CS483 Midterm Review: Analysis of Algorithms

General Motivation:
Objective of the course: how to design and analyze algorithms
Algorithms: recipes for solving problems, they are realized by programs
Programs: sequences of instructions

One alternative: take a program and count the number of instructions (this may depend on implementation)

We need some measure how to compare them which would be independent of machine, details, and it would depend on the method used to solve the problem

In general given a program the number of instructions will depend on input:
- depends on the size of input (number of input elements n)
- depends on particular input

Comparing algorithms/programs

We want to compare algorithms in the same class:
(trying to solve the same problems, without worrying about details of speed of computer, implementation)

Running time -> Number of instructions as a function of input size n:
Algorithm 1 takes f(n) steps, algorithm 2 takes g(n) steps, we want to compare them -> compare functions f, g, for large n

Order of growth of functions:
Definitions of classes of functions
Upper bound, lower bound, tight bound, not tight upper (lower) bound

\( O(f(n)), \Theta(f(n)), \Omega(f(n)) \)

Asymptotic Efficiency

- Measuring input size
- What is elementary operation
- Definition of asymptotic notations
- Basic Efficiency classes
- Asymptotic running time of some elementary algorithms (sequential search, binary search, brute force algorithms, closest pair, convex hull etc)
- (computing summations - see appendix)

Techniques how to compare functions

Facts: polynomials are dominated by higher order terms
\( n^3 + n^2 - 3n = \Theta(n^3) \)

To show it from definition choose \( c \) and \( n \) such that inequalities hold
\( f(n) = 3n^3, g(n) = 30n \)

Facts: exponential functions grow faster than polynomials
\[ \lim_{n \to \infty} \frac{n^b}{n^a} = 0 \]

Comparing functions using limits (L’Hospital rule):
\[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = a; f(n) = \Theta(g(n)) \quad (1) \]
\[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0; f(n) = O(g(n)) \quad (2) \]
\[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty; f(n) = \Omega(g(n)) \quad (3) \]
**Worst case running-time**

Longest running time on any input
Regardless of the input it will take \textit{at most} \( w(n) \) steps
It is the \textit{upper bound} of the algorithm on any input

\textbf{Lower bound} on the worst-case running time
There is some input that the algorithm will perform \textit{at least} \( f(n) \) steps

Worst case result on a particular array
Best case result on a particular array
If we make statement about the algorithm, it has to be for any input

---

**Design and analysis of algorithms**

Types of algorithms:
- Iterative (multiple loops)
- Recursive algorithms – partition into smaller sub-problems

To establish the running time, we need to solve the recurrent equations.
Techniques for solving recurrences:
1. Expansion (expand and compute (or bound summations)
2. Master’s Theorem (for divide and conquer problems)

Examples of recursive algorithms:
(Merge Sort, Quick Sort, Binary Search, Selection, Tower of Hanoi etc., number of bits in \( n \)'s binary represent.)

---

**Brute force algorithms**

- Selection Sort – worst case running time
- Bubble Sort – worst case running time
- Sequential Search, String Matching
- Closest Pair, Convex Hull problems
- Exhaustive Search – knapsack, traveling salesman, assignment problems
Divide and Conquer strategy
leads to recurrences of special form
\[ T(n) = aT\left(\frac{n}{b}\right) + f(n) \]
- Merge Sort
- Quick Sort
- Binary Search
- Matrix Multiplication
- Closest Pair and Convex Hull

Master’s theorem
\[ T(n) = aT\left(\frac{n}{b}\right) + f(n) \]
Examples:
- \( T(n) = 7T(n/2) + n^2 \) Case 1, matrix multiplication alg
- \( T(n) = 2T(n/2) + n \) Case 2, balanced
- \( T(n) = 4T(n/2) + n^3 \) Case 3 (made up)
  1. cost dominated by root
  2. balanced
  3. cost dominated by leaves

Sorting algorithms

**Merge Sort**
Recursive algorithm split-sort-merge \( \Theta(n \lg n) \)

**Heap Sort**
Combination of recursive and iterative \( \Theta(n \lg n) \)

**Quick Sort** average-time complexity
Recursive algorithm partition-split-sort \( \Theta(n \lg n) \)
Worst-case partitioning (for particular input) \( \Theta(n^2) \)
Best-case partitioning (for particular input)-lower bound
In general it is good to have an upper bound and lower bound on worst-time complexity as close as possible

Decrease and Conquer

- Graph representations
- DFS - (computed times vertices were first and last visited, label edges as tree, back, forward, cross edges), connected component - stack
- BFS - (the same) + their efficiency - queue
- Directed vs undirected graphs
- DAGs - properties of DAGs - no back edges
- Topological Sort (reverse popping # or source removal)
- Generating Permutations (minimal # of operations)
- Generating Subsets
Transform and Conquer

- Pre-sorting
- AVL trees - properties of AVL trees
  - efficiency of insert operation
- How to do rotations
- Heaps and heapsort

Median and order statistics

- Selection problem
  - expected average running time $O(n)$
  - randomized partition
- Min
  - worst case (at most) and best case $n - 1$ $O(n)$
- Max
- Selection problem
  - average case running time
  - quick sort idea $O(n)$

Heapsort

Implicit data structure (correspondence heap - array)

Heap properties

- Loose bound upper bound $O(n \log n)$
- Tighter bound $O(n)$

Build Heap $O(\log n)$

Heapify $O(\log n)$

Heap Sort
- extract max, replace root, Heapify(A,1)

Heapsort is $\Theta(n \log n)$

Need to know how these work
Does it depend on whether array is sorted or not?