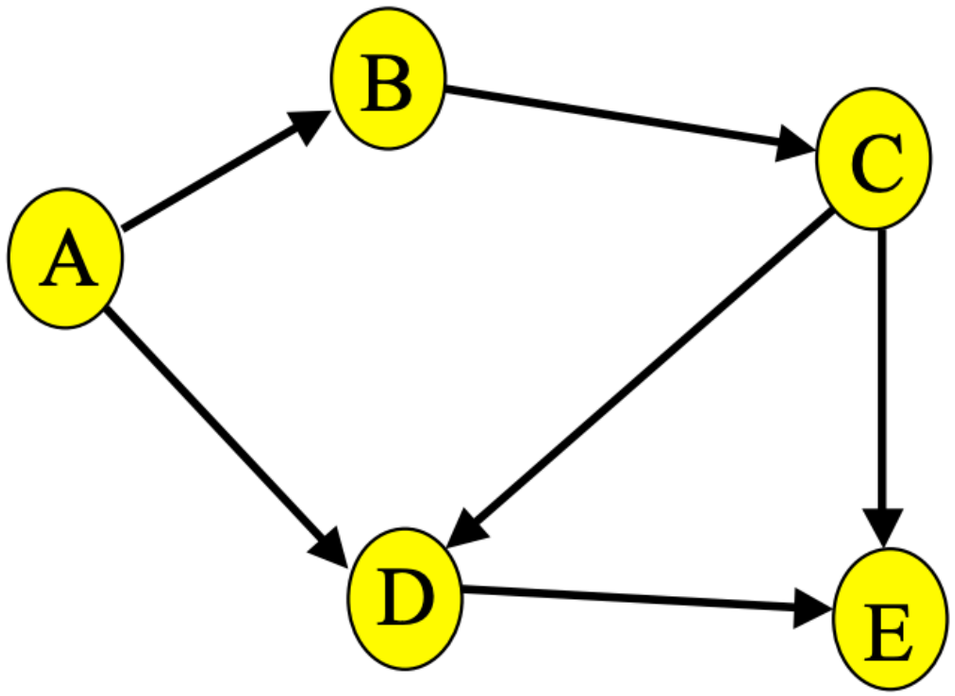
# Topological Sort and Greedy Algorithms

# Admin

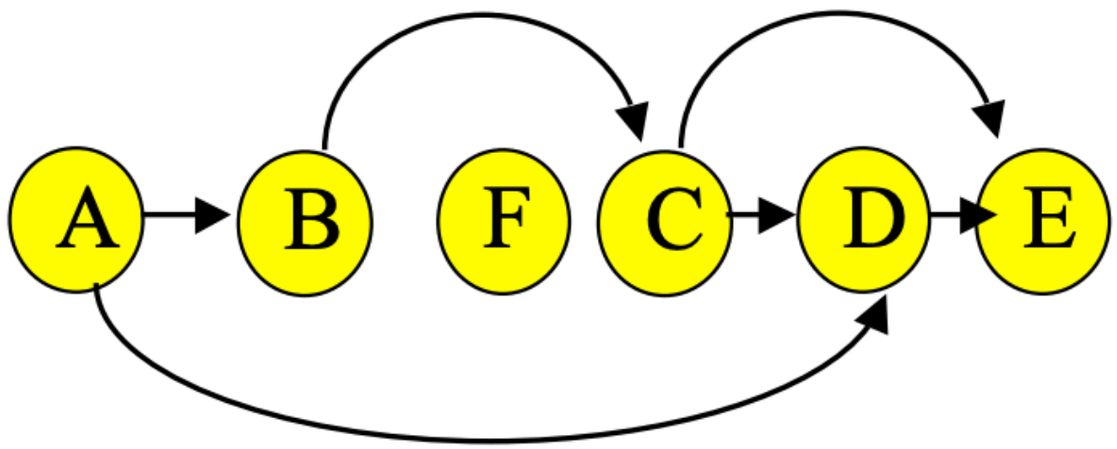
- Assignment 1 due tomorrow
- Office hours on google calendar
  - Mine 1-3 today, 3:30-5:30 tomorrow
- Assignment 0 back soon!

# *Assignment 0 Discussion*

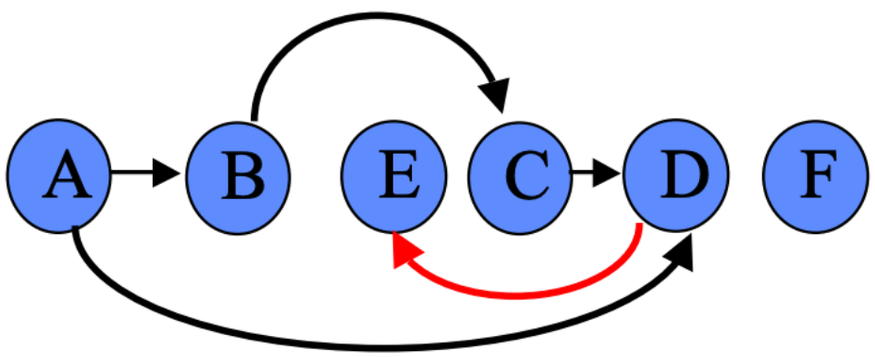
# Topological Ordering: Example




*Any linear ordering in which all the arrows go to the right is a valid solution*




*Not a valid topological sort!*



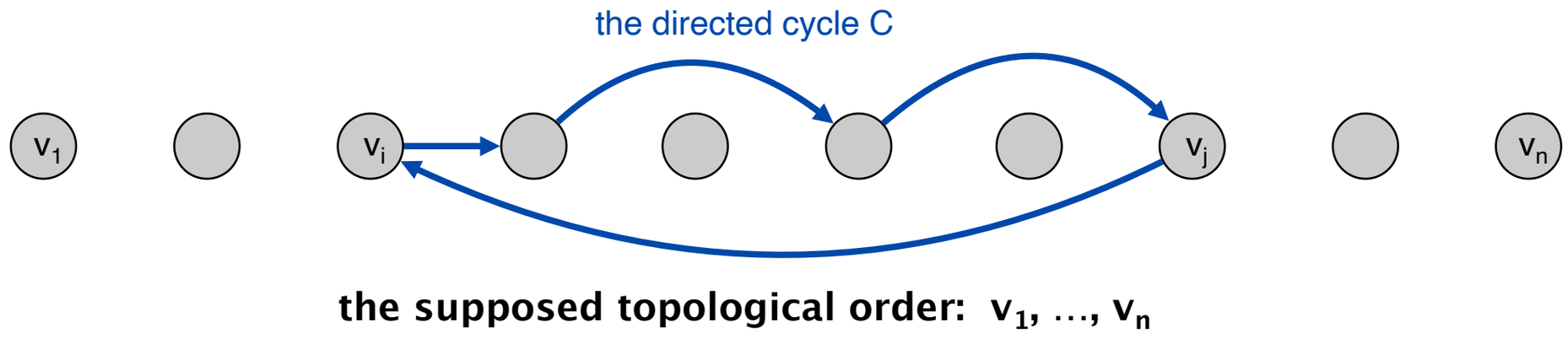

Taken from: https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf

# Topological Ordering and DAGs

**Lemma.** If $G$ has a topological ordering, then $G$ is a DAG.

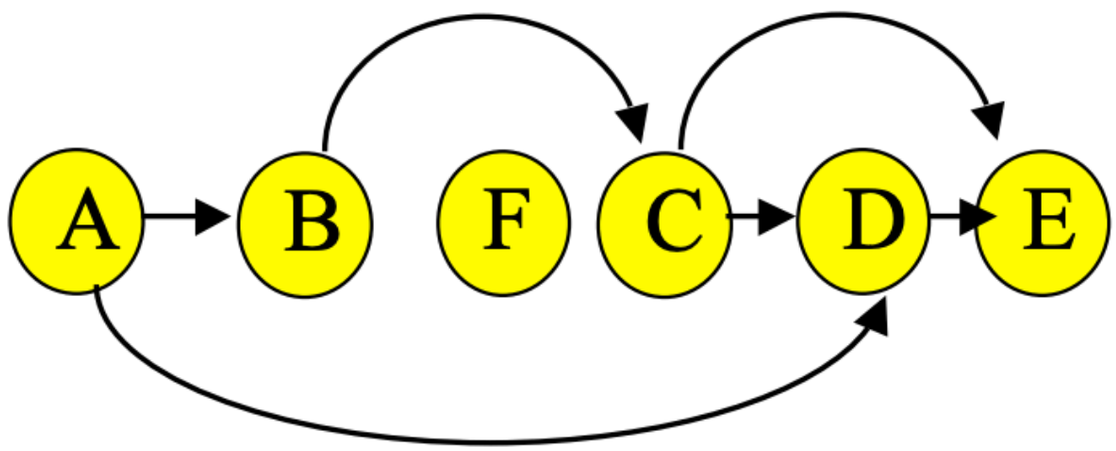
**Proof.** [By contradiction] Suppose $G$ has a cycle $C$. Let $v_1, v_2, \ldots, v_n$ be the topological ordering of $G$

- Let $v_i$ be the lowest-indexed node in $C$, and let $v_j$ be the node just before $v_i$; thus $(v_j, v_i)$ is an edge
- By our choice of $i$, we have $i < j$.
- On the other hand, since $(v_j, v_i)$ is an edge and $v_1, v_2, \ldots, v_n$ is a topological order, we must have $j < i$ $(\Rightarrow\Leftarrow)$ ∎



# Topological Ordering and DAGs

- No directed **cyclic** graph can have a topological ordering
- Does every DAG have a topological ordering?
    - Yes, can prove by induction (and construction)
- How do we compute a topological ordering?
    - What property should the first node in any topological ordering satisfy?
        - Cannot have incoming edges, i.e., indegree = 0
    - Can we use this idea repeatedly?



# Finding a Topological Ordering

**Claim.** Every DAG has a vertex with in-degree zero.

**Proof.** [By contradiction] Suppose every vertex has an incoming edge. Show that the graph must have a cycle.

- Pick any vertex $v$, there must be an edge $(u, v)$.
- Walk backwards following these incoming edges for each vertex
- After $n + 1$ steps, we must have visited some vertex $w$ twice (why?)
- Nodes between two successive visits to $w$ form a cycle ( $\Rightarrow\Leftarrow$ ) ∎

**Idea for finding topological ordering.** Build order by repeatedly removing a vertex of in-degree 0 from $G$.

# Topological Sorting Algorithm

```
TopologicalSorting(G) ◃ G = (V,E) is a DAG
    Initialize T[1..n]← 0 and i ← 0
    while V is not empty do
        i←i+1
        Find a vertex v ∈ V with indeg(v) = 0
        T[i] ← v
        Delete v (and its edges) from G
```

**Analysis:**

- Correctness,  any ideas how to proceed?
- Running time

# Topological Sorting Algorithm

**Analysis (Correctness).** Proof by induction on number of vertices $n$:

- $n = 1$, no edges, the vertex itself forms topological ordering
- Suppose our algorithm is correct for any graph with less than $n$ vertices
- Consider an arbitrary DAG on $n$ vertices
  - Must contain a vertex $v$ with in-degree $0$ (we proved it)
  - Deleting that vertex and all outgoing edges gives us a graph $G'$ with less than $n$ vertices that is still a DAG
  - Can invoke induction hypothesis on $G'$ !
- Let $u_1, u_2, \ldots, u_{n-1}$ be a topological ordering of $G'$, then $v, u_1, u_2, \ldots, u_{n-1}$ must be a topological ordering of $G$ ∎

# Topological Sorting Algorithm

**Running time:**

- (Initialize) In-degree array `ID[1..n]` of all vertices
  - $O(n + m)$ time
- Find a vertex with in-degree zero
  - $O(n)$ time
  - Need to keep doing this till we run out of vertices! $O(n^2)$
- Reduce in-degree of vertices adjacent to a vertex
  - $O(\text{outdegree(v)})$ time for each $v$: $O(n + m)$ time
- **Bottleneck step:** finding vertices with in-degree zero

**Can we do better?**

# Linear-Time Algorithm

- Need a faster way to find vertices with in-degree 0 instead of searching through entire in-degree array!
- **Idea:** Maintain a queue (or stack) $S$ of in-degree 0 vertices
- Update $S$: When $v$ is deleted, decrement `ID[u]` for each neighbor $u$; if `ID[u]` = 0, add $u$ to $S$:
  - $O$(outdegree(v)) time
- Total time for previous step over all vertices:
  - $\sum_{v \in V} O(\text{outdegree}(v)) = O(n + m)$ time
- Topological sorting takes $O(n + m)$ time and space!

# Topological Ordering by DFS

- Call DFS and maintain finish times of all vertices
  - Finish($u$): time DFS($v$) completed for all neighbors of $u$
- Return the list of vertices **in reverse order of finish times**
    - Vertex finished last will be first in topological ordering
- **New.** This generates the topological ordering all all nodes reachable from the root of the DFS

- **Claim.** If a DAG $G$ contains an edge $u \rightarrow v$, then the finish time of $u$ must be larger than the finish time of $v$.
    - $u$ is finished only after all its neighbors are finished

# Traversals: Many More Applications

BFS and/or DFS can also be used to solve many other problems

- Find a (directed) cycle in a (directed) graph (or a cycle containing a specified vertex $v$)
- Find all cut vertices of a graph (A cut vertex is one whose removal increases the number of connected components)
- Find all bridges of a graph (A bridge is an edge whose removal increases the number of connected components
- Find all biconnected components of a graph (A biconnected component is a maximal subgraph having no cut vertices)

All of this can be done in $O(|V| + |E|)$ space and time!

# 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

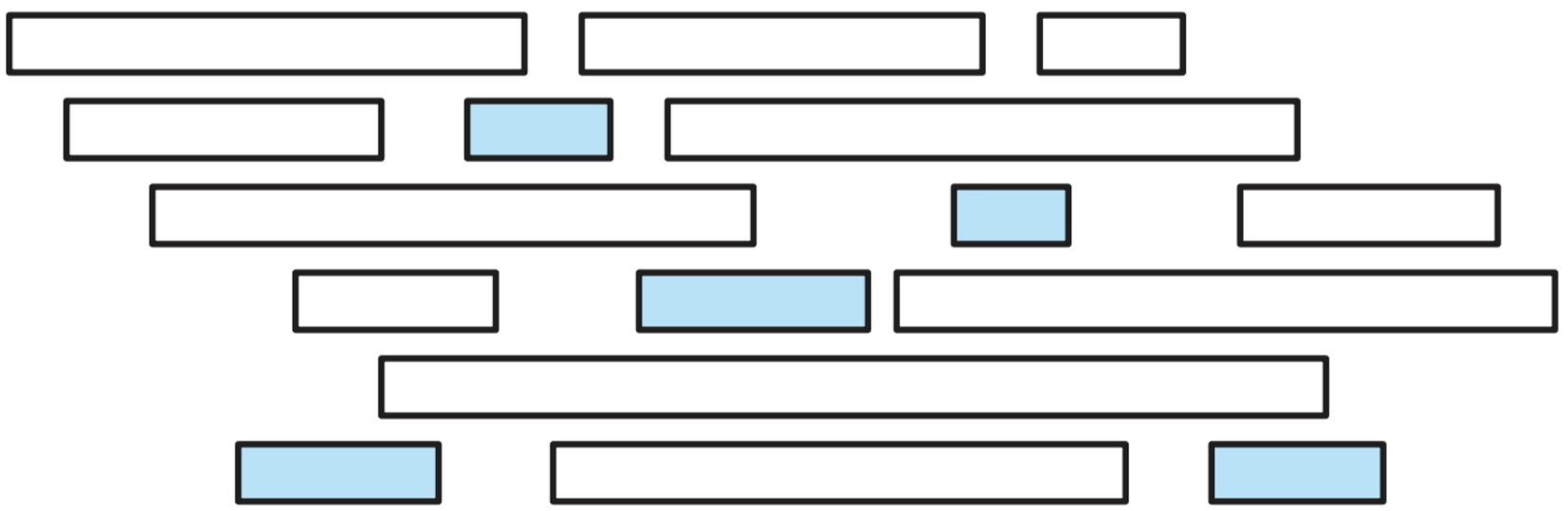
# Greedy: Proof Techniques

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?

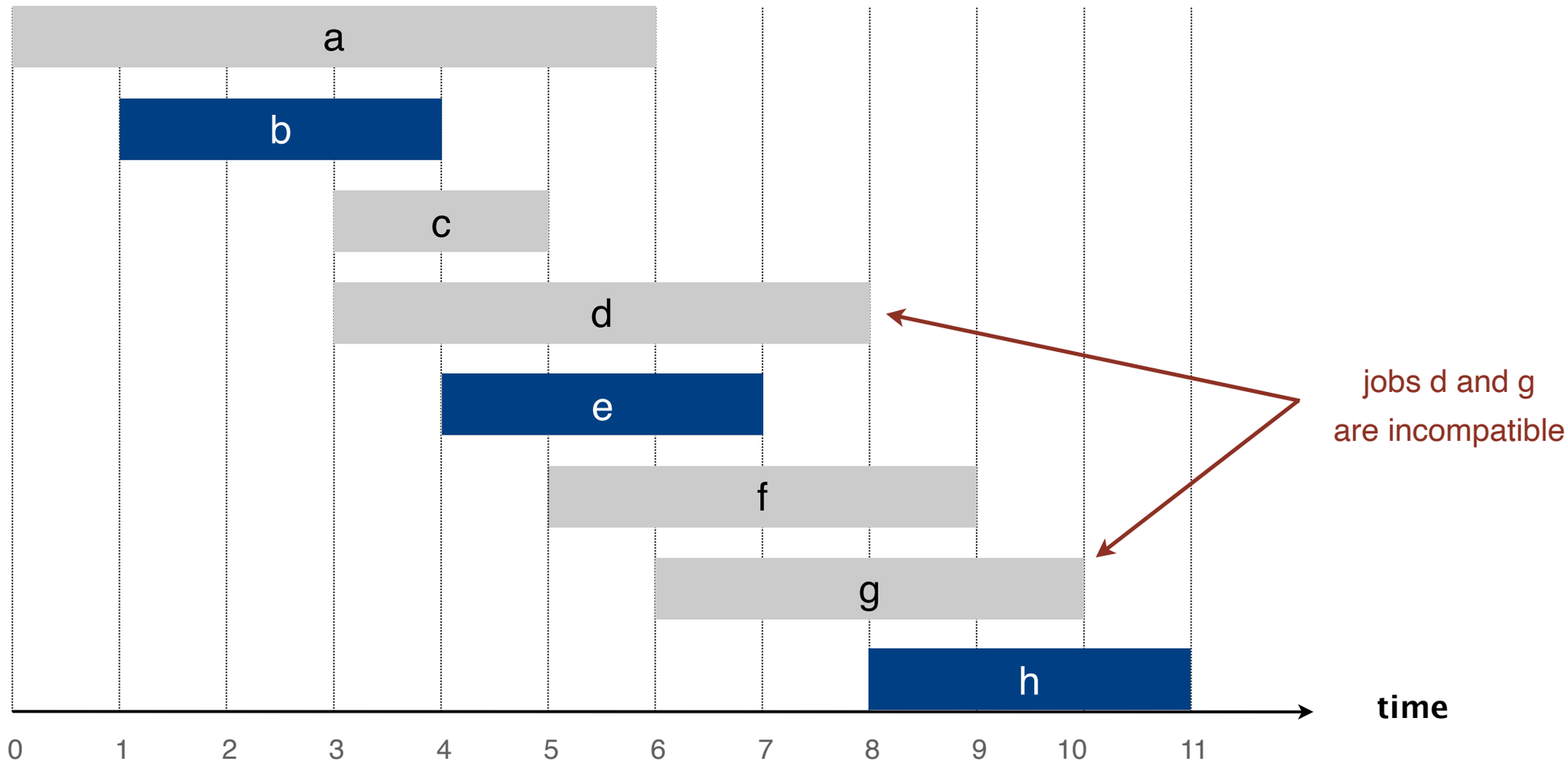


A maximum conflict-free schedule for a set of classes.

From Erickson's Algorithms Book

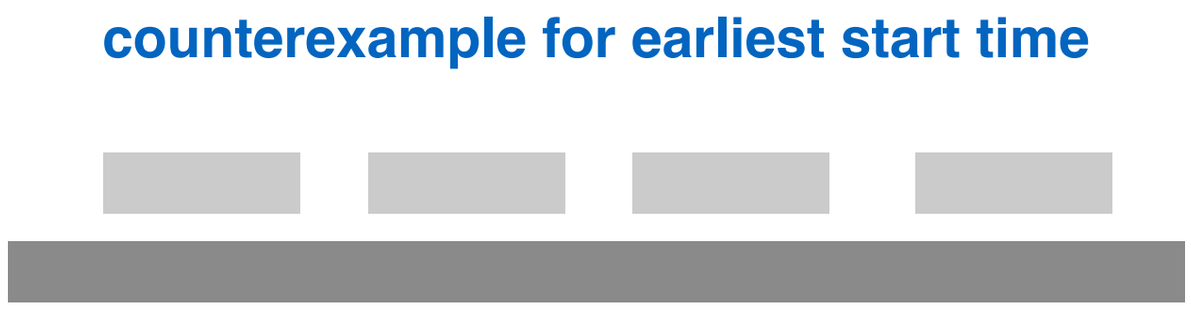
# Interval Scheduling

**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 $k$th 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
  - $O(n)$
- 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

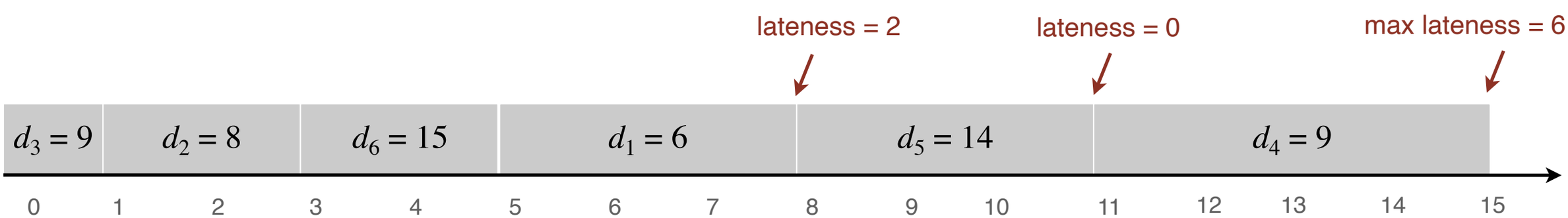
# Minimizing Lateness: Problem

**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 (triage!)

**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 (triage!)

- How all computer scientists schedule their work
- 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 $d_1 \leq d_2 \leq \ldots \leq d_n$.

$t \leftarrow 0$.

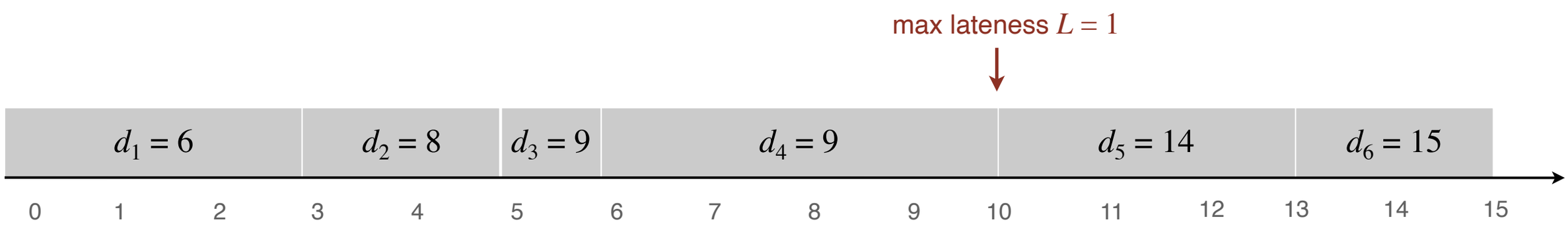
FOR $j = 1$ TO $n$

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

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

  $t \leftarrow t + t_j$.

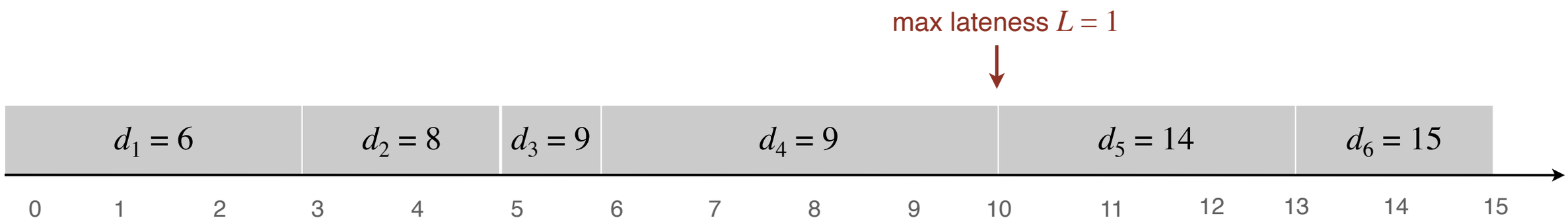
RETURN intervals $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$.



# 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 $[t, t + t_j]$.

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

  $t \leftarrow t + t_j$.

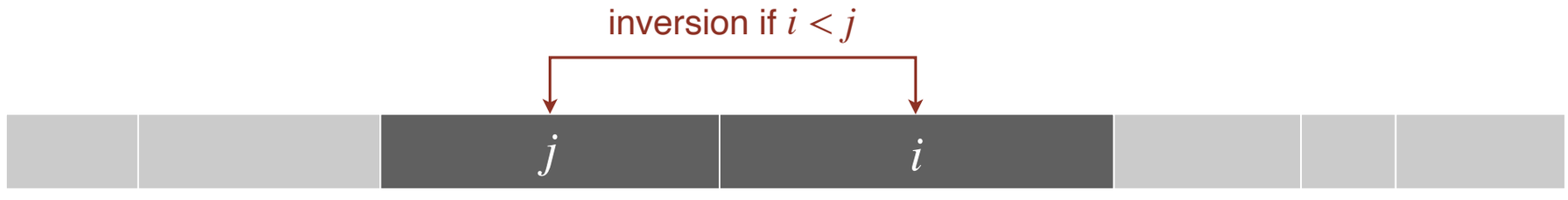
RETURN intervals $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$.

# 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: $\max\{0,(s + t_i + t_j) - d\}$

Continued in Next Lecture

# 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)