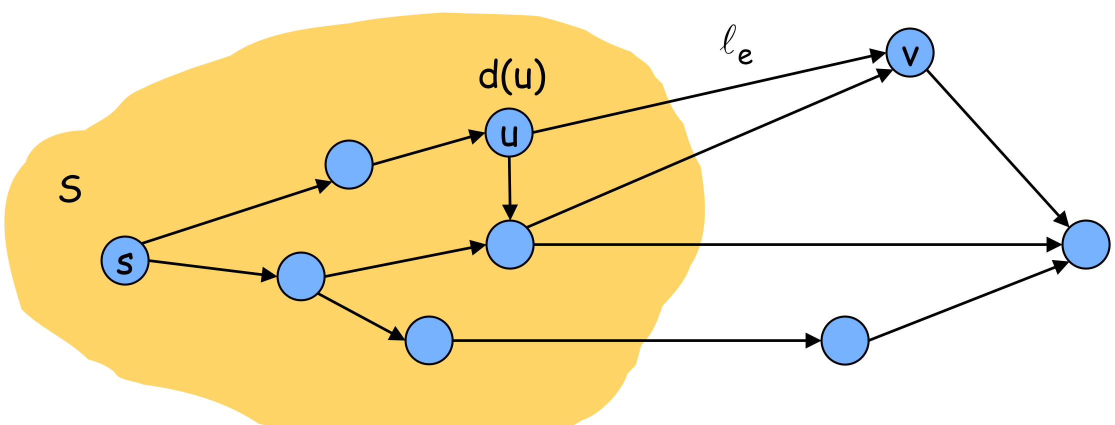
Dijkstra's Algorithm

– Dijkstra's algorithm.
  – Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
  – Initialize S = { s }, d(s) = 0.
  – Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e ,$$

add v to S, and set d(v) = π(v).

shortest path to some u in explored part, followed by a single edge (u, v)



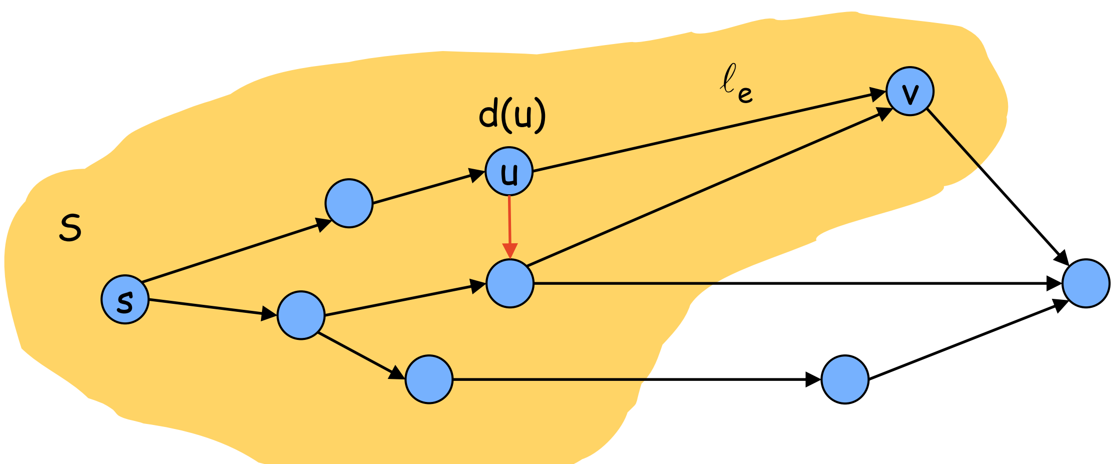
Dijkstra's Algorithm

- Dijkstra's algorithm.
  - Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
  - Initialize S = { s }, d(s) = 0.
  - Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e ,$$

add v to S, and set d(v) = π(v).

shortest path to some u in explored part, followed by a single edge (u, v)



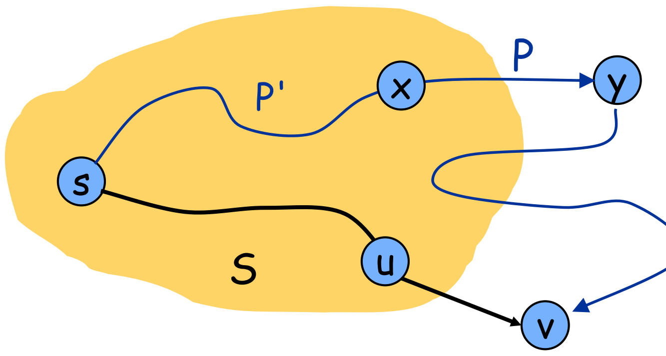
Dijkstra's Algorithm: Proof of Correctness

- **Invariant:** For each node $u \in S$, $d(u)$ is the length of the shortest s-u path.
- **Proof:** (by induction on $|S|$)

  Base case: $|S| = 1$ is trivial.

  Inductive hypothesis: Assume true for $|S| = k \geq 1$.

  - Let v be next node added to S, and let u-v be the chosen edge.
  - The shortest s-u path plus (u, v) is an s-v path of length $\pi(v)$.
  - Consider any s-v path P. We'll see that it's no shorter than $\pi(v)$.
  - Let x-y be the first edge in P that leaves S, and let P' be the subpath to x.
  - P is already too long as soon as it leaves S.



$$\ell(P) = \underbrace{\ell(P') + \ell(x,y) \geq d(x) + \ell(x,y)}_{\text{inductive hypothesis}} \underbrace{\geq \pi(y)}_{\text{defn of } \pi(y)} \underbrace{\geq \pi(v)}_{\text{Dijkstra chose v instead of y}}$$

Dijkstra's Algorithm: Implementation

- For each unexplored node, explicitly maintain

$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e .$$

  - Next node to explore = node with minimum $\pi(v)$.
  - When exploring v, for each incident edge e = (v, w), update

$$\pi(w) = \min \{ \ \pi(w), \ \pi(v) + \ell_e \}.$$

- Efficient implementation. Maintain a priority queue of unexplored nodes, prioritized by $\pi(v)$.

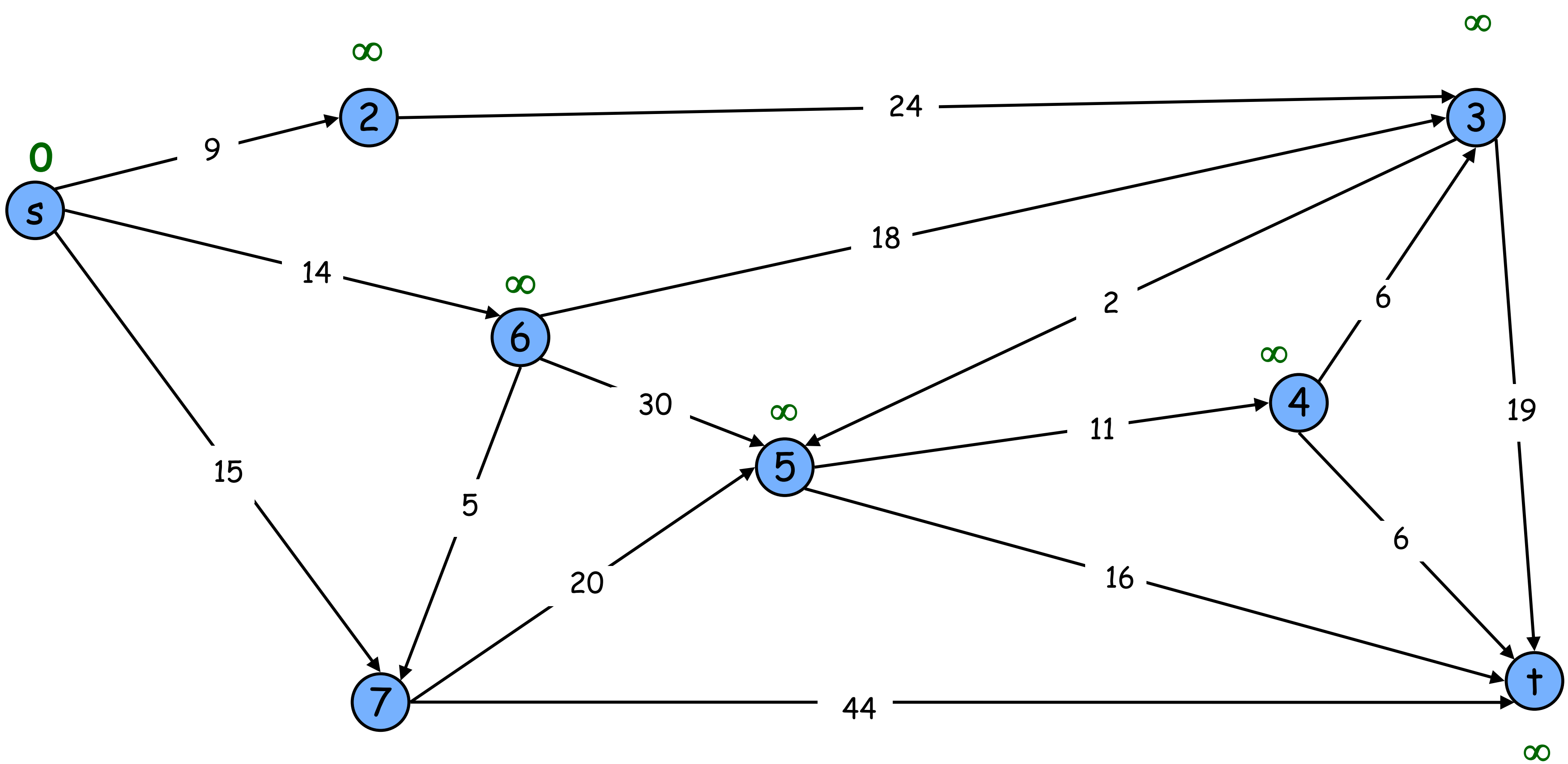
		Priority Queue			
PQ Operation	Dijkstra	Array	Binary heap	d-way Heap	Fib heap †
Insert	n	n	log n	$d \log_d n$	1
ExtractMin	n	n	log n	$d \log_d n$	log n
ChangeKey	m	1	log n	$\log_d n$	1
IsEmpty	n	1	1	1	1
Total		n^2	m log n	$m \log_{m/n} n$	m + n log n

† Individual ops are amortized bounds

Dijkstra's Shortest Path Algorithm

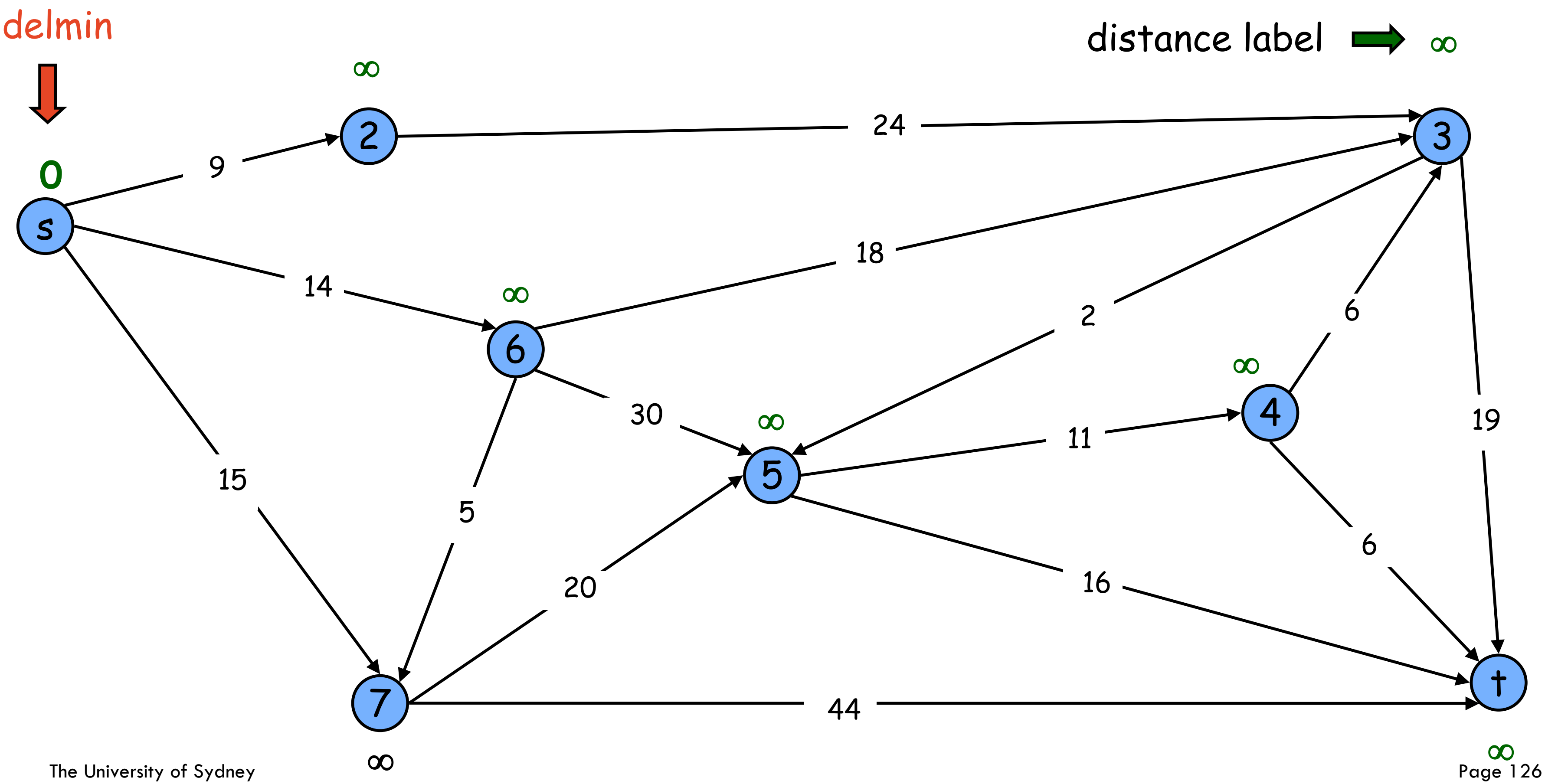
S = { }
PQ = { s, 2, 3, 4, 5, 6, 7, t }

distance label ➡ ∞



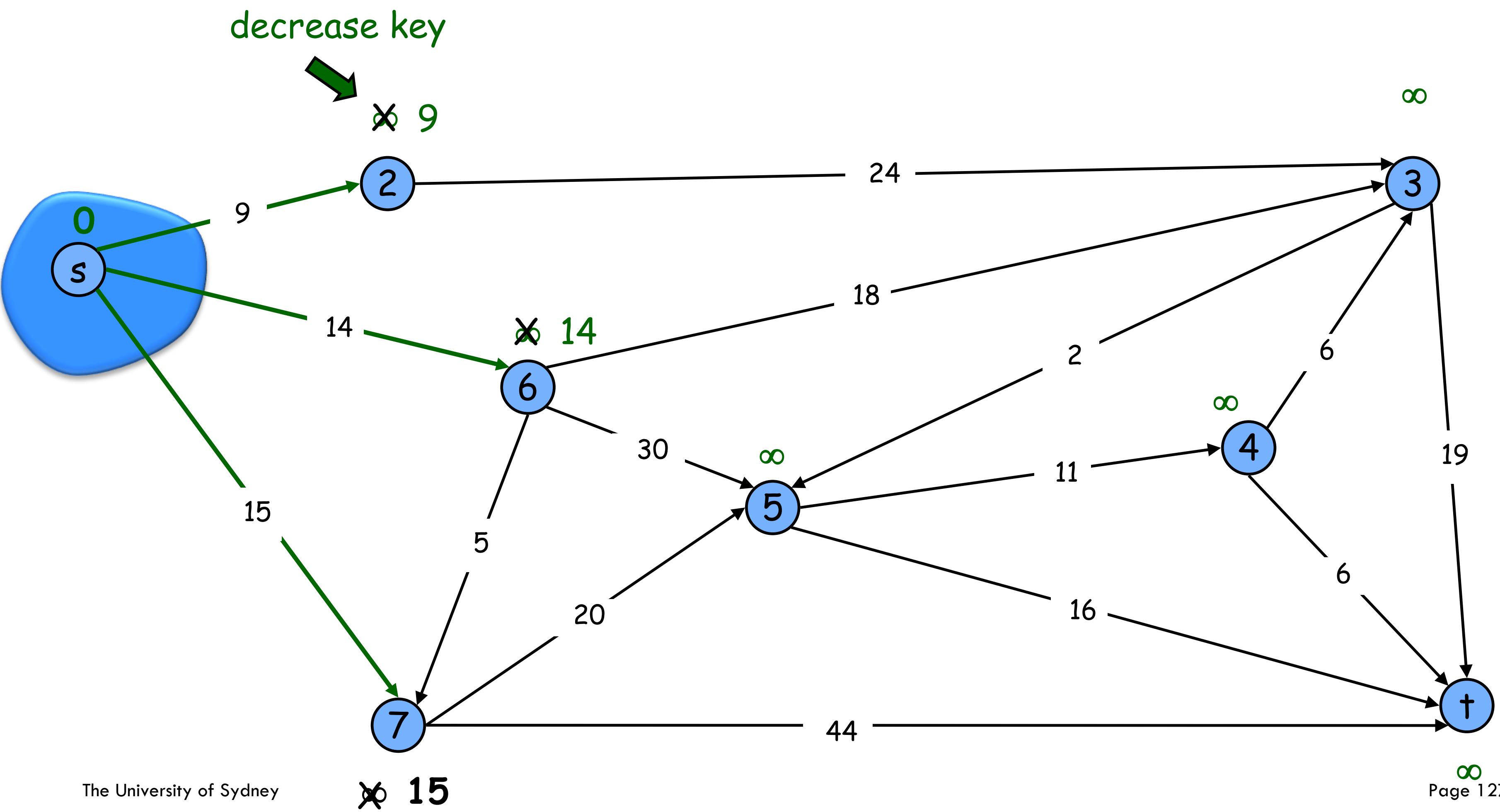
Dijkstra's Shortest Path Algorithm

S = { }
PQ = { s, 2, 3, 4, 5, 6, 7, t }



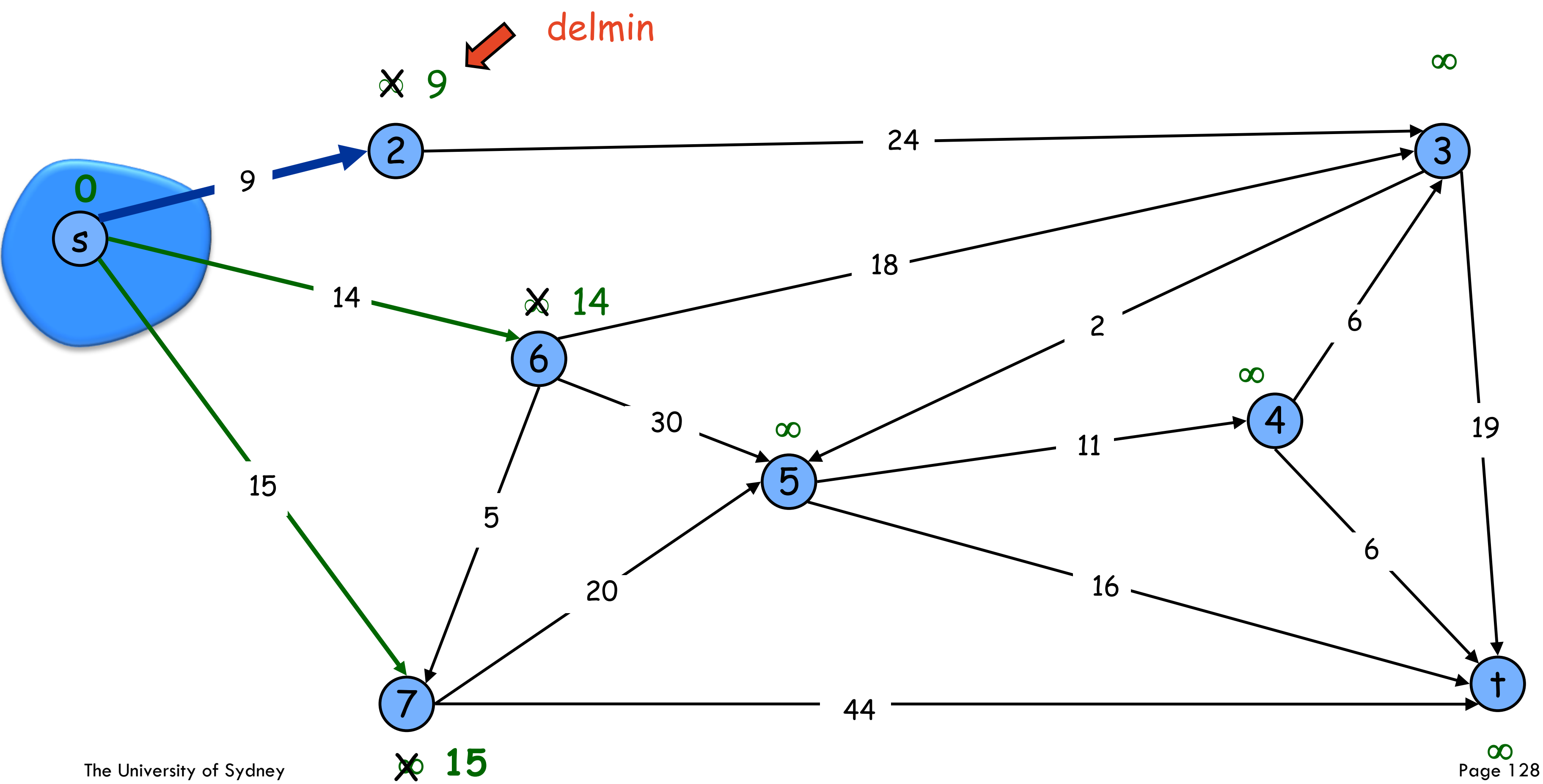
Dijkstra's Shortest Path Algorithm

S = { s }
PQ = { 2, 3, 4, 5, 6, 7, t }



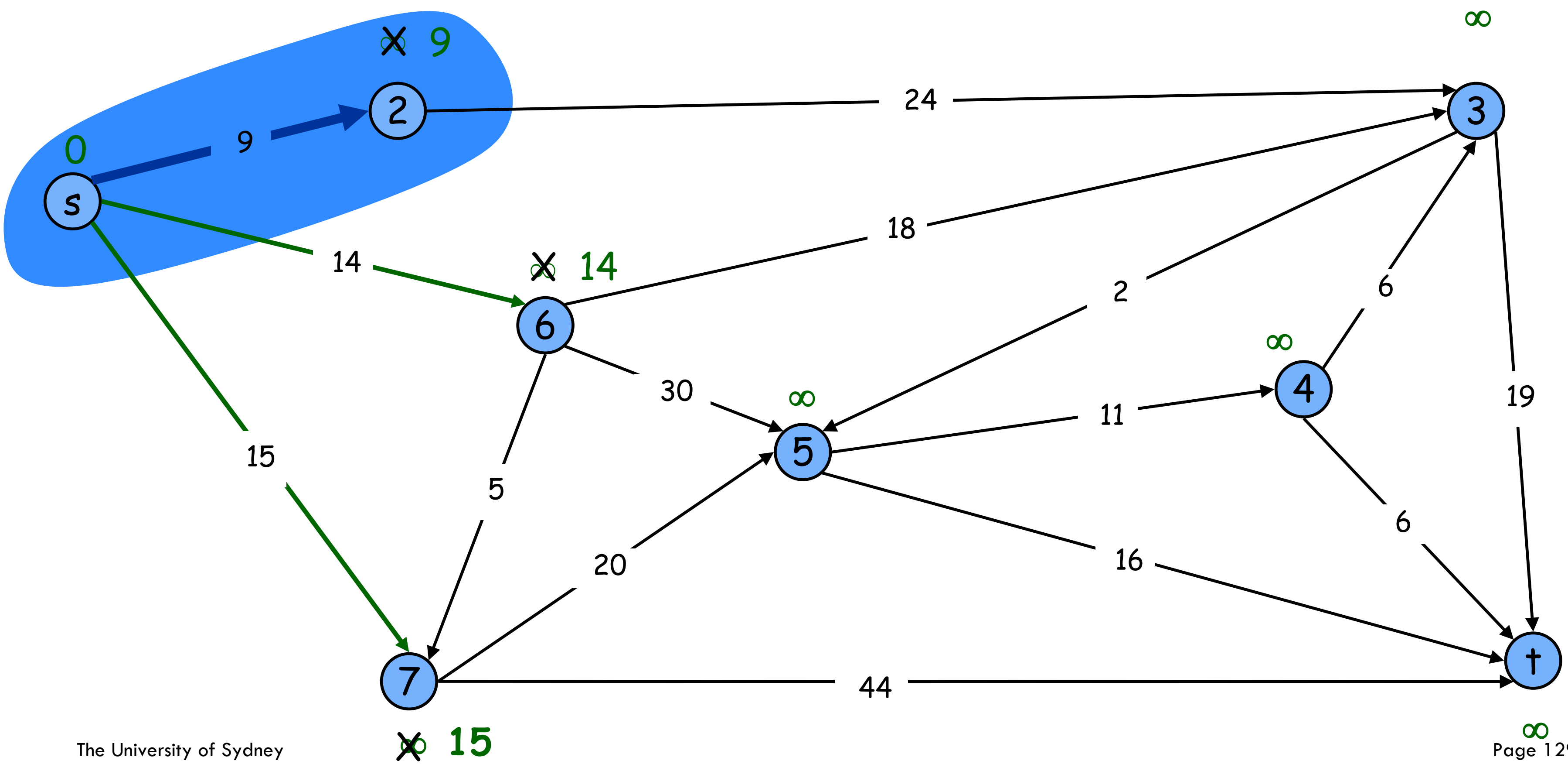
Dijkstra's Shortest Path Algorithm

S = { s }
PQ = { 2, 3, 4, 5, 6, 7, t }



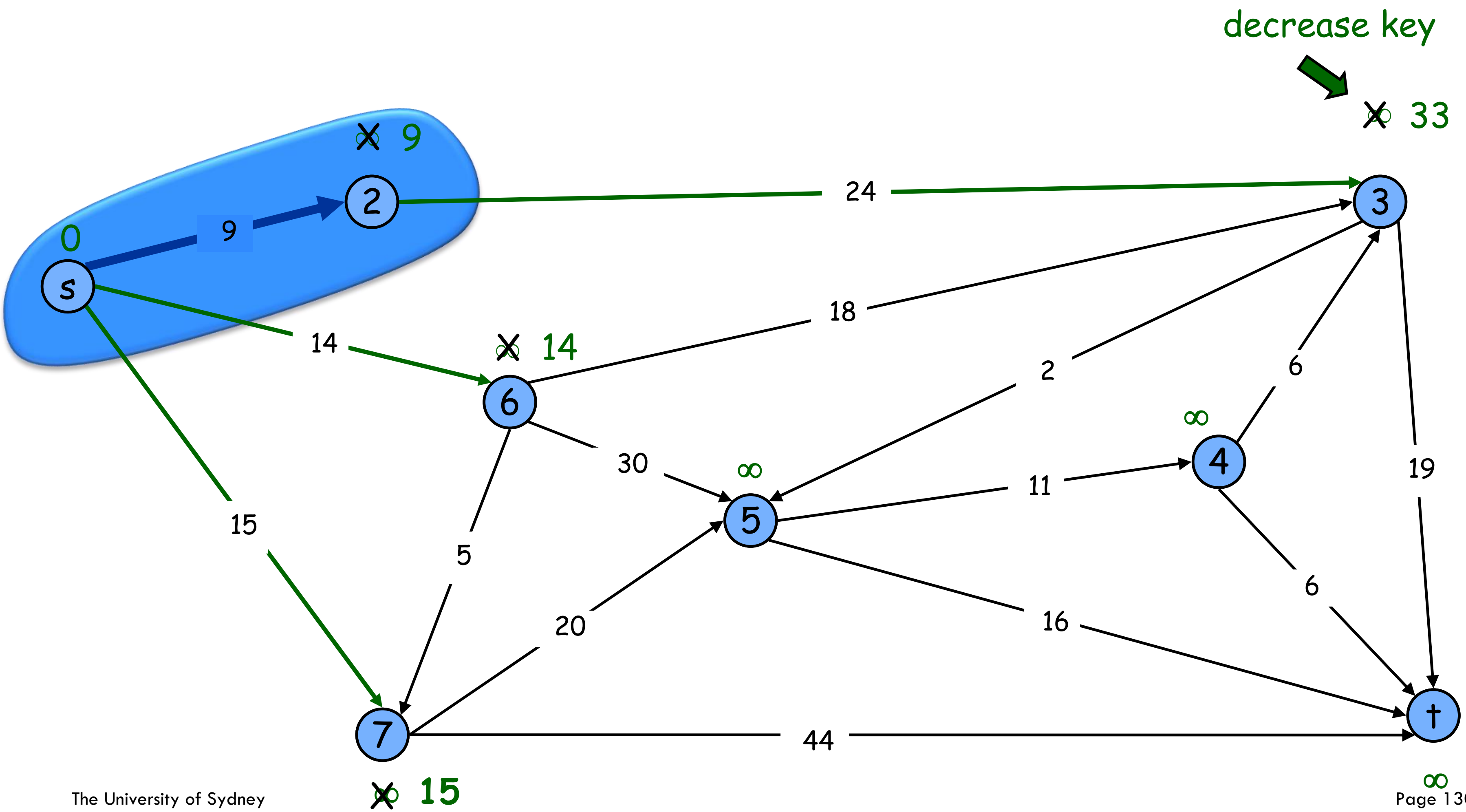
Dijkstra's Shortest Path Algorithm

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



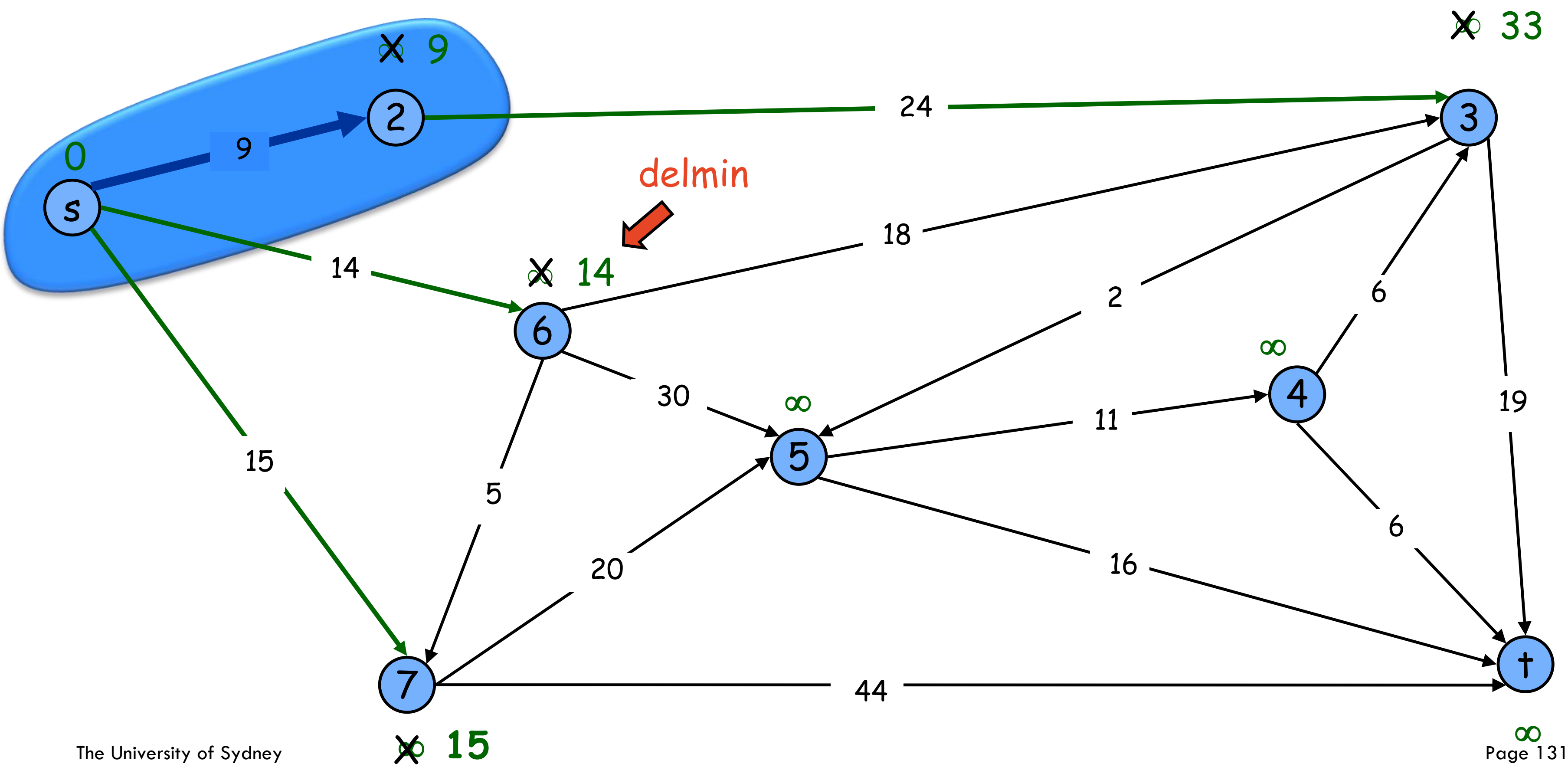
Dijkstra's Shortest Path Algorithm

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



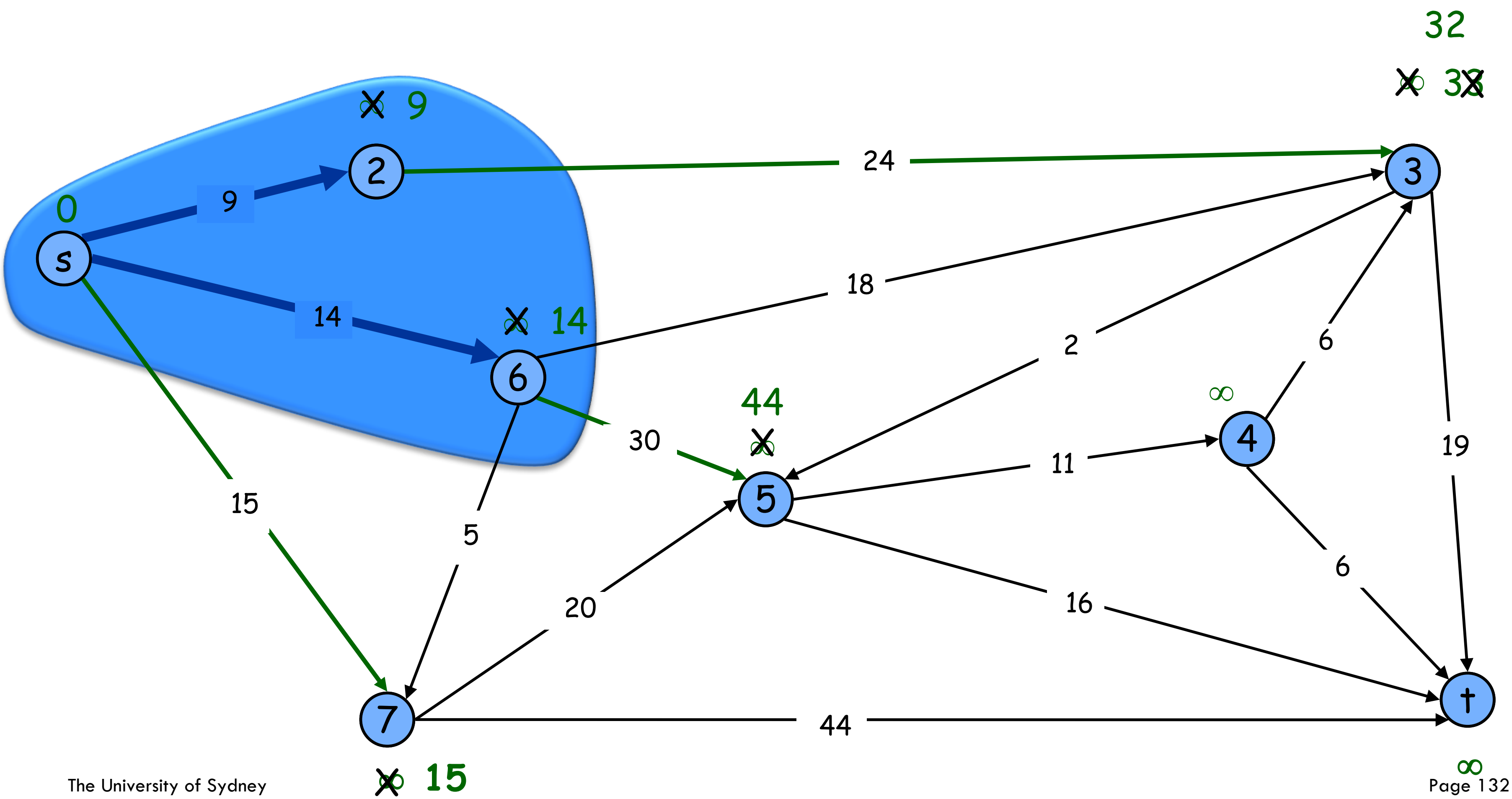
Dijkstra's Shortest Path Algorithm

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



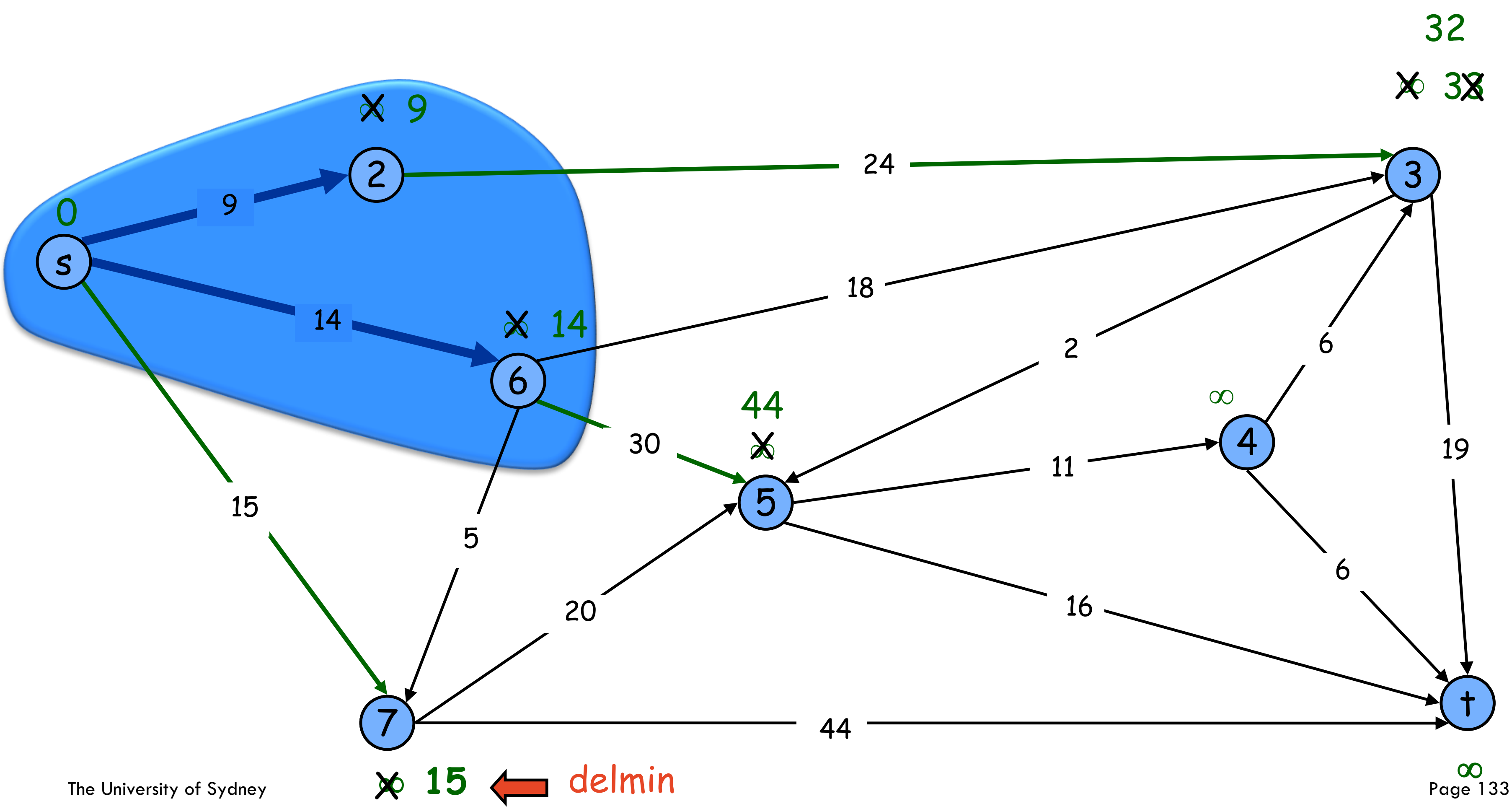
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }



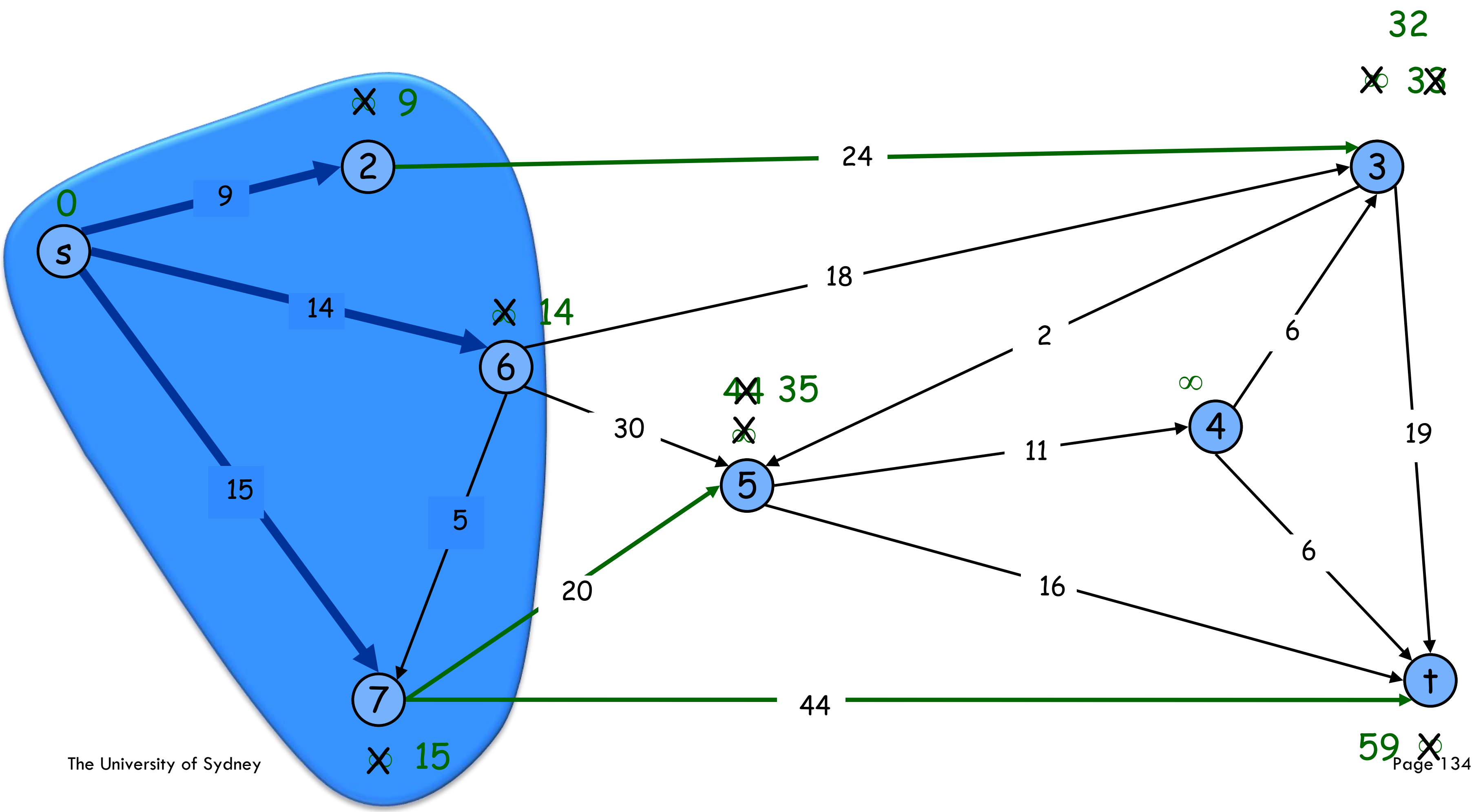
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }



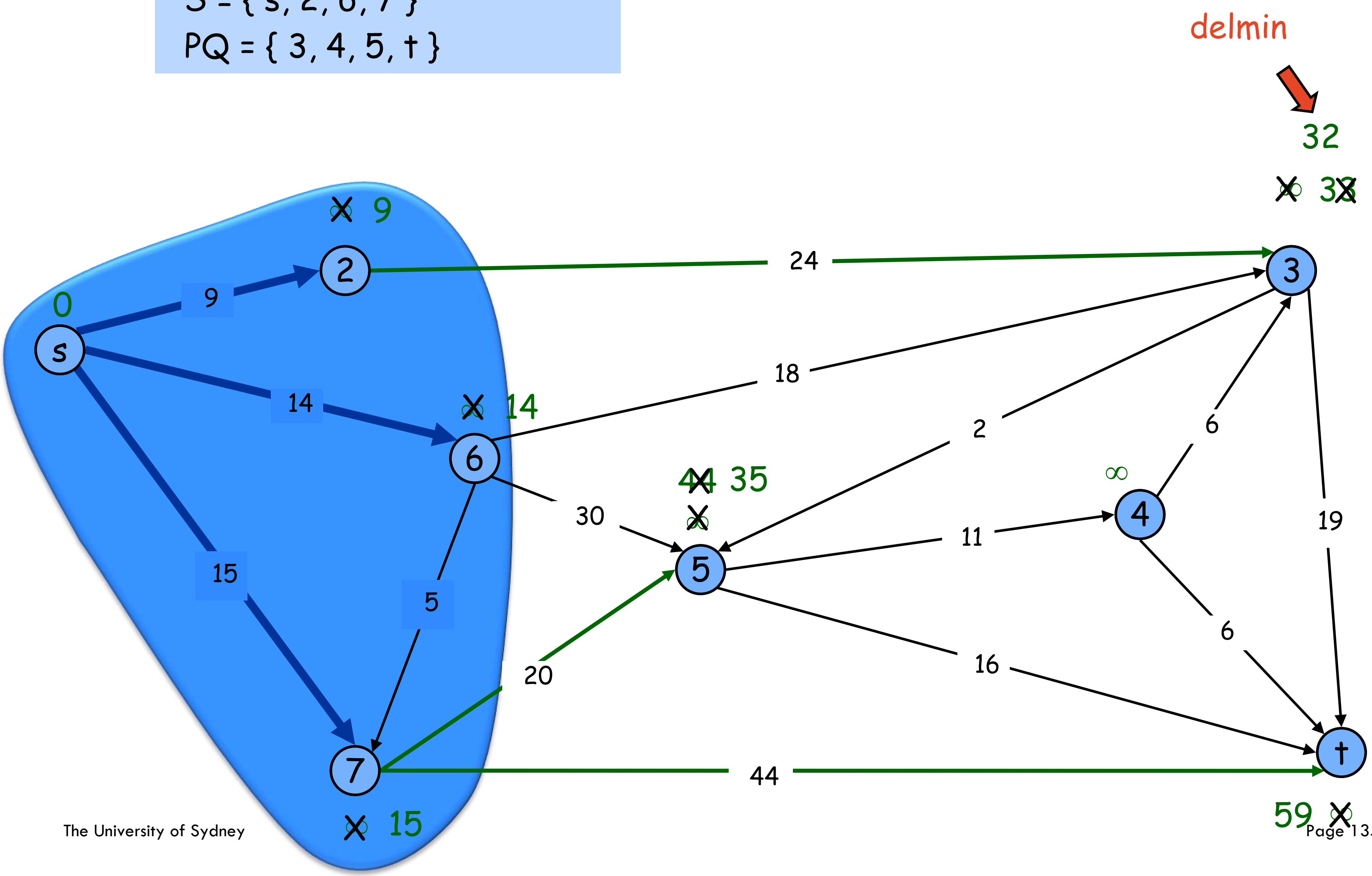
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6, 7 }
PQ = { 3, 4, 5, t }



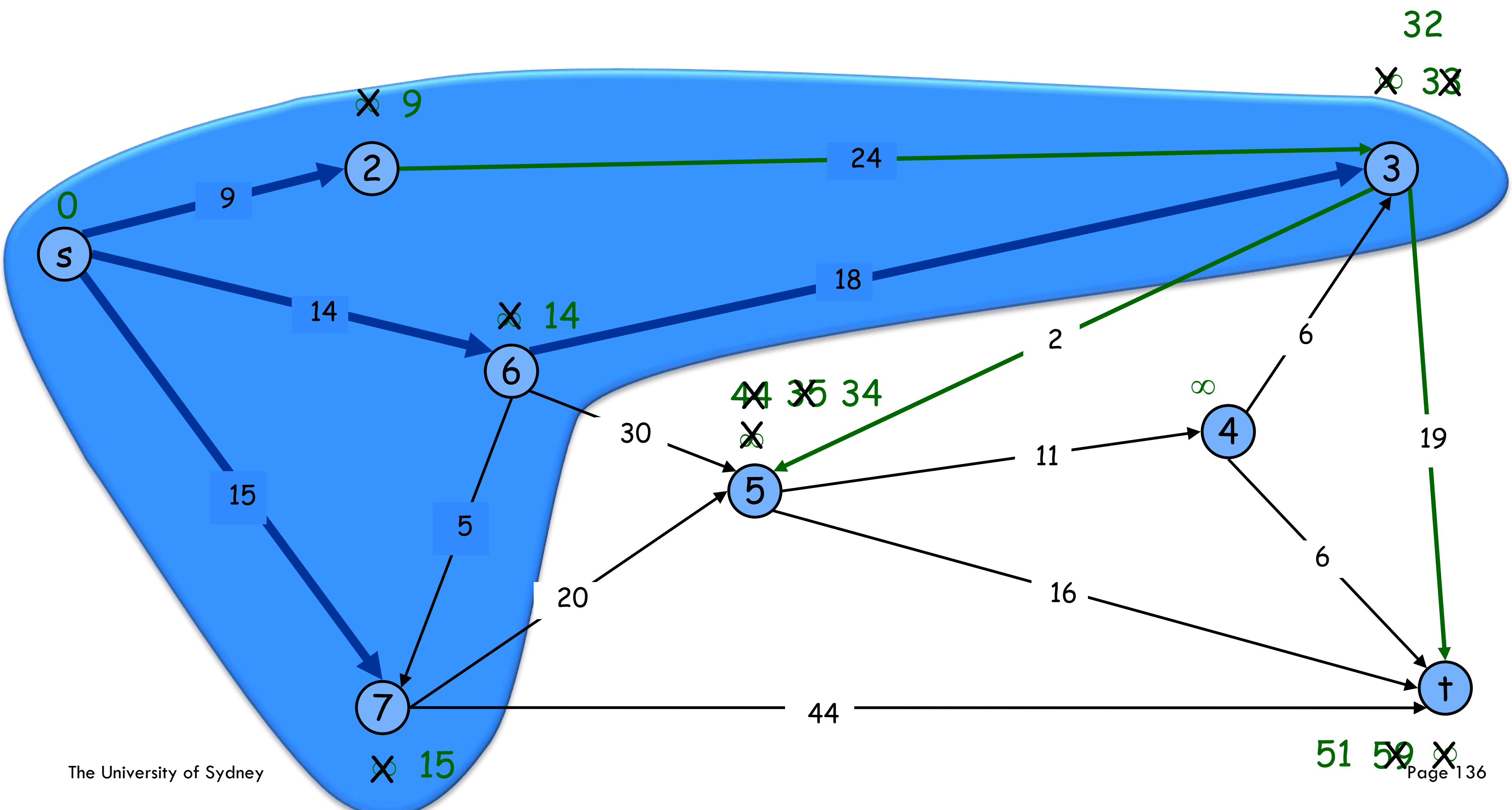
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6, 7 }
PQ = { 3, 4, 5, t }



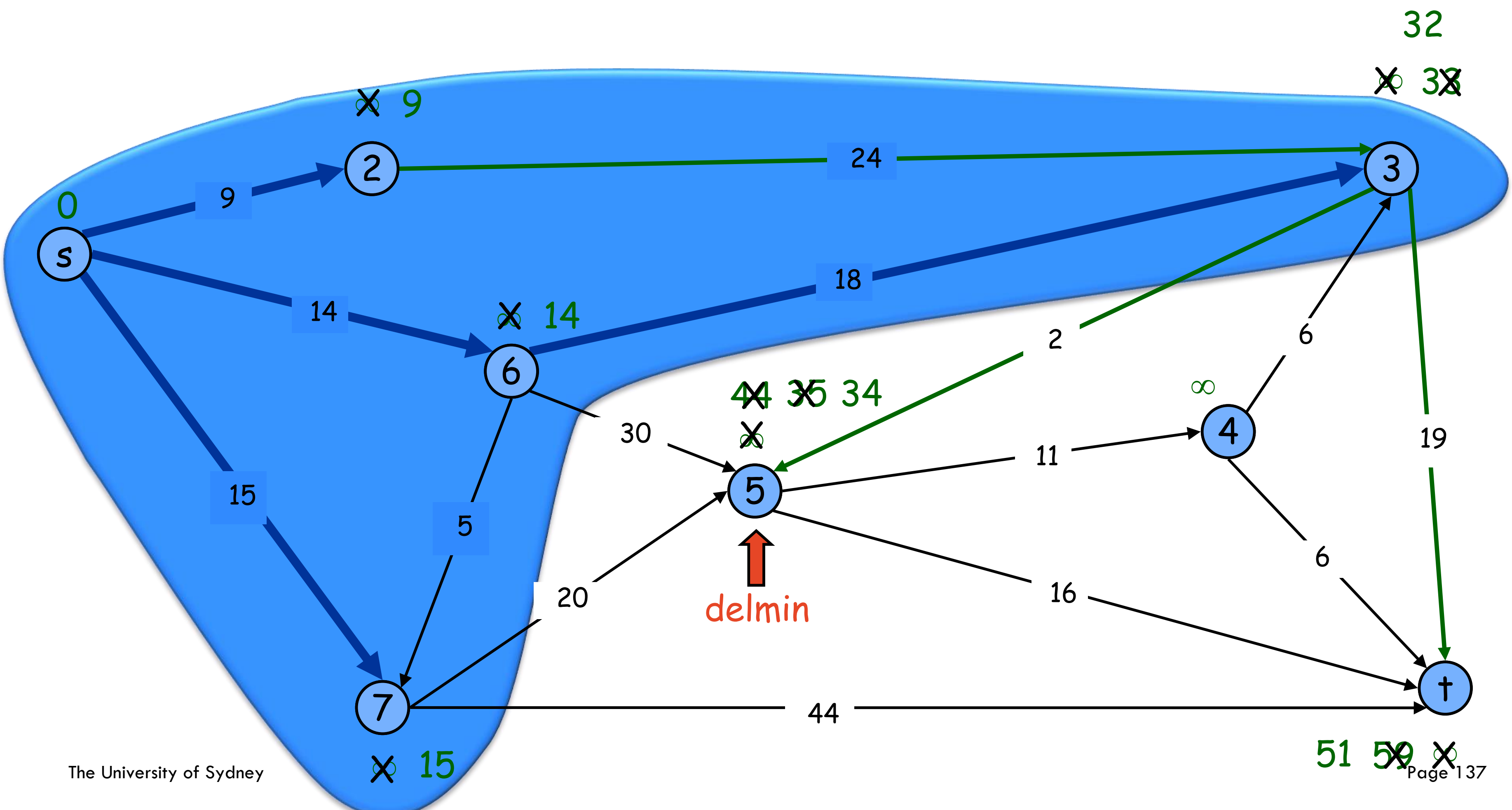
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 6, 7 }
PQ = { 4, 5, t }



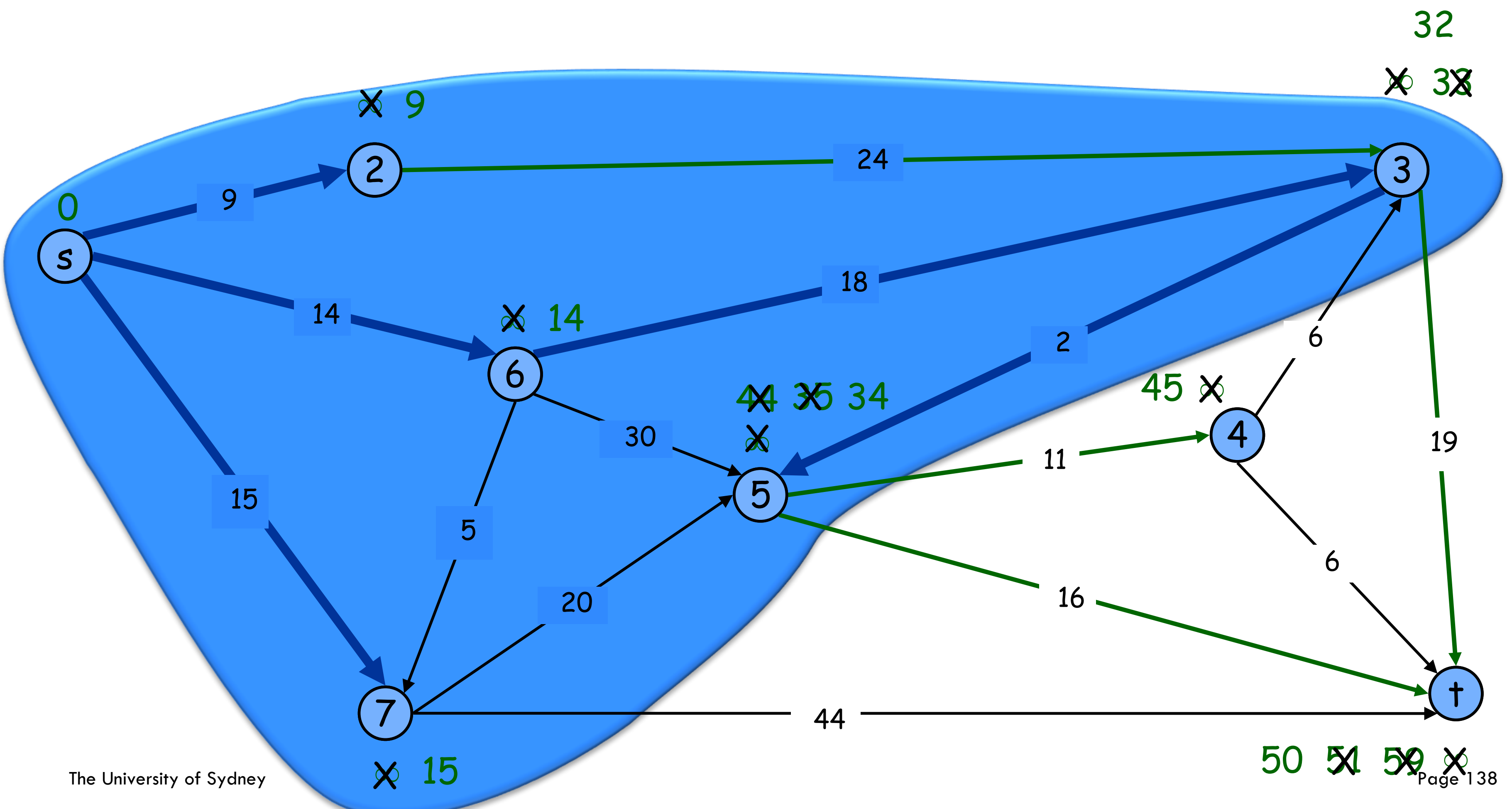
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 6, 7 }
PQ = { 4, 5, t }



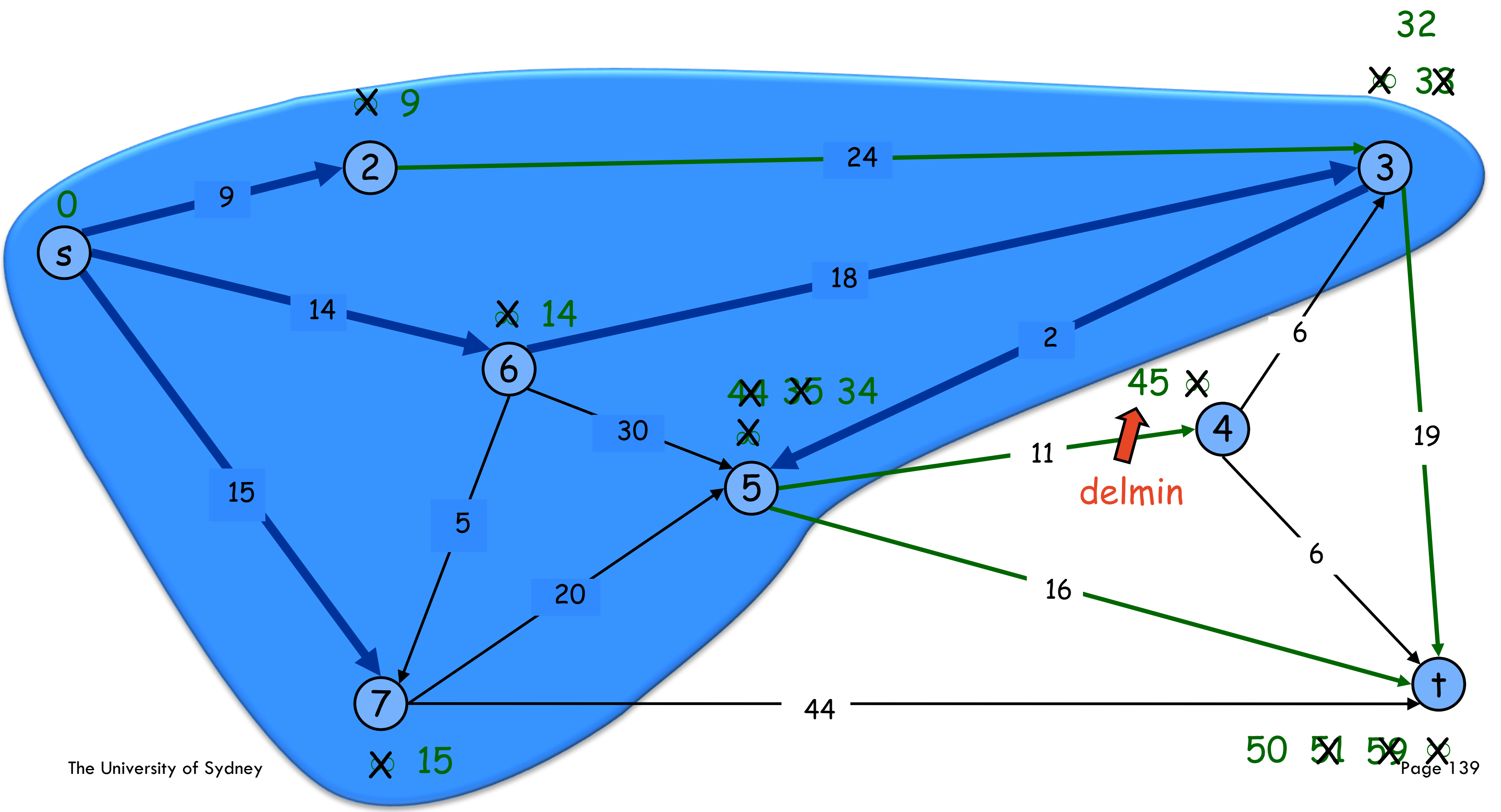
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 5, 6, 7 }
PQ = { 4, t }



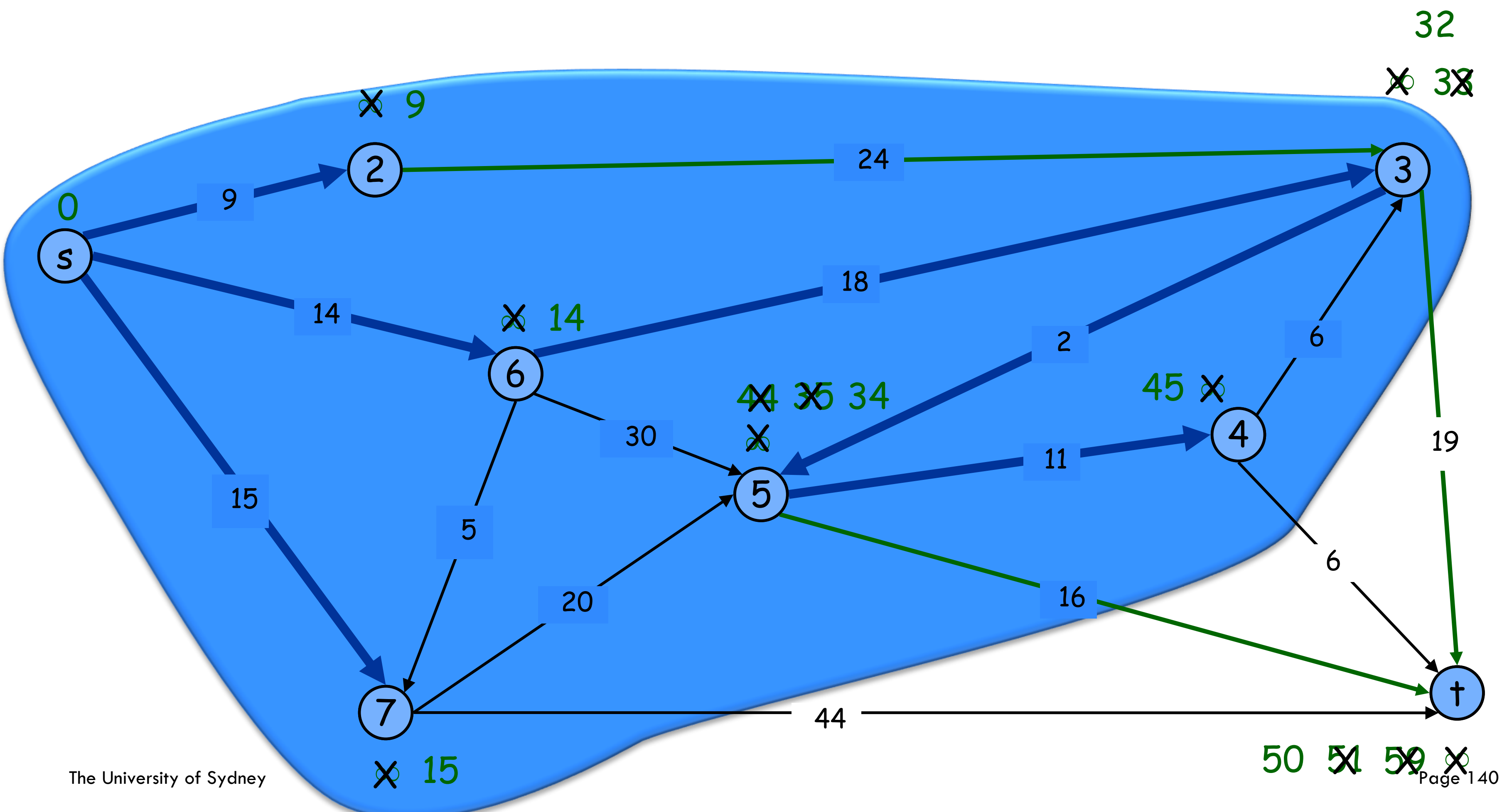
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 5, 6, 7 }
PQ = { 4, t }



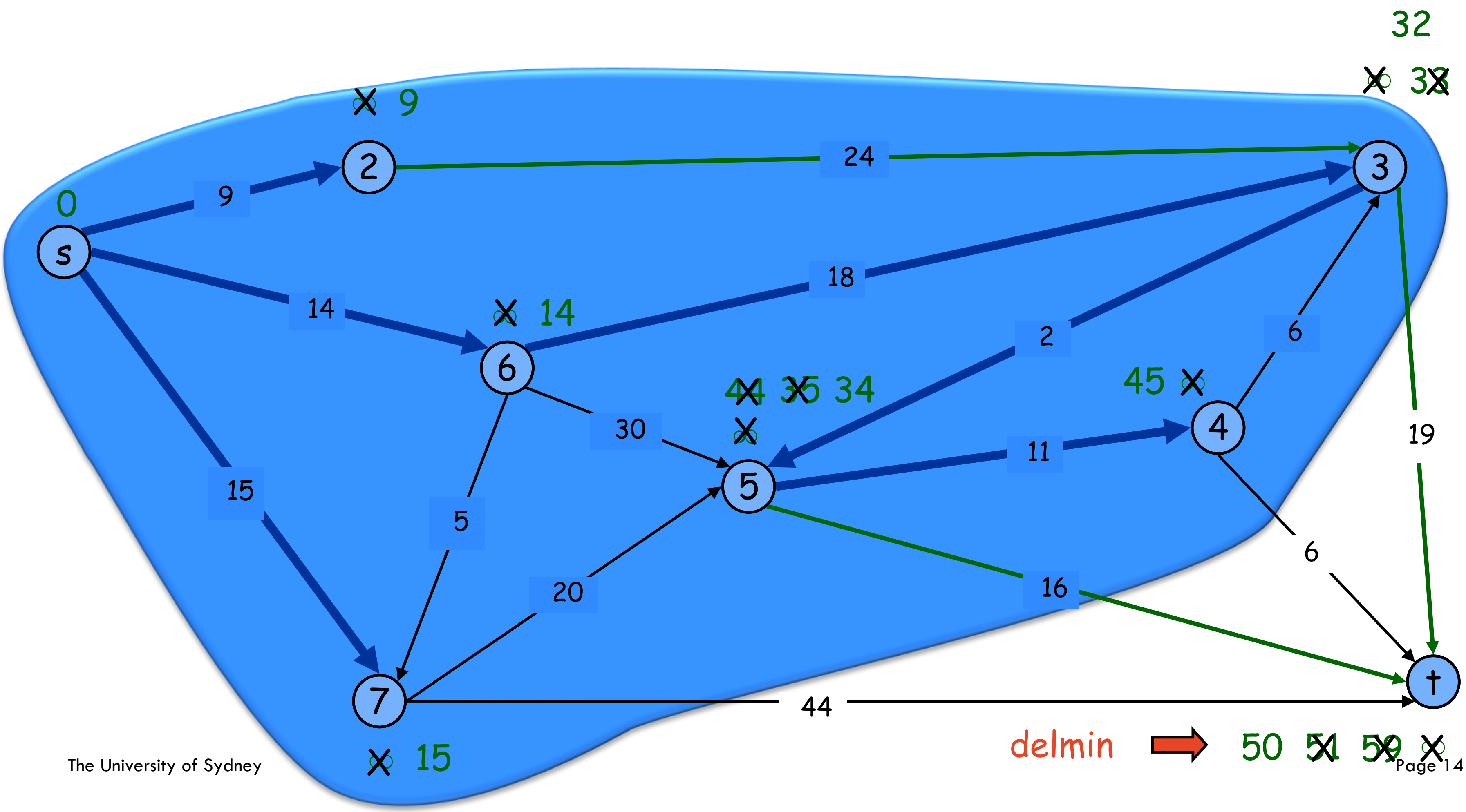
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }



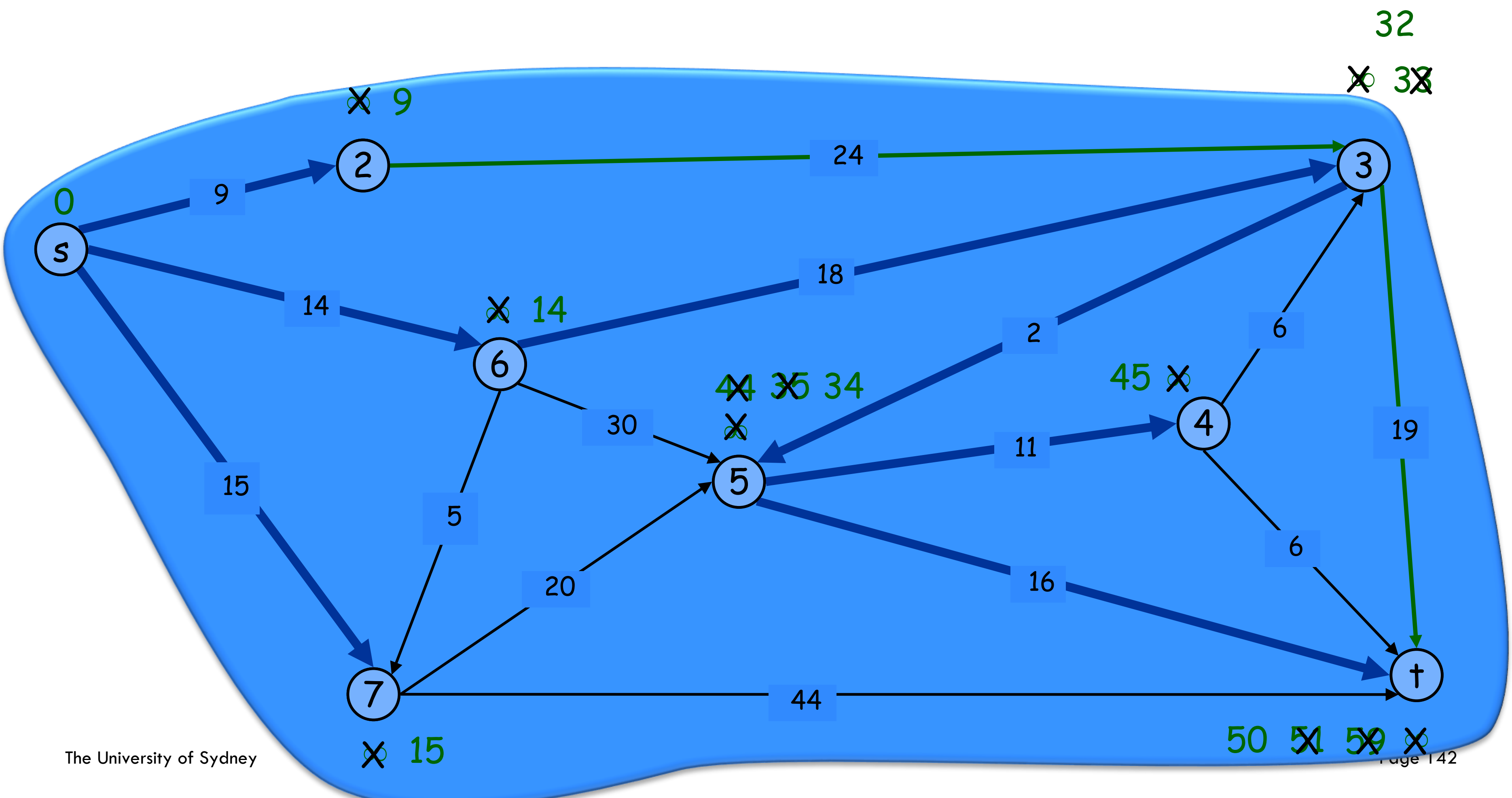
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }



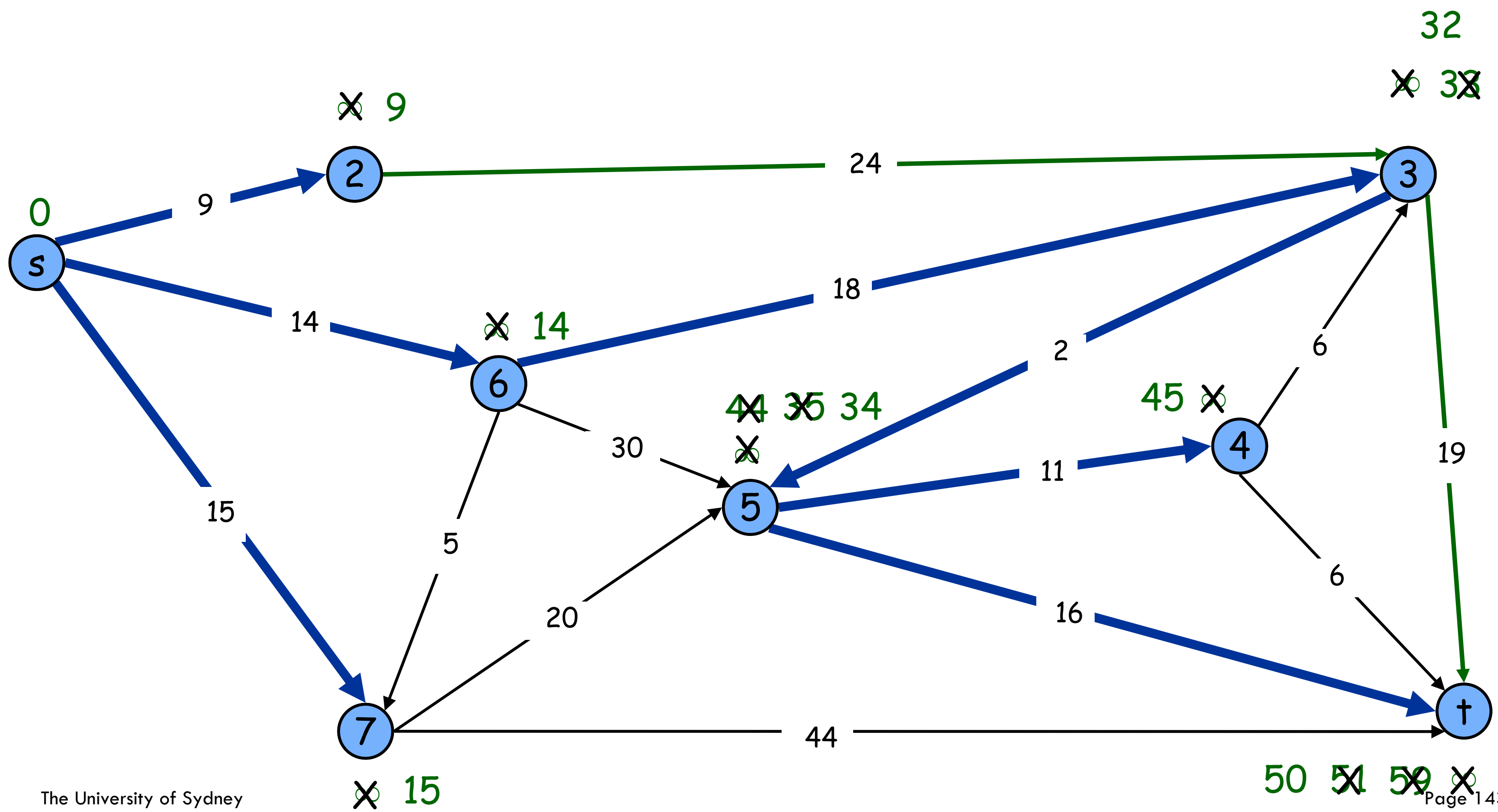
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7, t }
PQ = { }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7, t }
PQ = { }



Shortest Path

The shortest path between two vertices in a graph G with n vertices and m nodes can be computed in O(m+n log n) time.

Summary: Greedy algorithms

A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

Problems

- Interval scheduling/partitioning
- Scheduling: minimize lateness
- Minimum spanning tree (Prim's algorithm)
- Shortest path in graphs (Dijkstra's algorithms)
- …