CSE 101 Exam 3 Review
Greedy Algorithms (Ch 5)

- Basics
- Change making
- Interval scheduling
- Exchange arguments
- Optimal caching
- Huffman codes
- Minimal spanning trees
Greedy Algorithms

General Algorithmic Technique:
1. Find decision criterion
2. Make best choice according to criterion
3. Repeat until done

Surprisingly, this sometimes works.
Things to Keep in Mind about Greedy Algorithms

• Algorithms are very natural and easy to write down.
• However, not all greedy algorithms work.
• Proving correctness is important.
Problem: Given a collection C of intervals, find a subset $S \subseteq C$ so that:

1. No two intervals in $S$ overlap.
2. Subject to (1), $|S|$ is as large as possible.
Algorithm

IntervalScheduling(C)
S ← {}
While(some interval in C doesn’t overlap any in S)
    Let J be the non-overlapping interval with smallest max
    Add J to S
Return S
Proof of Correctness

- Algorithm produces \( J_1, J_2, \ldots, J_s \) with \( J_i = [x_i, y_i] \).
- Consider some other solution
  \( K_1, K_2, \ldots, K_t \) with \( K_i = [w_i, z_i] \).

**Claim:** For each \( m \leq t \), \( y_m \leq z_m \).
In particular, \( s \geq t \).
Proof of Claim

Use Induction on m.

**Base Case:** $m = 1$.

$J_1$ is the interval with smallest max, so $y_1 \leq z_1$.

**Inductive Step:** Assume $y_m \leq z_m$.

- $J_{m+1}$ has smallest $y$ for any $[x,y]$ with $x > y_m$.
- $K_{m+1} = [w_{m+1}, z_{m+1}]$ has
  
  $w_{m+1} > z_m \geq y_m$

- Therefore, $y_{m+1} \leq z_{m+1}$. 
Exchange Argument

- Greedy algorithm makes a sequence of decisions $D_1, D_2, D_3, \ldots, D_n$ eventually reaching solution $G$.
- Need to show that for arbitrary solutions $A$ that $G \geq A$.
- Find sequence of solutions $A=A_0, A_1, A_2, \ldots, A_n = G$ so that:
  - $A_i \leq A_{i+1}$
  - $A_i$ agrees with $D_1, D_2, \ldots, D_i$
Exchange Argument

In particular, we need to show that given any $A_i$ consistent with $D_1,...,D_i$ we can find an $A_{i+1}$ so that:

• $A_{i+1}$ is consistent with $D_1,...,D_{i+1}$
• $A_{i+1} \geq A_i$

Then we inductively construct sequence

$A=A_0 \leq A_1 \leq A_2 \leq ... \leq A_n = G$

Thus, $G \geq A$ for any $A$. So $G$ is optimal.
Model

- $k$ words in cache at a time.
- CPU asks for memory access.
- If in cache already, easy.
- Otherwise, need to load into cache replacing something else, slow.
**Optimal Caching**

**Problem:** Given sequence of memory accesses and cache size, find a cache schedule that involves fewest possible number of swaps with disk.

<table>
<thead>
<tr>
<th>CPU</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>D</th>
<th>E</th>
<th>C</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8 Cache misses.
**Furthest In The Future (FITF)**

- For each cell consider the next time that memory location will be called for.
- Replace cell whose next call is the furthest in the future.

<table>
<thead>
<tr>
<th>X</th>
<th>A</th>
<th>Y</th>
<th>A</th>
<th>B</th>
<th>B</th>
<th>X</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Proof of Optimality

• Exchange argument
• $n^{th}$ decision: What to do at $n^{th}$ time step.
• Given schedule $S$ that agrees with FITF for first $n$ time steps, create schedule $S'$ that agrees for $n+1$ and has no more cache misses.
Case 1: S agrees with FITF on step n+1

Nothing to do. $S' = S.$
Case 2: S Makes Unnecessary Replacement

If S replaces some element of memory that was not immediately called for, put it off.

Can assume that S only replaces elements if there’s a cache miss.
Case 3

The remaining case is that there is a cache miss at step \( n+1 \) and that \( S \) replaces the *wrong* thing.
Case 3a: S throws out B before using it
Case 3b: S keeps B until it is used

• B is FITF
• A is used sometime before B.
• A must be loaded into memory somewhere else.
Case 3b: S keeps B until it is used

Instead of replacing A and then bringing it back, we can replace B and then bring it back.
Huffman Codes

**Definition:** An encoding is prefix-free if the encoding of no letter is a prefix of the encoding of any other.

**Lemma:** Any prefix-free encoding can be uniquely decoded.
Optimal Encoding

**Problem:** Given a string, S, find a prefix-free encoding that encodes S using the fewest number of bits.
How Long is the Encoding?

If for each letter $x$ in our string, $x$ appears $f(x)$ times and if we encode $x$ as a string of length $\ell(x)$, the total encoding length is:

$$\sum f(x) \cdot \ell(x).$$
Tree Representation

Can represent prefix-free encoding as a tree.

Letters are leaves.
Length of encoding = Depth of leaf.
Siblings

- No matter what the tree structure, two of the deepest leaves are siblings.
- Can assume filled by two least frequent elements.
- Can assume that two least frequent elements are siblings!
Algorithm

HuffmanTree(L)

While(at least two left)
    x, y ← Two least frequent
    z new node f(z) ← f(x)+f(y)
    x and y children of z
    Replace x and y with z in L
Return remaining elt of L
Minimum Spanning Trees

**Note:** In this problem, you will never want to build more roads than necessary. This means, you will never want to have a cycle.

**Definition:** A *tree* is a connected graph, with no cycles. A *spanning tree* in a graph $G$, is a subset of the edges of $G$ that connect all vertices and have no cycles.

If $G$ has weights, a *minimum spanning tree* is a spanning tree whose total weight is as small as possible.
Basic Facts about Trees

**Lemma:** For an undirected graph $G$, any two of the below imply the third:

1. $|E| = |V| - 1$
2. $G$ is connected
3. $G$ has no cycles

**Corollary:** If $G$ is a tree, then $|E| = |V| - 1$. 
Greedy Idea

How do you make an MST?
• Try using the cheapest edges.

**Proposition:** In a graph $G$, let $e$ be an edge of lightest weight. Then there exists an MST of $G$ containing $e$. Furthermore, if $e$ is the unique lightest edge, then *all* MSTs contain $e$. 
One Step

• After picking lightest edge, what then?
• Contract edge and repeat.
• Equivalent to picking lightest edge that doesn’t form a cycle.
Algorithm

Kruskal(G)

\[ T \leftarrow \{\} \]
\[ \text{While}(|T| < |V|-1) \]
  \[ \text{Find lightest edge } e \text{ that doesn’t create cycle with } T \]
  \[ \text{Add } e \text{ to } T \]
\[ \text{Return } T \]
Optimized Kruskal

Kruskal(G)
Sort edges by weight
T ← {}
Create Union Find
For v ∈ V, New(v)
For (v,w) ∈ E in increasing order
    If Rep(v) ≠ Rep(w)
        Add (v,w) to T
        Join(v,w)
Return T

Runtime:O(|E|log|E|)
Other Algorithms

There are many other ways to create MST algorithms. Kruskal searches the whole graph for light edges, but you can also grow from a point.

**Proposition:** In a graph $G$, with vertex $v$, let $e$ be an edge of lightest weight adjacent to $v$. Then there exists an MST of $G$ containing $e$. Furthermore, if $e$ is the unique lightest edge, then *all* MSTs contain $e$. 
Prim’s Algorithm

**Prim’s Algorithm:** Add lightest edge that connects \( v \) to a new vertex.

Implementation very similar to Dijkstra.
Prim’s Algorithm

Prim(G,w)

Pick vertex s \hspace{1cm} \text{\textbackslash\hspace{0.5cm} doesn’t matter which}
For v ∈ V, b(v) ← ∞ \hspace{1cm} \text{\textbackslash\hspace{0.5cm} lightest edge into v}
T ← {}, b(s) ← 0
Priority Queue Q, add all v with key=b(v)
While(Q not empty)
    u ← DeleteMin(Q)
    If u ≠ s, add (u,Prev(u)) to T
    For (u,v) ∈ E
        If w(u,v) < b(v)
            b(v) ← w(u,v)
            Prev(v) ← u
            DecreaseKey(v)
Return T

Runtime:
O(|V|\log|V| + |E|)
Slightly better than Kruskal
Analysis

At any stage, have some set $S$ of vertices connected to $s$. Find cheapest edge connecting $S$ to $S^C$.

**Proposition:** In a graph $G$, with a cut $C$, let $e$ be an edge of lightest weight crossing $C$. Then there exists an MST of $G$ containing $e$. Furthermore, if $e$ is the unique lightest edge, then all MSTs contain $e$. 
Dynamic Programming (Ch 6)

• Background and past examples
• Longest Common Subsequence
• Knapsack
• Chain Matrix Multiplication
• All-Pairs Shortest Paths
• Maximum Independent Sets of Trees
• Travelling Salesman
Dynamic Programming

Our final general algorithmic technique:
1. Break problem into smaller subproblems.
2. Find recursive formula solving one subproblem in terms of simpler ones.
3. Tabulate answers and solve all subproblems.
Longest Common Subsequence

We say that a sequence is a common subsequence of two others, if it is a subsequence of both.

**Problem:** Given two sequences compute the longest common subsequence. That is the subsequence with as many letters as possible.
Case Analysis

How do we compute $\text{LCSS}(A_1A_2...A_n, B_1B_2...B_m)$?

Consider cases for the common subsequence:
1. It does not use $A_n$.
2. It does not use $B_m$.
3. It uses both $A_n$ and $B_m$ and these characters are the same.
Recursion

On the other hand, the longest common subsequence must come from one of these cases. In particular, it will always be the one that gives the biggest result.

\[
\text{LCSS}(A_1A_2...A_n, B_1B_2...B_m) = \\
\text{Max}(\text{LCSS}(A_1A_2...A_{n-1}, B_1B_2...B_m), \\
\text{LCSS}(A_1A_2...A_n, B_1B_2...B_{m-1}), \\
[\text{LCSS}(A_1A_2...A_{n-1}, B_1B_2...B_{m-1})+1])
\]

[where the last option is only allowed if \(A_n = B_m\)]
Recursion

Key Point: Subproblem reuse
Only ever see $\text{LCSS}(A_1A_2...A_k, B_1B_2...B_\ell)$
Base Case

Our recursion also needs a base case.

In this case we have:

\[ \text{LCSS}(\emptyset, B_1B_2...B_m) = \text{LCSS}(A_1A_2...A_n, \emptyset) = 0. \]
Algorithm

LCSS(A_1 A_2 ... A_n, B_1 B_2 ... B_m)

Initialize Array T[0...n, 0...m]

\[ T[i,j] \text{ will store } \text{LCSS}(A_1 A_2 ... A_i, B_1 B_2 ... B_j) \]

For i = 0 to n

For j = 0 to m

If (i = 0) OR (j = 0)

\[ T[i,j] \leftarrow 0 \]

Else If A_i = B_j

\[ T[i,j] \leftarrow \max(T[i-1,j], T[i,j-1], T[i-1,j-1]+1) \]

Else

\[ T[i,j] \leftarrow \max(T[i-1,j], T[i,j-1]) \]

Return T[n,m]
Example

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>∅</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>∅</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|   | A | B | A | C |
|---|---|---|---|
| ∅|   |   |   |

|   | A | B | A |
|---|---|---|
| ∅|   |   |

|   | A |
|---|
| ∅|   |

String: ABCA
Notes about DP

• General Correct Proof Outline:
  – Prove by induction that each table entry is filled out correctly
  – Use base-case and recursion

• Runtime of DP:
  – Usually
    [Number of subproblems]x[Time per subproblem]
More Notes about DP

• Finding Recursion
  – Often look at first or last choice and see what things look like without that choice

• Key point: Picking right subproblem
  – Enough information stored to allow recursion
  – Not too many
Knapsack

You have an available list of items. Each has a (non-negative integer) weight, and value. Your sack also has a capacity.

The goal is to find the collection of items so that:

1. The total weight of all the items is less than the capacity
2. Subject to 1, the total value is as large as possible.
Variations

There are two slight variations of this problem:
1. Each item can be taken as many times as you want.
2. Each item can be taken at most once.
Recursion

What is BestValue(C)?

Possibilities:

• No items in bag
  – Value = 0

• Item i in bag
  – Value = BestValue(C-weight(i)) + value(i)

Recursion: $\text{BestValue}(C) = \max(0, \max_{\text{wt}(i) \leq C} (\text{val}(i) + \text{BestValue}(C-\text{wt}(i))))$
Algorithm

Knapsack(Wt, Val, Cap)
    Create Array T[0...Cap]
    For C = 0 to Cap
        T[C] ← 0
        For items i with Wt(i) ≤ C
            If T[C] < Val(i) + T[C - Wt(i)]
                T[C] ← Val(i) + T[C - Wt(i)]
    Return T[Cap]

Runtime:
O([Cap] [#Items])
Non Repeating Items

BestValue\(_{\leq k}(\text{Cap})\) = Highest total value of items with total weight at most Cap using only items from the first k.

**Base Case:** BestValue\(_{\leq 0}(\text{C})\) = 0

**Recursion:** BestValue\(_{\leq k}(\text{C})\) is the maximum of

1. BestValue\(_{\leq k-1}(\text{C})\)

2. BestValue\(_{\leq k-1}(\text{C} - \text{Wt}(k)) + \text{Val}(k)\)
   [where this is only used if Wt(k) ≤ Cap]
Runtime

- Number of Subproblems: $O([\text{Cap}] [\#\text{items}])$
- Time per subproblem $O(1)$
  - Only need to compare two options.
- Final runtime $O([\text{Cap}][\#\text{items}])$. 
Chain Matrix Multiplication

**Problem:** Find the order to multiply matrices $A_1$, $A_2$, $A_3$, ..., $A_m$ that requires the fewest total operations.

Multiplying $(n \times m)$ by $(m \times k)$ takes $nmk$ time.

Assume $A_1$ is an $n_0 \times n_1$ matrix, $A_2$ is $n_1 \times n_2$, generally $A_k$ is an $n_{k-1} \times n_k$ matrix.
Recursion

• We need to find a recursive formulation.
• Often we do this by considering the last step.
• For some value of k, last step:
  \((A_1A_2\ldots A_k)(A_{k+1}A_{k+2}\ldots A_m)\)
• Number of steps:
  – \(CMM(A_1,A_2,\ldots,A_k)\) to compute first product
  – \(CMM(A_{k+1},\ldots,A_m)\) to compute second product
  – \(n_0n_kn_m\) to do final multiply
• Recursion \(CMM(A_1,\ldots,A_m) = \min_k[CMM(A_1,\ldots,A_k)+CMM(A_{k+1},\ldots,A_m)+n_0n_kn_m]\)
Subproblems

• Only need subproblems \( C(i,j) = CMM(A_i,A_{i+1},...,A_j) \) for \( 1 \leq i \leq j \leq m \).
  – Fewer than \( m^2 \) total subproblems.
  – Critical: Subproblem reuse.
Full Recursion

**Base Case:** $C(i,i) = 0$.

(With a single matrix, we don’t have to do anything)

**Recursive Step:**

$$C(i,j) = \min_{i \leq k < j} [C(i,k)+C(k+1,j)+n_in_kn_j]$$

**Solution order:** Solve subproblems with smaller $j-i$ first. This ensures that the recursive calls will already be in your table.
Runtime

**Number of Subproblems:** One for each $1 \leq i \leq j \leq m$. Total: $O(m^2)$.

**Time per Subproblem:** Need to check each $i \leq k < j$. Each check takes constant time. $O(m)$.

**Final Runtime:** $O(m^3)$
Problem: Given a graph $G$ with (possibly negative) edge weights, compute the length of the shortest path between every pair of vertices.

Note: Bellman–Ford computes single-source shortest paths. Namely, for some fixed vertex $s$ it computes all of the shortest paths lengths $d(s,v)$ for every $v$. 
Repeated Bellman-Ford

**Easy Algorithm:** Run Bellman-Ford with source s for each vertex s.

**Runtime:** $O(|V|^2|E|)$
Dynamic Program

• Let $d_k(u,v)$ be the length of the shortest $u$-$v$ path using at most $k$ edges.

• Consider last edge.
  • Length $k-1$ path from $u$ to $w$, edge from $w$ to $v$.
  • $d_k(u,v) = \min_w[d_{k-1}(u,w) + \ell(w,v)]$
Matrix Multiplication Method

• Bellman-Ford is slow in part because we can only increase \( k \) by one step at a time.

• This happens because we cut off only the last edge of the optimal path.

• What if instead we cut it in the middle?
Recursion

\[ d_{2k}(u, v) = \min_{w \in V} (d_k(u, w) + d_k(w, v)) . \]
Algorithm

**Base Case:**

\[d_1(u, v) = \begin{cases} 
0 & \text{if } u = v \\
\ell(u, v) & \text{if } (u, v) \in E \\
\infty & \text{otherwise}
\end{cases}\]

**Recursion:** Given \(d_k(u,v)\) for all \(u, v\) compute \(d_{2k}(u,v)\) using

\[d_{2k}(u, v) = \min_{w \in V} (d_k(u, w) + d_k(w, v)).\]

**End Condition:** Compute \(d_1, d_2, d_4, \ldots, d_m\) with \(m > |V|\).

*Runtime: \(O(|V|^3 \log |V|)\)*
Floyd-Warshall Algorithm

• Label vertices \( v_1, v_2, \ldots, v_n \).
• Let \( d_k(u,w) \) be the length of the shortest \( u \)-\( w \) path using only \( v_1, v_2, \ldots, v_k \) as intermediate vertices.
• **Base Case:**

\[
d_0(u, w) = \begin{cases} 
0 & \text{if } u = w \\
\ell(u, w) & \text{if } (u, w) \in E \\
\infty & \text{otherwise}
\end{cases}
\]
Recursion

Break into cases based on whether shortest path uses $v_k$.

- The shortest path not using $v_k$ has length $d_{k-1}(u,w)$.
- The shortest path using $v_k$ has length $d_{k-1}(u,v_k) + d_{k-1}(v_k,w)$.
Algorithm

**Base Case:**

\[ d_0(u, w) = \begin{cases} 
0 & \text{if } u = w \\
\ell(u, w) & \text{if } (u, w) \in E \\
\infty & \text{otherwise}
\end{cases} \]

**Recursion:** For each \( u, w \) compute:

\[ d_k(u, w) = \min(d_{k-1}(u, w), d_{k-1}(u, v_k) + d_{k-1}(v_k, w)) \]

**End Condition:** \( d(u, w) = d_n(u, w) \) where \( n = |V| \).

**Runtime:** \( O(|V|^3) \)
Maximum Independent Set

**Definition:** In an undirected graph G, an *independent set* is a subset of the vertices of G, no two of which are connected by an edge.

**Problem:** Given a graph G compute the largest possible size of an independent set of G.

Call answer I(G).
Simple Recursion

Is vertex $v$ in the independent set?

**If not:** Maximum independent set is an independent set of $G-v$.
$I(G) = I(G-v)$.

**If so:** Maximum independent set is $v$ plus an independent set of $G-N(v)$.
$I(G) = 1 + I(G-N(v))$.

**Recursion:** $I(G) = \max(I(G-v), 1 + I(G-N(v)))$
**Lemma:** If G has connected components $C_1, C_2, ..., C_k$ then

$$I(G) = I(C_1) + I(C_2) + ... + I(C_k).$$

**Proof:** An independent set for G is exactly the union of an independent set for each of the $C_i$. Can pick the biggest set for each $C_i$. 
Recursion For Trees

**Root not used:**
\[ I(G) = \sum I(\text{children’s subtrees}) \]

**Root is used:**
\[ I(G) = 1 + \sum I(\text{grandchildren’s subtrees}) \]
Travelling Salesman

**Problem:** Given a weighted (undirected) graph $G$ with $n$ vertices find a cycle that visits each vertex exactly once whose total weight is as small as possible.
Naïve Algorithm

• Try all possible paths and see which is cheapest.

• How many paths?
  – n possible options for first city.
  – (n-1) possible options for second city.
  – (n-2) for third city
  – ...
  – Total n!

• Runtime ≈ n!
Dynamic Program

Best_{st,L}(G) = Best s-t path that uses exactly the vertices in L.

• Last edge is some (v,t) ∈ E for some v ∈ L.
• Cost is Best_{sv,L-t}(G) + ℓ(v,t).

Full Recursion:

Best_{st,L}(G) = \min_v [Best_{sv,L-t}(G) + ℓ(v,t)].
Runtime Analysis

Number of Subproblems:
L can be any subset of vertices (2^n possibilities) 
s and t can be any vertices (n^2 possibilities) 
n^22^n total.

Time per Subproblem:
Need to check every v (O(n) time).

Final Runtime:
O(n^32^n)
[can improve to O(n^22^n) with a bit of thought]