COMP 3011
DESIGN AND ANALYSIS OF ALGORITHMS
FALL 2024

Graph Algorithms

LI Bo
Department of Computing
The Hong Kong Polytechnic University

GREEDY ALGORITHMS

INTERVAL SCHEDULING PROBLEM

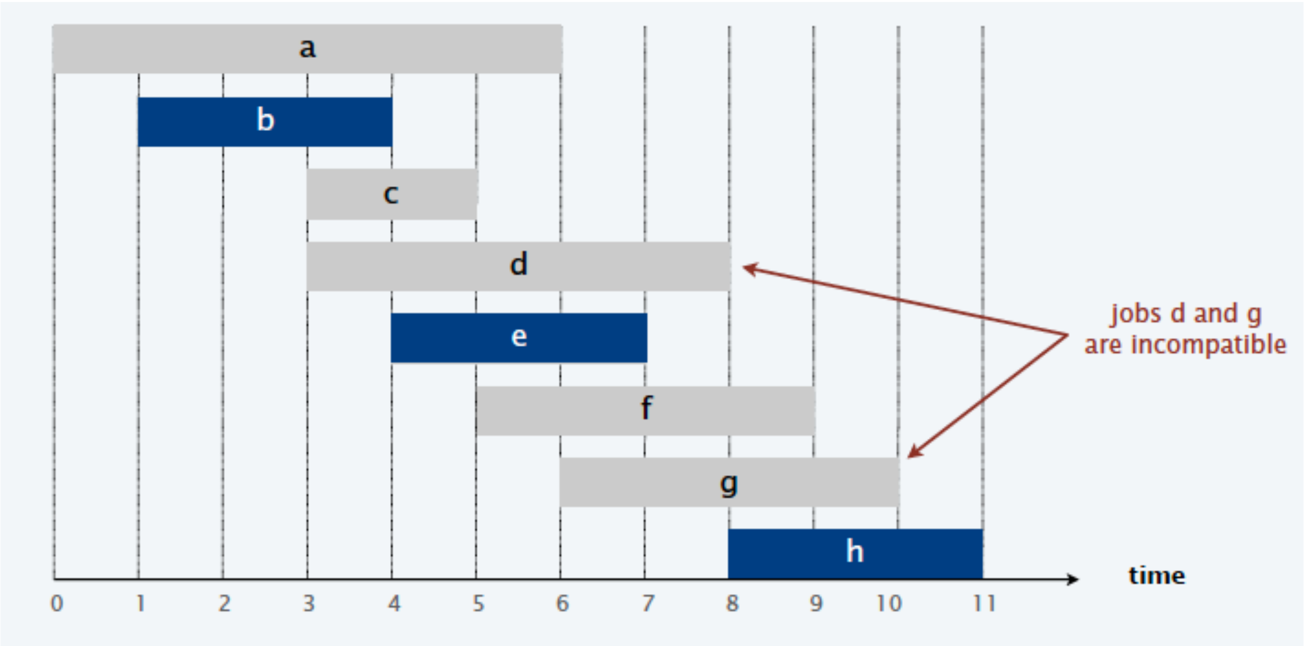
INTERVAL SCHEDULING PROBLEM

Given a set of jobs $J = \{1, 2, \ldots, n\}$

➢ Job j starts at s_j and finishes at $f_j \geq s_j$.

➢ Two jobs (open intervals) are compatible if they don't overlap.

Goal: find maximum subset of mutually compatible jobs.



Intuition: shorter is better

INTERVAL SCHEDULING PROBLEM

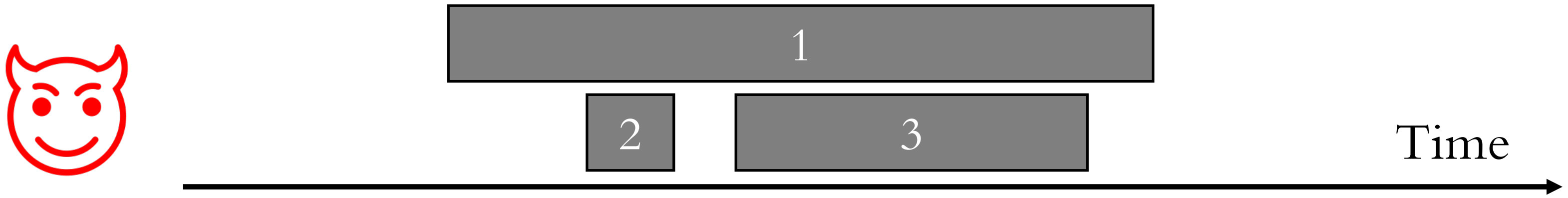
Idea 1:

➢ Repeatedly pick shortest compatible, unscheduled job (i.e. that does not conflict with any scheduled job).

Idea 2:

Intuition: earlier is better

➢ Repeatedly pick compatible job with earliest starting time.



GREEDY ALGORITHM

- Repeatedly pick an item until no more feasible choices.
- Among all feasible choices, we always pick the one that minimizes (or maximizes) ***some property***.
  - length, starting time, …
- Such algorithms are called ***greedy***.

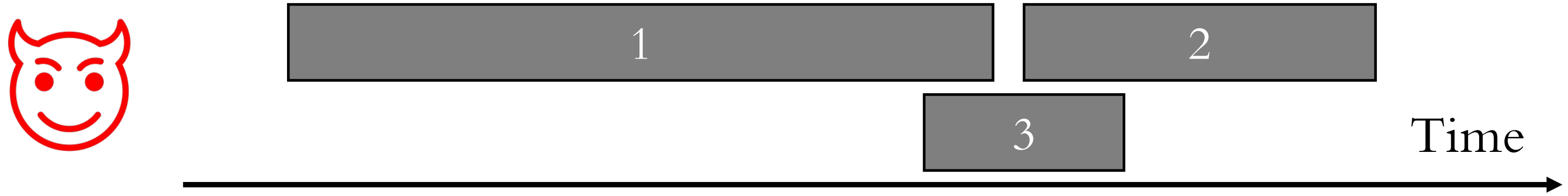
- Greedy algorithms may not be optimal.
- But maybe we have been using the wrong property!

INTERVAL SCHEDULING PROBLEM

What about earliest-finish-time-first?

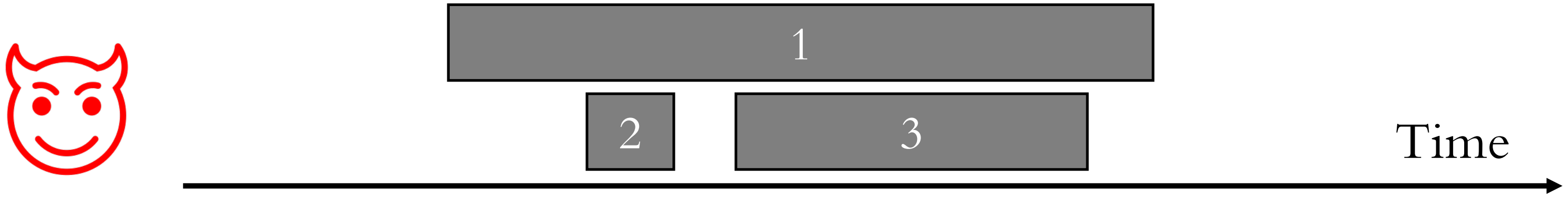
Idea 1:

- Repeatedly pick shortest compatible, unscheduled job (i.e. that does not conflict with any scheduled job).



Idea 2:

- Repeatedly pick compatible job with earliest starting time.



EARLIEST-FINISH-TIME-FIRST ALGORITHM

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

$S \leftarrow \varnothing$. ⟵ set of jobs selected

FOR $j = 1$ TO n

    IF (job j is compatible with S)

        $S \leftarrow S \cup \{ j \}$.

RETURN S.

Proposition. Can implement earliest-finish-time first in $O(n \log n)$ time.

EARLIEST-FINISH-TIME-FIRST ALGORITHM

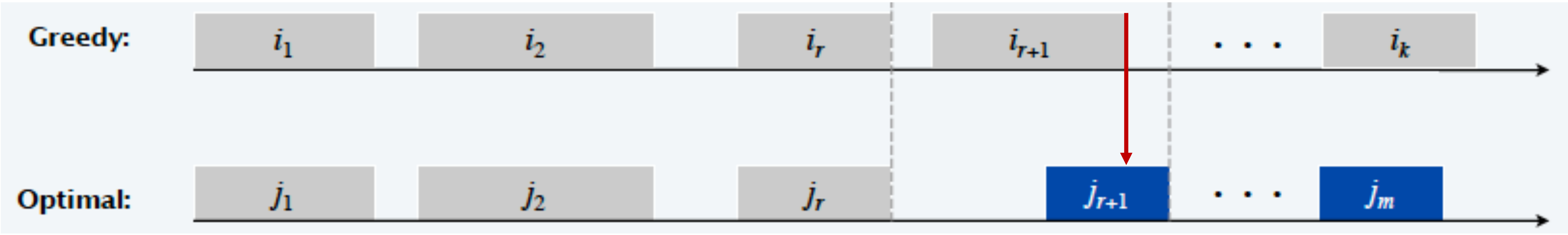
Theorem. The earliest-finish-time-first algorithm is optimal.

Proof. [by contradiction]

Switching j_{r+1} by i_{r+1} in O:
Still ***feasible*** and ***optimal***!

- Assume Greedy is not optimal.
- Let $A = \{i_1, i_2, \ldots, i_k\}$ be set of jobs selected by Greedy.
- Let $O = \{j_1, j_2, \ldots, j_m\}$ be set of jobs in an optimal solution. Then $m > k$.
- Let $r+1$ be first index such that $i_{r+1} \neq j_{r+1}$.

such a job exists ➡ $f_{i_{r+1}} \leq f_{j_{r+1}}$



$i_1 = j_1$ $\quad$ $i_2 = j_2$ $\quad$ $i_r = j_r$ $\quad$ $i_{r+1} \neq j_{r+1}$

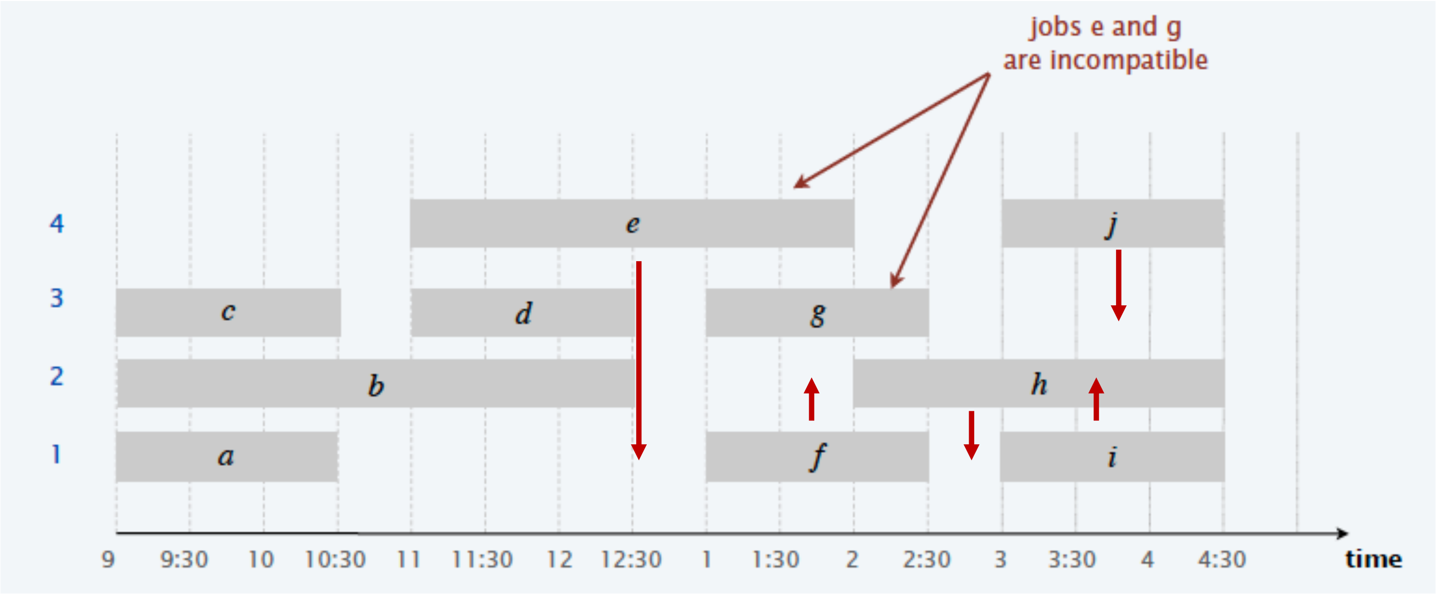
INTERVAL PARTITIONING

INTERVAL PARTITIONING

Given a set of lectures (jobs) $L = \{1, 2, \ldots, n\}$;

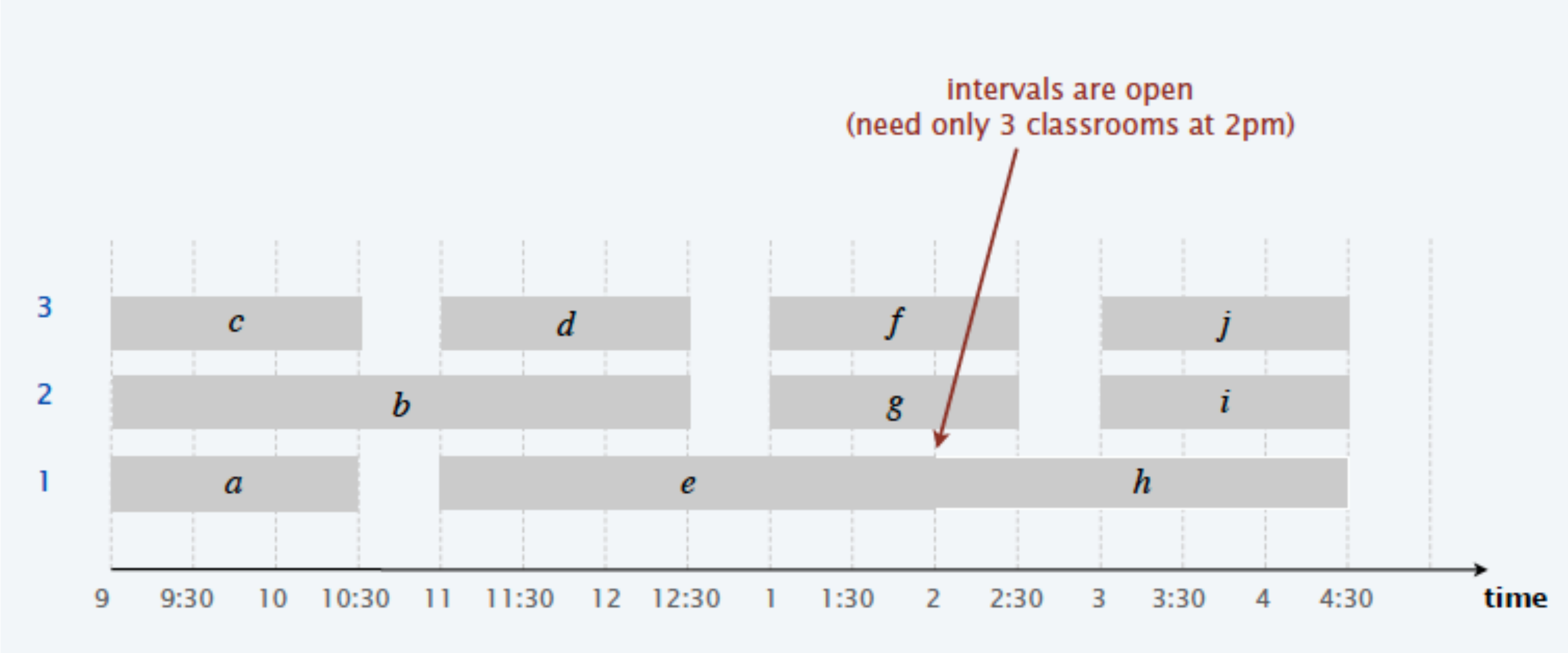
- Lecture j starts at s_j and finishes at $f_j \geq s_j$.
- Two lectures are compatible if they don't overlap.

Goal: find minimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same room



INTERVAL PARTITIONING

- Optimal is 3 classrooms.

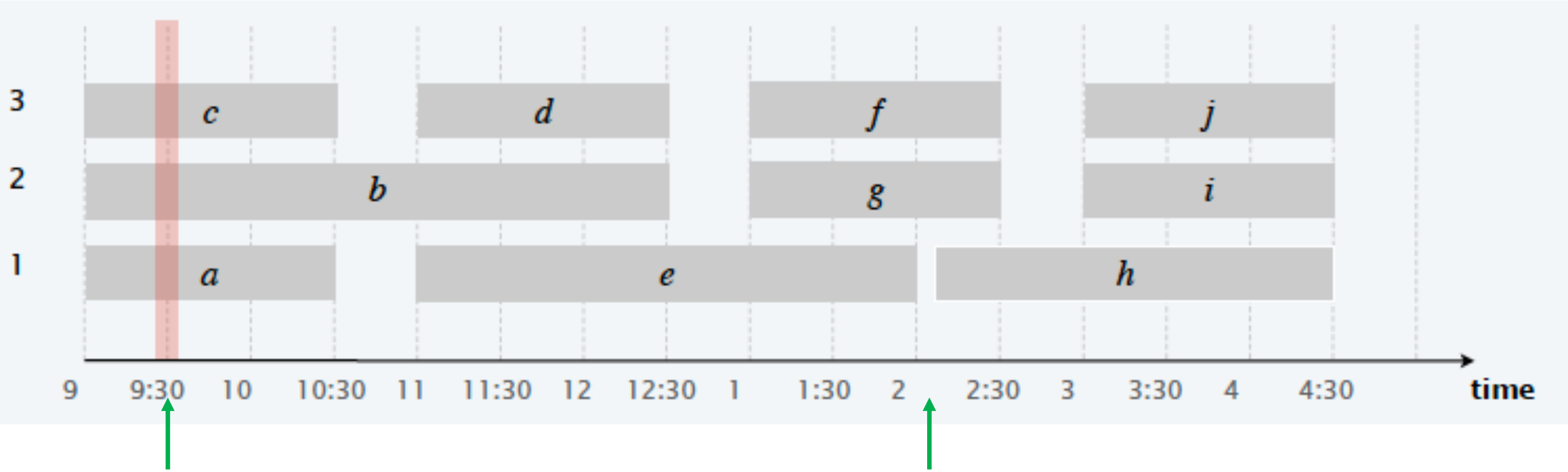


INTERVAL PARTITIONING

Definition. The depth of a set of open intervals is the maximum number of intervals that contain any given point.

Key observation. #rooms needed ≥ depth.

Is depth enough???

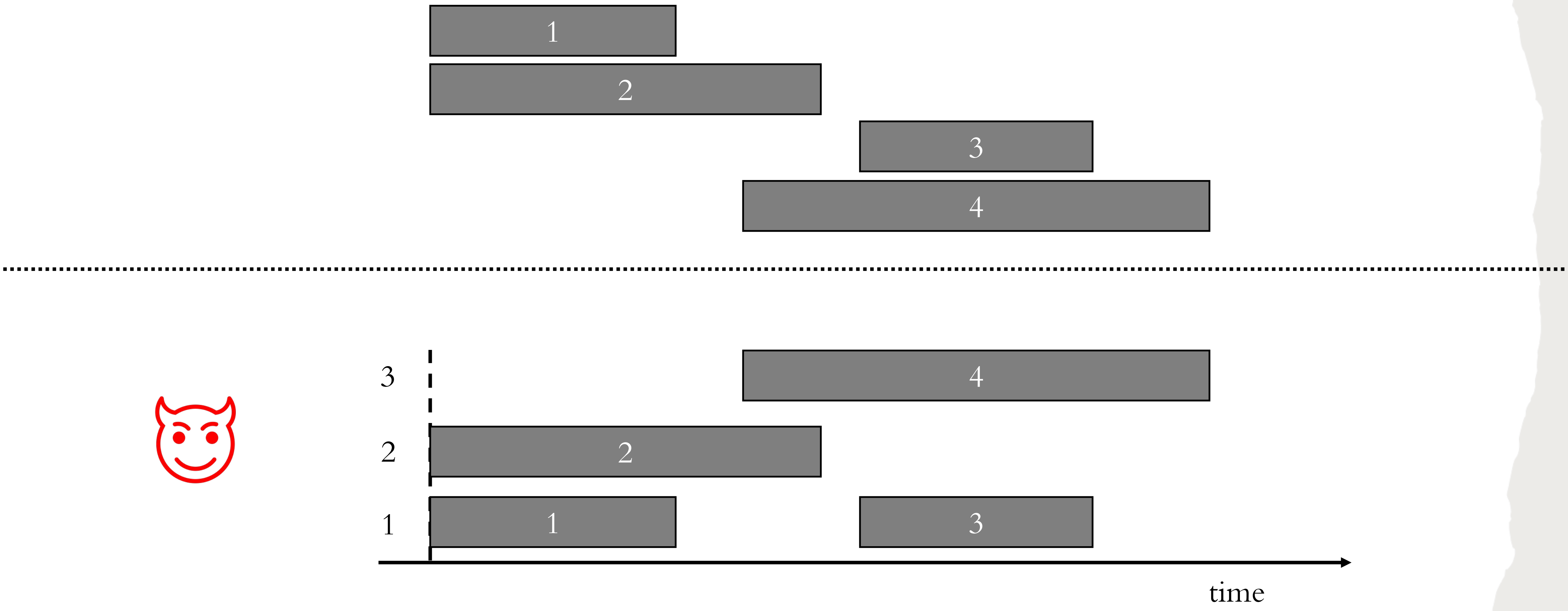


3 classrooms are needed

2 classrooms are needed

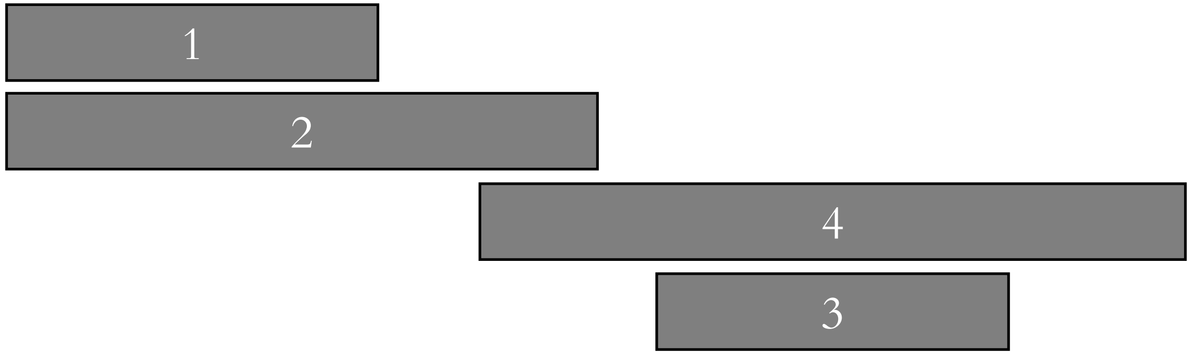
INTERVAL PARTITIONING

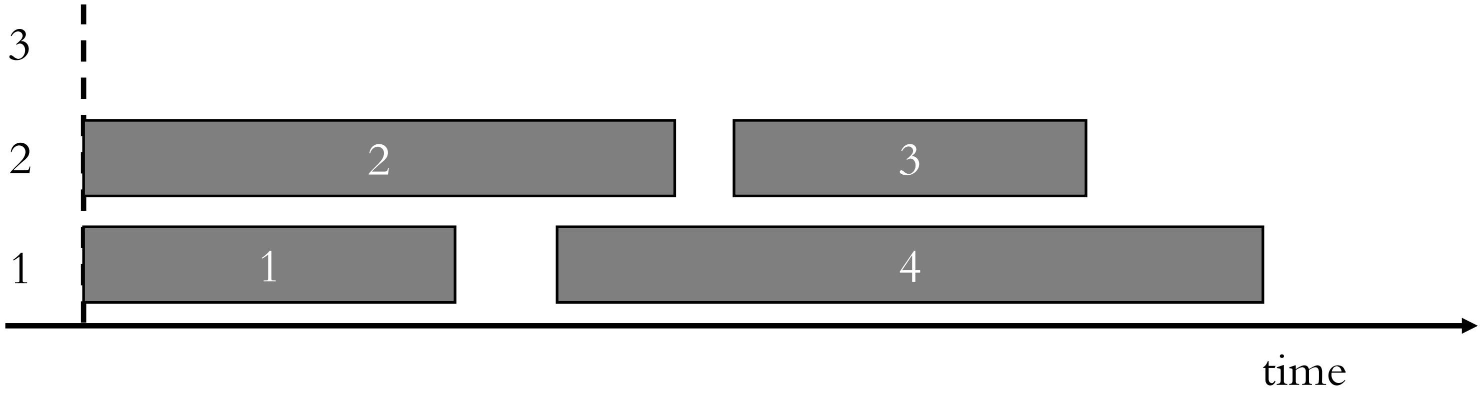
Can we do earliest-finish-time-first?



INTERVAL PARTITIONING

Can we do earliest-start-time-first?





INTERVAL PARTITIONING: EARLIEST-START-TIME-FIRST ALGORITHM

EARLIEST-START-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT lectures by start times and renumber so that $s_1 \leq s_2 \leq \ldots \leq s_n$.

$d \leftarrow 0$. ⟵ number of allocated classrooms

FOR $j = 1$ TO n

  IF (lecture j is compatible with some classroom)

    Schedule lecture j in any such classroom k.

  ELSE

    Allocate a new classroom $d + 1$.

    Schedule lecture j in classroom $d + 1$.

    $d \leftarrow d + 1$.

RETURN schedule.

***Lemma*.**

The earliest-start-time-first algorithm can be implemented in $O(n \log n)$ time.

***Lemma*.**

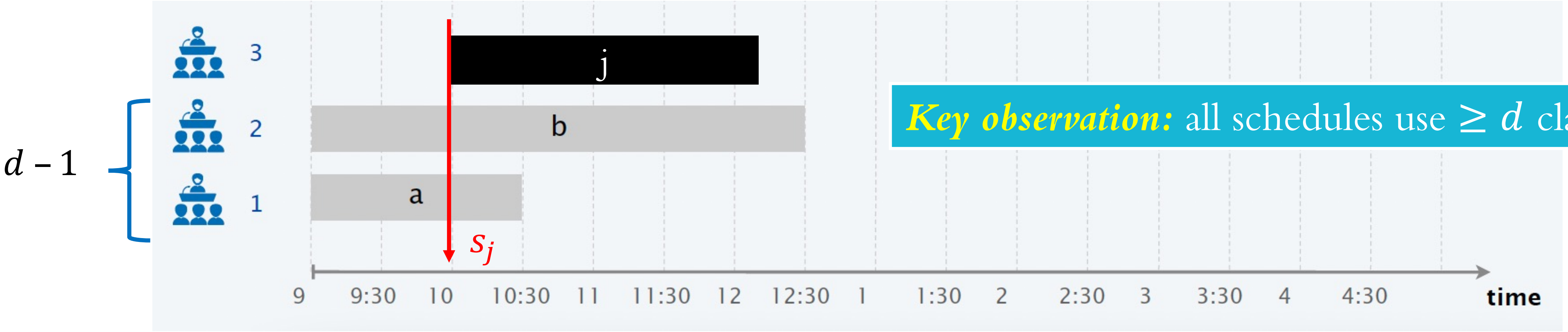
The earliest-start-time first algorithm never schedules two incompatible lectures in the same classroom.

INTERVAL PARTITIONING: EARLIEST-START-TIME-FIRST ALGORITHM

Theorem. Earliest-start-time-first algorithm uses #depth rooms and thus is optimal.

- Let d = number of classrooms that the algorithm allocates.
- Classroom d is opened because we needed to schedule a lecture, say j, that is incompatible with a lecture in each of $d - 1$ other classrooms.
- Thus, these d lectures each end after s_j.
- Since we sorted by start time, each of these incompatible lectures start no later than s_j. ■

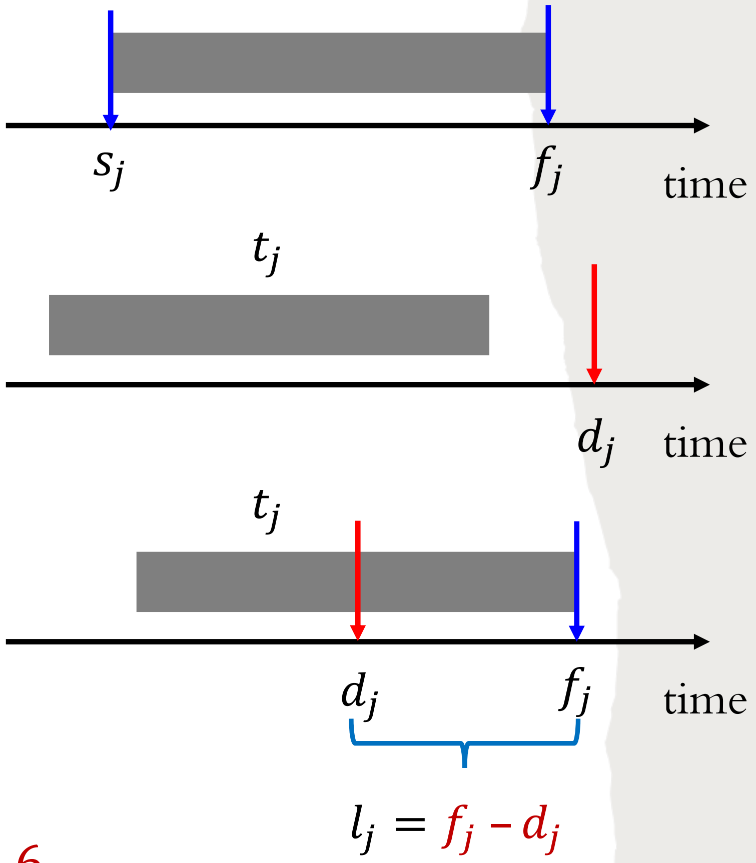
The d lectures are incompatible.



Key observation: all schedules use $\geq d$ classrooms.

SCHEDULING TO MINIMIZING LATENESS

SCHEDULING TO MINIMIZING LATENESS



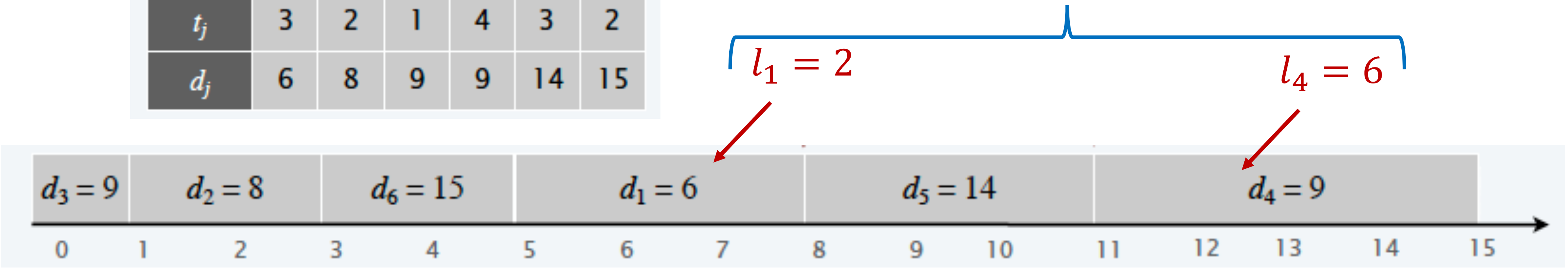
Single resource processes one job at a time.

- Job j requires t_j units of processing time and is due at time d_j.
- If j starts at time s_j, it finishes at time $f_j = s_j + t_j$.
- Lateness: $l_j = \max\{0, f_j - d_j\}$.

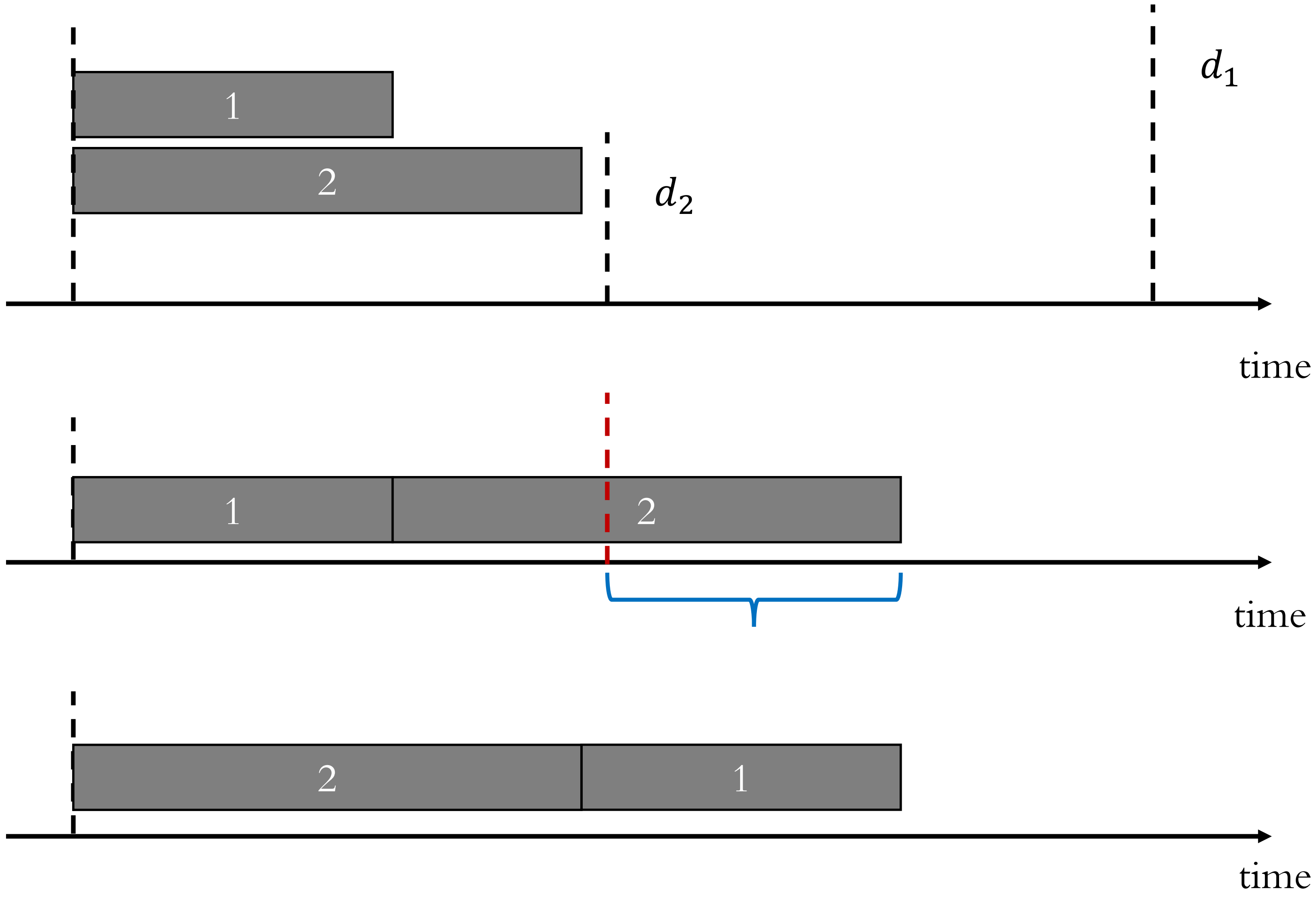
Goal: schedule all jobs to minimize maximum lateness $L = \max_j l_j$.

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15

Maximum latency $L = 6$



SCHEDULING TO MINIMIZING LATENESS



SCHEDULING TO MINIMIZING LATENESS

EARLIEST-DEADLINE-FIRST $(n, t_1, t_2, \ldots, t_n, d_1, d_2, \ldots, d_n)$

SORT jobs by due times and renumber so that $d_1 \le d_2 \le \ldots \le d_n$.

$t \leftarrow 0$.

FOR $j = 1$ TO n ← Process the ordered jobs one by one (immediately)

  Assign job j to interval $[t, t + t_j]$.

  $s_j \leftarrow t$; $f_j \leftarrow t + t_j$.

  $t \leftarrow t + t_j$.

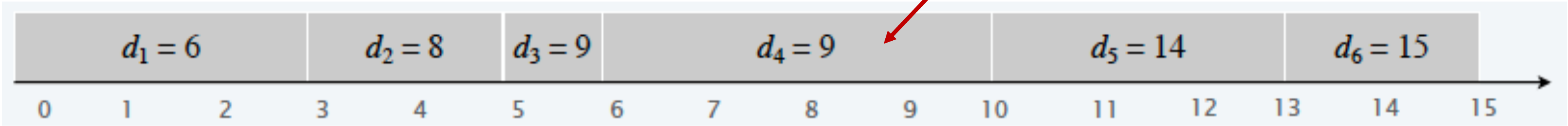
RETURN intervals $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$.

$\Downarrow$

t

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15

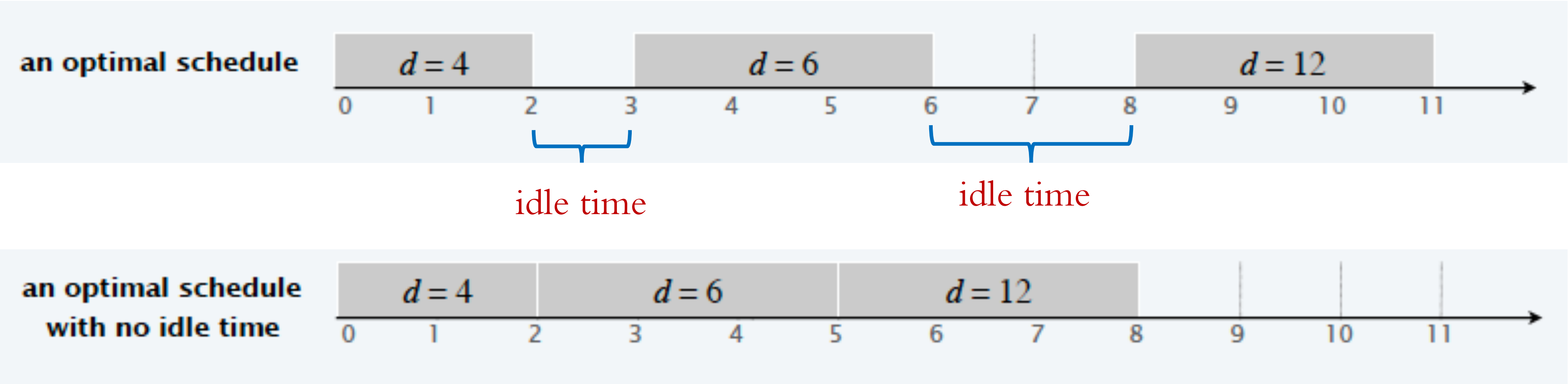
$l_4 = 1$



SCHEDULING TO MINIMIZING LATENESS

Properties for optimal schedules.

Observation 1. There exists an optimal schedule with no idle time.



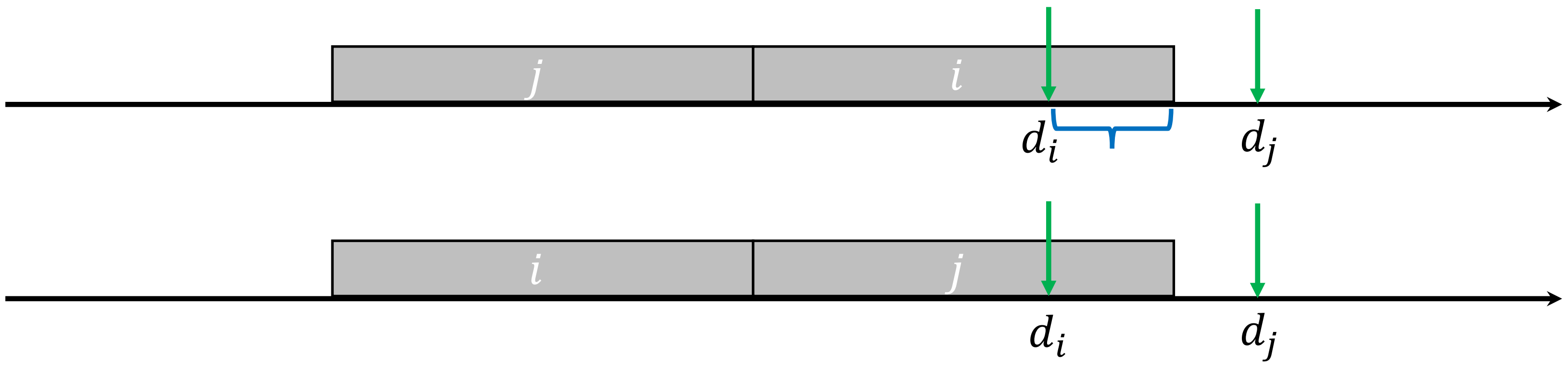
Observation 2. The earliest-deadline-first schedule has no idle time.

SCHEDULING TO MINIMIZING LATENESS

or $i < j$ for ordered jobs

Definition. Given a schedule S, an inversion is a pair of jobs i and j such that: $d_i < d_j$ but j is scheduled before i.

inversion



swap makes the schedule better!

Observation 3. The earliest-deadline-first schedule is the ***unique*** idle-free schedule with no inversions.

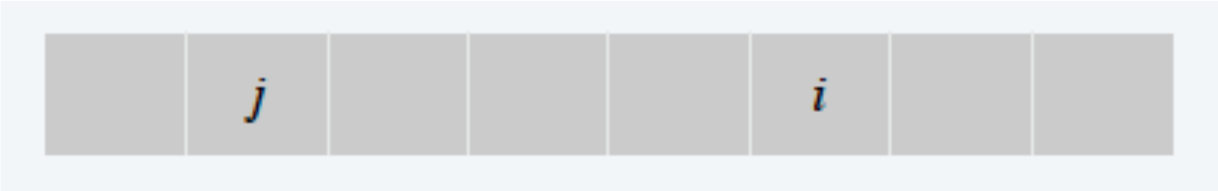
SCHEDULING TO MINIMIZING LATENESS

Observation 4. If an idle-free schedule has an inversion, then it has an adjacent inversion.

two inverted jobs scheduled consecutively

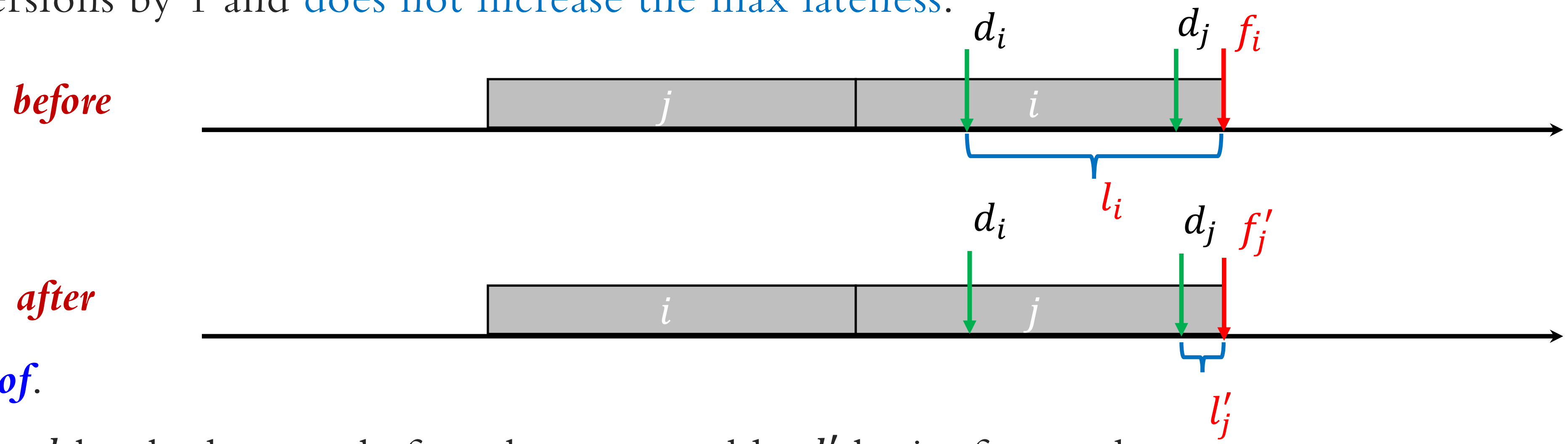
Proof.

- Let i–j be a closest inversion. $d_j > d_i$
- Let k be element immediately to the right of j.
  - Case 1: $d_j > d_k$. Then $j - k$ is an adjacent inversion.
  - Case 2. $d_j < d_k$. Then $i - k$ is a closer inversion. ■



SCHEDULING TO MINIMIZING LATENESS

Key Claim. Exchanging two adjacent, inverted jobs i and j reduces the number of inversions by 1 and does not increase the max lateness.



Proof.

- Let l be the lateness before the swap, and let l' be it afterwards.
- $l'_k = l_k$ for all $k \neq i, j$.
- $l'_i \leq l_i$

$f'_j = f_i$ $\quad$ $i < j$: $d_i \leq d_j$

- If job j is late, $l'_j = f'_j - d_j = f_i - d_j \leq f_i - d_i \leq l_i$. ■

SCHEDULING TO MINIMIZING LATENESS

Theorem. The earliest-deadline-first schedule S is optimal.

Proof. [by contradiction]

- Define S^* to be an optimal schedule with the fewest inversions.
- Can assume S^* has no idle time. $\longrightarrow$ Observation 1
- Case 1: S^* has no inversions. Then $S = S^*$. $\longrightarrow$ Observation 3
- Case 2: S^* has an inversion.
  - Let $i - j$ be an adjacent inversion $\longrightarrow$ Observation 4
  - Exchanging jobs i and j decreases the number of inversions by 1 without increasing the max lateness $\longrightarrow$ Key Claim
  - Contradicts "fewest inversions" part of the definition of S^*. ■

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.
- [Interval scheduling]

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.
- [Interval partitioning]

Exchange argument.

- Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.
- [Minimizing lateness, Interval scheduling]

SHORTEST PATH PROBLEM

Shortest Path Problem

Single-pair:

Given a digraph $G = (V, E)$, edge lengths $l_e \geq 0$, source $s \in V$, and destination $t \in V$, find a shortest directed path from s to t.

Single-source:

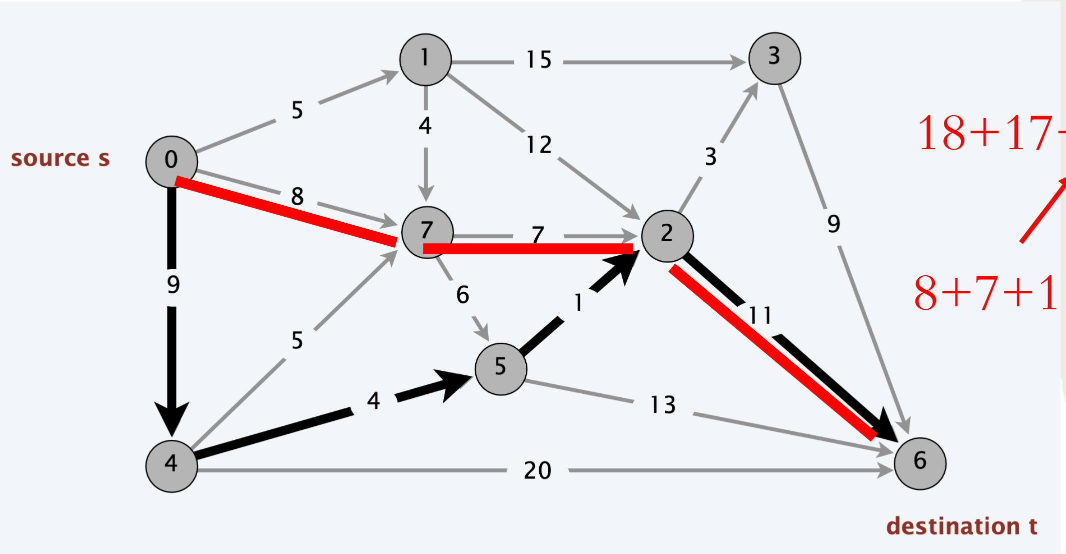
Given a digraph $G = (V, E)$, edge lengths $l_e \geq 0$, source $s \in V$, find a shortest directed path from s to every node.

Question:

Suppose that you change the length of every edge of G as follows. For which is every shortest path in G a shortest path in G'?

A. Multiply by 10.

B. Add 10.

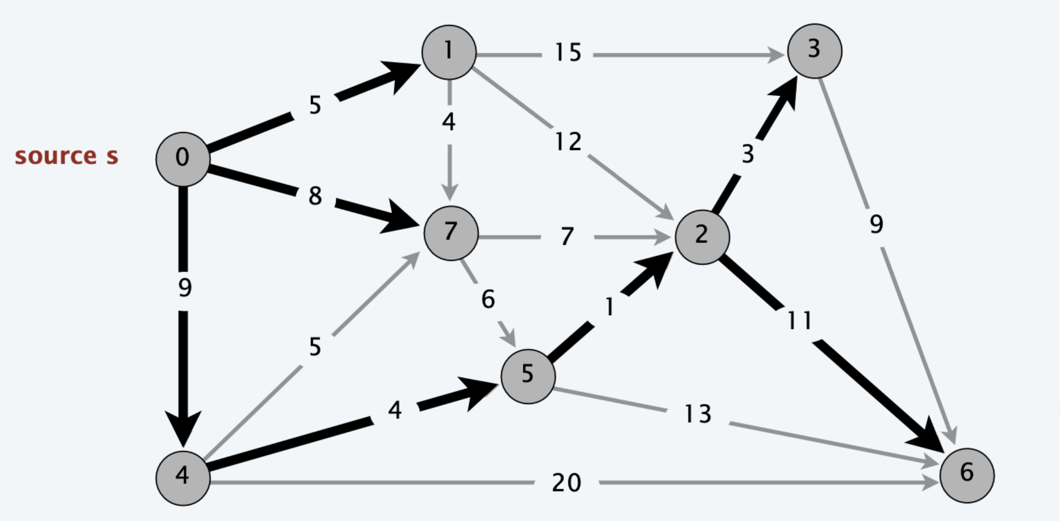


18+17+21=56

8+7+11=26

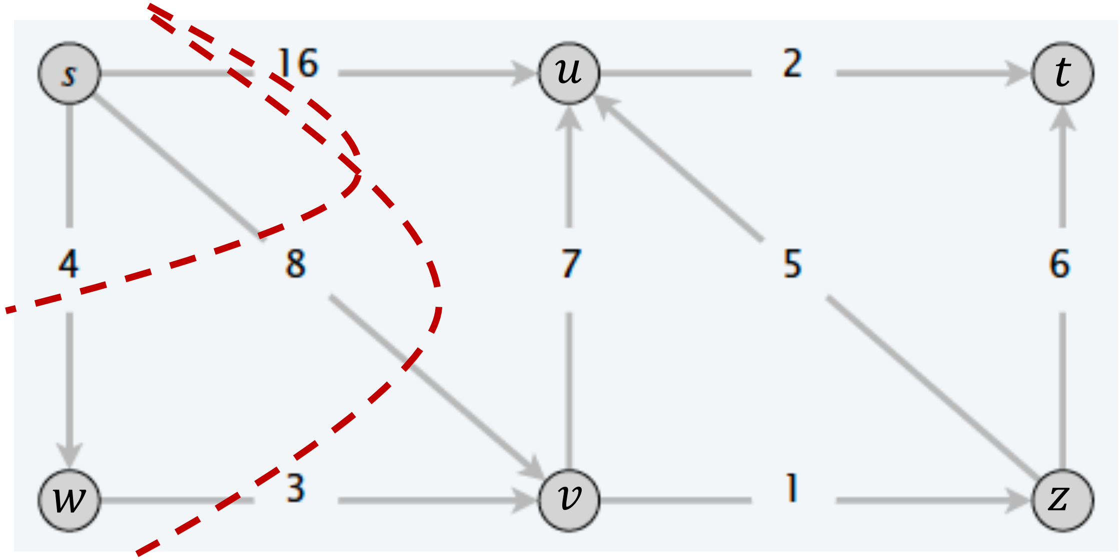
length of path = 9 + 4 + 1 + 11 = 25

19 + 14 + 11 + 21 = 65



Dijkstra's algorithm

$d[s] = 0$



$d[w] = 4$

Why? $w = \min_{x \neq s} \boxed{d[s] + l_{(s,x)}}$

The minimum distance from s to x without using other nodes

For $u \in V$, $d[u]$ = length of a shortest path from s to u.

$d[u] = 16?$

$s \to v \to u$ has length $15 < 16$

$d[u] = 15?$

$s \to w \to v \to u$ has length $14 < 15$

$d[u] = 14?$ $\Rightarrow d[v] \neq 8?$

$s \to w \to v \to z \to u$ has length $13 < 14$

$d[u] = 13?$

$s \to w \to v$ has length $7 < 8$

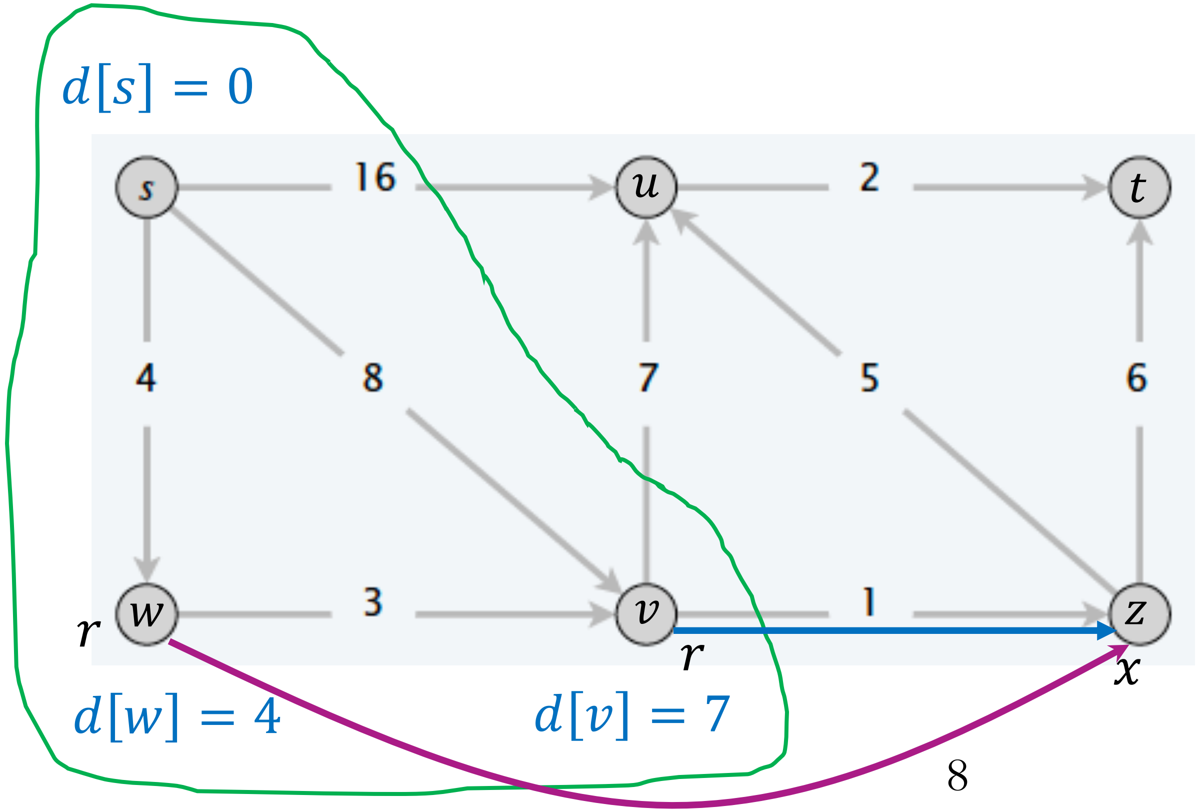
$d[v] = 7?$

$\pi(x) = \min_{r \in \{s,w\}} d[r] + l_{(r,x)}$

$v = \min_{x \neq s,w} \pi(x)$

The minimum distance from s, w to others

Dijkstra's algorithm



S → explored nodes

For $u \in S$, $d[u]$ = length of a shortest path from s to u.

the length of a shortest path from s to some node r in explored part S, followed by a single edge $e = (r, x)$.

For $x \notin S$, $\pi(x) = \min_{(r,x): r \in S} d[r] + l_{(r,x)}$

The closest distance to x without using $V \setminus S$

$\pi(z) = \min \{d[w] + l_{(w,z)}, d[v] + l_{(v,z)}\}$

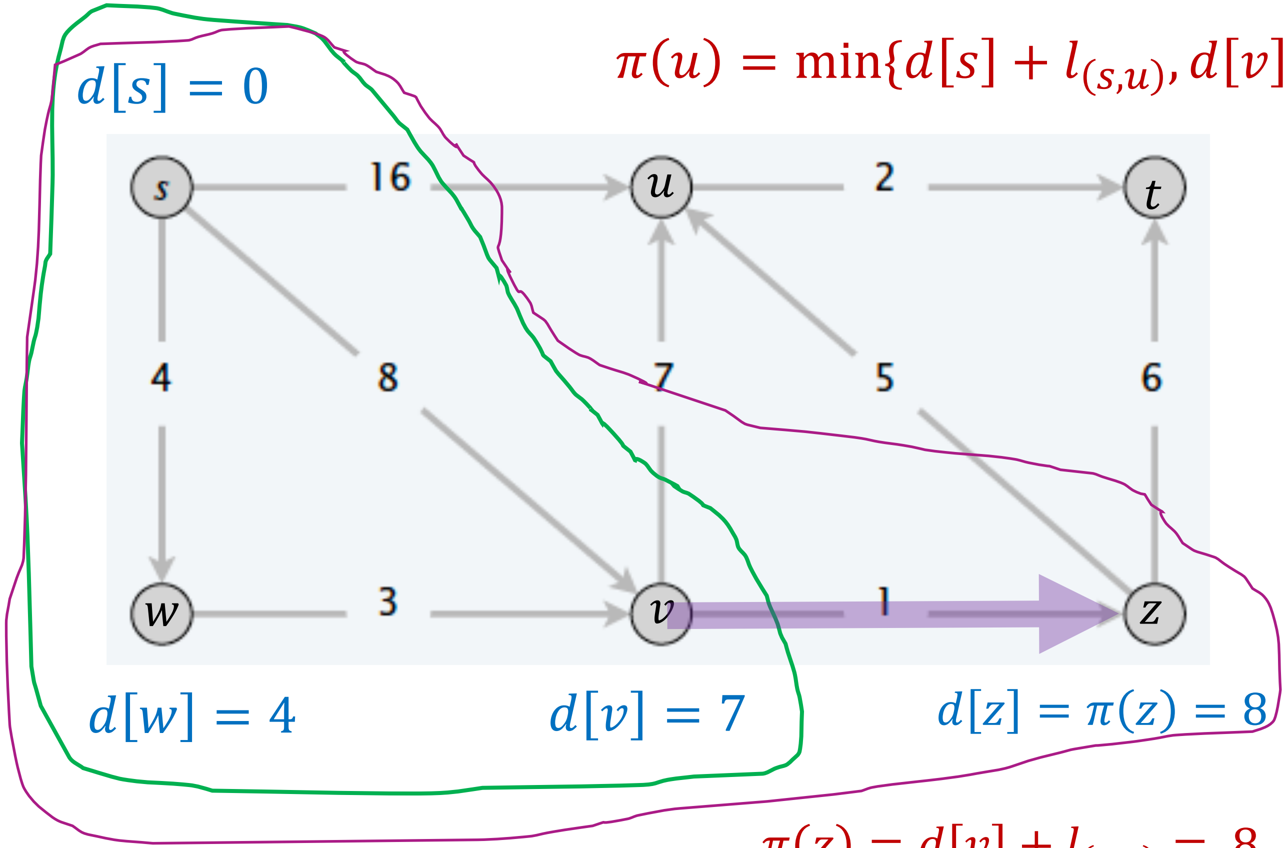
$\pi(u) = \min \{d[s] + l_{(s,u)}, d[v] + l_{(v,u)}\}$

Action

- Choose unexplored node $v \notin S$ which minimizes $\pi(v)$.
- Add v to S.

Dijkstra's algorithm

$\pi(u) = \min\{d[s] + l_{(s,u)}, d[v] + l_{(v,u)}\} = 14$



$\pi(z) = d[v] + l_{(v,z)} = 8$

Dijkstra's algorithm

- Maintain a set of explored nodes S for which algorithm has determined $d[u] =$ "length of a shortest path from s to u".
- Initialize $S \leftarrow \{s\}$, $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $x \notin S$ which ***minimizes***

$$\pi(x) = \min_{(r,x):r \in S} d[r] + l_{(r,x)}$$

add x to S and set $d[x] \leftarrow \pi(x)$.

Invariant.
For each node $u \in S$: $d[u]$ = length of a shortest path from s to u.

Invariant. For each node $u \in S$: $d[u]$ = length of a shortest path from s to u.

Proof [by induction on $|S|$]

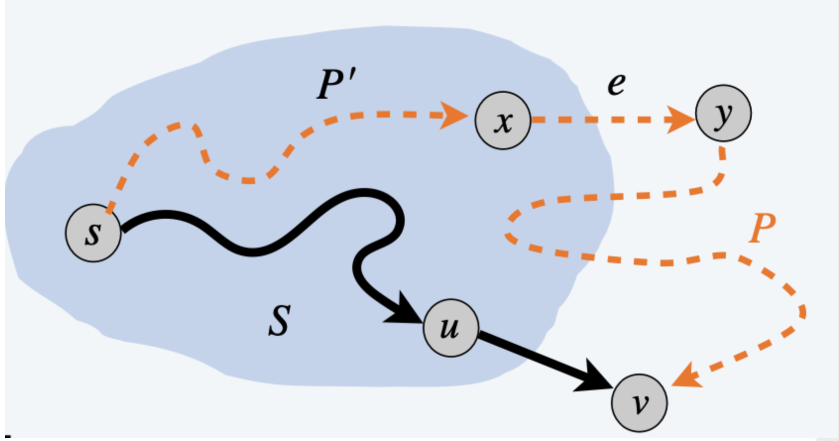
- Base case: $|S|=1$ is easy since $S = \{s\}$ and $d[s] = 0$.
- Inductive hypothesis: Assume true for $|S| \geq 1$.

- Let v be next node added to S, and let (u, v) be the final edge.
- A shortest $s \to u$ path plus (u, v) is an $s \to v$ path of length $\pi(v)$.
- Consider any other $s \to v$ path P. We show that it is no shorter than $\pi(v)$.
- Let $e = (x, y)$ be the first edge in P that leaves S, and let P' be the subpath from s to x.
- The length of P is already $\geq \pi(v)$ as soon as it reaches y:

$$l(P) \geq l(P') + l_e \geq d[x] + l_e \geq \pi(y) \geq \pi(v).$$

non-negative lengths | inductive hypothesis | definition of $\pi(y)$ | Dijkstra chose v instead of y

$$\pi(y) = \min_{(r,y):r \in S} d[r] + l_{(r,y)}$$

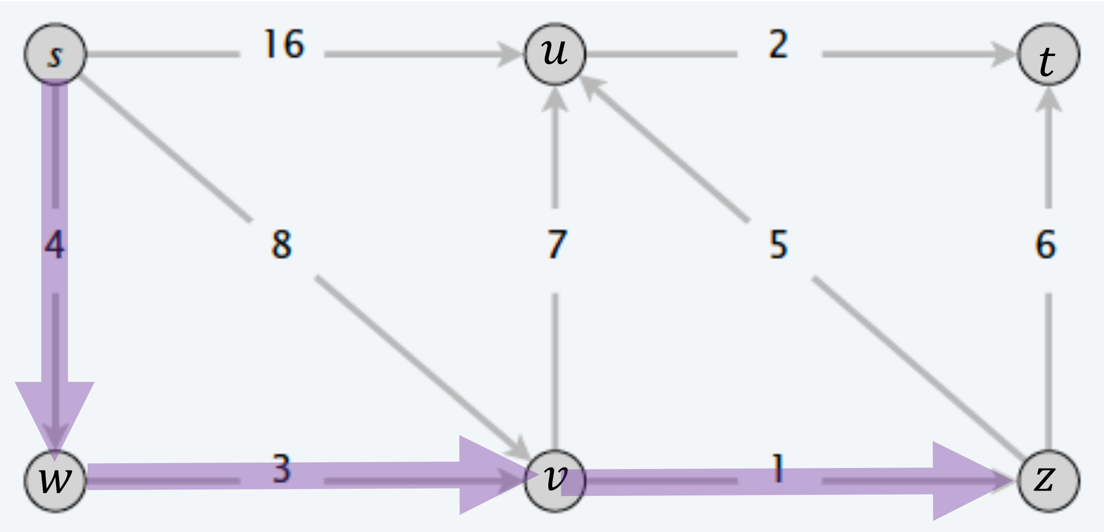


$$\pi(v) = d[u] + l_{(u,v)}$$

$$v = \min_{a \notin S} \pi(a)$$

Dijkstra's algorithm

$d[s] = 0$



$d[w] = 4$ $\quad$ $d[v] = 7$ $\quad$ $d[z] = \pi(z) = 8$

$pred[w] = s$ $\quad$ $pred[v] = w$ $\quad$ $pred[z] = v$

1. Tracking the shortest paths.
2. Running time = $O(n^2)$

Dijkstra's algorithm

- Maintain a set of explored nodes S for which algorithm has determined $d[u]$ = "length of a shortest path from s to u".
- Initialize $S \leftarrow \{s\},\ d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $x \notin S$ which ***minimizes***

$$\pi(x) = \min_{(r,x):r\in S} d[r] + l_{(r,x)}$$

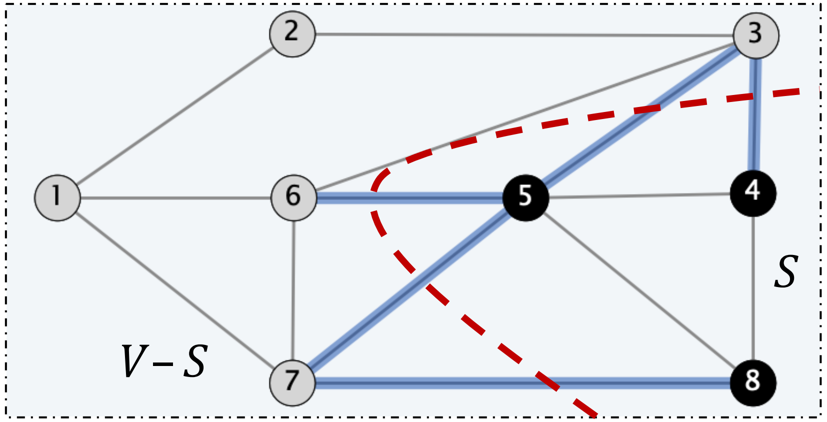
add x to S and set $d[x] \leftarrow \pi(x)$.

MINIMUM SPANNING TREES

Basic Definitions

A ***cut*** is a partition of the nodes into two nonempty subsets S and $V - S$, denoted by $(S, V - S)$.

The ***cutset*** of a cut S is the set of edges with exactly one endpoint in S.

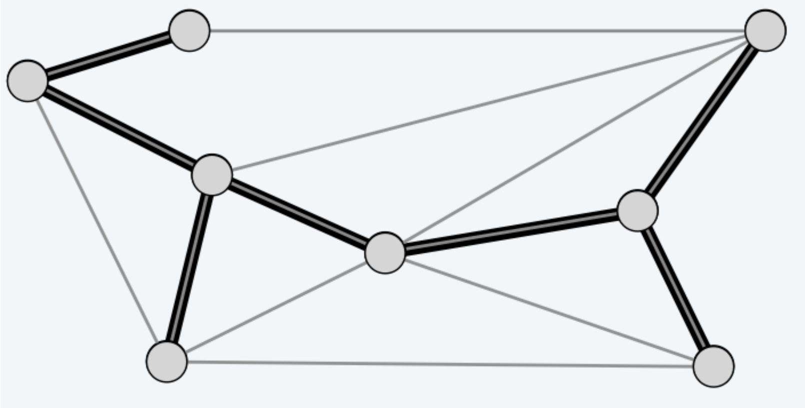


Cut $S = \{4,5,8\}$

Cutset $D = \{(3,4), (3,5), (5,6), (5,7), (8,7)\}$

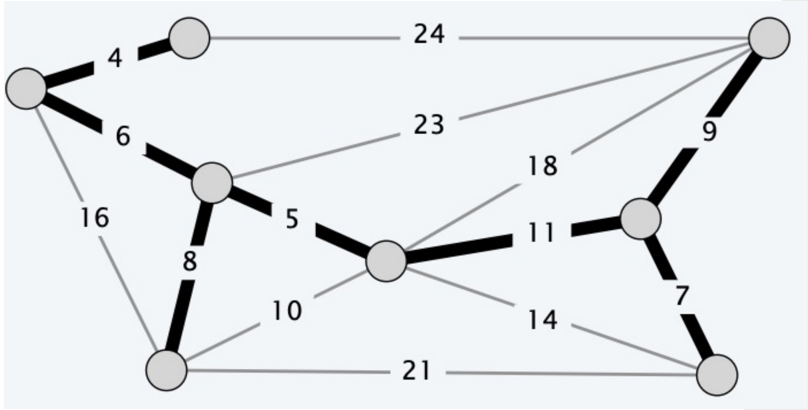
Spanning Tree

Let $H = (V, T)$ be a subgraph of an undirected graph $G = (V, E)$.
H is a ***spanning tree*** of G if H is both acyclic and connected.



***Proposition*.**
Let $H = (V, T)$ be a subgraph of an undirected graph $G = (V, E)$.
Then, the following are equivalent:

- H is a spanning tree of G.
- H is acyclic and connected.
- H is connected and has $|V| - 1$ edges.
- H is acyclic and has $|V| - 1$ edges.
- H is minimally connected: removal of any edge disconnects it.
- H is maximally acyclic: addition of any edge creates a cycle.



Minimum spanning tree (MST)
Given a connected, undirected graph $G = (V, E)$ with edge costs c_e, a ***minimum spanning tree*** (V, T) is a spanning tree of G such that the sum of the edge costs in T is minimized.

Tree cost = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7

Minimum spanning tree (MST)

Cayley's theorem. The complete graph on n nodes has n^{n-2} spanning trees.

can't solve by brute force

Both give the optimal solution!

Kruskal's Algorithm

Idea.

- Starts without any edges and insert edges from E in order of increasing cost:

$$c_1 < c_2 < \cdots < c_i < \cdots < c_m$$

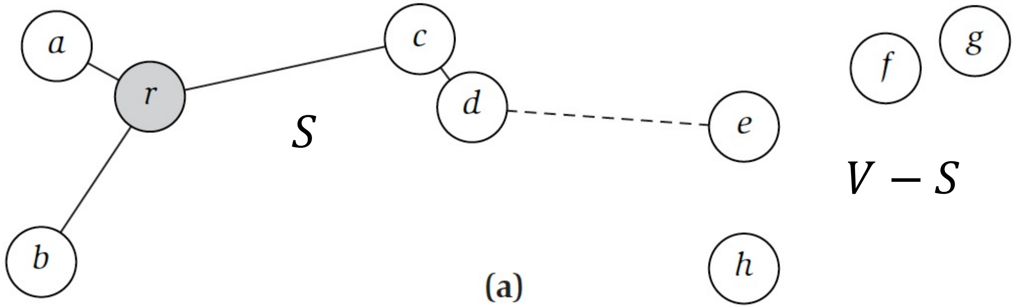
- For edge e_i, insert it if it does not create a cycle with all inserted edges, and discard otherwise.

Prim's Algorithm

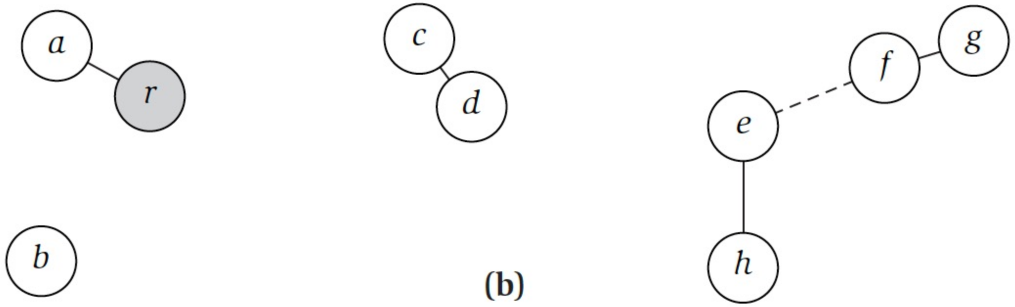
Idea (inspired by Dijkstra's Algorithm).

- Start with a root node $S = \{s\}$, and try to greedily grow a tree from S outward.
- At each step, we add the node v connected with S that can be attached as cheaply as possibly.

$$\min_{e=(u,v):u\in S} c_e$$



(a)



(b)

When Is It Safe to Include an Edge in the Minimum Spanning Tree?

Cut Property (Assume that all edge costs are distinct.)
Let S be ***any*** subset of nodes $S \neq V$ or $\emptyset$.
Let edge $e = (v, w)$ be the minimum cost edge with one end in S and the other in $V - S$.
Then ***every*** MST contains the edge e.

Proof. [contradiction + exchange argument]

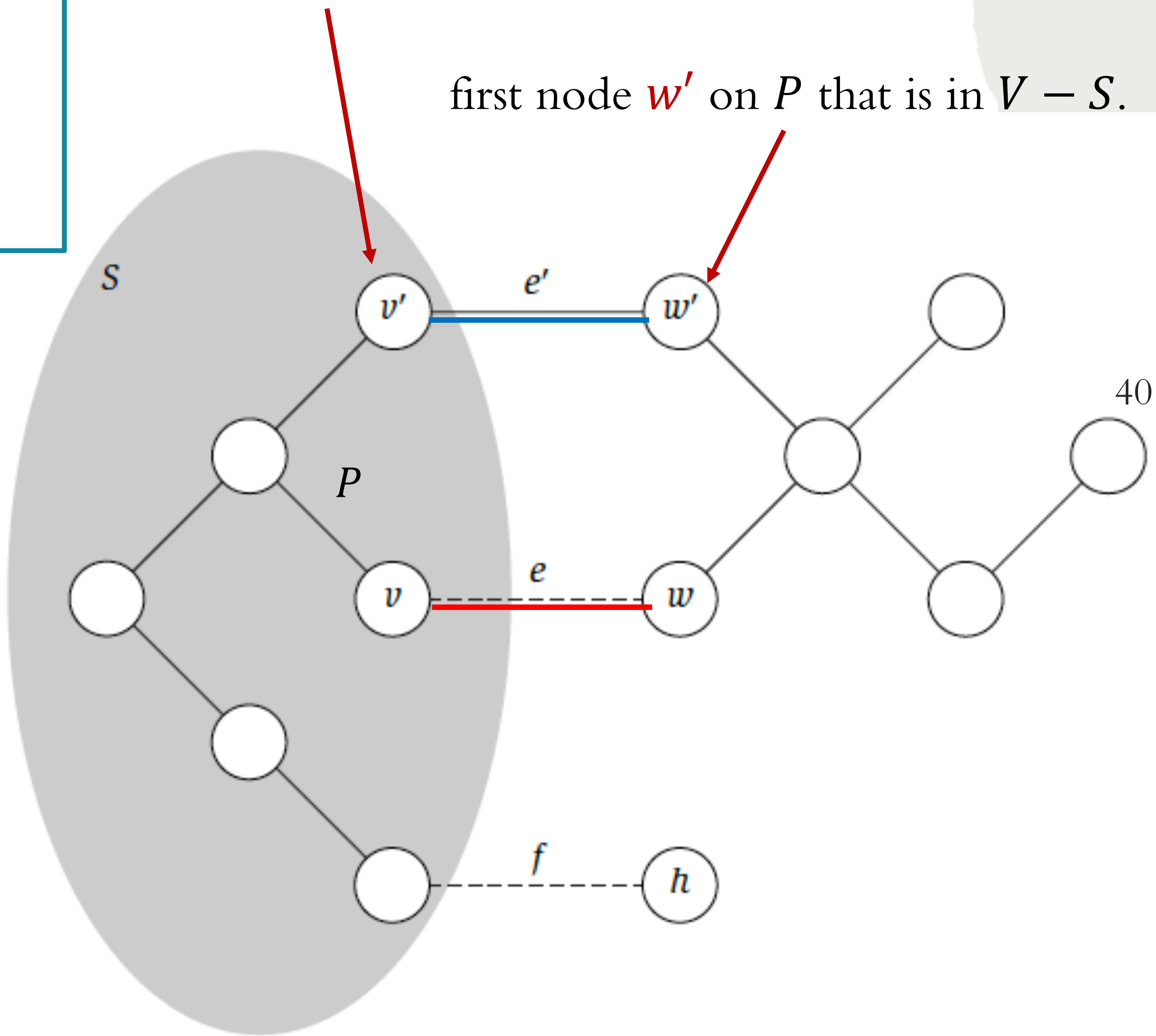
- Let T be an MST that does not contain e.
- There must be a path P in T from v to w.
- Exchange e' for e, get a set of edges

$$T' = T - \{e'\} \cup \{e\}.$$

- T' is a spanning tree:
  - Connected: any path in (V, T) that used e' can now be "rerouted" by using e.
  - Contains $|V| - 1$ edges.
- $c_e < c_e$: cost of T' < cost of T -- a contradiction.

$v' \in S$ is the node just before w' on P

first node w' on P that is in $V - S$.

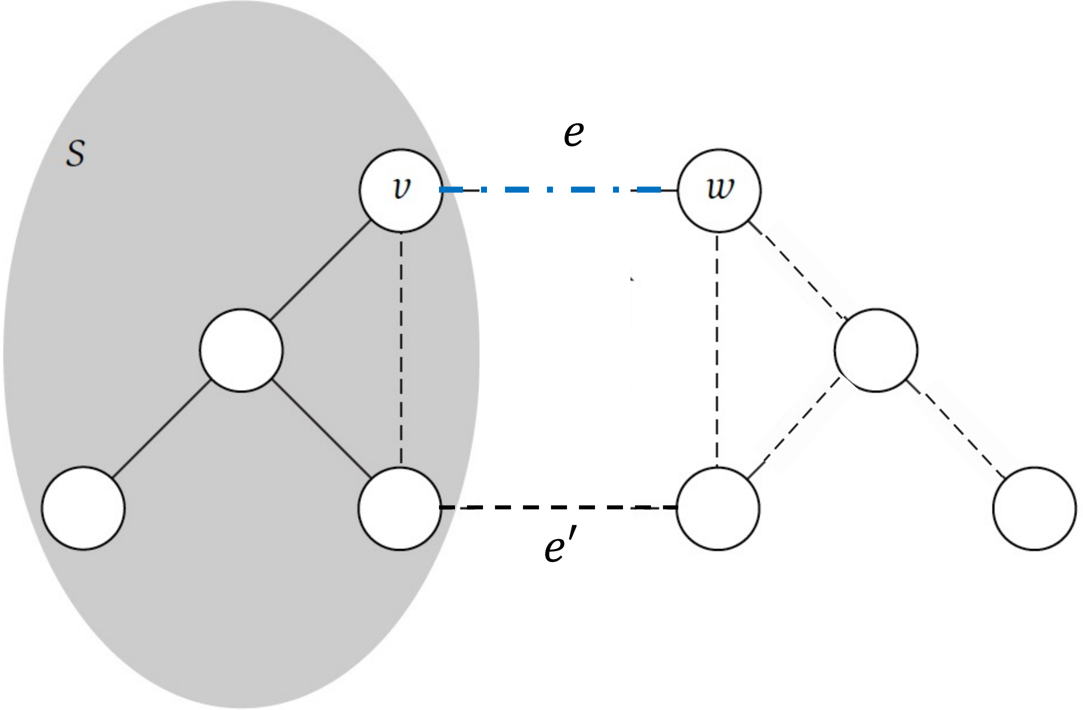


Prim's Algorithm

- Start with a root node $S = \{s\}$, and try to greedily grow a tree from S outward.
- At each step, we add the node v connected with S that can be attached as cheaply as possibly.

$$\min_{e=(u,v):u \in S} c_e$$

Theorem. Prim's Algorithm produces an MST of G.



Prim's Algorithm outputs a spanning tree.

- Contains no cycles: by the design
- Connected: otherwise can add an edge between two components.

Prim's Algorithm outputs an MST.

- At each step, we add the node v connected with S that can be attached as cheaply as possibly.

$$\min_{e=(u,v):u \in S} c_e$$

- Thus, e is the cheapest edge connecting S and $V - S$.
- By ***Cut Property***, e belongs to every MST.

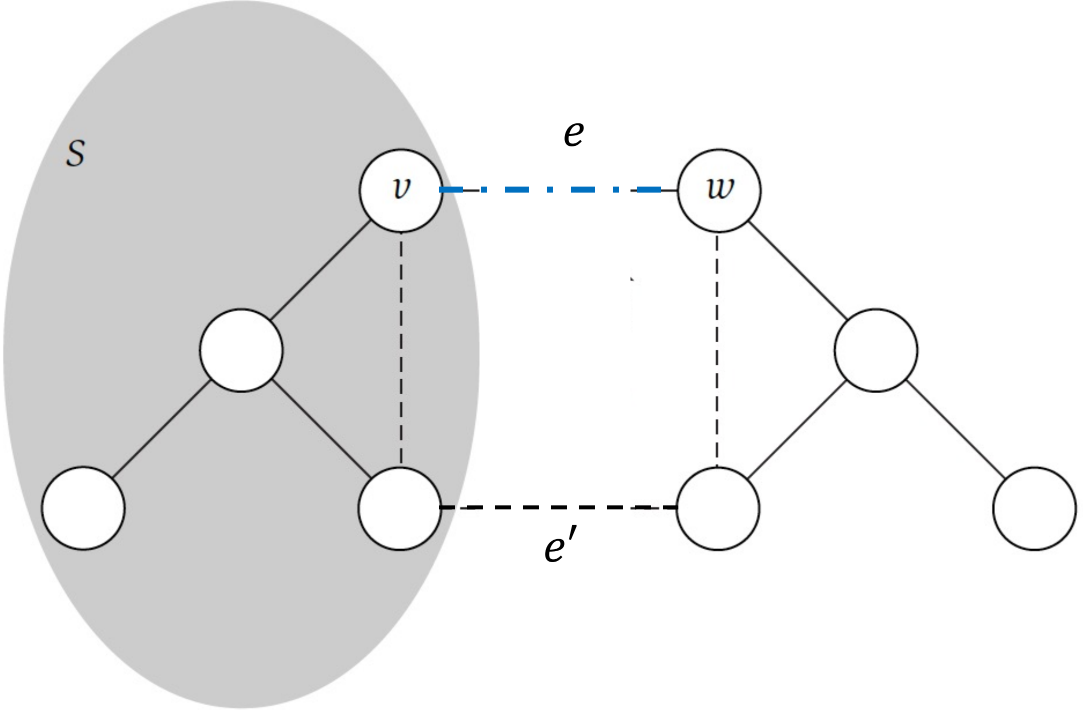
Kruskal's Algorithm

- Starts without any edges and insert edges from E in order of increasing cost:

$$c_1 < c_2 < \cdots < c_i < \cdots < c_m$$

- For edge e_i insert it if it does not create a cycle with all inserted edges, and discard otherwise.

Theorem. Kruskal's Algorithm produces an MST of G.



Kruskal's Algorithm outputs a spanning tree.

- Contains no cycles: by the design
- Connected; otherwise can add an edge between two components.

Kruskal's Algorithm outputs an MST.

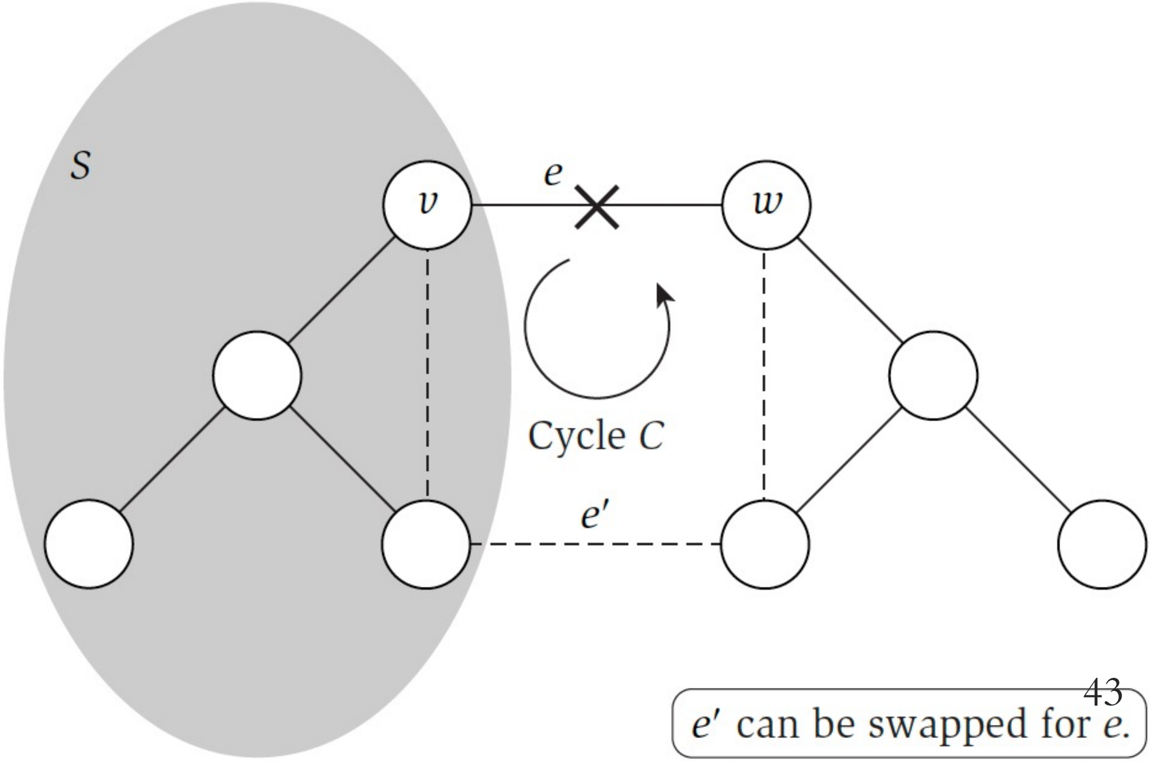
- Consider any edge $e = (v, w)$ added by Kruskal's Algorithm.
- Let S be the set of nodes to which v has a path before e is added. Clearly $v \in S$, but $w \notin S$.
- No edge from S to $V - S$ has been considered: any such edge could have been added without creating a cycle.
- Thus, e is the cheapest edge connecting S and $V - S$.
- By ***Cut Property***, e belongs to every MST.

Reverse-Delete Algorithm

- Start with the full graph (V, E) and begin deleting edges in order of decreasing cost.
$$c_1 > c_2 > \cdots > c_i > \cdots > c_m$$
- As we get to each edge e (starting from the most expensive), we delete it as long as doing so would not actually disconnect the graph we currently have.

Theorem.
The Reverse-Delete Algorithm produces an MST of G.

Cycle Property
(Assume that all edge costs are distinct.)
Let C be any cycle in G.
Let edge $e = (v, w)$ be the most expensive edge on C.
Then e does not belong to any MST of G.



e' can be swapped for e.

Proof [by contradiction].
- Let T be an MST that contains $e = (v, w)$.
- Deleting e from T and partition the nodes into S and $V - S$.
- There is another edge e' crosses from S to $V - S$.
- Consider the set of edges
$$T = T - \{e\} \cup \{e'\}$$
which is a spanning tree of G with smaller cost.

DIVIDE AND CONQUER

Divide and Conquer

- Divide up problem into several subproblems (of the same kind).
- Solve (conquer) each subproblem recursively.
- Combine solutions to subproblems into overall solution

Most common usage:

- Divide problem of size n into two subproblems of size $n/2$.
- Solve (conquer) each subproblem recursively.
- Combine two solutions into overall solution.

Consequence:

- Brute force: $\Theta(n^2)$.
- Divide-and-conquer: $O(n\log n)$.

Brute-force algorithm may already be polynomial time, and the divide and conquer strategy is to reduce the running time to a lower polynomial.

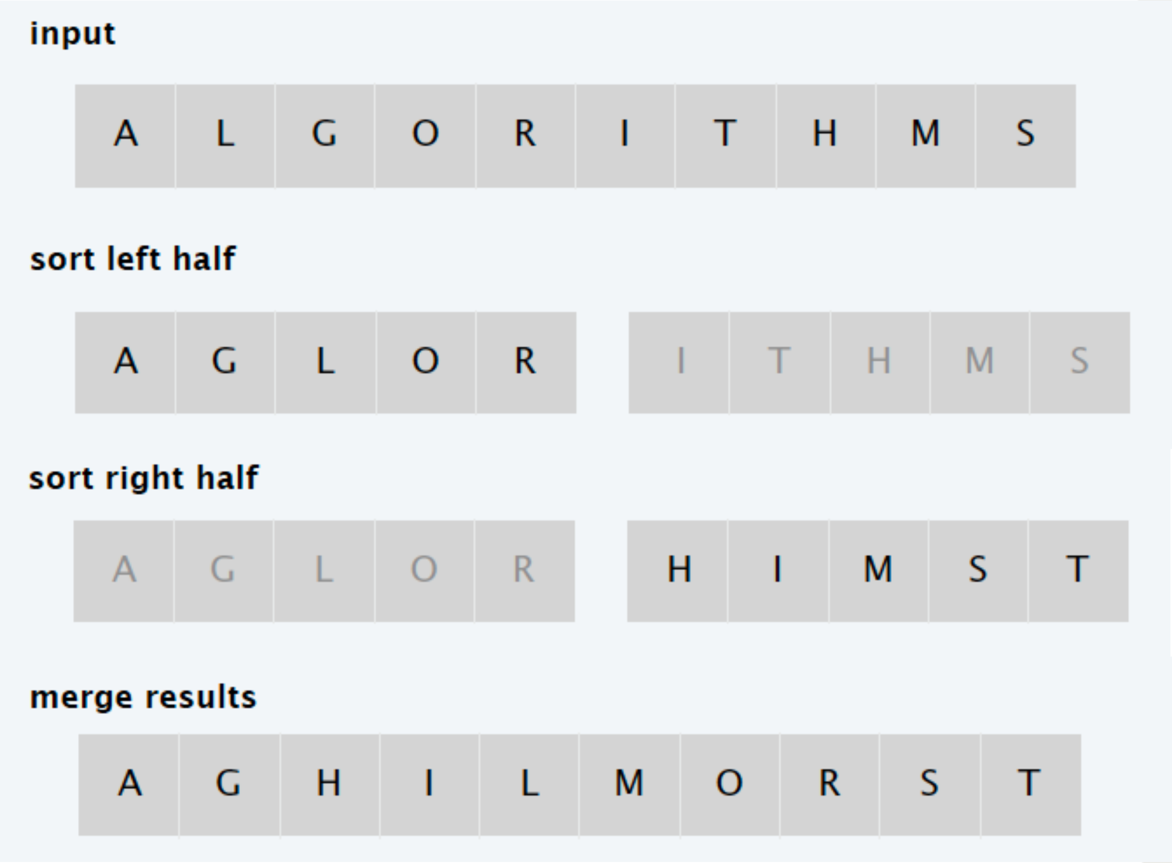
THE MERGESORT ALGORITHM

The Mergesort Algorithm

Problem. Given a list L of n elements from an ordered universe, rearrange them in ascending order.

The algorithm

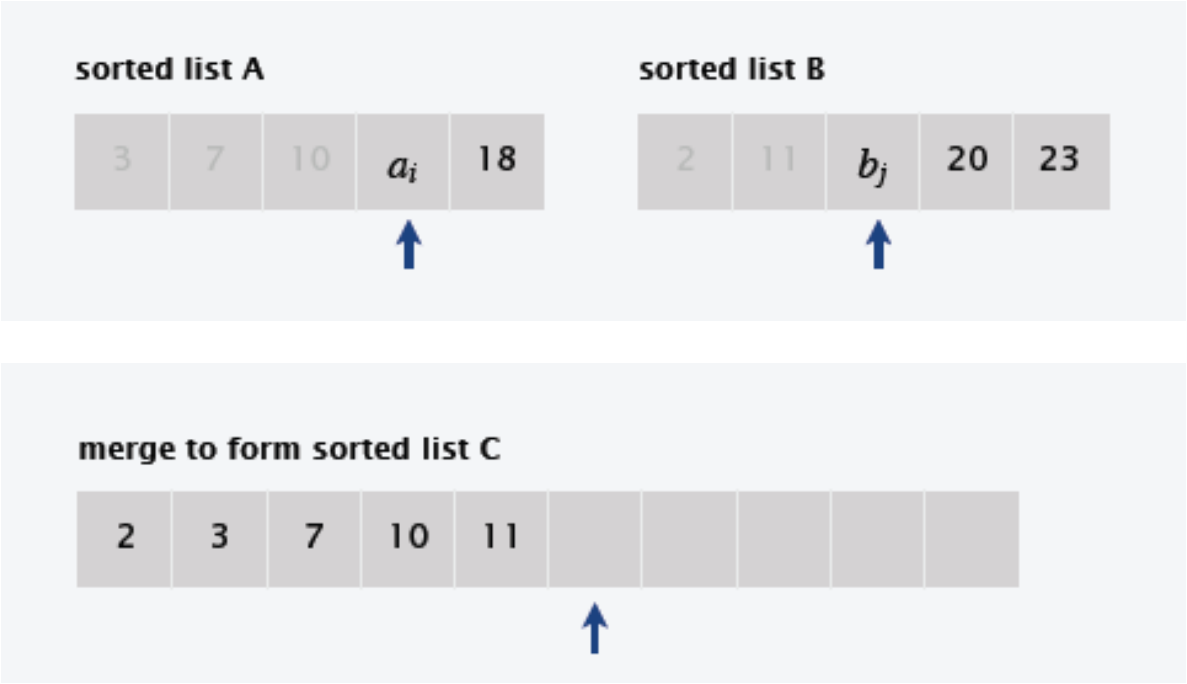
- Divide into left and right smaller problems.
- Recursively sort left half.
- Recursively sort right half.
- Merge two halves to make sorted whole.



How to do this?

The Mergesort Algorithm

Goal. Combine two sorted lists A and B into a sorted whole C.



The algorithm

- Scan A and B from left to right.
- Compare a_i and b_j.
- If $a_i < b_j$, append a_i to C (no larger than any remaining element in B).
- If $a_i > b_j$, append b_j to C (smaller than every remaining element in A).

$\Theta(n)$

The Mergesort Algorithm

Definition. $T(n)$ = max number of compares to Mergesort a list of length n.

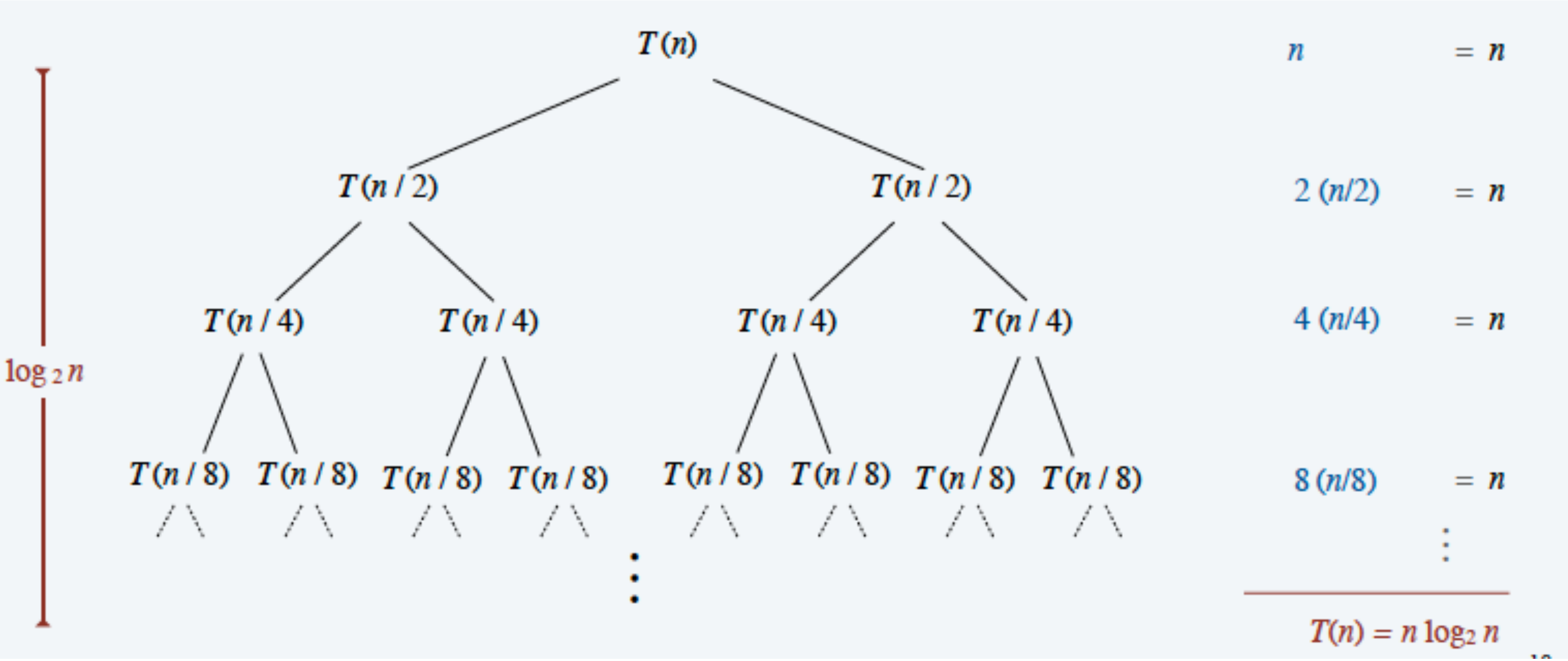
$$T(n) \le \begin{cases} 0 & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n & \text{if } n > 1 \end{cases}$$

Solving this recurrence: assume n is a power of 2 and replace $\le$ with $=$ in the recurrence.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

The Mergesort Algorithm

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2\,T(n/2) + n & \text{if } n > 1 \end{cases}$$



The Mergesort Algorithm

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2\,T(n/2) + n & \text{if } n > 1 \end{cases}$$

Proposition. If $T(n)$ satisfies the recurrence, then $T(n) = n\log_2 n$.

Proof. [by induction on n]

- ***Base case***: when $n = 1, T(1) = 0 = n\log_2 n$.
- ***Inductive hypothesis***: assume $T(n) = n\log_2 n$.
- ***Goal***: show that $T(2n) = 2n\log_2 2n$.

What if n is not a power of 2??

$$\begin{aligned} T(2n) &= 2T(n) + 2n \\ &= 2n\log_2 n + 2n \\ &= 2n[\log_2 2n - 1] + 2n \\ &= 2n\log_2 2n \end{aligned}$$

inductive hypothesis →

Thank You!