Greedy Algorithms

Admin

- Assignment 2 out
- Assignment 0 graded (full points for all, but check comments)
- Tip at home: pin video. Can swap me to be big using the button on the top-right
- Any questions or comments before we begin?

Tablet “board” or black board?

Greedy: Examples

- Cashier's algorithm to return change in coins?
  - Greedy! To make change for $\$r$, start with biggest denomination less than r, and so on
  - Optimal for US coins!
  - (Not in general)

Greedy: Locally Optimal

Greedy algorithms build solutions by making locally optimal choices

Surprisingly, sometimes this also leads to globally optimal solutions!

We start with greedy algorithms as the first design paradigm because

- They are natural and intuitive
- Proving they are optimal is the hard part

Greedy Algorithms Takeaway

- The takeaway is that **greedy algorithms *do not usually work***

- When greedy algorithms work, it is because the problem has structure that greedy can take advantage of

- The question is not “can I use greedy”—-it’s “**what structure does the problem have?** Does it lead to a greedy approach?”

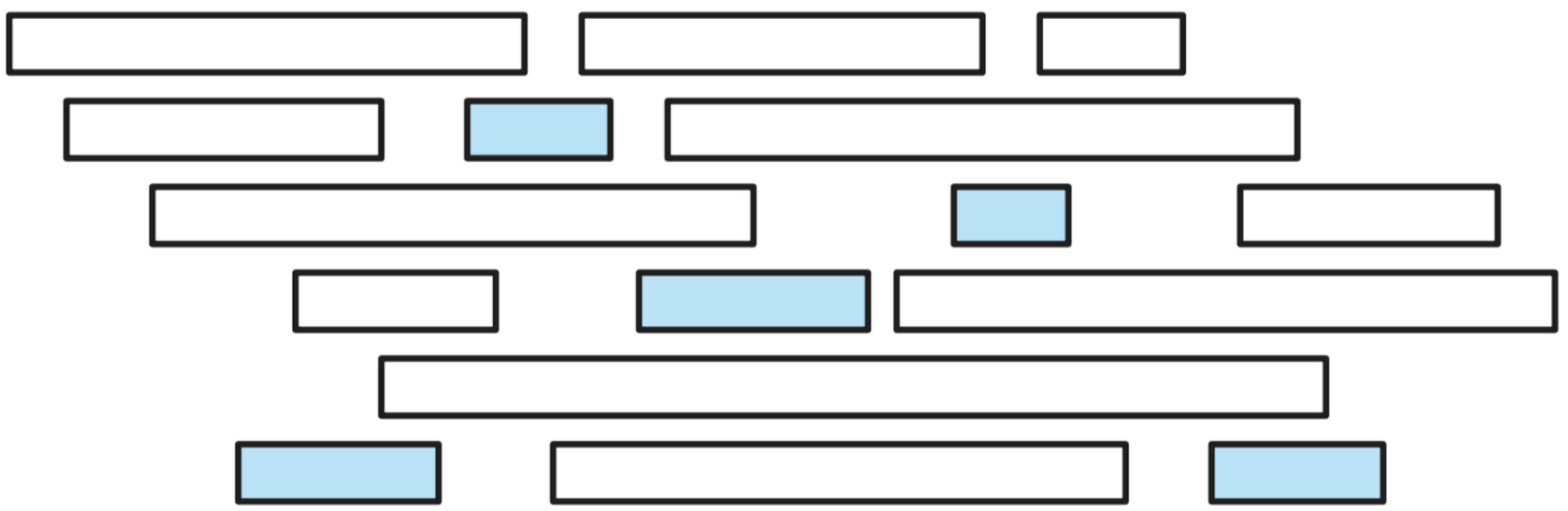
Greedy: Proof Techniques

Two fundamental approaches to proving correctness of greedy algorithms

- **Greedy stays ahead**: Partial greedy solution is, at all times, as good as an "equivalent" portion of any other solution
- **Exchange Property**: An optimal solution can be transformed into a greedy solution without sacrificing optimality.

Class Scheduling

Problem. Given the list of start times $s_1, \ldots, s_n$ and finish times $f_1, \ldots, f_n$ of n classes (labeled $1,\ldots,n$), what is the maximum number of non-conflicting classes you can schedule?

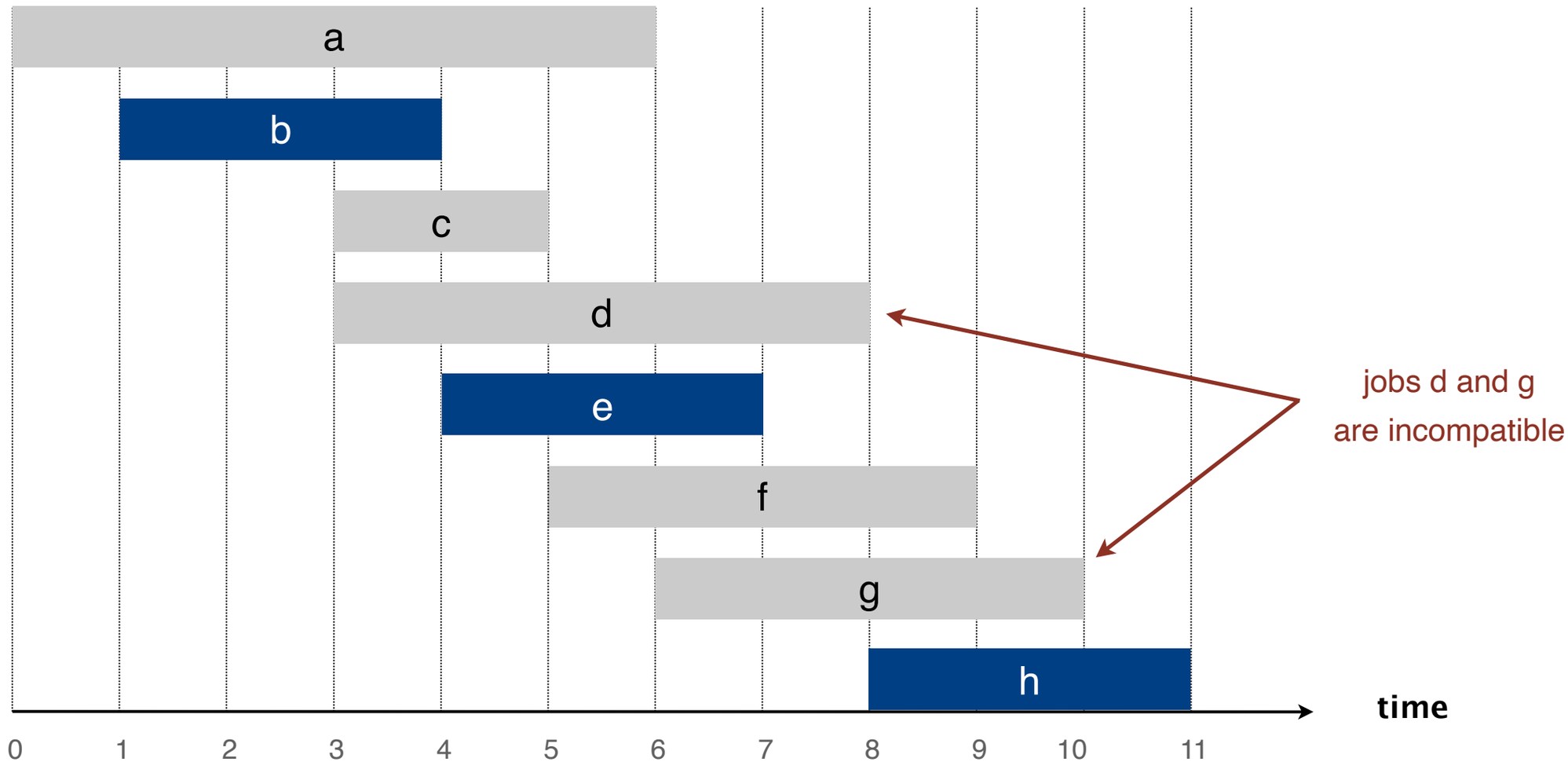


A maximum conflict-free schedule for a set of classes.

From Erickson's Algorithms Book

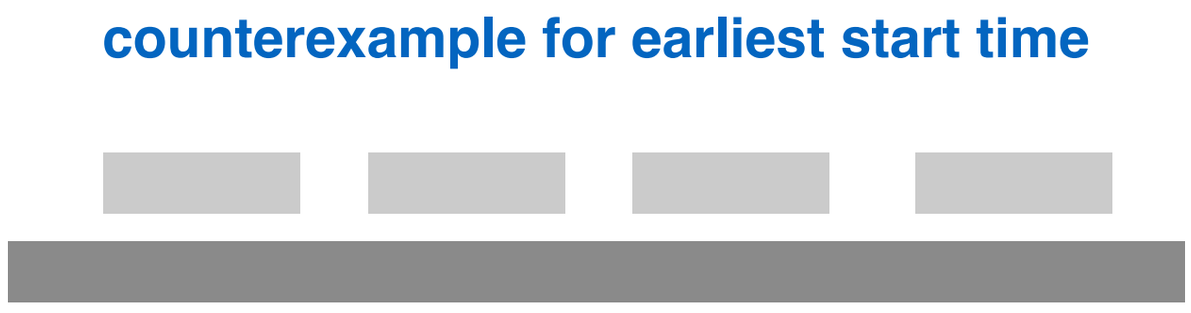
Interval Scheduling

Job scheduling. This is a general job scheduling problem. Suppose you have a machine that can run one job at a time and n job requests with start and finish times: $s_1, \ldots, s_n$ and $f_1, \ldots, f_n$. How to determine the most number of compatible requests?



What to be Greedy About?

- Algorithmic idea: Pick a criterion to be greedy about. Keep choosing compatible jobs based on it
- Lets start with obvious one: **start times**
  - Schedule jobs with **earliest start time** first
- Is this the best way?
  - If not, can we come up with a counter example?



Many Ways to be Greedy

- Algorithmic idea: Pick a criterion to be greedy about. Keep choosing compatible jobs based on it
- Another possible criterion:
  - Schedule jobs with **shortest interval** first
  - That is, smallest value of $f_i - s_i$

counterexample for shortest interval

Many Ways to be Greedy

- Algorithmic idea: Pick a criterion to be greedy about. Keep choosing compatible jobs based on it
- Another possible criterion:
  - **Fewest conflict**
  - Schedule that conflict with fewest other jobs first

counterexample for fewest conflicts

Many Ways to be Greedy

- Algorithmic idea: Pick a criterion to be greedy about. Keep choosing compatible jobs based on it
- Criteria that do not work:
    - Earliest start time first
    - Shortest interval works
    - Fewest conflict first
- How about: **earliest finish time first?**
    - Surprisingly optimal
    - Need to prove why it is optimal
    - Idea: Free your resource as soon as possible!

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 $f_1 \leq f_2 \leq \ldots \leq f_n$.

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

FOR $j = 1$ TO n

  IF job j is compatible with S

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

RETURN S.

Correctness of Algorithm

- Set S output consists of compatible requests
    - By construction!
- We want to prove our solution S is optimal (schedules the maximum number of jobs)
- Let $\mathcal{O}$ be an optimal set of jobs. **Goal:** show $|S| = |\mathcal{O}|$,i.e., greedy also selects the same number of jobs and thus is optimal
- **Proof technique to prove optimality:**
    - Greedy always “stays ahead” (or rather never falls behind)
    - We will compare partial solutions of greedy vs an optimal and show that greedy is doing better or just as well
    - **Intuition:** greedy frees up the resource as soon as possible
    - Lets use this metric to compare greedy and optimal

Get Ahead Stay Ahead Proof

Correctness proof. Let $g_1, \ldots, g_k$ and $o_1, \ldots, o_m$ be the sequence of compatible jobs selected by the greedy and optimal algorithm respectively, ordered by increasing finish time.

Lemma 1. For all $i \leq k$, we have: $f_{g_i} \leq f_{o_i}$.

Proof. (By induction) Base case: $i = 1$ is true, why?

- Assume holds for $k-1$: $f_{g_{k-1}} \leq f_{o_{k-1}}$
- For kth job, note that $f_{o_{k-1}} \leq s_{o_k}$ (why?)
- Using inductive hypothesis: $f_{g_{k-1}} \leq f_{o_{k-1}} \leq s_{o_k}$
- Greedy picks earliest finish time among compatible jobs (which includes o_k) thus $f_{g_k} \leq f_{o_k}$

∎

Are We Done? Almost

Let $g_1, \ldots, g_k$ and $o_1, \ldots, o_m$ be the sequence of compatible jobs selected by the greedy and optimal algorithm respectively, ordered by finish times.

Lemma 1. For all $i \leq k$, we have: $f_{g_i} \leq f_{o_i}$.

Lemma 2. The greedy algorithm returns an optimal set of jobs S, that is, $k = m$.

Proof. (By contradiction)

Suppose S is not optimal, then the optimal set $\mathcal{O}$ must select more jobs, that is, $m > k$.

That is, there is a job o_{k+1} that starts after o_k ends

What is the contradiction? Greedy keeps selecting jobs until no more compatible jobs left. Since $f_{g_k} \leq f_{o_k}$ by Lemma 1, greedy would also select compatible job o_{k+1} ($\Rightarrow\!\!\!\!\!\!\Leftarrow$) ∎

Implementation & Running Time

Analysis (Running time):

Let's analyze all the steps:

- Sorting jobs by finish times
    - $O(n \log n)$
- Permuting start times in the order of finish times
    - $O(n)$ total (how can we do this bookkeeping?)
- For each selected job i, find next job j such that $s_j \geq f_i$
    - Iterate through the list until you reach the right interval j
    - This part of the algorithm is $O(1)$ per interval, so $O(n)$
- Overall $O(n \log n)$ time

Greedy Algorithms: Class Quiz

Question.

- Suppose that each job also has a positive weight and the goal is to find a maximum weight subset of mutually compatible intervals.
- Is the earliest-finish-time-first algorithm still optimal?
- If no, can we design a simple counter example?

Minimizing Lateness: Problem

Given: A list of processes needs to be scheduled

- Only one process can be executed at a time
- A process must run to completion before another can be executed
- Each process has a duration t_i and a deadline d_i

Goal: Schedule tasks: $(t_i, d_i) \rightarrow (s_i, f_i)$ (start & finish times), where $f_i = s_i + t_i$, to minimize maximum lateness

- Satisfy all requests but optimize max lateness
- Lateness of process i : $L_i = \max\{0, f_i - d_i\}$
- Resource is first available at time 0

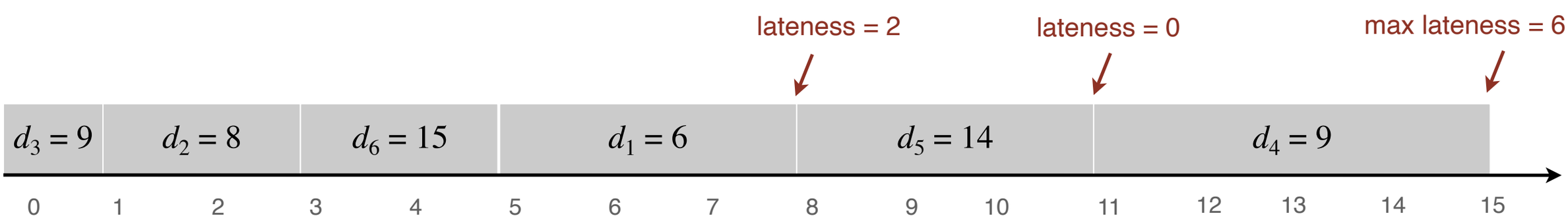
Minimizing Lateness: Problem

Given: A list of processes needs to be scheduled, each process has a duration t_i and a deadline d_i

Goal: Schedule tasks: $(t_i, d_i) \rightarrow (s_i, f_i)$ (start & finish times), where $f_i = s_i + t_i$, to minimize maximum lateness

- Lateness of process i : $L_i = \max\{0, f_i - d_i\}$

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



Minimizing Lateness: Problem

Possible strategies?

- Shortest jobs first (get more done faster!)
- Do jobs with shortest slack time first (slack of job i is $d_i - t_i$)
- Earlier deadlines first

Shortest job first:

	1	2
t_j	1	10
d_j	100	10

counterexample

Gives max lateness: 1
OPT max lateness: 0

Minimizing Lateness: Problem

Possible strategies

- Shortest jobs first (get more done faster!)
- Do jobs with shortest slack time first (slack of job i is $d_i - t_i$)
- Earlier deadlines first

Shortest slack first:

	1	2
t_j	1	10
d_j	2	10

counterexample

Gives max lateness: 9
OPT max lateness: 1

Minimizing Lateness: Problem

Possible strategies

- Shortest jobs first (get more done faster!)
- Do jobs with shortest slack time first (slack of job i is $d_i - t_i$)
  - Earlier deadlines first

- Order jobs by their deadline and schedule them in that order
- **Intuition:** get the jobs due first done first
- Surprisingly optimal (We will show this)
- Disregards job lengths! (Seems counter-intuitive)

Earliest Deadline First

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 $d_1 \leq d_2 \leq \ldots \leq d_n$.

$t \leftarrow 0$.

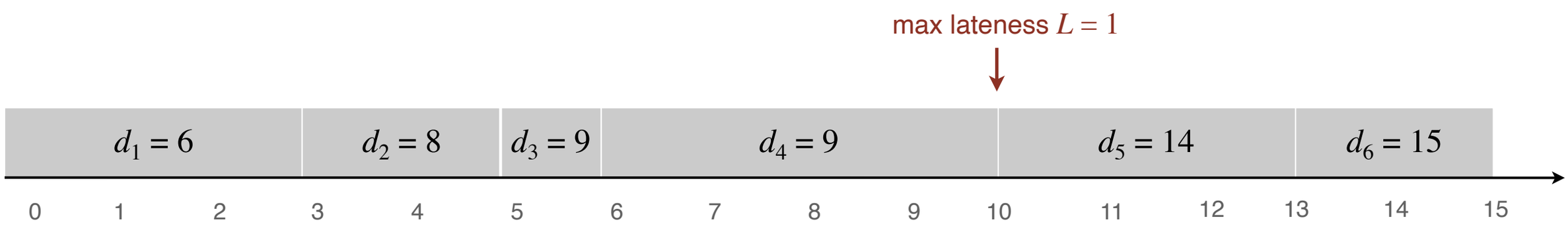
FOR $j = 1$ TO n

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

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

  $t \leftarrow t + t_j$.

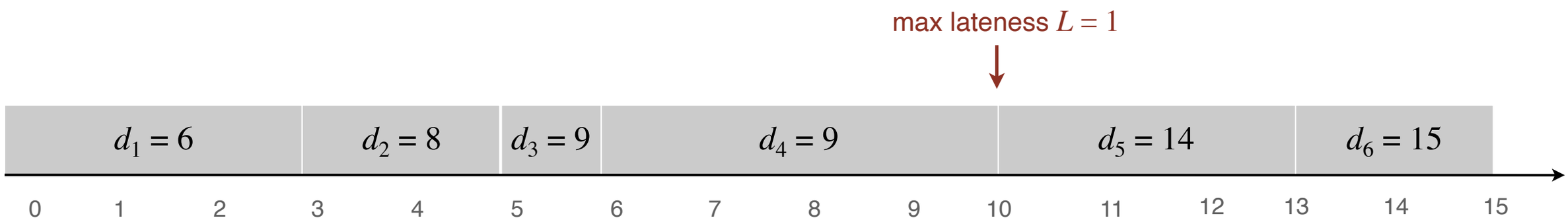
RETURN intervals $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$.



Recall: Scheduling with Deadlines

Given interval length t_i and deadline d_i for $i \in \{1, \ldots, n\}$ jobs, schedule all tasks, that is, assign start and finish times $(t_i, d_i) \rightarrow (s_i, f_i)$, where $f_i = s_i + t_i$, so as to minimize the maximum lateness.

- Lateness of process i : $L_i = \max\{0, f_i - d_i\}$



Greedy: Earliest-Deadline First

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 $d_1 \leq d_2 \leq \ldots \leq d_n$.

$t \leftarrow 0$.

FOR $j = 1$ TO n

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

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

  $t \leftarrow t + t_j$.

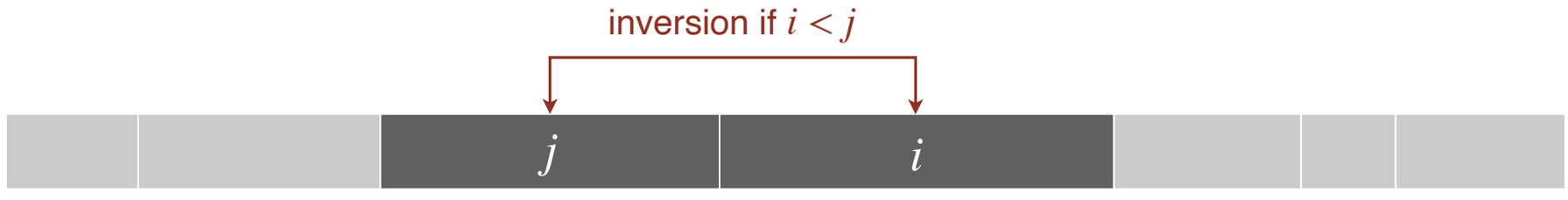
RETURN intervals $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$.

Minimizing Lateness: Greedy

Observations about our greedy algorithm

- It produces a schedule with no idle time
- It produces a schedule with no inversions
  - i, j is an inversion if job j is scheduled before i but i's deadline is earlier ($d_i < d_j$)

a schedule with an inversion



recall: we assume the jobs are numbered so that $d_1 \leq d_2 \leq \ldots \leq d_n$

Structure of the Solution

- **Notice:** All schedules with no inversions and no idle time have the same maximum lateness
  - Distinct deadlines, unique schedule
  - Non-distinct deadlines: Consider two jobs with deadline d; the maximum lateness does not depend on the order in which they are scheduled
- Say the two jobs have duration t_i, t_j and same deadline d
- If i is scheduled first at time s, the max lateness is: $\max\{0,(s + t_i + t_j) - d\}$
- If j is scheduled first at time s, the max lateness is the same: $\max\{0,(s + t_i + t_j) - d\}$

Minimizing Lateness: Review

Given: A list of processes with a duration t_i and a deadline d_i

Goal: Schedule tasks: $(t_i, d_i) \rightarrow (s_i, f_i)$ (start & finish times), where $f_i = s_i + t_i$, to minimize maximum lateness

- Lateness of process i : $L_i = \max\{0,\ f_i - d_i\}$

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

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

Structure of the Solution

- **Notice:** All schedules with no inversions and no idle time have the same maximum lateness
  - Distinct deadlines, unique schedule
  - Non-distinct deadlines: Consider two jobs with deadline d; the maximum lateness does not depend on the order in which they are scheduled
- Say the two jobs have duration t_i, t_j and same deadline d
- If i is scheduled first at time s, the max lateness is: $\max\{0,(s + t_i + t_j) - d\}$
- If j is scheduled first at time s, the max lateness is the same: $\max\{0,(s + t_i + t_j) - d\}$

Where We Are Going

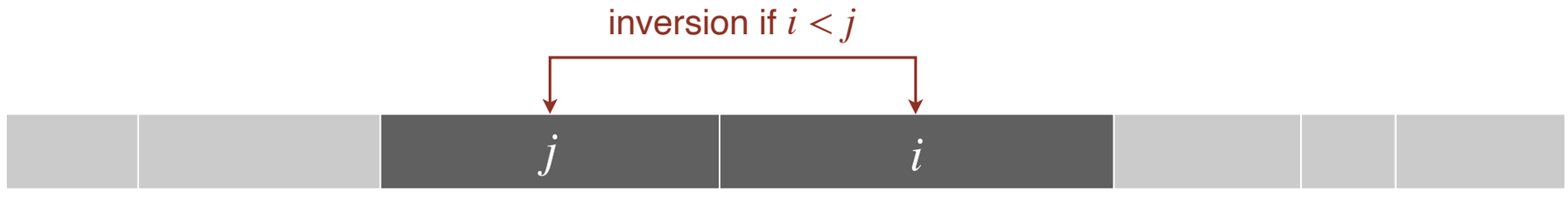
- **Notice:** All schedules with no inversions and no idle time have the same maximum lateness
  - Distinct deadlines, unique schedule
  - Non-distinct deadlines: Consider two jobs with deadline d; the maximum lateness does not depend on the order in which they are scheduled
- **Goal.** show there exists an optimal schedule with no inversions and no idle time
- Then, we have shown that the optimal schedule has the same maximum lateness as greedy!
- We will show this via **an exchange argument**
- Second proof technique to prove greedy is optimal

Minimizing Lateness: Greedy

Observations about our greedy algorithm

- It produces a schedule with no idle time
- It produces a schedule with no inversions
  - i, j is an inversion if job j is scheduled before i but i's deadline is earlier ($d_i < d_j$)

a schedule with an inversion

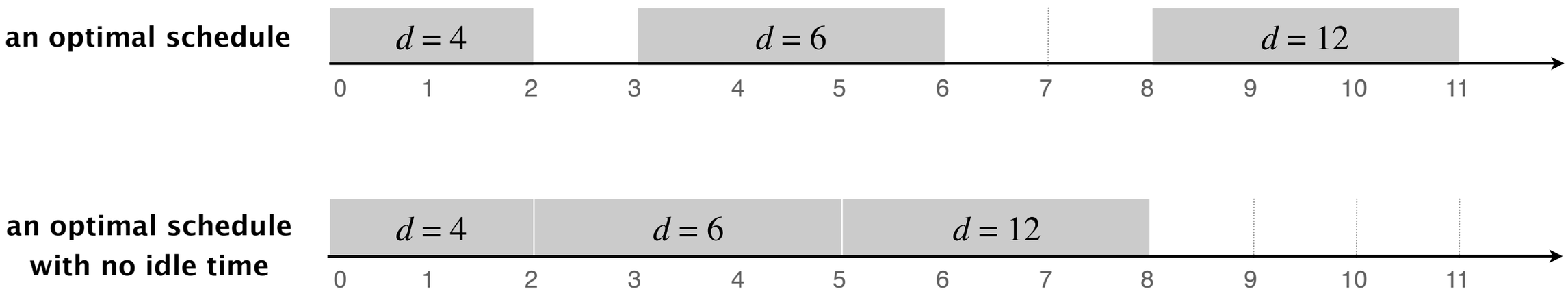


recall: we assume the jobs are numbered so that $d_1 \leq d_2 \leq \ldots \leq d_n$

Structure of Optimal

Observation about optimal.

- There exists an optimal schedule with no idle time.
- (Can always schedule jobs earlier to prevent idleness!)



Structure of Optimal: Inversions

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

Recall. i, j is an inversion if job j is scheduled before i but i's deadline is earlier ($d_i < d_j$)

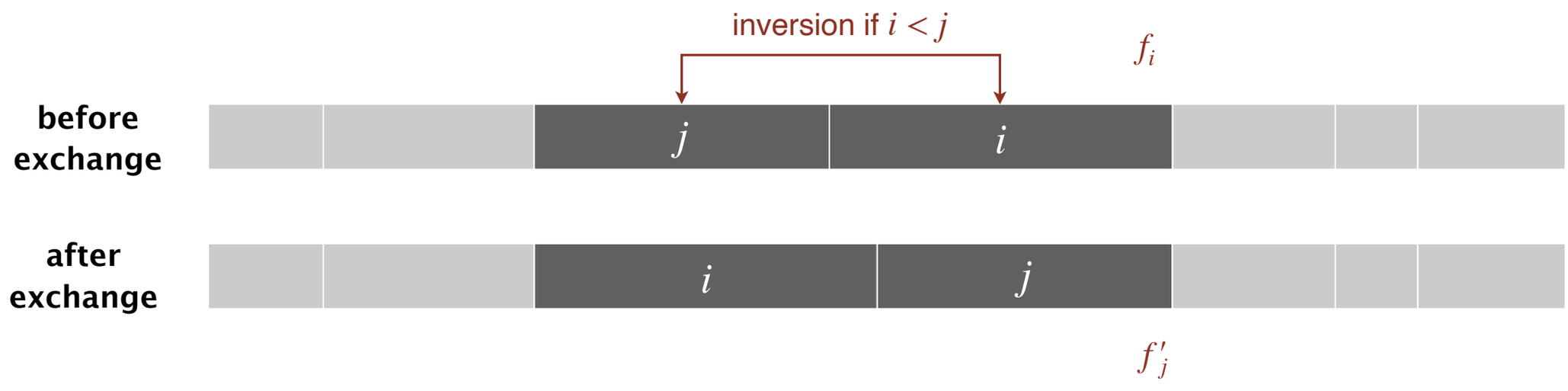
Proof. [Contradiction]

- Let i, j be any two non-adjacent inversions without another inversion between them
- Let k be element immediately to the right of j.
- Case 1. $[d_j > d_k]$ Then j, k is an adjacent and closer inversion ($\Rightarrow\Leftarrow$)
- Case 2. $[d_j < d_k]$ Since $d_i < d_j$, this means that i, k is a closer inversion ($\Rightarrow\Leftarrow$) ∎

Structure of Optimal: Inversions

Claim. Given a schedule with k inversions, we can modify it to schedule with $k - 1$ inversions (without increasing the maximum lateness).

Proof. (Key Idea) Exchanging two adjacent, inverted jobs i and j reduces the number of inversions by 1 and does not increase the maximum lateness.

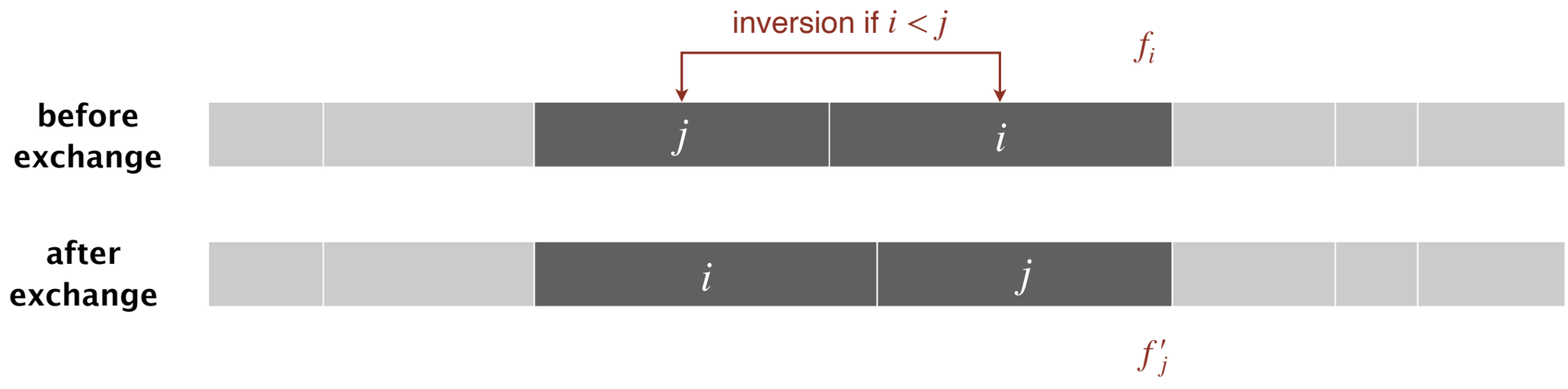


Structure of Optimal: Inversions

Claim. Given a schedule with k inversions, we can modify it to schedule with $k - 1$ inversions (without increasing the maximum lateness).

Proof. Let i, j be inverted jobs with $i < j$. Let ℓ be the lateness before swapping them and ℓ' after the swap.

- $\ell_k = \ell'_k \qquad \forall k \neq i, j$ (swap doesn't affect other jobs)
- $\ell'_i \leq \ell_i$ (lateness of i improves after swap)
- $\ell'_j = f'_j - d_j = f_i - d_j \leq f_i - d_i \leq \ell_i$ ∎



Optimality of Greedy

Summarizing the proof.

- All schedules with no inversions and no idle time have the same maximum lateness
- Greedy schedule has no inversions and no idle time
- Consider an optimal schedule $\mathcal{O}$, without loss of generality, we can assume that
  - $\mathcal{O}$ has no idle time
  - $\mathcal{O}$ has no inversions, why?
  - **[Iterate and exchange].** If there is an inversion, must be adjacent, exchanging them decreases # of inversions by 1 without increasing max lateness (repeat until no inversions)
- Greedy and $\mathcal{O}$ have same max lateness. ∎

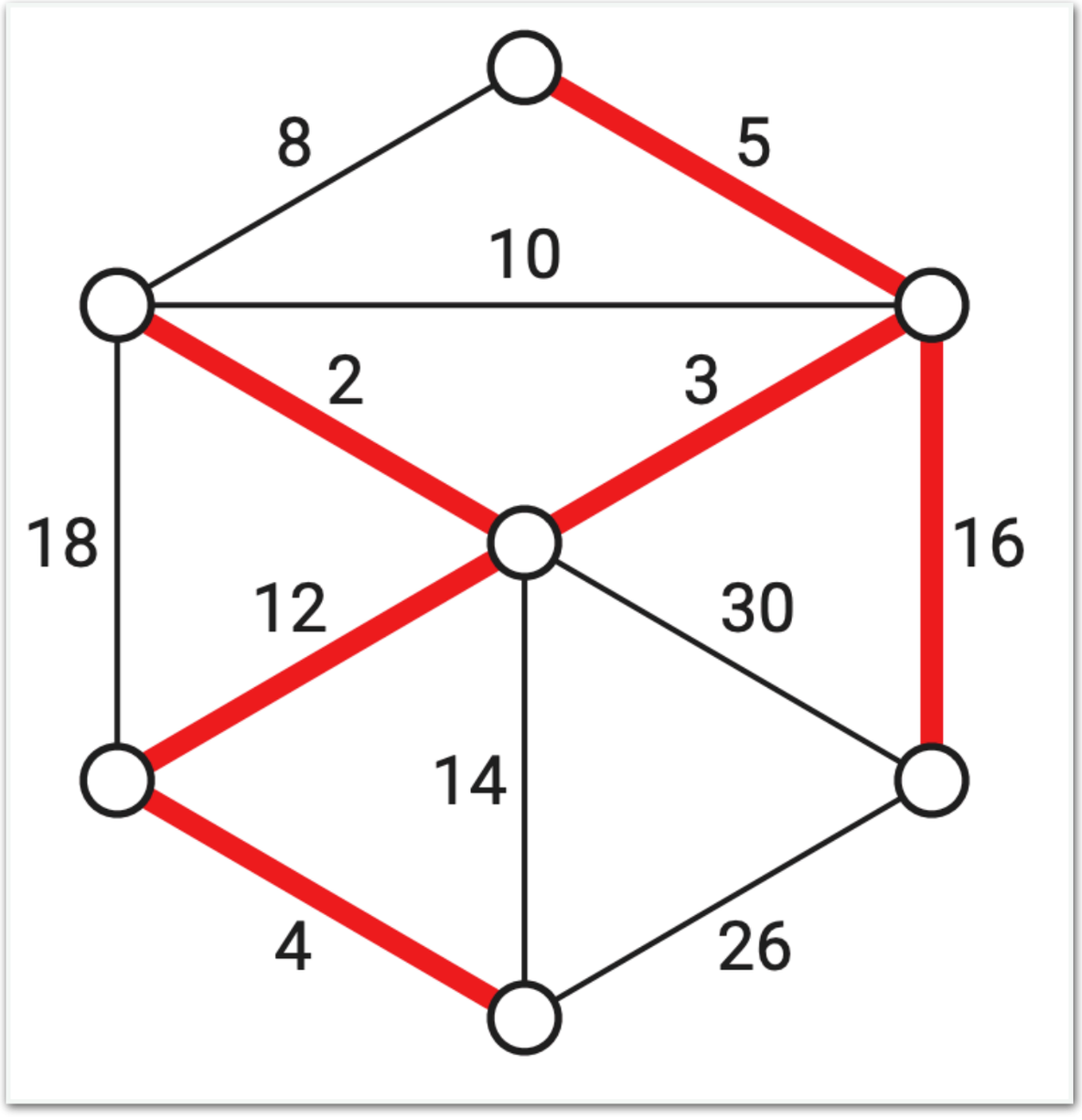
Exchange Argument

General Pattern. An inductive exchange argument

- You start with an arbitrary optimal solution
- Prove that WLOG it must have certain nice properties
- Assume there is an optimal solution that is different from the greedy solution
- Find the “first” difference between the two solutions
- Argue that we can exchange the optimal choice for the greedy choice without making the solution worse (although the exchange may make it better)
- Show that you can iteratively perform the exchange step until you get the greedy solution

Greedy Graph Algorithms: Minimum Spanning Trees

Minimum Cost Spanning Trees



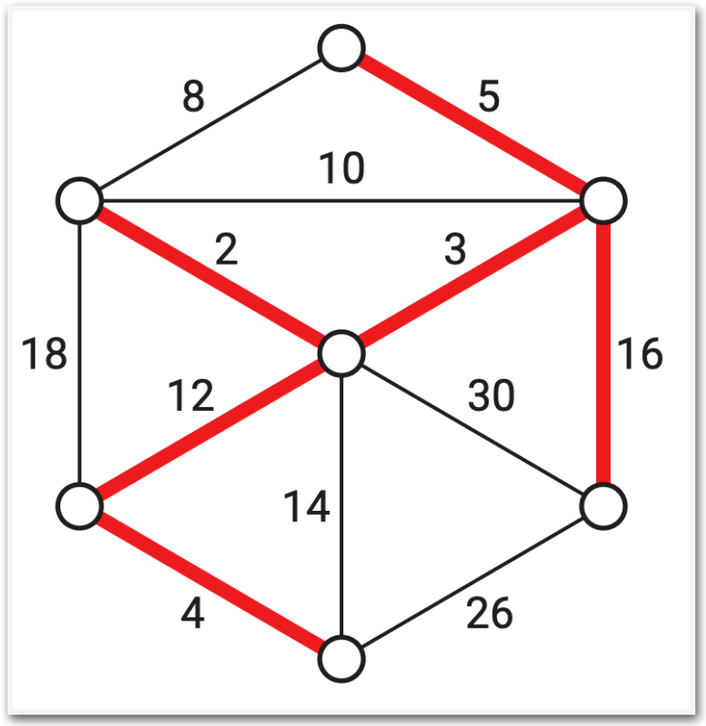
Minimum Spanning Trees

- Many applications!
  - Classic application:
    - Underground cable (Power, Telecom, etc)
  - Efficient broadcasting on a computer network (Note: different from shortest paths)
  - Approximate solutions to harder problems, such at TSP
  - Real-time face verification

Minimum Spanning Trees

Problem. Given a connected, undirected graph $G = (V, E)$ with edge costs w_e, output **a minimum spanning tree**, i.e., set of edges $T \subseteq E$ such that

- (a spanning tree of G): T connects all vertices
- (has minimum weight): for any other spanning tree T' of G, we have $\sum_{e \in T} w_e \leq \sum_{e \in T'} w_e$



Distinct Edge Weights

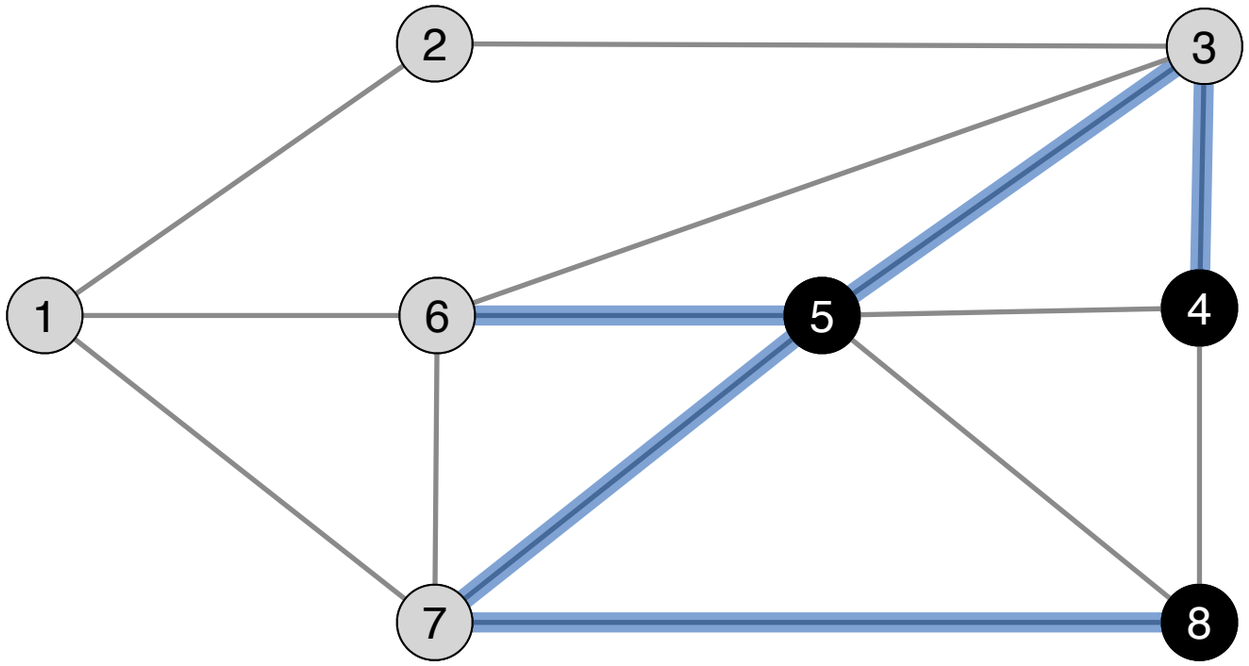
- Annoying subtlety in the problem statement is there may be multiple minimum spanning trees
    - If a graph has edges with same edge, e.g., all edges have weight 1: all spanning trees are min!
- To simplify discussion in our algorithm design, we will assume distinct edge weights

Lemma. If all edge weights in a connected graph are distinct, then it has a unique minimum spanning tree.

We will relax the distinct-edge-weight assumption later.

Spanning Trees and Cuts

A **cut** is a partition of the vertices into two **nonempty** subsets S and $V - S$. A **cut edge** of a cut S is an edge with one end point in S and another in $V - S$.



cut S = { 4, 5, 8 }

Cut edges = { (3, 4), (3, 5), (5, 6), (5, 7), (8, 7) }

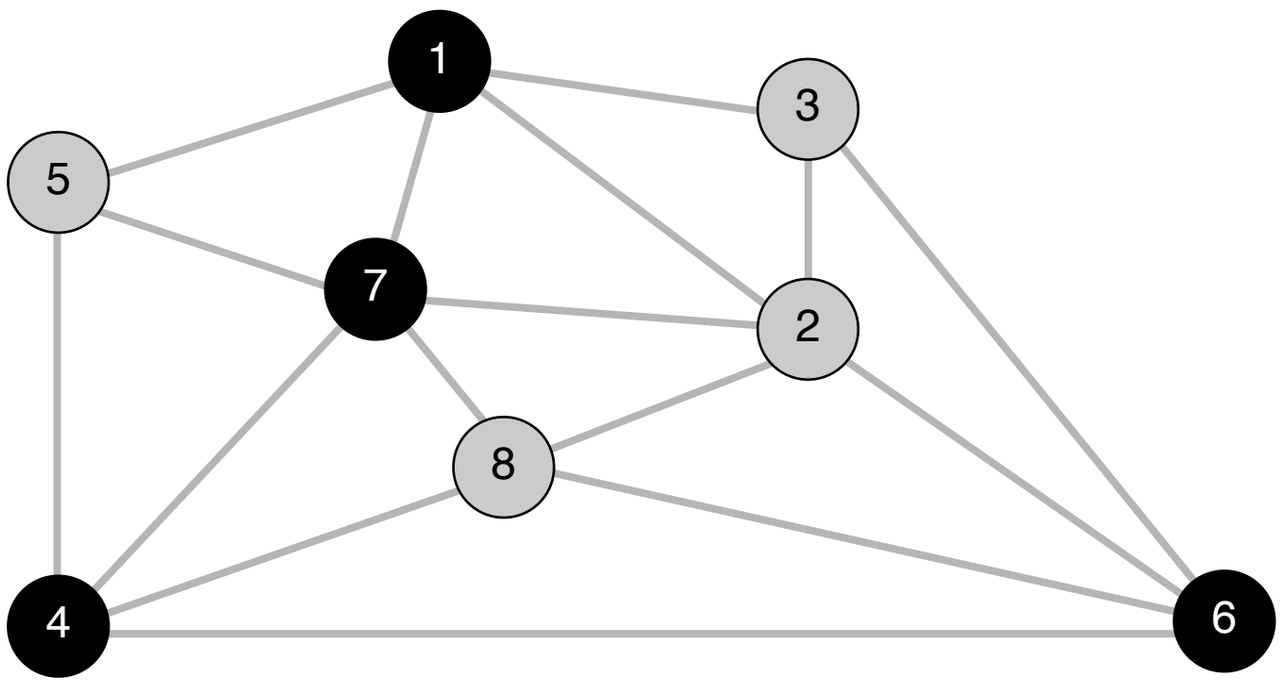
Spanning Trees and Cuts

Question. Consider the cut $S = \{1,4,6,7\}$. Which of the following edges are cut edges with respect to this cut?

A. (1, 7)

B. (5, 7)

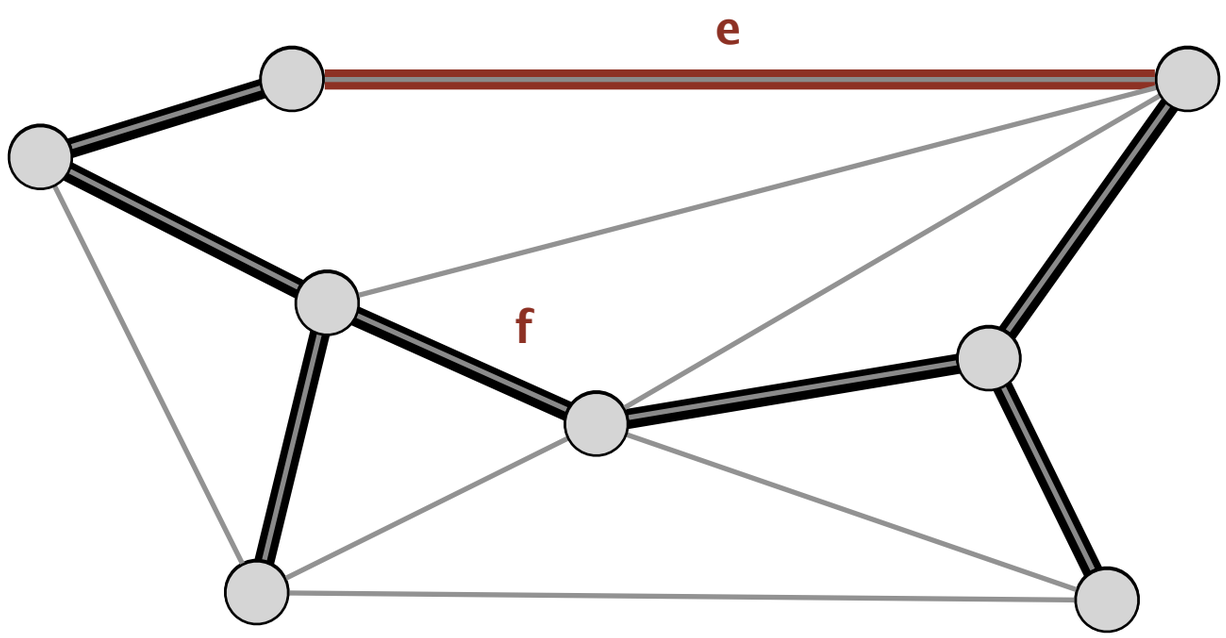
C. (2, 3)



Fundamental Cycle

Let T be a spanning tree of G.

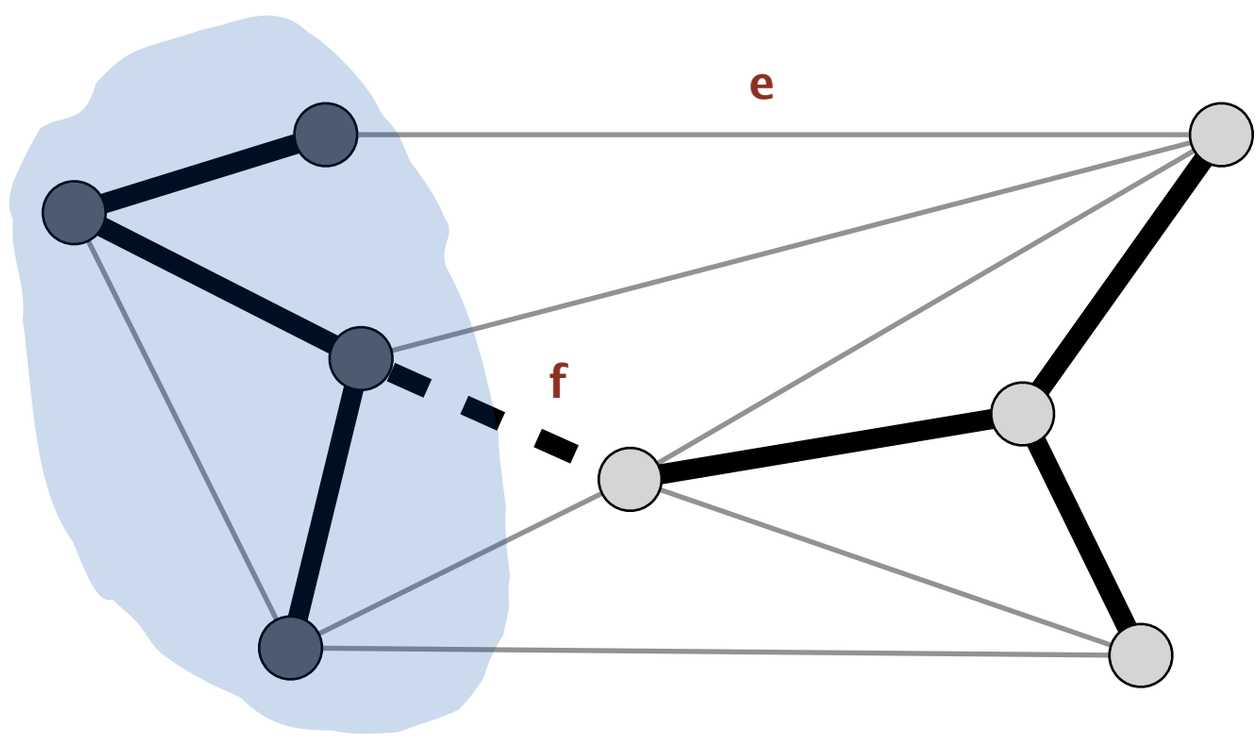
- For any edge $e \notin T$, $T \cup \{e\}$ creates a unique cycle C
- For any edge $f \in C$: $T \cup \{e\} - \{f\}$ is a spanning tree



Fundamental Cut

Let T be a spanning tree of G.

- For any edge $f \in T$, $T - \{f\}$ breaks the graph into two connected components, let D be the set of cut edges with end points in each component
- For any edge $e \in D : T - \{f\} \cup \{e\}$ is a spanning tree



Spanning Trees and Cuts

Lemma (Cut Property). For any cut $S \subset V$, let $e = (u, v)$ be the minimum weight edge connecting any vertex in S to a vertex in $V - S$, then every minimum spanning tree must include e.

Proof. (By contradiction via an exchange argument)

Suppose T is a spanning tree that does not contain $e = (u, v)$.

Main Idea: We will construct another spanning tree $T' = T \cup e - e'$ with weight less than T ($\Rightarrow\Leftarrow$)

How to find such an edge e'?

Exchange argument

Spanning Trees and Cuts

Proof (Cut Property).

Suppose T is a spanning tree that does not contain $e = (u, v)$.

- Adding e to T results in a unique cycle C (why?)
- C must "enter" and "leave" cut S, that is, $\exists e' = (u', v') \in C$ s.t. $u' \in S, v' \in V - S$
- $w(e') > w(e)$ (why?)
- $T' = T \cup e - e'$ is a spanning tree (why?)
- $w(T') < w(T)$

$(\Rightarrow\Leftarrow)$ ∎

Spanning Trees and Cycles

Lemma (Cycle Property). For any cycle C in G, its highest cost edge e is in no MST of G.

Proof. (By contradiction via an exchange argument)

Suppose a MST T contains e.

- Main Idea: We will construct another spanning tree $T' = T - \{e\} \cup \{e'\}$ with weight less than T ($\Rightarrow\Leftarrow$)
- How to find such an e'?

Exchange argument

Spanning Trees and Cycles

Lemma (Cycle Property). For any cycle C in G, its highest cost edge e is in no MST of G.

Proof. (By contradiction via an exchange argument)

Suppose a MST T contains $e \in C$.

- Let $e' \in C$ be an edge of weight less than e
- Exchanging e with e' in T creates another spanning tree $T - \{e\} \cup \{e'\}$
- Weight of $T - \{e\} \cup \{e'\}$ is less than weight of T
- But we assumed T was the minimum weight spanning tree

$(\Rightarrow\Leftarrow)$ ∎

Class Exercises

- Often the cut edge of minimum weight is called a *light edge*.
- **Exercise.** Show that
  - If for every cut of a graph there is a unique light edge crossing the cut, then the graph has a unique minimum spanning tree.
  - Show that the converse is not true by giving a counterexample.
- Remark: Do not assume distinct edge weights.

Class Exercise: Solution

- **Exercise.** Show that if for every cut of a graph there is a unique light edge crossing the cut, then the graph has a unique min spanning tree.

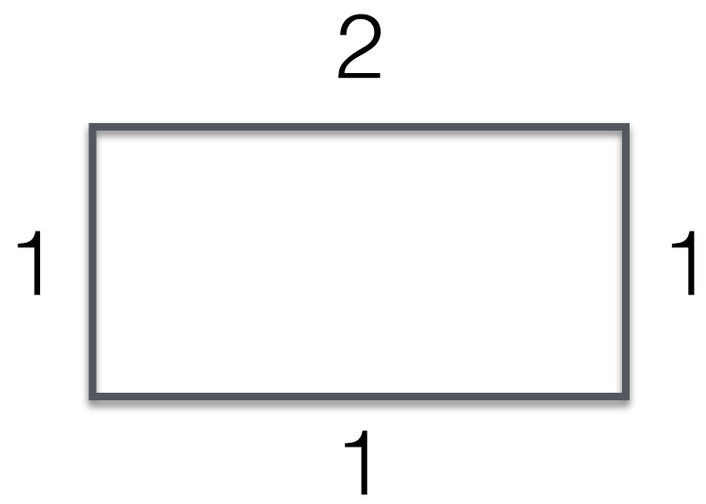
Proof. Suppose there is a unique light edge crossing every cut and there are two minimum spanning trees T_1 and T_2 of the graph.

- WLOG, as the trees are different there must exist an edge $e \in T_1$ such that $e \notin T_2$.
- $T_1 - \{e\}$breaks the graph into two connected components $(S, V - S)$
- Consider the unique light edge f going across cut $(S, V - S)$
  - **Case 1.** Suppose $e \neq f$, since e is also a cut edge of this cut, it must be that $w(f) < w(e)$. Then, $T_1 - \{e\} \cup \{f\}$ is a spanning tree of weight less than T_1. $(\Rightarrow\Leftarrow)$
  - **Case 2.** Suppose $e = f \notin T_2$, let e' be the cut edge in T_2 that crosses the cut $(S, V - S)$, then in $T_2 - \{e'\} \cup \{e\}$ is a spanning tree of weight less than T_2. $(\Rightarrow\Leftarrow)$ ∎

Class Exercises: Solution

- **Exercise.**

  Show that the converse is not true by giving a counterexample (that is, if a graph has a unique minimum spanning tree, then every cut must have a unique light edge.



Acknowledgments

- The pictures in these slides are taken from
    - Kleinberg Tardos Slides by Kevin Wayne (https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf)
    - Jeff Erickson's Algorithms Book (http://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf)