CSE 250
Lecture 37
Final Review
Day 1
Logarithms
Logarithms (refresher)

- Let $a, b, c, n > 0$
- Exponent rule: $\log(n^a) = a \log(n)$
- Product rule: $\log(an) = \log(a) + \log(n)$
- Division rule: $\log\left(\frac{n}{a}\right) = \log(n) - \log(a)$
- Change of base from b to c: $\log_b(n) = \frac{\log_c(n)}{\log_c(b)}$
  - Base changes are only a constant factor off
- Log/Exponent are inverses: $b^{\log_b(n)} = \log_b(b^n) = n$
Asymptotic Analysis
Growth Functions

A growth function must be a non-decreasing function of the form

\[ f : \mathbb{Z}^+ \cup \{0\} \rightarrow \mathbb{R}^+ \]

- \( f \) is a function from ...
- \( \mathbb{Z}^+ \cup \{0\} = \{0, 1, 2, 3, \ldots\} \) (non-negative integers)
- \( \mathbb{R}^+ = \{ x \mid x \in \mathbb{R}, x > 0 \} \) (positive real numbers)
- ... to ...
Classify Functions by their Scaling

- Functions that grow faster
- Functions that grow "at the same rate"
- Functions that grow slower
Big-$\Theta$

$\Theta(g)$ is the set of functions where $f \overset{\text{=}}{\sim} g$
Big-O

$O(g)$ is the set of functions where $f \preceq g$
Big-$\Omega$

$\Omega(g)$ is the set of functions where $f \geq g$
Types of Bounds

- **[no qualifier] Runtime**: The **guaranteed** runtime of the function
  - $O(g(n))$: The algorithm never runs slower than $c \cdot g(n)$
  - $\Omega(g(n))$: The algorithm never runs faster than $c \cdot g(n)$
  - $\Theta(g(n))$: The algorithm always runs within $[a \cdot g(n), b \cdot g(n)]$

- **Amortized Runtime**: Guaranteed per-call runtime over n calls
  - $O(g(n))$: n invocations of the algorithm take at most $c \cdot n \cdot g(n)$

- **Expected Runtime**: ‘Typical’ runtime without guarantees
  - $O(g(n))$: The algorithm usually takes no more than $c \cdot g(n)$
    - ... but it’s random, it could take longer if you’re unlucky.
Runtime Terminology

- “Worst-case” runtime
  - The $O()$ runtime of the function
- “Tight” runtime
  - A bound ($O$ or $\Omega$) with no better bound of the same type.
    - Remember that $n = O(n^2)$ (although it’s not tight)
  - A $\Theta$ bound is always tight.
Big-O

- Big-O (big oh) is an upper-bound on functions for any two functions $f, g : \mathbb{Z}^+ \cup \{0\} \to \mathbb{R}^+$

$$f(n) \in O(g(n)) \iff \exists c, n_0 \text{ s.t. } \forall n \geq n_0, f(n) \leq cg(n)$$

$O(g(n))$ is a set of functions and $f(n)$ is in it if (and only if) there's some constant $c$ and some “low” $n$ value $n_0$

$f(n)$ is lower than $g(n)$ scaled by $c$... where for every $n$ bigger than $n_0$
Big-Ω

- Big-Ω (big omega) is a lower-bound on functions for any two functions \( f, g : \mathbb{Z}^+ \cup \{0\} \to \mathbb{R}^+ \)

\[ f(n) \in \Omega(g(n)) \iff \exists c, n_0 \text{ s.t. } \forall n \geq n_0, f(n) \geq c g(n) \]

\( \Omega(g(n)) \) is a set of functions and \( f(n) \) is in it if (and only if) there’s some constant \( c \) and some “low” \( n \) value \( n_0 \) and \( f(n) \) is greater than \( g(n) \) scaled by \( c \) ... where for every \( n \) bigger than \( n_0 \).
Big-Θ

- Big-Θ (big theta) is a joint bound on functions for any two functions \( f, g : \mathbb{Z}^+ \cup \{0\} \rightarrow \mathbb{R}^+ \)

\[ f(n) \in \Theta(g(n)) \iff [f(n) \in O(g(n))] \land [f(n) \in \Omega(g(n))] \]

\( \Theta(g(n)) \) is a set of functions and \( f(n) \) is in it if (and only if) \( f(n) \) is upper-bounded by \( g(n) \) and \( f(n) \) is also lower-bounded by \( g(n) \)
Dominant Terms

exponential $\gg$ polynomial $\gg$ log $\gg$ constant
Common Runtimes

- **Constant Time**: $\Theta(1)$
  - e.g., $T(n) = c$ (for some constant $c > 0$)
- **Logarithmic Time**: $\Theta(\log(n))$
  - e.g., $T(n) = c\log(n)$ (for some constant $c > 0$)
- **Linear Time**: $\Theta(n)$
  - e.g., $T(n) = c_1 n + c_0$ (for some constants $c_1, c_0$ where $c_1 > 0$)
- **Quadratic Time**: $\Theta(n^2)$
  - e.g., $T(n) = c_2 n^2 + c_1 n + c_0$
- **Polynomial Time**: $\Theta(n^k)$ (for some $k \in \mathbb{Z}^+$)
  - e.g., $T(n) = c_k n^k + \ldots + c_2 n^2 + c_1 n + c_0$
- **Exponential Time**: $\Theta(c^n)$ (for some $c > 0$)
Indexing into a Linked List

- Runtime to retrieve the $i$th element is linear in $i$
  - $O(i)$ is a tight bound: $i \leq O(i)$
  - $O(i^2)$ is a bound; $i \leq O(i^2)$ (but not a tight one)
  - $\Omega(i)$ is a tight bound: $i \geq \Omega(i)$
  - Since the runtime is $O(i)$ and $\Omega(i)$, it is also $\Theta(i)$
Appending to an ArrayBuffer

• Runtime is either constant [typical case] OR linear [if resizing]
  - $O(n)$ is a tight bound: $1 \leq O(n), n \leq O(n)$
  - $\Omega(1)$ is a tight bound: $1 \geq \Omega(1), n \geq \Omega(1)$
  - There is no $\Theta$ bound (the tight $O$ bound $\neq$ the tight $\Omega$ bound)
• Runtime of $n$ appends is provably $O(n)$ (and $\Theta(n)$, $\Omega(n)$)
  - Amortized runtime of $\frac{O(n)}{n} = O(1)$
Observation

- The only time when tight bounds $O(f) \neq \Omega(f)$ is when $f$ is
  - ...defined by cases.
  - as in appending to an array buffer
  - ...has variable runtimes
  - e.g., indexing into a linked list is $O(n)$, but $\Theta(i)$
Quick Sort

- Each level of splits takes $O(n)$ total runtime
  - Typically, each split will cut the input array in (nearly) half
    - Will need $\log(n)$ levels of splits
  - **No guarantees**: Unlikely, but might accidentally always pick the lowest value as a pivot for each split.
    - Might need as many as $n$ levels of splits
- **Runtime**: $O(n^2)$
- **Expected Runtime**: $O(n \cdot \log(n))$
Sequences
Immutable Sequence ADTs

- apply(idx: Int): A
  - Get the element (of type A) at position idx.
- iterator: Iterator[A]
  - Get access to view all elements in the seq, in order, once.
- length: Int
  - Count the number of elements in the seq.
Mutable Sequence ADTs

- **apply**($\text{idx}$: Int): A
  - Get the element (of type A) at position $\text{idx}$.

- **iterator**: Iterator[A]
  - Get access to view all elements in the seq, in order, once.

- **length**: Int
  - Count the number of elements in the seq.

- **insert**($\text{idx}$: Int, $\text{elem}$: A): Unit
  - Insert the element at position $\text{idx}$ with the value $\text{elem}$.

- **remove**($\text{idx}$: Int): Unit
  - Remove the element at position $\text{idx}$.
Runtime Cost for Appends

- $T(n) = \text{insert cost} + \text{reserve cost} = \Theta(n) + \Theta(n) = \Theta(n)$
- Append runtime is **Amortized** $O(1)$
  - Runtime for one append is $O(n)$
  - Runtime for $n$ appends is $\Theta(n)$
- “Amortized” describes runtime over the long run.
  - reserve is only called $\log(n)$ times (very infrequently)
  - Not quite the same as the “average” case
    - Average case is the expected runtime over any input
    - Here, $\Theta(n)$ is the runtime.

**Amortized → Upfront costs paid off over time**
## Overview

<table>
<thead>
<tr>
<th>Function</th>
<th>Array</th>
<th>LL by Index</th>
<th>LL by Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply</td>
<td>$\Theta(1)$</td>
<td>$\Theta(i)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>update</td>
<td>$\Theta(1)$</td>
<td>$\Theta(i)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(n)$</td>
<td>$\Theta(i)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(n)$</td>
<td>$\Theta(i)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>append</td>
<td>Amortized $O(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>
Bubble Sort for Mutable Sequences

1. `def sort(seq: mutable.Seq[Int]): Unit = {
2.   val n = seq.length
3.   for(i ← n – 2 to 0 by -1; j ← i to n)
4.     if(seq(j+1) < seq(j))
5.       val temp = seq(j+1)
6.       seq(j+1) = seq(j)
7.       seq(j) = temp
4. }
5. }

Is the runtime $T(n) = \Theta(n^2)$?
- What is the cost of $\text{seq}(j+1) < \text{seq}(j)$?
- What is the cost of each $\text{seq}(k)$?
Bubble Sort for Immutable Sequences

1. def sort(seq: Seq[Int]): Seq[Int] =
   {
2.   val newSeq = seq.toArray
3.   val n = seq.length
4.   for(i ← n – 2 to 0 by -1; j ← 0 to i)
   {
5.     if(newSeq(j+1) < newSeq(j))
   {
6.       val temp = seq(j+1)
7.       seq(j+1) = seq(j)
8.       seq(j) = temp
   }
9.   return newSeq.toList
   }

Is the runtime $T(n) = \Theta(n^2)$?

- What is the cost of seq.toArray?
- What is the cost of newSeq.toList?
Searching Sequences

1. def indexOf[T](seq: Seq[T], value: T, from: Int): Int = {
2.   for(i ← from 0 until seq.length) {
3.     if(seq(i).equals(value)) { return i }
4.   }
5.   return -1
}

Expected runtime is \(T(n) = \Theta(n)\)

1. def count[T](seq: Seq[T], value: T): Int = {
2.   var count = 0; var i = indexOf(seq, value, 0)
3.   while(i != -1) {
4.     count += 1; indexOf(seq, value, i+1)
5.   }
6.   return count
}

Expected runtime is \(T(n) = \Theta(n)\)
Recursion
Fibonacci Sequence Runtime

The runtime of a recursive function is easiest to represent with a recurrence relation

```scala
def fib(n: Int) = {
  if(n == 0 || n == 1) { n }
  else { fib(n-1) + fib(n-2) }
}
```

\[
T(n) = \begin{cases} 
  \Theta(1) & \text{if } n \leq 1 \\
  T(n - 1) + T(n - 2) + \Theta(1) & \text{otherwise}
\end{cases}
\]

(this specific recurrence has a closed form, but ask on Piazza)
Factorial

```java
def fact(n: Int): Long = {
    if(n <= 0) { 1 }
    else { n * fact(n-1) }
}
```

\[ T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq 0 \\
T(n - 1) + \Theta(1) & \text{otherwise}
\end{cases} \]

**What is the closed form?**

**How much space is used?**
Tail-Recursive Factorial

def fact(n: Int): Long = {
    if(n <= 0) { 1 }
    else { n * fact(n-1) }
}

def fact(n: Int): Long = {
    var total = 1l
    for(i ← 1 to n) {
        total *= i
    }
    return total
}
Divide and Conquer

• Recursive Solutions
  – Solve a problem building from solution(s) to smaller versions of the same problem.

• The Divide and Conquer Strategy
  – **Divide** problem into smaller subproblem(s)
  – **Conquer** subproblem(s) by solving recursively
  – **Combine** solutions to subproblem(s) into final solution
Divide and Conquer

- **Towers of Hanoi**
  - $n = 1$: Move disk directly
  - $n > 1$: Solve $n-1$ subproblem 2 times (Conquer)

- **Factorial**
  - $n = 0$: 1
  - $n > 0$:
    - Compute $(n-1)!$ (Conquer)
    - Multiply by $n$ (Merge)

No real “divide” step in any of these examples.
Merge Sort

- If the sequence has 1 or 0 values: Done!
- If n > 1
  - Divide: “Split” the sequence in half
  - Conquer: Sort each half independently
  - Combine: Merge halves together
Merge Sort Analysis

• Suppose data is a sequence of size n
  - Assume n is a power of 2 to simplify analysis
• Divide: “Split” the sequence in half \( D(n) = \Theta(n) \)
• Conquer: Sort left and right halves \( a = 2, b = 2, c = 1 \)
• Combine: Merge sorted halves together \( C(n) = \Theta(n) \)

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq 1 \\
2 \cdot T\left(\frac{n}{2}\right) + \Theta(n) + \Theta(n) = 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n) & \text{otherwise} 
\end{cases}
\]
Merge Sort: Recursion Tree

There are \( \log(n) \) levels in the tree.

At level \( i \), there are \( 2^i \) tasks, each with runtime \( \Theta\left(\frac{n}{2^i}\right) \).

\[
T(N) = \sum_{i=0}^{\log(n)} \sum_{j=1}^{2^i} \Theta\left(\frac{n}{2^i}\right) \\
= \sum_{i=0}^{\log(n)} (2^i - 1 + 1) \Theta\left(\frac{n}{2^i}\right) \\
= \sum_{i=0}^{\log(n)} 2^i \Theta\left(\frac{n}{2^i}\right) \\
= \sum_{i=0}^{\log(n)} \Theta(n) \\
= (\log(n) - 0 + 1) \Theta(n) \\
= \Theta(n) \log(n) + \Theta(n) \\
= \Theta(n \log(n))
\]
Merge Sort: Inductive Analysis

- Base Case: $n = 1$
  \[ T(n) = \Theta(1) = c' \]
- True for any $n_0 > 1$, $c > c'$
Merge Sort: Inductive Analysis

- Inductive step for step $n > 1$: assume for all $m < n$
  \[ T(m) = c \cdot m \log(m) \]
- Now use that to show \[ T(n) = c \cdot n \log(n) \]

\[
T(n) = T\left(\frac{n}{2}\right) + \Theta(n) \\
\leq 2\left(c \frac{n}{2} \log\left(\frac{n}{2}\right) + \Theta(n)\right) \\
= cn \log(n) - cn \log(2) + \Theta(n) \\
\leq cn \log(n) - cn + \Theta(n) \\
= cn \log(n) - cn + dn \text{ (for some constant } d > 0) \\
\leq cn \log(n) \text{ (as long as } c \geq d)\]
Stacks and Queues
Stacks vs Queues

**Stack**

- `push(item)`
  - Insert at end of list
- `pop`
  - Remove from **end** of list
- `top`
  - Retrieve **end** of list

**Queue**

- `enqueue(item)`
  - Insert at end of list
- `dequeue`
  - Remove from **front** of list
- `front`
  - Retrieve **front** of list
Edge Types

- Directed Edge
  - Ordered pair of vertices \((u, v)\)
  - origin \((u)\) → destination \((v)\)
  - e.g., transmit bandwidth
- Undirected Edge
  - Unordered pair of vertices \((u, v)\)
  - e.g., round-trip latency
- Directed Graph: All edges are directed
- Undirected Graph: All edges are undirected
Terminology

- **Endpoints** (end-vertices) of an edge
  - U, V are the endpoints of a
- Edges **incident** on a vertex
  - a, b, d are incident on V
- **Adjacent** Vertices
  - U, V are adjacent
- **Degree** of a vertex (# of incident edges)
  - X has degree 5
- **Parallel Edges**
  - h, i are parallel
- **Self-Loop**
  - j is a self-loop
- **Simple Graph**
  - A graph without parallel edges or self-loops
Edge List Summary

- addEdge, addVertex: $O(1)$
- removeEdge: $O(1)$
- removeVertex: $O(1) + O(\text{vertex.incidentEdges})$
- vertex.outEdges, vertex.inEdges, vertex.incidentEdges: $O(m)$
  - (total cost to visit all out/in/incident edges)
- vertex.edgeTo: $O(m)$
- Space Used: $O(n+m)$
Add an Adjacency List

class DirectedGraphV3[LV, LE]
{
    def addEdge(orig: Vertex, dest: Vertex, label: LE): Edge =
    {
        val edge = new Edge(label)
        edge._listNode = edges.append(edge)
        orig._outEdges.append(edge)
        dest._inEdges.append(edge)
        return edge
    }

class Vertex(_label: LV){
    val _outEdges: LinkedList[Edge]
    val _inEdges: LinkedList[Edge]
    // ...
}
}
Adjacency List Summary

- addEdge, addVertex: $O(1)$
- removeEdge: $O(1)$
- removeVertex: $O(d_{eg}(vertex))$
- vertex.outEdges: $O(|outEdges|)$ to visit all outEdges
  - Same for vertex.inEdges, vertex.incidentEdges
- vertex.edgeTo: $O(|outEdges|)$
- Space Used: $O(n+m)$
A few more terms...

- A subgraph $S$ of a graph $G$ is a graph where
  - $S$’s vertices are a subset of $G$’s vertices
  - $S$’s edges are a subset of $G$’s edges
- A spanning subgraph of $G$ is a subgraph that contains all of $G$’s vertices
A few more terms…

- A graph is **connected** if there is a path between every pair of vertices.
- A **connected component** is a maximal connected subgraph of $G$.
  - Maximal means you can’t add any new vertex without breaking the property.
  - Any subset of $G$’s edges that connects the subgraph is fine.
A few more terms...

- A spanning tree of a connected graph is a spanning subgraph that is a tree.
  - not unique unless the graph is a tree.
Recall...

- Searching the maze with a Stack  
  - Try out every path, one at a time...
  - ... repeatedly backtrack and try another
- Searching the maze with a Queue  
  - Try out every path in parallel...
  - ... repeatedly pick a path and expand it by one step

Depth-First Search

Breadth-First Search
Depth-First Search

- DFS Marking Vertices UNVISITED: $O(|\text{vertices}|)$
- DFS Marking Edges UNVISITED: $O(|\text{edges}|)$
- DFS Vertex Loop: $O(|\text{vertices}|)$
- All Calls to DFSOne:

$$O\left(\sum_v 1 + \deg(v)\right) = O(|\text{vertices}| + |\text{edges}|)$$

$O(|\text{vertices}| + |\text{edges}|)$
Breadth-First Search

- Primary Goals
  - Visit every vertex in the graph in increasing order of distance from the starting vertex
  - Construct a spanning tree for every connected component
    - Side effect: Compute connected components
    - Side effect: Compute paths between pairs of vertices
    - Side effect: Determine if the graph is connected
    - Side effect: Identify cycles
    - Side effect: Identify shortest paths to the starting vertex
  - Complete in time $O(|\text{vertices}|+|\text{edges}|)$
  - Complete with memory overhead $O(|\text{vertices}|)$
Breadth-First Search

- BFS Marking Vertices UNVISITED: $O(\lvert \text{vertices} \rvert)$
- BFS Marking Edges UNVISITED: $O(\lvert \text{edges} \rvert)$
- BFS Vertex Loop: $O(\lvert \text{vertices} \rvert)$
- All connected components:
  $$O(\sum_v 1 + \deg(v)) = O(\lvert \text{vertices} \rvert + \lvert \text{edges} \rvert) = O(\lvert \text{vertices} \rvert + \lvert \text{edges} \rvert)$$
# DFS vs BFS

<table>
<thead>
<tr>
<th>Application</th>
<th>DFS</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spanning Trees</td>
<td>✅</td>
<td></td>
</tr>
<tr>
<td>Connected Components</td>
<td>✅</td>
<td></td>
</tr>
<tr>
<td>Paths/Connectivity</td>
<td>✅</td>
<td></td>
</tr>
<tr>
<td>Cycles</td>
<td>✅</td>
<td></td>
</tr>
<tr>
<td>Shortest Paths</td>
<td></td>
<td>✅</td>
</tr>
<tr>
<td>Articulation Points</td>
<td></td>
<td>✅</td>
</tr>
</tbody>
</table>