CSE 101 Exam 2 Review
Shortest Paths (Ch 4)

- BFS
- Dijkstra
  - Priority Queues
- Bellman-Ford
- Shortest Paths in DAGs
Fundamental Shortest Paths Formula

For $w \neq s$,
\[
\text{dist}(w) = \min_{(v,w) \in E} \text{dist}(v) + \ell(v, w).
\]
**Observation**

If there is a length $\leq d$ s-v path, then there is some w adjacent to v with a length $\leq (d-1)$ s-w path.

**Proof:** w is the next to last vertex on the path. This means that if we know all of the vertices at distance $\leq (d-1)$, we can find all of the vertices at distance $\leq d$. 
BFS(G,s)

For v ∈ V, dist(v) ← ∞
Initialize Queue Q
Q.enqueue(s)
dist(s) ← 0

While (Q nonempty)
    u ← front(Q)
    For (u,v) ∈ E
        If dist(v) = ∞
            dist(v) ← dist(u) + 1
            Q.enqueue(v)
            v.prev ← u

Total runtime: O(|V| + |E|)
Edge Lengths

The number of edges in a path is not always the right measure of distance. Sometimes, taking several shorter steps is preferable to taking a few longer ones.

We assign each edge \((u,v)\) a non-negative length \(\ell(u,v)\). The length of a path is the sum of the lengths of its edges.
Problem: Shortest Paths

**Problem:** Given a Graph $G$ with vertices $s$ and $t$ and a length function $\ell$, find the shortest path from $s$ to $t$. 
Algorithm

Distances(G,s,ℓ)

\[ \text{dist}(s) \leftarrow 0 \]

\[ \text{While(not all distances found)} \]

\[ \text{Find minimum over } (v,w) \in E \text{ with } v \text{ discovered } w \text{ not } \]

\[ \text{of } \text{dist}(v)+ℓ(v,w) \]

\[ \text{dist}(w) \leftarrow \text{dist}(v)+ℓ(v,w) \]

\[ \text{prev}(w) \leftarrow v \]
Why does this work?

**Claim:** Whenever the algorithm assigns a distance to a vertex $v$ that is the length of the shortest path from $s$ to $v$.

**Proof by Induction:**
- $\text{dist}(s) = 0$ [the empty path has length 0]
- When assigning distance to $w$, assume that all previously assigned distances are correct.
Inductive Step

Correctly Assigned Distances

This is the shortest path from s to any vertex outside the bubble.
Priority Queue

A **Priority Queue** is a datastructure that stores elements sorted by a **key** value.

**Operations:**

- **Insert** – adds a new element to the PQ.
- **DecreaseKey** – Changes the key of an element of the PQ to a specified *smaller* value.
- **DeleteMin** – Finds the element with the smallest key and removes it from the PQ.
Dijkstra(G, s, ℓ)

Initialize Priority Queue Q
For v ∈ V
  dist(v) ← ∞
  Q.Insert(v)

dist(s) ← 0

While(Q not empty)
  v ← Q.DeleteMin()
  For (v, w) ∈ E
    If dist(v) + ℓ(v, w) < dist(w)
      dist(w) ← dist(v) + ℓ(v, w)
      Q.DecreaseKey(w)

Runtime:
O(|V|) Inserts +
O(|V|) DeleteMins +
O(|E|) DecreaseKeys
Binary Heap

Store elements in a balanced binary tree with each element having smaller key than its children.

$log(n)$ levels

Smallest Key
Operations

• Insert
  – Add new element at bottom
  – Bubble up
• DecreaseKey
  – Change key
  – Bubble up
• DeleteMin
  – Remove root
  – Move bottom to root
  – Bubble down

Runtime
O(log(n)) per operation
d-ary Heap

- Like binary heap, but each node has d children
- Only \( \frac{\log(n)}{\log(d)} \) levels.
- Bubble up faster!
- Bubble down slower – need to compare to more children.
# Summary of Priority Queues

<table>
<thead>
<tr>
<th></th>
<th>Insert/DecreaseKey</th>
<th>DeleteMin</th>
<th>Dijkstra</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>$O(\log(n))$</td>
<td>$O(\log(n))$</td>
<td>$O(\log</td>
</tr>
<tr>
<td>$d$-ary Heap</td>
<td>$O\left(\frac{\log(n)}{\log(d)}\right)$</td>
<td>$O\left(\frac{d \log(n)}{\log(d)}\right)$</td>
<td>$O\left(\frac{\log</td>
</tr>
<tr>
<td>Fibonacci Heap</td>
<td>$O(1)^*$</td>
<td>$O(\log(n))^*$</td>
<td>$O(</td>
</tr>
</tbody>
</table>
Negative Edge Weights

• Unusual case, but some applications
• Dijkstra no longer works
  – Sometimes need to be able to look ahead to find savings
Negative Weight Cycles

**Definition:** A negative weight cycle is a cycle where the total weight of edges is negative.

- If G has a negative weight cycle, then there are probably no shortest paths.
  - Go around the cycle over and over.

- **Note:** For undirected G, a single negative weight edge gives a negative weight cycle by going back and forth on it.
Fundamental Shortest Paths Formula

For \( w \neq s \),

\[
\text{dist}(w) = \min_{(v,w) \in E} \text{dist}(v) + \ell(v,w).
\]

- System of equations to solve for distances.
- When \( \ell \geq 0 \), Dijkstra gives an order to solve in.
- With \( \ell < 0 \), might be no solution.
Algorithm Idea

Instead of finding shortest paths (which may not exist), find shortest paths of length at most $k$.

For $w \neq s$,

$$\text{dist}_k(w) = \min_{(v,w) \in E} \text{dist}_{k-1}(v) + \ell(v, w).$$
Algorithm

Bellman-Ford\((G, s, \ell)\)
\[ \text{dist}_0(v) \leftarrow \infty \text{ for all } v \]
// cant reach
\[ \text{dist}_0(s) \leftarrow 0 \]
For \( k = 1 \) to \( n \)
\[
\text{For } w \in V
\begin{cases}
\text{dist}_k(w) \leftarrow \min(\text{dist}_{k-1}(v) + \ell(v,w)) \\
\text{dist}_k(s) \leftarrow \min(\text{dist}_k(s), 0)
\end{cases}
// s has the trivial path
\]
\( O(|E|) \)

What value of \( k \) do we use?
Analysis

**Proposition:** If \( n \geq |V|-1 \) and if \( G \) has no negative weight cycles, then for all \( v \),
\[
\text{dist}(v) = \text{dist}_n(v).
\]

- If there is a negative weight cycle, there probably is no shortest path.
- If not, we only need to run our algorithm for \(|V|\) rounds, for a final runtime \( O(|V| |E|) \).
Proof

• If we have a path with $|V|$ edges:
  – Must repeat some vertex
  – Contains a loop
  – Removing loop makes path shorter

• Thus, there is a shortest path with at most $|V|-1$ edges.
Proposition: For any \( n \geq |V| - 1 \), there are no negative weight cycles reachable from \( s \) if and only if for every \( v \in V \)

\[
\text{dist}_n(v) = \text{dist}_{n+1}(v)
\]

- Detect by running one more round of Bellman-Ford.
- Need to see if \textit{any} \( v \)'s distance changes.
Proof of “Only If”

• Suppose no negative weight cycles.
• For any $n \geq |V| - 1$, $\text{dist}_n(v) = \text{dist}(v)$.
• So

\[ \text{dist}_n(v) = \text{dist}(v) = \text{dist}_{n+1}(v) \]
Proof of “If”

Suppose \( \text{dist}_n(v) = \text{dist}_{n+1}(v) \) for all \( v \).

\[
\begin{align*}
\text{dist}_{n+2}(w) &= \min_{(v,w) \in E} (\text{dist}_{n+1}(v) + \ell(v, w)) \\
&= \min_{(v,w) \in E} (\text{dist}_n(v) + \ell(v, w)) \\
&= \text{dist}_{n+1}(w).
\end{align*}
\]

So

\[
\text{dist}_n(v) = \text{dist}_{n+1}(v) = \text{dist}_{n+2}(v) = \text{dist}_{n+3}(v) = \ldots
\]

But if there were a negative weight cycle, distances would decrease eventually.
Algorithm

ShortestPathsInDAGs(G, s, ℓ)

TopologicalSort(G)

For w ∈ V in topological order
  If w = s, dist(w) ← 0
  Else
    dist(w) ← min(dist(v) + ℓ(v, w))

\ \ \ dist(v) for all upstream v already computed

Runtime
O(|V| + |E|)
Divide & Conquer (Ch 2)

• General Technique
• Master Theorem
• Karatsuba Multiplication
• Merge Sort
• Order Statistics
• Binary Search
• Closest Pair of Points
Divide and Conquer

This is the first of our three major algorithmic techniques.

1. Break problem into pieces
2. Solve pieces recursively
3. Recombine pieces to get answer
Master Theorem

**Theorem:** Let \( T(n) \) be given by the recurrence:

\[
T(n) = \begin{cases} 
O(1) & \text{if } n = O(1) \\
aT(n/b + O(1)) + O(n^d) & \text{otherwise}
\end{cases}
\]

Then we have that

\[
T(n) = \begin{cases} 
O(n^{\log_b(a)}) & \text{if } a > b^d \\
O(n^d \log(n)) & \text{if } a = b^d \\
O(n^d) & \text{if } a < b^d
\end{cases}
\]
Karatsuba Multiplication

Want to multiply N and M:

1. Let \( X \approx \sqrt{(N+M)} \) be a power of 2.
2. Write \( N = AX+B, \ M = CX+D \)
   - This can be done by just taking the high and low bits.
3. \( N \cdot M = AC \cdot X^2 + (AD+BC)X + BD \)
   = \( AC \cdot X^2 + [(A+B)(C+D)-AC-BD]X + BD \)
   - The multiplications by \( X \) are just bit shifts.
Algorithm

KaratsubaMult\((N, M)\)

If \(N+M<99\), Return Product\((N, M)\)

Let \(X\) be a power of \(2^{\lceil \log(N+M)/2 \rceil}\)

Write \(N = AX + B, M = CX + D\)

\(P_1 \leftarrow \text{KaratsubaMult}(A,C)\)

\(P_2 \leftarrow \text{KaratsubaMult}(B,D)\)

\(P_3 \leftarrow \text{KaratsubaMult}(A+B,C+D)\)

Return \(P_1X^2 + [P_3-P_1-P_2]X + P_3\)
Runtime Recurrence

Karatsuba multiplication on inputs of size $n$ spends $O(n)$ time, and then makes three recursive calls to problems of (approximately) half the size.

If $T(n)$ is the runtime for $n$-bit inputs, we have the recursion:

$$T(n) = \begin{cases} 
O(1) & \text{if } n = O(1) \\
3T(n/2 + O(1)) + O(n) & \text{otherwise}
\end{cases}$$

Master Theorem: $T(n) = O(n^{\log_2(3)})$. 
Note

In divide and conquer, it is important that the recursive subcalls are a constant fraction of the size of the original.
Note on Proving Correctness

There’s a general procedure for proving correctness of a D&C algorithm:

**Use Induction:** Prove correctness by induction on problem size.

**Base Case:** Your base case will be the non-recursive case of your algorithm (which your algorithm does need to have).

**Inductive Step:** Assuming that the (smaller) recursive calls are correct, show that algorithm works.
Sorting

**Problem:** Given a list of n numbers, return those numbers in ascending order.
Divide and Conquer

How do we make a divide and conquer algorithm?

1. Split into Subproblems   - Split list in two
2. Recursively Solve       - Sort each sublist
3. Combine Answers         - ???
Merge

**Problem:** Given two sorted lists, combine them into a single sorted list.
Merge

Merge(A,B)
  C ← List of length Len(A)+Len(B)
  a ← 1, b ← 1
  For c = 1 to Len(C)
    If (b > Len(B))
      C[c] ← A[a], a++
    Else if (a > Len(A))
      C[c] ← B[b], b++
    Else if A[a] < B[b]
      C[c] ← A[a], a++
    Else
      C[c] ← B[b], b++
  Return C

Runtime: O(|A|+|B|)
MergeSort

MergeSort(L)
   If Len(L) = 1 \ Base Case
       Return L
   Split L into equal L_1 and L_2
   A ← MergeSort(L_1)
   B ← MergeSort(L_2)
   Return Merge(A,B)

Runtime
T(n) = 2T(n/2)+O(n)
T(n) = O(n \log(n))
Order Statistics

**Problem:** Given a list \( L \) of numbers and an integer \( k \), find the \( k \)th largest element of \( L \).
Divide Step

Select a pivot $x \in L$. Compare it to the other elements of $L$.

\[
\begin{array}{ccc}
< x & = x & > x \\
\end{array}
\]

Which list is our answer in?

- Answer is $> x$ if there are $\geq k$ elements bigger than $x$.
- Answer is $x$ if there are $< k$ elements bigger and $\geq k$ elements bigger than or equal to $x$.
- Otherwise answer is less than $x$. 
Order Statistics

Select(L, k)

Pick x ∈ L
Sort L into L_{>x}, L_{<x}, L_{=x}
If Len(L_{>x}) ≥ k
    Return Select(L_{>x}, k)
Else if Len(L_{>x})+Len(L_{=x}) ≥ k
    Return x
Return
Select(L_{<x}, k-Len(L_{>x})-Len(L_{=x}))
**Randomization**

**Fix:** Pick a *random* pivot.

- There’s a 50% chance that $x$ is selected in the middle half.
- If so, no matter where the answer is, recursive call of size at most $3n/4$.
- On average need two tries to reduce call.
Runtime

$T(n) = T(3n/4) + O(n)$

So...

$T(n) = O(n)$
Note

The algorithm discussed does give the correct answer in *expected* $O(n)$ time.

There are deterministic $O(n)$ algorithms using similar ideas, but they are substantially more complicated.
Search

Problem: Given a sorted list L and a number x, find the location of x in L.
Binary Search

BinarySearch(L, i, j, x)
Search between L[i] and L[j]
If j < i, Return 'error'
k ← [(i+j)/2]
If L[k] = x, Return k
If L[k] > x
    Return BinarySearch(L, i, k-1, x)
If L[k] < x
    Return BinarySearch(L, k+1, j, x)
Runtime

\[ T(n) = T(n/2) + O(1) \]

So...

\[ T(n) = O(\log(n)) \]
Binary Search Puzzles

You have 27 coins one of which is heavier than the others, and a balance. Determine the heavy coin in 3 weightings.

Lots of puzzles have binary search-like answers. As long as you can spend constant time to divide your search space in half (or thirds). You can use binary search in $O(\log(n))$ time.
Problem: Given $n$ points in the plane $(x_1, y_1) \ldots (x_n, y_n)$ find the pair $(x_i, y_i)$ and $(x_j, y_j)$ whose Euclidean distance is as small as possible.
Divide and Conquer Outline

• Divide points into two sets by drawing a line.
• Compute closest pair on each side.
• What about pairs that cross the divide?
Observation

• Suppose closest pair on either side at distance $\delta$.
• Only need to care about points within $\delta$ of dividing line.
• Need to know if some pair closer than $\delta$. 
Main Idea

**Proposition:** Take the points within $\delta$ of the dividing line and sort them by $y$-coordinate. Any one of these points can only be within $\delta$ of the 8 closest points on either side of it.
Proof

- Nearby points must have y-coordinate within $\delta$.
- Subdivide region into $\delta/2$-sided squares.
- At most one point in each square.
Algorithm

CPP(S)

If $|S| \leq 3$

    Return closest distance

Find line $L$ evenly dividing points

Sort $S$ into $S_{\text{left}}, S_{\text{right}}$

$\delta \leftarrow \min(\text{CPP}(S_{\text{left}}), \text{CPP}(S_{\text{right}}))$

Let $T$ be points within $\delta$ of $L$

Sort $T$ by y-coordinate

Compare each element of $T$ to 8 closest on either side. Let $\text{min dist}$ be $\delta'$.

Return $\min(\delta, \delta')$
Runtime

We have a recurrence

\[ T(n) = O(n \log(n)) + 2 \, T(n/2). \]

This is not quite covered by the Master Theorem, but can be shown to give

\[ T(n) = O(n \log^2(n)). \]

Alternatively, if you are more careful and have \texttt{CPP} take points already sorted by y-coordinate, you can reduce to \( O(n \log(n)) \).