CS 580: Algorithm Design and Analysis

Announcement: Homework 1 released!
Due: January 25th at midnight (Blackboard)
Recap: Graphs

Definition of a Graph
- Representations
  - Adjacency Matrix
  - Adjacency List
- Connectivity, Cycles
- Trees (Connected + No Cycles)
- Rooted Trees
  - Binary Trees
  - Balanced Trees

Breadth First Search
- BFS Tree
  - $O(m+n)$ algorithm

Finding Connected Components
3.4 Testing Bipartiteness
Bipartite Graphs

**Def.** An undirected graph $G = (V, E)$ is **bipartite** if the nodes can be colored red or blue such that every edge has one red and one blue end.

**Applications.**
- Stable marriage: men = red, women = blue.
- Scheduling: machines = red, jobs = blue.
Testing Bipartiteness

Testing bipartiteness. Given a graph $G$, is it bipartite?

- Many graph problems become:
  - easier if the underlying graph is bipartite (matching)
  - tractable if the underlying graph is bipartite (independent set)
- Before attempting to design an algorithm, we need to understand structure of bipartite graphs.

![a bipartite graph $G$](image1)

![another drawing of $G$](image2)
An Obstruction to Bipartiteness

**Lemma.** If a graph $G$ is bipartite, it cannot contain an odd length cycle.

**Pf.** Not possible to 2-color the odd cycle, let alone $G$. 

![Bipartite and Not Bipartite Graphs](image)
Lemma. Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).
Lemma. Let \( G \) be a connected graph, and let \( L_0, \ldots, L_k \) be the layers produced by BFS starting at node \( s \). Exactly one of the following holds.

(i) No edge of \( G \) joins two nodes of the same layer, and \( G \) is bipartite.
(ii) An edge of \( G \) joins two nodes of the same layer, and \( G \) contains an odd-length cycle (and hence is not bipartite).

Pf. (i)
- Suppose no edge joins two nodes in adjacent layers.
- By previous lemma, this implies all edges join nodes on same level.
- Bipartition: red = nodes on odd levels, blue = nodes on even levels.

\[
\begin{array}{c}
\text{Case (i)} \\
L_1 \quad L_2 \quad L_3
\end{array}
\]
Bipartite Graphs

Lemma. Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.

(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

Pf. (ii)

- Suppose $(x, y)$ is an edge with $x, y$ in same level $L_j$.
- Let $z = \text{lca}(x, y) =$ lowest common ancestor.
- Let $L_i$ be level containing $z$.
- Consider cycle that takes edge from $x$ to $y$, then path from $y$ to $z$, then path from $z$ to $x$.
- Its length is $1 + (j-i) + (j-i)$, which is odd. •

\[ 
\begin{align*}
&\text{Layer } L_i \\
&\text{Layer } L_j \\
&z = \text{lca}(x, y) \\
&\text{Path from } x \text{ to } y \\
&\text{Path from } y \text{ to } z \\
&\text{Path from } z \text{ to } x \\
\end{align*} \]
Obstruction to Bipartiteness

**Corollary.** A graph $G$ is bipartite iff it contain no odd length cycle.

![Bipartite and non-bipartite graphs](image)
3.5 Connectivity in Directed Graphs
Directed Graphs

Directed graph. \( G = (V, E) \)
- Edge \((u, v)\) goes from node \(u\) to node \(v\).

Ex. Web graph - hyperlink points from one web page to another.
- Directedness of graph is crucial.
- Modern web search engines exploit hyperlink structure to rank web pages by importance.
Graph Search

Directed reachability. Given a node \( s \), find all nodes reachable from \( s \).

Directed \( s-t \) shortest path problem. Given two node \( s \) and \( t \), what is the length of the shortest path between \( s \) and \( t \)?

Graph search. BFS extends naturally to directed graphs.

Web crawler. Start from web page \( s \). Find all web pages linked from \( s \), either directly or indirectly.
Strong Connectivity

**Def.** Node \( u \) and \( v \) are **mutually reachable** if there is a path from \( u \) to \( v \) and also a path from \( v \) to \( u \).

**Def.** A graph is **strongly connected** if every pair of nodes is mutually reachable.

**Lemma.** Let \( s \) be any node. \( G \) is strongly connected iff every node is reachable from \( s \), and \( s \) is reachable from every node.

**Pf.** \( \Rightarrow \) Follows from definition.

**Pf.** \( \Leftarrow \) Path from \( u \) to \( v \): concatenate \( u-s \) path with \( s-v \) path.
Path from \( v \) to \( u \): concatenate \( v-s \) path with \( s-u \) path.

\( \checkmark \)

ok if paths overlap
**Theorem.** Can determine if $G$ is strongly connected in $O(m + n)$ time.

**Pf.**
- Pick any node $s$.
- Run BFS from $s$ in $G$.
- Run BFS from $s$ in $G^{rev}$.
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma.

---

---

**Diagram:**
- **strongly connected**
- **not strongly connected**
3.6 DAGs and Topological Ordering
Directed Acyclic Graphs

**Def.** An **DAG** is a directed graph that contains no directed cycles.

**Ex.** Precedence constraints: edge \((v_i, v_j)\) means \(v_i\) must precede \(v_j\).

**Def.** A **topological order** of a directed graph \(G = (V, E)\) is an ordering of its nodes as \(v_1, v_2, \ldots, v_n\) so that for every edge \((v_i, v_j)\) we have \(i < j\).
Precedence Constraints

Precedence constraints. Edge \((v_i, v_j)\) means task \(v_i\) must occur before \(v_j\).

Applications.
- Course prerequisite graph: course \(v_i\) must be taken before \(v_j\).
- Compilation: module \(v_i\) must be compiled before \(v_j\). Pipeline of computing jobs: output of job \(v_i\) needed to determine input of job \(v_j\).
Directed Acyclic Graphs

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

**Pf.** (by contradiction)
- Suppose that $G$ has a topological order $v_1, \ldots, v_n$ and that $G$ also has a directed cycle $C$. Let's see what happens.
- 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 $v_1, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction.  

$$\begin{array}{cccccccc}
  & & & & \text{the directed cycle } C \\
  v_1 & \quad & v_i & \quad & v_j & \quad & \cdots & \quad & v_n \\
\end{array}$$

the supposed topological order: $v_1, \ldots, v_n$

Directed Acyclic Graphs

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

Q. Does every DAG have a topological ordering?

Q. If so, how do we compute one?
Directed Acyclic Graphs

Lemma. If $G$ is a DAG, then $G$ has a node with no incoming edges.

Pf. (by contradiction)
- Suppose that $G$ is a DAG and every node has at least one incoming edge. Let’s see what happens.
- Pick any node $v$, and begin following edges backward from $v$. Since $v$ has at least one incoming edge $(u, v)$ we can walk backward to $u$.
- Then, since $u$ has at least one incoming edge $(x, u)$, we can walk backward to $x$.
- Repeat until we visit a node, say $w$, twice.
- Let $C$ denote the sequence of nodes encountered between successive visits to $w$. $C$ is a cycle. □
Directed Acyclic Graphs

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

Pf. (by induction on $n$)

- Base case: true if $n = 1$.
- Given DAG on $n > 1$ nodes, find a node $v$ with no incoming edges.
- $G - \{v\}$ is a DAG, since deleting $v$ cannot create cycles.
- By inductive hypothesis, $G - \{v\}$ has a topological ordering.
- Place $v$ first in topological ordering; then append nodes of $G - \{v\}$ in topological order. This is valid since $v$ has no incoming edges.

To compute a topological ordering of $G$:
Find a node $v$ with no incoming edges and order it first
Delete $v$ from $G$
Recursively compute a topological ordering of $G - \{v\}$ and append this order after $v$
**Theorem.** Algorithm finds a topological order in $O(m + n)$ time.

**Pf.**

- Maintain the following information:
  - $\text{count}[w] =$ remaining number of incoming edges
  - $S =$ set of remaining nodes with no incoming edges
- Initialization: $O(m + n)$ via single scan through graph.
- Update: to delete $v$
  - remove $v$ from $S$
  - decrement $\text{count}[w]$ for all edges from $v$ to $w$, and add $w$ to $S$ if $c \text{count}[w]$ hits 0
  - this is $O(1)$ per edge
Greedy Algorithms
4.1 Interval Scheduling
Interval Scheduling

Interval scheduling.
- Job j starts at $s_j$ and finishes at $f_j$.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.
Interval Scheduling: Greedy Algorithms

**Greedy template.** Consider jobs in some natural order. Take each job provided it's compatible with the ones already taken.

- [Earliest start time] Consider jobs in ascending order of $s_j$.
- [Earliest finish time] Consider jobs in ascending order of $f_j$.
- [Shortest interval] Consider jobs in ascending order of $f_j - s_j$.
- [Fewest conflicts] For each job $j$, count the number of conflicting jobs $c_j$. Schedule in ascending order of $c_j$. 
Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some natural order. Take each job provided it's compatible with the ones already taken.

counterexample for earliest start time

counterexample for shortest interval

counterexample for fewest conflicts
Interval Scheduling: Greedy Algorithm

**Greedy algorithm.** Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

Sort jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

- set of jobs selected

\[ A \leftarrow \emptyset \]

for \( j = 1 \) to \( n \) {
  if (job \( j \) compatible with \( A \))
    \[ A \leftarrow A \cup \{ j \} \]
}

return \( A \)

**Implementation.** \( O(n \log n) \).

- Remember job \( j^* \) that was added last to \( A \).
- Job \( j \) is compatible with \( A \) if \( s_j \geq f_{j^*} \).
Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots, i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots, j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of $r$.

why not replace job $j_{r+1}$ with job $i_{r+1}$?
**Theorem.** Greedy algorithm is optimal.

**Pf.** (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots, i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots, j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of $r$.

![Diagram showing greedy and optimal solutions](image)

- Job $i_{r+1}$ finishes before $j_{r+1}$.
- Solution still feasible and optimal, but contradicts maximality of $r$. 
4.1 Interval Partitioning
Interval Partitioning

Interval partitioning.
- Lecture j starts at $s_j$ and finishes at $f_j$.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Ex: This schedule uses 4 classrooms to schedule 10 lectures.
Interval Partitioning

Interval partitioning.
- Lecture $j$ starts at $s_j$ and finishes at $f_j$.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Ex: This schedule uses only 3.
**Def.** The depth of a set of open intervals is the maximum number that contain any given time.

**Key observation.** Number of classrooms needed $\geq$ depth.

**Ex:** Depth of schedule below $= 3 \Rightarrow$ schedule below is optimal.

Q. Does there always exist a schedule equal to depth of intervals?
Interval Partitioning: Greedy Algorithm

**Greedy algorithm.** Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

Sort intervals by starting time so that \( s_1 \leq s_2 \leq \ldots \leq s_n \).

\[
d \leftarrow 0
\]

number of allocated classrooms

\[
\text{for } j = 1 \text{ to } n \{ \\
\quad \text{if (lecture } j \text{ is compatible with some classroom } k) \\
\quad \quad \text{schedule lecture } j \text{ in classroom } k \\
\quad \text{else} \\
\quad \quad \text{allocate a new classroom } d + 1 \\
\quad \quad \text{schedule lecture } j \text{ in classroom } d + 1 \\
\quad \quad d \leftarrow d + 1 \\
\}
\]

**Implementation.** \( O(n \log n) \).

- For each classroom \( k \), maintain the finish time of the last job added.
- Keep the classrooms in a priority queue.
Interval Partitioning: Greedy Analysis

**Observation.** Greedy algorithm never schedules two incompatible lectures in the same classroom.

**Theorem.** Greedy algorithm is optimal.

**Pf.**
- Let d = number of classrooms that the greedy algorithm allocates.
- Classroom d is opened because we needed to schedule a job, say j, that is incompatible with all d-1 other classrooms.
- These d jobs each end after $s_j$.
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than $s_j$.
- Thus, we have d lectures overlapping at time $s_j + \varepsilon$.
- Key observation $\Rightarrow$ all schedules use $\geq d$ classrooms. •
shortest route from Wang Hall to Miami Beach
Shortest Path Problem

Shortest path network.
- Directed graph $G = (V, E)$.
- Source $s$, destination $t$.
- Length $\ell_e = \text{length of edge } e$.

Shortest path problem: find shortest directed path from $s$ to $t$.

Cost of path $s$-2-3-5-$t$
\[ = 9 + 23 + 2 + 16 \\
= 50.\]
Dijkstra's Algorithm

Dijkstra's algorithm.

- Maintain a set of explored nodes \( S \) for which we have determined the shortest path distance \( d(u) \) from \( s \) to \( u \).
- Initialize \( S = \{ s \} \), \( d(s) = 0 \).
- Repeatedly choose unexplored node \( v \) which minimizes

\[
\pi(v) = \min_{e = (u, v) : u \in S} d(u) + \ell_e,
\]

add \( v \) to \( S \), and set \( d(v) = \pi(v) \).

![Diagram of Dijkstra's Algorithm](https://via.placeholder.com/150)
Dijkstra's Algorithm

Dijkstra's algorithm.

- Maintain a set of explored nodes $S$ for which we have determined the shortest path distance $d(u)$ from $s$ to $u$.
- Initialize $S = \{ s \}$, $d(s) = 0$.
- Repeatedly choose unexplored node $v$ which minimizes $\pi(v) = \min_{e = (u, v) : u \in S} d(u) + \ell_e$, add $v$ to $S$, and set $d(v) = \pi(v)$. shortest path to some $u$ in explored part, followed by a single edge $(u, v)$.

\[ \pi(v) = \min_{e = (u, v) : u \in S} d(u) + \ell_e, \]

\[ add \ v \ to \ S, \ and \ set \ d(v) = \pi(v). \]
Dijkstra's Algorithm: Proof of Correctness

Invariant. For each node \( u \in S \), \( d(u) \) is the length of the shortest \( s-u \) path.

Pf. (by induction on \( |S| \))

Base case: \( |S| = 1 \) is trivial.

Inductive hypothesis: Assume true for \( |S| = k \geq 1 \).

- Let \( v \) be next node added to \( S \), and let \( u-v \) be the chosen edge.
- The shortest \( s-u \) path plus \( (u, v) \) is an \( s-v \) path of length \( \pi(v) \).
- Consider any \( s-v \) path \( P \). We'll see that it's no shorter than \( \pi(v) \).
- Let \( x-y \) be the first edge in \( P \) that leaves \( S \), and let \( P' \) be the subpath to \( x \).
- \( P \) is already too long as soon as it leaves \( S \).

\[
\ell(P) + \ell(x,y) \geq d(x) + \ell(x, y) \geq \pi(y) \geq \pi(v)
\]

\( \uparrow \) nonnegative weights  \( \uparrow \) inductive hypothesis  \( \uparrow \) defn of \( \pi(y) \)  \( \uparrow \) Dijkstra chose \( v \) instead of \( y \)
Dijkstra's Algorithm: Implementation

For each unexplored node, explicitly maintain \( \pi(v) = \min_{e=(u,v): u \in S} d(u) + \ell_e \).

- Next node to explore = node with minimum \( \pi(v) \).
- When exploring \( v \), for each incident edge \( e=(v,w) \), update
  \[
  \pi(w) = \min \{ \pi(w), \pi(v) + \ell_e \}.
  \]

Efficient implementation. Maintain a priority queue of unexplored nodes, prioritized by \( \pi(v) \).

<table>
<thead>
<tr>
<th>PQ Operation</th>
<th>Dijkstra</th>
<th>Array</th>
<th>Binary heap</th>
<th>d-way Heap</th>
<th>Fib heap †</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>( n )</td>
<td>( n )</td>
<td>( \log n )</td>
<td>( d \log_d n )</td>
<td>1</td>
</tr>
<tr>
<td>ExtractMin</td>
<td>( n )</td>
<td>( n )</td>
<td>( \log n )</td>
<td>( d \log_d n )</td>
<td>( \log n )</td>
</tr>
<tr>
<td>ChangeKey</td>
<td>( m )</td>
<td>( 1 )</td>
<td>( \log n )</td>
<td>( \log_d n )</td>
<td>1</td>
</tr>
<tr>
<td>IsEmpty</td>
<td>( n )</td>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>( n^2 )</td>
<td>( m \log n )</td>
<td>( m \log_m n )</td>
<td>( m + n \log n )</td>
<td></td>
</tr>
</tbody>
</table>

† Individual ops are amortized bounds
Extra Slides
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>

$d_3 = 9$, $d_2 = 8$, $d_6 = 15$, $d_1 = 6$, $d_5 = 14$, $d_4 = 9$

Lateness: $\ell_3 = 2$, $\ell_2 = 0$, max lateness $= 6$. 
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 \( t_j \).
  
  \[
  \begin{array}{c|c|c}
  \hline
  & 1 & 2 \\
  \hline
  t_j & 1 & 10 \\
  \hline
  d_j & 100 & 10 \\
  \hline
  \end{array}
  \]

  counterexample

- **[Smallest slack]** Consider jobs in ascending order of slack \( d_j - t_j \).
  
  \[
  \begin{array}{c|c|c}
  \hline
  & 1 & 2 \\
  \hline
  t_j & 1 & 10 \\
  \hline
  d_j & 2 & 10 \\
  \hline
  \end{array}
  \]

  counterexample
Minimizing Lateness: Greedy Algorithm

**Greedy algorithm.** Earliest deadline first.

Sort n jobs by deadline so that \( d_1 \leq d_2 \leq \ldots \leq d_n \)

\[
t \leftarrow 0 \\
\text{for } j = 1 \text{ to } n \\
\quad \text{Assign job } j \text{ to interval } [t, t + t_j] \\
\quad s_j \leftarrow t, f_j \leftarrow t + t_j \\
\quad t \leftarrow t + t_j \\
\text{output intervals } [s_j, f_j]
\]

max lateness = 1
Minimizing Lateness: No Idle Time

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

<table>
<thead>
<tr>
<th>d = 4</th>
<th>d = 6</th>
<th>d = 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Observation.** The greedy schedule has no idle time.
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 \( i \).

![Diagram of job sequence with inversion marked]

[as before, we assume jobs are numbered so that \( d_1 \leq d_2 \leq ... \leq d_n \)]

**Observation.** Greedy schedule has no inversions.

**Observation.** 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: $i < j$ 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 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, Dijkstra, 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: \[\text{a b c d e f}\]

future queries: \[\text{g a b c e d a b b a c d e a f a d e f g h ...}\]

\[\uparrow\text{cache miss}\]

\[\uparrow\text{eject this one}\]

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.

- **Unreduced Schedule:**
  
  a | a | b | c
  ---|---|---|---
  a | a | x | c
  c | a | d | c
  d | a | d | b
  a | a | c | b
  b | a | x | b
  c | a | c | b
  a | a | b | c
  a | a | b | c

- **Reduced Schedule:**
  
  a | a | b | c
  ---|---|---|---
  a | a | b | c
  a | a | b | c
  a | a | b | c
  a | a | b | c
  a | a | c | b
  a | a | c | b
  a | a | c | b
  a | a | c | b
**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**

```
S  |  c  
---|-----
 t  |     
 d  |     
 t' |     
    |     
    | e   
    |     
```

**Case 2**

```
S  |  c  
---|-----
 t  |     
 d  |     
 t' |     
    |     
    | e   
    |     
```

$S'$

```
S' |  c  
---|-----
 t' |     
    |     
    | d   
    |     
    |     
```

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

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

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

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). $S' = S$ satisfies invariant.
- **Case 2:** ($d$ is not in the cache and $S$ and $S_{FF}$ evict the same element). $S' = S$ satisfies invariant.
Pf. (continued)

- Case 3: \((d \text{ is not in the cache; } S_{FF} \text{ evicts } e; \ S \text{ evicts } f \neq e)\).
  - begin construction of \(S'\) from \(S\) by evicting \(e\) instead of \(f\)

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

- 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 \(e\)
Farthest-In-Future: Analysis

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 $j'$.

![Diagram](image)

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

- **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 $j+1$. 

[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 \( j' \).

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

\[\uparrow\]

must involve \( e \) or \( f \) (or both)

otherwise \( S' \) would take the same action

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

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

Greed is good. Greed is right. Greed works. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit.

- Gordon Gecko (Michael Douglas)
Coin Changing

**Goal.** Given currency denominations: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins.

**Ex:** 34¢.

**Cashier’s algorithm.** At each iteration, add coin of the largest value that does not take us past the amount to be paid.

**Ex:** $2.89.
Coin-Changing: Greedy Algorithm

Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

Sort coins denominations by value: \( c_1 < c_2 < \ldots < c_n \).

\[
\begin{align*}
S & \leftarrow \emptyset \\
\text{while } (x \neq 0) \{ \\
\quad & \text{let } k \text{ be largest integer such that } c_k \leq x \\
\quad & \text{if } (k = 0) \\
\quad & \quad \text{return } "\text{no solution found}" \\
\quad & \quad x \leftarrow x - c_k \\
\quad & \quad S \leftarrow S \cup \{k\} \\
\} \\
\text{return } S
\end{align*}
\]

Q. Is cashier's algorithm optimal?
Coin-Changing: Analysis of Greedy Algorithm

**Theorem.** Greed is optimal for U.S. coinage: 1, 5, 10, 25, 100.

**Pf.** (by induction on x)

- Consider optimal way to change $c_k \leq x < c_{k+1}$: greedy takes coin $k$.
- We claim that any optimal solution must also take coin $k$.
  - if not, it needs enough coins of type $c_1, \ldots, c_{k-1}$ to add up to $x$
  - table below indicates no optimal solution can do this
- Problem reduces to coin-changing $x - c_k$ cents, which, by induction, is optimally solved by greedy algorithm.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$c_k$</th>
<th>All optimal solutions must satisfy</th>
<th>Max value of coins 1, 2, ..., k-1 in any OPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$P \leq 4$</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>$N \leq 1$</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>$N + D \leq 2$</td>
<td>$4 + 5 = 9$</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>$Q \leq 3$</td>
<td>$20 + 4 = 24$</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>no limit</td>
<td>$75 + 24 = 99$</td>
</tr>
</tbody>
</table>
Observation. Greedy algorithm is sub-optimal for US postal denominations: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

Counterexample. 140¢.
- Greedy: 100, 34, 1, 1, 1, 1, 1.
- Optimal: 70, 70.
Selecting Breakpoints
Selecting Breakpoints

Selecting breakpoints.
- Road trip from Princeton to Palo Alto along fixed route.
- Refueling stations at certain points along the way.
- Fuel capacity = $C$.
- Goal: makes as few refueling stops as possible.

**Greedy algorithm.** Go as far as you can before refueling.
Selecting Breakpoints: Greedy Algorithm

Truck driver's algorithm.

Sort breakpoints so that: 0 = b₀ < b₁ < b₂ < ... < bₙ = L

S ← {0} ← breakpoints selected
x ← 0 ← current location

while (x ≠ bₙ)
    let p be largest integer such that bₚ ≤ x + C
    if (bₚ = x)
        return "no solution"
    x ← bₚ
    S ← S ∪ {p}
return S

Implementation. O(n log n)

- Use binary search to select each breakpoint p.
Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)
- Assume greedy is not optimal, and let's see what happens.
- Let \( 0 = g_0 < g_1 < \ldots < g_p = L \) denote set of breakpoints chosen by greedy.
- Let \( 0 = f_0 < f_1 < \ldots < f_q = L \) denote set of breakpoints in an optimal solution with \( f_0 = g_0, f_1 = g_1, \ldots, f_r = g_r \) for largest possible value of \( r \).
- Note: \( g_{r+1} > f_{r+1} \) by greedy choice of algorithm.

Greedy:
\[
\begin{array}{ccccccc}
g_0 & g_1 & g_2 & \cdots & g_r & g_{r+1} \\
\end{array}
\]

OPT:
\[
\begin{array}{ccccccc}
f_0 & f_1 & f_2 & \cdots & f_r & f_{r+1} \\
\end{array}
\]

\[\text{why doesn't optimal solution drive a little further?}\]
Selecting Breakpoints: Correctness

**Theorem.** Greedy algorithm is optimal.

**Pf.** (by contradiction)
- Assume greedy is not optimal, and let’s see what happens.
- Let \(0 = g_0 < g_1 < \ldots < g_p = L\) denote set of breakpoints chosen by greedy.
- Let \(0 = f_0 < f_1 < \ldots < f_q = L\) denote set of breakpoints in an optimal solution with \(f_0 = g_0, f_1 = g_1, \ldots, f_r = g_r\) for largest possible value of \(r\).
- Note: \(g_{r+1} > f_{r+1}\) by greedy choice of algorithm.
The question of whether computers can think is like the question of whether submarines can swim.

Do only what only you can do.

In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.

The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.

APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.