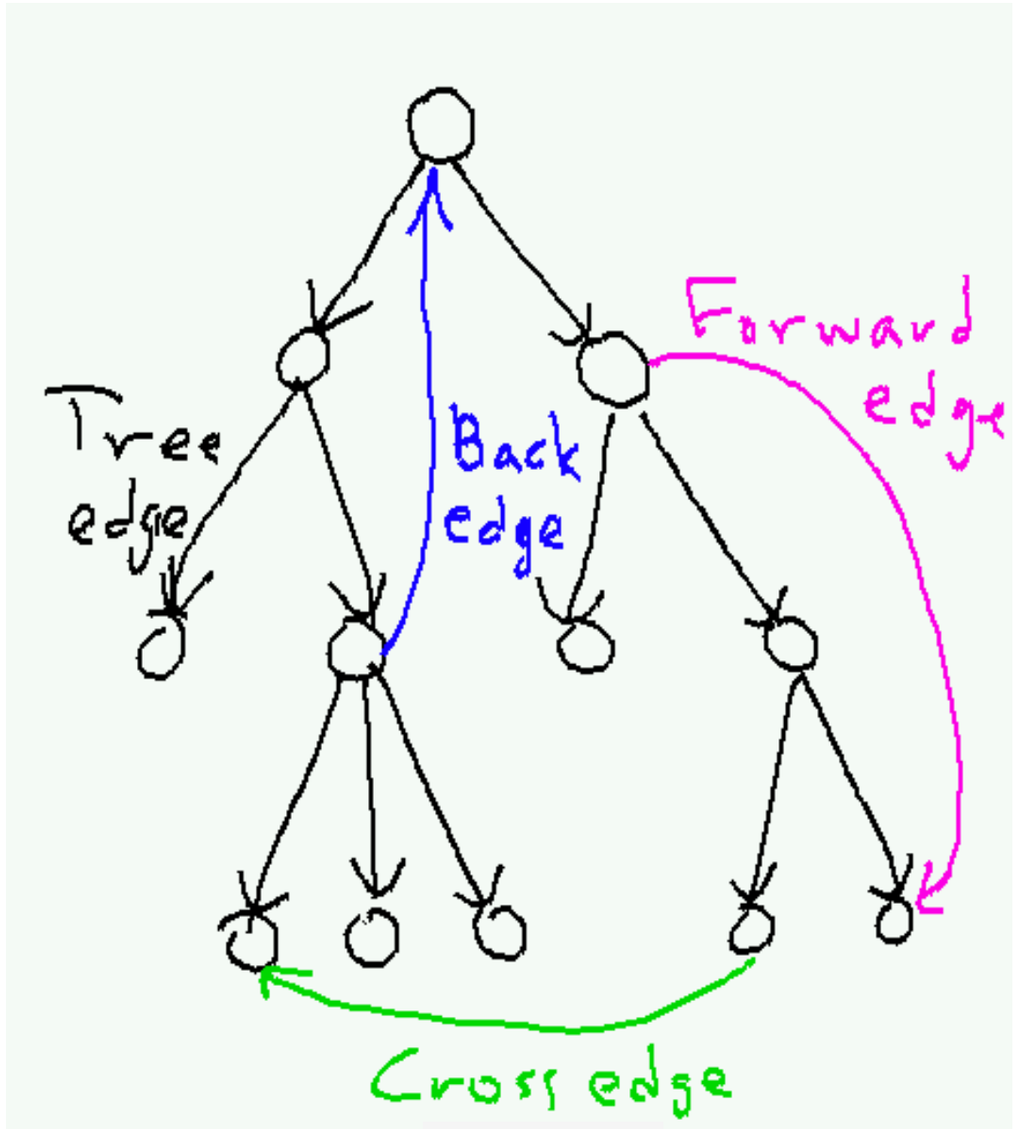
Greedy Algorithms

Reminders

- CS Colloquium Today:
    - Thesis Proposal Presentations
    - Zoom, **3.15 - 4.30 pm**
    - Good to attend if you are considering doing a CS thesis

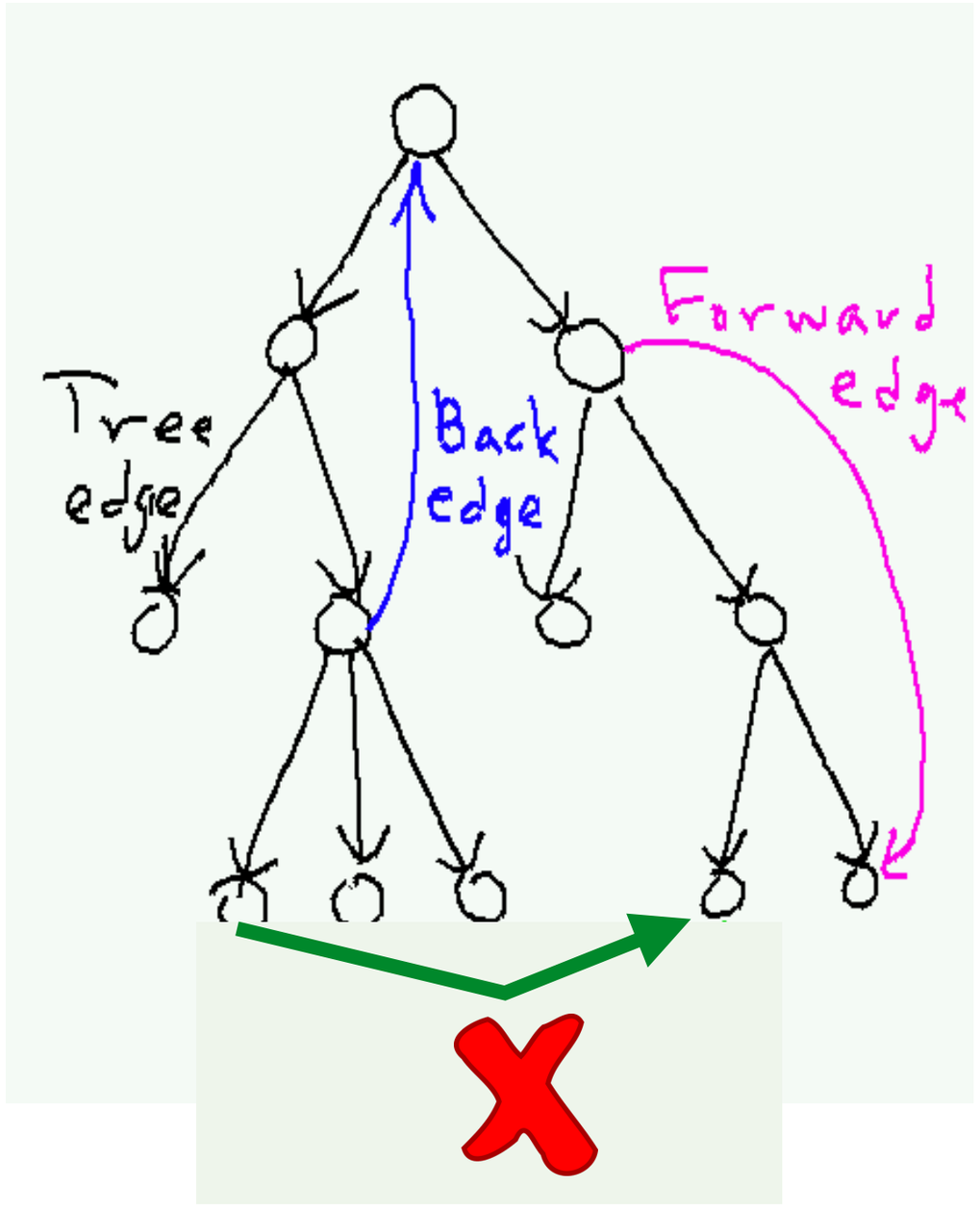
Clarification: Last Lecture

- In **undirected** graphs, each edge is a back edge or tree edge wrt to DFS tree (**cannot** have cross edges)
- In **directed** graphs, **can have** cross edges of the type
  - (u, v) is a cross edge if DFS(v) is finished before DFS u is called
- However, these (leftward) cross edges **cannot** contribute to a cycle
- Can't have edges (u, v) where DFS(u) is finished before DFS(v) is called



Clarification: Last Lecture

- In **undirected** graphs, each edge is a back edge or tree edge wrt to DFS tree (**cannot** have cross edges)
- In **directed** graphs, can have cross edges of the type
  - (u, v) is a cross edge if DFS(v) is finished before DFS u is called
- However, these (leftward) cross edges **cannot** contribute to a cycle
- Can't have edges (u, v) where DFS(u) is finished before DFS(v) is called



Wrapping Up

Topological Sorting

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 $v, u_1, u_2, \ldots, u_{n-1}$ is a valid 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
- This generates the topological order of nodes reachable from root, can keep going with start time of next DFS set to finish time of last
- **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 Algorithms

First Design Paradigm

- Greedy Algorithms 
- Divide and Conquer
- Dynamic Programming
- Network flow

Greedy: Locally Optimal

- We already saw a greedy algorithm that works! Which one?
- Greedy algorithms build solutions by making locally optimal choices.
- Surprisingly, sometimes this also leads to globally optimal solutions!
- 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)

Filling Up on Gas

- Suppose you are on a road trip on a long straight highway
- **Goal**: minimize the number of times you stop to get gas
- Many possible ways to choose which gas station to stop at
- Greedy: wait until just about to run out of gas, stop for gas
  - Turns out this is an optimal solution

Typical Problem Structure

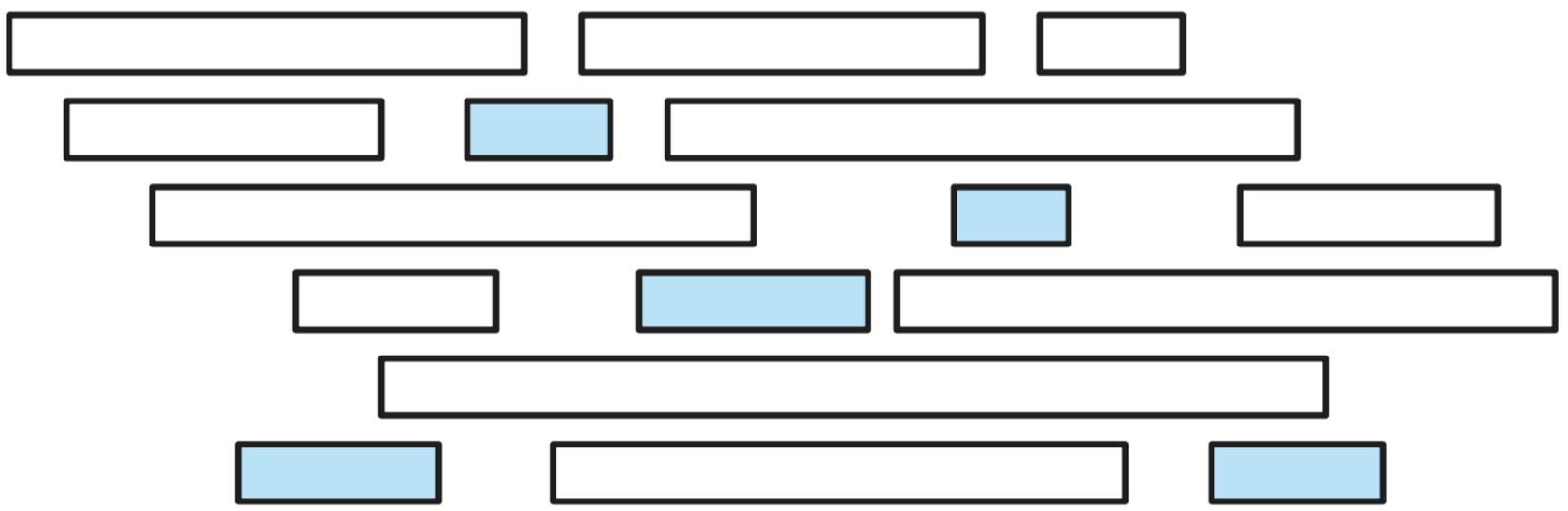
- **Global objective:** minimize or maximize a quantity
- **Local optimization.** At every step, an algorithm can make several choices; a greedy algorithm makes this choice myopically
- For some problems, a greedy algorithm ends up being optimal
    - Greedy happens to be *one way* to reach the optimal solution

Problem Solving Steps

- Formalize the problem
- Design the algorithm to solve the problem
    - Usually natural/ intuitive/ easy for greedy
- Proving that the algorithm is correct: for greedy, this is proving that greedy is optimal: it minimizes or maximizes the objective
    - The hard part: **our focus** on this topic!
- Analyze running time
    - Often straightforward

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?

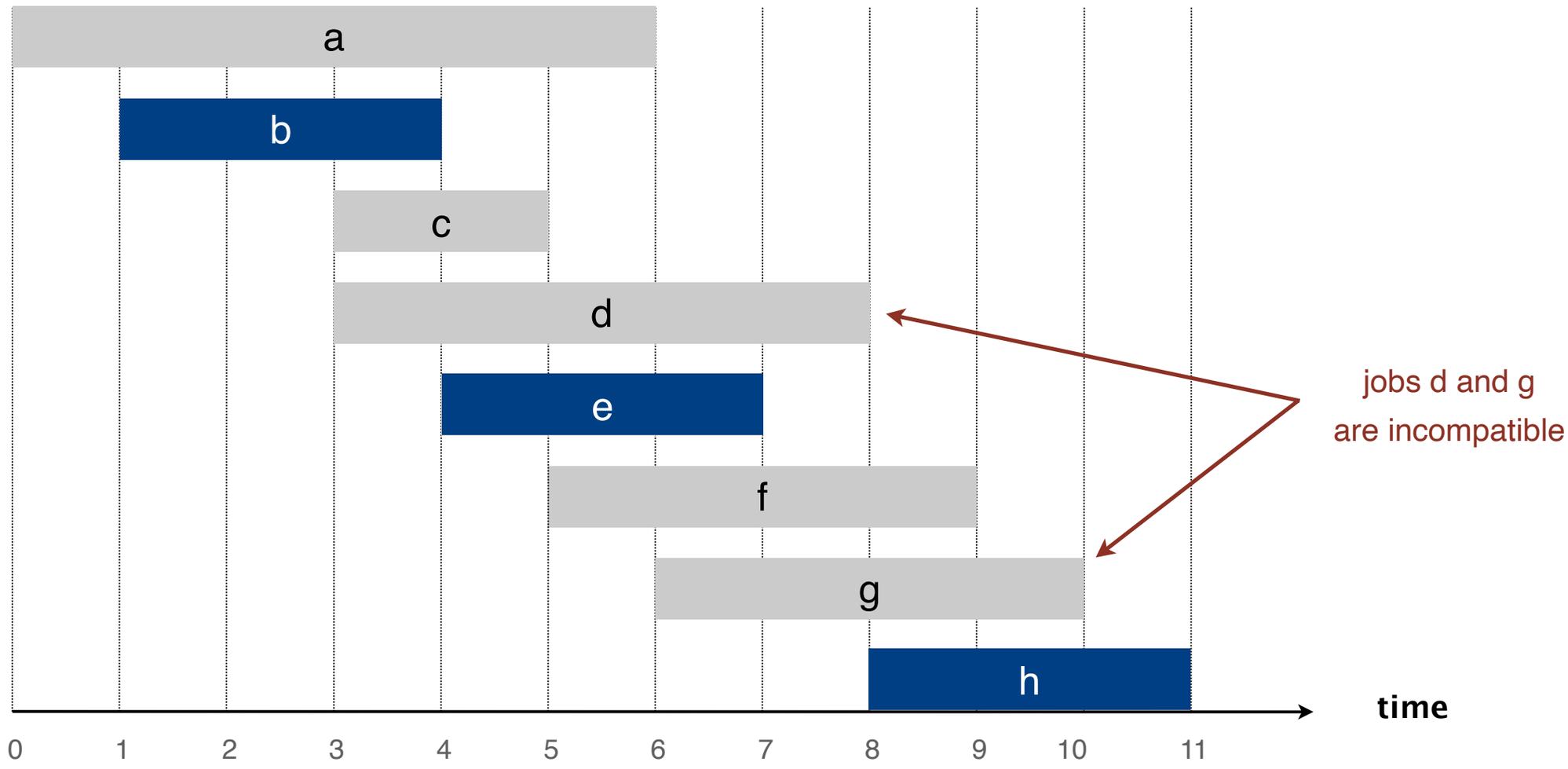


A maximum conflict-free schedule for a set of classes.

From Erickson's Algorithms Book

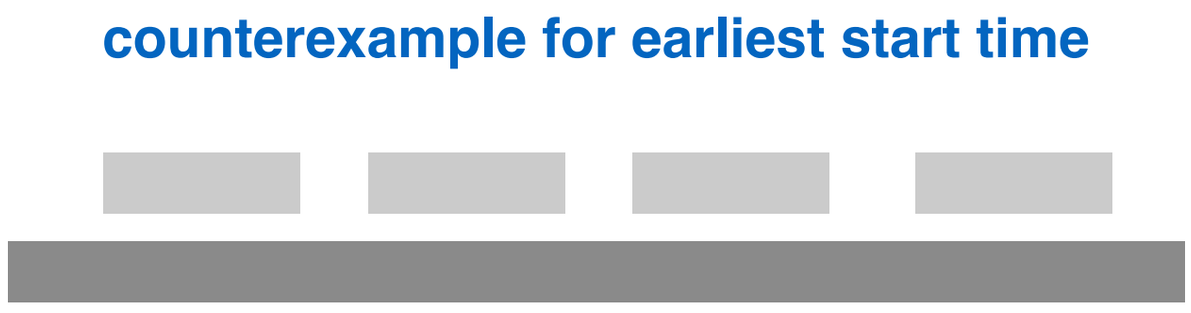
Interval Scheduling

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,
    - That is it schedules the maximum number of jobs
- Let $\mathcal{O}$ be be some optimal set of jobs
    - Note: there can be more than one optimal solutions!
- **Goal:** show $|S| = |\mathcal{O}|$,i.e., greedy also selects the same number of jobs and thus is optimal

Exchange Argument

- Let $O = o_1, o_2, \ldots, o_m$ be the sequence of jobs scheduled by the optimal algorithm, and $G = g_1, g_2, \ldots, g_k$ be the sequence of jobs scheduled by greedy, such that $O \neq G$
- We can modify O to produce a new solution O' that is:
  - No worse than O
  - Closer to G in some measurable way

Idea behind proof by exchange argument:

- Transform O into G one step at a time, without hurting solution (that is, preserving optimality)

O (optimal) $\rightarrow$ O' (optimal) $\rightarrow$ O'' (optimal) $\rightarrow \cdots \rightarrow$ G (optimal)

Exchange Argument

- Let $O = o_1, o_2, \ldots, o_m$ be the sequence of jobs scheduled by the optimal algorithm, and $G = g_1, g_2, \ldots, g_k$ be the sequence of jobs scheduled by greedy, both ordered by increasing finish time
- By induction, we will show that we exchange each job scheduled by optimal with a non-conflicting job scheduled by greedy to create a new optimal schedule
- (Base case: $j = 1$). In the beginning, greedy picks the job with the earliest finish time, so $f_{g_1} \leq f_{o_1}$, thus g_1 does not conflict with any of the jobs $o_2, \ldots, o_m$
- We can exchange o_1 with g_1 to get a new conflict-free optimal schedule $g_1, o_2, o_3, \ldots, o_m$

Exchange Argument

- Inductive hypothesis: Let us say we now have an optimal conflict-free schedule that is the same as greedy up to job $j-1$
  - $O' = g_1, g_2, \ldots, g_{j-1}, o_j, \ldots, o_m$
- Because both G and O' consist on non-conflicting jobs, neither g_j nor o_j conflict with $g_1, g_2, \ldots, g_{j-1}$
- Greedy picks earliest finish time among non-conflicting jobs
  - Since $f_{g_j} \leq f_{o_j} \leq s_{o_{j+1}}$ which means g_j does not conflict with any remaining jobs $o_{j+1}, \ldots o_m$
- We can exchange o_j with the greedy choice g_j to construct a new optimal schedule $g_1, g_2, \ldots, g_j, o_{j+1}, \ldots, o_m$

Are We Done? Almost

- We can keep replacing every job scheduled by the optimal algorithm with a non-conflicting job scheduled by greedy until we have an optimal schedule that contains all the greedy jobs

Lemma 2. Greedy is optimal, that is, $k = m$.

Proof. (By contradiction) Suppose $m > k$.

- That is, there is a job o_{k+1} that starts after g_k ends
- What is the contradiction?
    - Greedy keeps selecting jobs until no more compatible jobs left. Since $f_{g_k} \leq f_{o_k}$, greedy would also select compatible job o_{k+1}

($\Rightarrow\!\!\!\Leftarrow$) ∎

Review: Exchange Argument

- Assume there is an optimal solution O that is different from the greedy solution G
- We can modify O to produce a new solution O' that is:
  - No worse than O
  - Closer to G in some measurable way

Idea behind proof by exchange argument:

- Transform O into G one step at a time, without hurting solution (that is, preserving optimality)

O (optimal) $\rightarrow O'$ (optimal) $\rightarrow O''$ (optimal) $\rightarrow \cdots \rightarrow G$ (optimal)

Caution: Not Uniquely Optimal

We did not prove that greedy was the only optimal solution: can be more than one

Greedy: Proof Techniques

The textbook (reading) talks about two 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
    - Simple induction, *often has an implicit exchange argument at its heart*
- **Exchange Property**: An optimal solution can be transformed into a greedy solution without sacrificing optimality
- Can use any approach that proves correctness

Running Time Analysis

Let's analyze all the steps:

- Sorting and relabelling jobs by finish times and
  - $O(n \log n)$
- 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?

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)