Announcement: Homework 1 released!

Due: January 24th at 11:59PM (Gradescope)

Recap: Graphs

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

Breadth First Search
- BFS Tree
  - O(m+n) algorithm
  - Applications: Connected Components, Shortest Path etc...

Testing Bipartiteness
- Directed Graphs and Strong Connectivity

Strong Connectivity: Algorithm

Theorem. Can determine if G is strongly connected in O(m+n) time.

Proof.
1. Pick any node s.
2. Run BFS from s in G.
3. Run BFS from s in Grev.
4. Return true iff all nodes reached in both BFS executions.
5. Correctness follows immediately from previous lemma.

Directed Acyclic Graphs

Definition. An DAG is a directed graph that contains no directed cycles.

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

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 \(v_j\).

Applying Topological Sorting

Function \(F(v)\)

\[
\begin{align*}
W &:= 2 * V; \\
X &:= W + V; \\
Y &:= X * W; \\
C &:= W * W; \\
Z &:= Y + V; \\
& \text{return } Z
\end{align*}
\]
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. □

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. □

Topological Sorting Algorithm: Running Time

Theorem. Algorithm finds a topological order in $O(m + n)$ time.

Pf.

- Maintain the following information:
  - $\text{count}[v]$ = 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 $\text{count}[v]$ for all edges from $v$ to $w$, and add $w$ to $S$ if $\text{count}[w]$ hits 0
- this is $O(1)$ per edge. □

Copyright © 2005 Pearson-Addison Wesley. All rights reserved.
# 4.1 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.

## 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 \( c_j \).

## Greedy Algorithm

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

1. Sort jobs by finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_n \).
2. Set of jobs selected:
   ```
   \begin{align*}
   &A \leftarrow \emptyset, \\
   &\text{for } j = 1 \text{ to } n \{ \\
   &\quad A \leftarrow A \cup \{j\} \\
   &\text{if (job } j \text{ compatible with } A) \\
   &\text{return } A
   \end{align*}
   ```

**Implementation.** \( O(n \log n) \).
- Remember job \( j^* \) that was added last to \( A \).
- Job \( j \) is compatible with \( A \) if \( s_j \geq f_j^* \).

## Analysis

**Theorem.** Greedy algorithm is optimal.

**Pf.** (by contradiction)
- Assume greedy is not optimal, and let’s see what happens.
- Let \( j_1, j_2, \ldots, j_r \) denote set of jobs selected by greedy.
- Let \( j_1^*, j_2^*, \ldots, j_n^* \) denote set of jobs in the optimal solution with \( s_{j_1^*} \leq s_{j_2^*} \leq \ldots \leq s_{j_n^*} \), for the largest possible value of \( r \).

- Greedy:
  ```
  \begin{align*}
  j_1 &< j_2 < \ldots < j_r
  \end{align*}
  ```
- OPT:
  ```
  \begin{align*}
  j_1^* &< j_2^* < \ldots < j_n^*
  \end{align*}
  ```

- why not replace job \( j_{r+1} \) with job \( j_{r+1}^* \)
Interval Scheduling: Analysis

Theorem. Greedy algorithm is optimal.

\textbf{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 \( r \).

**Greedy:**
\[
\begin{array}{cccccc}
\hline
& j_1 & j_2 & j_3 & \cdots & j_m \\
\hline
\text{OPT:} & \underline{\ } & \underline{\ } & \underline{\ } & \cdots & \underline{\ } \\
\end{array}
\]

\text{solution still feasible and optimal, but contradicts maximality of } r.

4.1 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.

\[
\begin{array}{cccccccccccc}
9 & 9:30 & 10 & 10:30 & 11 & 11:30 & 12 & 12:30 & 1 & 1:30 & 2 & 2:30 \\
\hline
a & b & c & d & e & f & g & h & i & j & & \\
\end{array}
\]

Interval Partitioning: Lower Bound on Optimal Solution

**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.

\[
\begin{array}{cccccccccccc}
9 & 9:30 & 10 & 10:30 & 11 & 11:30 & 12 & 12:30 & 1 & 1:30 & 2 & 2:30 \\
\hline
a & b & c & d & e & f & g & h & i & j & & \\
\end{array}
\]

Interval Partitioning: Greedy Algorithm

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

\[
\begin{aligned}
\text{Sort intervals by starting time so that } s_1 \leq s_2 \leq \ldots \leq s_n. \\
\text{\( d \) \( \triangleright \) \( \vdash \) number of allocated classrooms.} \\
\text{for } j = 1 \text{ to } n \\
\text{if (lecture } j \text{ is compatible with some classroom } k) } \\
\text{schedule lecture } j \text{ in classroom } k \\
\text{else allocate a new classroom } d + 1 \\
\text{schedule lecture } j \text{ in classroom } d + 1 \\
\text{d \( \triangleright \) d + 1} \\
\end{aligned}
\]

**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.

Proof. 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 (including j) 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.

4.4 Shortest Paths in a Graph

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 $d(v) = \min_{u \in S} d(u) + \ell_{uv}$.
- Add v to S, and set $d(v) = (v)$.

Dijkstra’s Algorithm: Proof of Correctness

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

Proof. (by induction on $|S|$.)

Base case: $|S| = 1$ is trivial.

Inductive hypothesis: Assume true for $|S| < k$.

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 $d(v)$.

Consider any s-v path P. We’ll see that it’s no shorter than $d(v)$.

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.

$\ell(P) \geq \ell(P') + \ell(x, y) = d(x) + \ell(x, y) = d(y) \geq d(v)$

Dijkstra chose v instead of y
Dijkstra’s Algorithm: Implementation

For each unexplored node, explicitly maintain $$\pi(v) = \min_{u \in \mathcal{N}(v) \setminus \{s\}} d(u) + \lambda_e.$$  

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

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>–</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>ExtractMin</td>
<td>–</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>ChangeKey</td>
<td>–</td>
<td>–</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Empty</td>
<td>–</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

† Individual ops are amortized bounds

4.2 Scheduling to Minimize 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: $$\lambda_j = \max\{0, f_j - d_j\}.$$  
- Goal: schedule all jobs to minimize maximum lateness $$L = \max \lambda_j.$$  

Ex:

<table>
<thead>
<tr>
<th>Job</th>
<th>$$d_j$$</th>
<th>$$t_j$$</th>
<th>Lateness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>14</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

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

Extra Slides
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 = 0
For j = 1 to n
  Assign job j to interval \([t, t + t_j]\)
  \( s_j = t \), \( f_j = t + t_j \)
  \( t = t + t_j \)
output intervals \([s_j, f_j]\)

Minimizing Lateness: No Idle Time

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

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.

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

Pf. Let \( \lambda \) be the lateness before the swap, and let \( \lambda' \) be it afterwards.

\( \lambda' = \lambda \) for all \( k \neq i, j \)
\( \lambda'_i \leq \lambda_i \)
\( \lambda'j \leq \lambda_j \)

If job j is late:

\[ f'_i = f_i - d_j \] (definition)
\[ = f_i - d_j \] (job finishes at time \( f_i \))
\[ \leq \lambda_j \] (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.

\( S^* \) 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, Dijkstra, Huffman, ...
4.3 Optimal Caching

Optimal Offline Caching

- Cache with capacity to store k items.
- Sequence of m item requests d1, d2, ..., dm.
- 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.

Farthest-In-Future

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

Theorem: [Bellady, 1960s] FF is optimal eviction schedule.
Proof: Algorithm and theorem are intuitive; proof is subtle.

Reduced Eviction Schedules

Claim: Given any unreduced schedule S, can transform it into a reduced schedule S’ with no more cache misses.
Proof: (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.

Theorem: FF is optimal eviction algorithm.
Proof: (by induction on number or requests j)

Invariant: There exists an optimal reduced schedule S that makes the same eviction schedule as SFF 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) request d = dj+1.
- Since S and SFF 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 SFF evict the same element). S’ = S satisfies invariant.
Farthest-In-Future: Analysis

PF. (continued)

- Case 3: \(d\) is not in the cache; \(S_F\) evicts \(e\); \(S\) evicts \(f\) ≠ \(e\).
- Begin construction of \(S'\) from \(S\) by evicting \(e\) instead of \(f\)

\[
\begin{array}{c|c|c}
\text{j} & \text{s} & \text{f} \\
\hline
\text{j+1} & \text{s} & \text{d} \\
\end{array}
\]

- Now \(S'\) agrees with \(S_F\) 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 first time after \(j+1\) that \(S\) and \(S'\) take a different action, and let \(g\) be item requested at time \(j'\).

- 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 \(e' ≠ e\), \(S'\) evicts \(e'\) and brings \(e\) into 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 \(S_F\) 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 \(j'\).

\[
\begin{array}{c|c|c}
\text{j} & \text{s} & \text{f} \\
\hline
\text{j+1} & \text{s} & \text{d} \\
\end{array}
\]

- \(S'\) would take the same action
- \(S\) must evict \(e\).
- Make \(S'\) evict \(f\); now \(S\) and \(S'\) have the same cache.

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.

PF with direction of time reversed!

Theorem. PF 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

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.

```plaintext
Sort coins denominations by value: c1 < c2 < ... < cn.
S  
while (x  0) {
  let k be largest integer such that ck ≤ x
  if (k = 0)
    return "no solution found"
  x  x - ck
  S  S  {k}
}
return S
```

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.

Proof (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</th>
<th>Table below indicates no optimal solution can do this</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>( P &lt; 4 )</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>( N + D &lt; 2 )</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td>( Q &lt; 3 )</td>
<td>20 + 4 = 24</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>( no limit )</td>
<td>75 + 24 = 99</td>
<td></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.**

- Road trip from Princeton to Palo Alto along fixed route.
- Refueling stations at certain points along the way.
- Fuel capacity = \( C \).
- Goal: make as few refueling stops as possible.

**Greedy algorithm.** Go as far as you can before refueling.

```
<table>
<thead>
<tr>
<th>Princeton</th>
<th>C</th>
<th>C</th>
<th>C</th>
<th>C</th>
<th>C</th>
<th>C</th>
<th>C</th>
<th>C</th>
<th>C</th>
<th>C</th>
<th>C</th>
<th>Palo Alto</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**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 \( b_p \leq x+C \)
  if (b_p = x)
    return "no solution"
  x  x - \( b_p \)
  S  S  \{p\}
}
return S
```

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

- Use binary search to select each breakpoint \( p \).
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.

Another optimal solution has one more breakpoint in common \(\Rightarrow\) contradiction.

Greedy:

\[
\begin{array}{cccccc}
  g_0 & g_1 & g_2 & \ldots & g_r & \text{Stop} \\
\end{array}
\]

Optimal:

\[
\begin{array}{cccccc}
  f_0 & f_1 & f_2 & \ldots & f_r & f_{r+1} \\
\end{array}
\]

Edsger W. Dijkstra

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.