Reminder: Homework 1 due tonight at 11:59PM!
Recap: Greedy Algorithms

Interval Scheduling
- **Goal**: Maximize number of meeting requests scheduled in single conference room
- **Greedy Algorithm**: Sort by earliest finish time
- **Running Time**: $O(n \log n)$

Interval Partitioning
- **Goal**: Minimize number of classrooms needed to assign all lectures
- **Greedy Algorithm**: Sort by earliest start time
- **Running Time**: $O(n \log n)$

Dijkstra's Shortest Path Algorithm
- **Invariant**: minimum distance $d(u)$ to all nodes in explored set $S$
- **Greedy Choice**: Add node $v$ to $S$ with minimum value $\pi(v)$
- **Running Time**: $O(m + n \log n)$ with Fibonacci Heap

\[
\pi(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e
\]
Remarks about Dijkstra’s Algorithm

Yields shortest path tree from origin $s$.
- Shortest path from $s$ to every other node $v$

shortest route from Wang Hall to Miami Beach
Maximum Capacity Path Problem

Each edge $e$ has capacity $c_e$ (e.g., maximum height)

Capacity of a path is Minimum capacity of any Edge in path

**Goal:** Find path from $s$ to $t$ with maximum capacity

**Solution:** Use Dijkstra!

Small Modification

$$\pi(v) = \max_{e=(u,v): u \in S} \min\{c_e, \pi(u)\}$$
Greedy Algorithms
4.2 Scheduling to Minimize Lateness
Scheduling to Minimizing Lateness

Minimizing lateness problem.
- 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: \( \ell_j = \max \{ 0, f_j - d_j \} \).
- Goal: schedule all jobs to minimize maximum lateness \( L = \max \ell_j \).

Ex:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_j )</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>( d_j )</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

\( d_3 = 9 \) \quad \text{lateness} = 2 \quad \text{lateness} = 0 \quad \text{max lateness} = 6

\( d_2 = 8 \) \quad \text{d}_6 = 15 \quad \text{d}_1 = 6 \quad \text{d}_5 = 14 \quad \text{d}_4 = 9 \)
Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

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

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

- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$. 
Minimizing Lateness: Greedy Algorithms

**Greedy template.** Consider jobs in some order.

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

  
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_j$</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>$d_j$</td>
<td>100</td>
<td>10</td>
</tr>
</tbody>
</table>

  counterexample

- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

  
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_j$</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>$d_j$</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

  counterexample
Minimizing Lateness: Greedy Algorithm

**Greedy algorithm.** Earliest deadline first.

Sort $n$ jobs by deadline so that $d_1 \leq d_2 \leq ... \leq d_n$

$t \leftarrow 0$

for $j = 1$ to $n$

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

$s_j \leftarrow t$, $f_j \leftarrow t + t_j$

$t \leftarrow t + t_j$

output intervals $[s_j, f_j]$

Max lateness $= 1$
Minimizing Lateness: No Idle Time

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

**Observation.** The greedy schedule has no idle time.
Def. Given a schedule $S$, an inversion is a pair of jobs $i$ and $j$ such that: $i < j$ (i.e., $d_i < d_j$) but $j$ scheduled before $i$. 

Observation. Greedy schedule has no inversions.

Claim. If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.
Minimizing Lateness: Inversions

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

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

**Proof:** Let $(i,j)$ be inversion with *smallest* number of jobs scheduled in between $j$ and $i$. Suppose for contradiction that some job $k$ is scheduled between $j$ and $i$.

- **Case 1:** $d_k < d_j \Rightarrow (k,j)$ is inversion (Contradiction!)
- **Case 2:** $d_k \geq d_j > d_i \Rightarrow (i,k)$ is inversion (Contradiction!)
Minimizing Lateness: Inversions

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

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

**Pf.** Let $\ell$ be the lateness before the swap, and let $\ell'$ be it afterwards.

- $\ell'_k = \ell_k$ for all $k \neq i, j$
- $\ell'_i \leq \ell_i$
- If job $j$ is late:

\[
\ell'_j = f'_j - d_j \quad \text{(definition)}
\]
\[
= f_i - d_j \quad \text{($j$ finishes at time $f_i$)}
\]
\[
\leq f_i - d_i \quad \text{($i < j$)}
\]
\[
\leq \ell_i \quad \text{(definition)}
\]
Minimizing Lateness: Analysis of Greedy Algorithm

**Theorem.** Greedy schedule $S$ is optimal.

**Pf.** 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 $i-j$ be an adjacent inversion.
  - swapping $i$ and $j$ does not increase the maximum lateness and strictly decreases the number of inversions
  - this contradicts 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.

**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.

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

**Other greedy algorithms.** Kruskal, Prim, Huffman, …
4.3 Optimal Caching
Optimal Offline Caching

**Caching.**
- Cache with capacity to store \( k \) items.
- Sequence of \( m \) item requests \( d_1, d_2, \ldots, d_m \).
- Cache hit: item already in cache when requested.
- Cache miss: item not already in cache when requested: must bring requested item into cache, and evict some existing item, if full.

**Goal.** Eviction schedule that minimizes number of cache misses.

**Ex:** \( k = 2 \), initial cache = ab,
requests: a, b, c, b, c, a, a, b.

**Optimal eviction schedule:** 2 cache misses.
Optimal Offline Caching: Farthest-In-Future

**Farthest-in-future.** Evict item in the cache that is not requested until farthest in the future.

current cache:  

```
| a | b | c | d | e | f |
```

future queries:  

```
g  a  b  c  e  d  a  b  b  a  c  d  e  a  f  a  d  e  f  g  h ...```

↑ cache miss  

↑ eject this one

**Theorem.** [Bellady, 1960s] FF is optimal eviction schedule.  
**Pf.** Algorithm and theorem are intuitive; proof is subtle.
Reduced Eviction Schedules

Def. A reduced schedule is a schedule that only inserts an item into the cache in a step in which that item is requested.

Intuition. Can transform an unreduced schedule into a reduced one with no more cache misses.

\[
\begin{array}{cccc}
a & a & b & c \\
a & a & x & c \\
c & a & d & c \\
d & a & d & c \\
a & a & c & b \\
b & a & x & b \\
c & a & c & b \\
a & a & b & c \\
a & a & b & c \\
\end{array}
\]

an unreduced schedule

\[
\begin{array}{cccc}
a & a & b & c \\
a & a & b & c \\
c & a & b & c \\
d & a & d & c \\
a & a & d & c \\
b & a & d & b \\
c & a & c & b \\
a & a & c & b \\
a & a & c & b \\
\end{array}
\]

a reduced schedule
Reduced Eviction Schedules

**Claim.** Given any unreduced schedule $S$, can transform it into a reduced schedule $S'$ with no more cache misses.

**Pf.** (by induction on number of unreduced items)

- Suppose $S$ brings $d$ into the cache at time $t$, without a request.
- Let $c$ be the item $S$ evicts when it brings $d$ into the cache.
- **Case 1:** $d$ evicted at time $t'$, before next request for $d$.
- **Case 2:** $d$ requested at time $t'$ before $d$ is evicted.

![Case 1 diagram](image1)

- $S$
  - $c$
  - $t$
  - $d$
  - $t'$
  - $e$
  - $d$ evicted at time $t'$, before next request for $d$.

![Case 2 diagram](image2)

- $S$
  - $c$
  - $t$
  - $d$
  - $t'$
  - $d$ requested at time $t'$
Farthest-In-Future: Analysis

**Theorem.** FF is optimal eviction algorithm.

**Pf.** (by induction on number or requests j)

**Invariant:** There exists an optimal reduced schedule $S$ that makes the same eviction schedule as $S_{FF}$ through the first j+1 requests.

Let $S$ be reduced schedule that satisfies invariant through j requests. We produce $S'$ that satisfies invariant after j+1 requests.

- Consider $(j+1)^{st}$ request $d = d_{j+1}$.
- Since $S$ and $S_{FF}$ have agreed up until now, they have the same cache contents before request j+1.
- **Case 1:** ($d$ is already in the cache).
  - $\rightarrow$ $S' = S$ satisfies invariant.
- **Case 2:** ($d$ is not in the cache and $S$ and $S_{FF}$ evict the same element).
  - $\rightarrow$ $S' = S$ satisfies invariant.
Farthest-In-Future: Analysis

**Pf. (continued)**
- **Case 3:** (d is not in the cache; $S_{FF}$ evicts e; S evicts f ≠ e).
  - begin construction of $S'$ from S by evicting e instead of f

  \[
  \begin{array}{ccc}
  j & \text{same} & e & f \\
  & S & & S' \\
  \hline
  j+1 & \text{same} & e & d \\
  & S & & S' \\
  \end{array}
  \]

  - now $S'$ agrees with $S_{FF}$ on first j+1 requests; we show that having element f in cache is no worse than having element e
Let $j'$ be the \textbf{first} time after $j+1$ that $S$ and $S'$ take a different action, and let $g$ be item requested at time $j'$.

- \textbf{Case 3a:} $g = e$. Can't happen with Farthest-In-Future since there must be a request for $f$ before $e$.

- \textbf{Case 3b:} $g = f$. Element $f$ can't be in cache of $S$, so let $e'$ be the element that $S$ evicts.
  - if $e' = e$, $S'$ accesses $f$ from cache; now $S$ and $S'$ have same cache
  - if $e' \neq e$, $S'$ evicts $e'$ and brings $e$ into the cache; now $S$ and $S'$ have the same cache

Note: $S'$ is no longer reduced, but can be transformed into a reduced schedule that agrees with $S_{FF}$ through step $j+1$.
Farthest-In-Future: Analysis

Let $j'$ be the **first** time after $j+1$ that $S$ and $S'$ take a different action, and let $g$ be item requested at time $j'$.

- **Case 3c:** $g \neq e, f$. $S$ must evict $e$.
  
  Make $S'$ evict $f$; now $S$ and $S'$ have the same cache.

\[
\begin{array}{c|c|c}
  j' & \text{same} & e \\
  S & & \text{same} & f \\
  S' & & & \\
\end{array}
\]

\[
\begin{array}{c|c|c}
  j' & \text{same} & g \\
  S & & \text{same} & g \\
  S' & & & \\
\end{array}
\]

otherwise $S'$ would take the same action
Caching Perspective

Online vs. offline algorithms.
- Offline: full sequence of requests is known a priori.
- Online (reality): requests are not known in advance.
- Caching is among most fundamental online problems in CS.

LIFO. Evict page brought in most recently.
LRU. Evict page whose most recent access was earliest.

Theorem. FF is optimal offline eviction algorithm.
- Provides basis for understanding and analyzing online algorithms.
- LRU is k-competitive. [Section 13.8]
- LIFO is arbitrarily bad.
4.5 Minimum Spanning Tree
**Minimum Spanning Tree**

**Minimum spanning tree.** 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.

$G = (V, E)$

$T, \sum_{e \in T} c_e = 50$

**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
  - model locality of particle interactions in turbulent fluid flows
  - autoconfig protocol for Ethernet bridging to avoid cycles in a network

- Cluster analysis.
Greedy Algorithms

**Kruskal's algorithm.** Start with $T = \emptyset$. Consider edges in ascending order of cost. Insert edge $e$ in $T$ unless doing so would create a cycle.

**Reverse-Delete algorithm.** Start with $T = E$. Consider edges in descending order of cost. Delete edge $e$ from $T$ unless doing so would disconnect $T$.

**Prim's algorithm.** Start with some root node $s$ and greedily grow a tree $T$ from $s$ outward. At each step, add the cheapest edge $e$ to $T$ that has exactly one endpoint in $T$.

**Remark.** All three algorithms produce an MST.
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 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 $f$. 

![Diagram showing e in the MST and f not in the MST]
Cycles and Cuts

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

![Graph](image)

**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.

![Graph](image)

Cut S = {4, 5, 8}
Cutset D = 5-6, 5-7, 3-4, 3-5, 7-8
**Claim.** A cycle and a cutset intersect in an even number of edges.

**Pf.** (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 $e$.

Pf. (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. □
Greedy Algorithms

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

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

**Pf.** (exchange argument)
- Suppose $f$ belongs to $T^*$, and let's see what happens.
- Deleting $f$ from $T^*$ creates a cut $S$ in $T^*$.
- Edge $f$ is both in the cycle $C$ and in the cutset $D$ corresponding to $S$.
  $\Rightarrow$ there exists another edge, say $e$, that is in both $C$ and $D$.
- $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree.
- Since $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$.
- This is a contradiction. ▪
Prim's Algorithm: Proof of Correctness

**Prim's algorithm.** [Jarník 1930, Dijkstra 1959, Prim 1957]

- Initialize $S = \text{any node}$.
- Apply cut property to $S$.
- Add min cost edge in cutset corresponding to $S$ to $T$, and add one new explored node $u$ to $S$. 

![Diagram of Prim's algorithm with a shaded area and a blue edge labeled 'S']
Implementation: Prim's Algorithm

Implementation. Use a priority queue ala Dijkstra.

- 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;
- $O(m + n \log n)$ with Fibonacci Heap

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

    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 < a[v]))
                decrease priority a[v] to c_e
    }
```
Kruskal's Algorithm: Proof of Correctness

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
Implementation: Kruskal's Algorithm

Implementation. Use the union-find data structure.

- Build set $T$ of edges in the MST.
- Maintain set for each connected component.
- $O(m \log n)$ for sorting and $O(m \alpha(m, n))$ for union-find.

\[
m \leq n^2 \Rightarrow \log m \text{ is } O(\log n)
\]

essentially a constant

Kruskal($G, c$) {
  Sort edges weights so that $c_1 \leq c_2 \leq \ldots \leq c_m$.
  $T \leftarrow \emptyset$

  foreach ($u \in V$) make a set containing singleton $u$

  for $i = 1$ to $m$
    $(u, v) = e_i$
    if ($u$ and $v$ are in different sets) {
      $T \leftarrow T \cup \{e_i\}$
      merge the sets containing $u$ and $v$
    }

  return $T$
}
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.

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

```java
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
}
```
Extra Slides
MST Algorithms: Theory

Deterministic comparison based algorithms.

- $O(m \log n)$  
  [Jarník, Prim, Dijkstra, Kruskal, Boruvka]
- $O(m \log \log n)$.  
  [Cheriton-Tarjan 1976, Yao 1975]
- $O(m \beta(m, n))$.  
  [Fredman-Tarjan 1987]
- $O(m \log \beta(m, n))$.  
  [Gabow-Galil-Spencer-Tarjan 1986]
- $O(m \alpha (m, n))$.  
  [Chazelle 2000]

Holy grail.  $O(m)$.

Notable.

- $O(m)$ randomized.  
  [Karger-Klein-Tarjan 1995]
- $O(m)$ verification.  
  [Dixon-Rauch-Tarjan 1992]

Euclidean.

- 2-d: $O(n \log n)$.  
  compute MST of edges in Delaunay
- k-d: $O(k n^2)$.  
  dense Prim