Exam 1 Study Guide - Algorithm Analysis & Foundations

Core Algorithm Analysis Concepts

Algorithm Properties

- **Correctness**: Produces correct output for all valid inputs
- **Efficiency**: Uses computational resources wisely (time/space)
- **Clarity**: Can be understood and implemented
- **Generality**: Solves a class of problems, not just one instance

Computational Model: Word-RAM

- **Fundamental Operations (O(1))**: Memory read/write, arithmetic (+,-,*,/,%), comparisons, logical operations, conditionals
- **NOT O(1)**: String operations, big integer arithmetic, dynamic memory allocation

Complexity Analysis

- **Time Complexity**: Operations as function of input size
- **Space Complexity**: Memory usage as function of input size
- **Best/Average/Worst Case**: Focus on worst-case for upper bounds

Asymptotic Notation

Big O Families

- **O(g(n))**: Upper bound - $f(n) \leq c \cdot g(n)$ for large n
- **Ω(g(n))**: Lower bound - $f(n) \geq c \cdot g(n)$ for large n
- **Θ(g(n))**: Tight bound - f(n) = O(g(n)) AND f(n) = Ω(g(n))

Common Growth Rates (fastest to slowest)

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$

Mathematical Foundations

Essential Summations

- **Triangular Numbers**: ∑(i=1 to n) i = n(n+1)/2
- **Sum of Squares**: ∑(i=1 to n) i^2 = n(n+1)(2n+1)/6
- **Geometric Series**: ∑(i=0 to n-1) r^i = (r^n - 1)/(r - 1)

Logarithm Rules

- **Change of Base**: log_a(n) = log_b(n)/log_b(a)
- **Product**: log(ab) = log(a) + log(b)

- **Quotient**: log(a/b) = log(a) - log(b)
- **Power**: $\log(a^b) = b \cdot \log(a)$
- **Key Insight**: $\log_2(n)$ = "How many times can you divide n by 2?"

Binary Trees

- **Perfect tree nodes**: 2^(h+1) - 1
- **Complete tree height**: $\lfloor \log_2(n) \rfloor$
- **Binary search iterations**: $\lfloor \log_2(n) \rfloor + 1$

Peak Finding

1D Peak Finding

- **Problem**: Find index where arr[i] ≥ arr[i-1] and arr[i] ≥ arr[i+1]
- **Brute Force**: O(n) - check every position
- **Divide & Conquer**: O(log n) - check middle, recurse on side with larger neighbor

Data Structures

Arrays

- **Access**: O(1) by index
- **Search**: O(n) unsorted, O(log n) sorted
- **Insert/Delete**: O(n) due to shifting

Linked Lists

- **Access**: O(n) to reach position
- **Insert/Delete**: O(1) if at known position
- **Search**: O(n) always

Dynamic Arrays

- **Amortized insert**: O(1) average, O(n) when resizing
- **Doubling strategy**: Maintains O(1) amortized time (if double capacity when resizing)

Recurrence Relations

Master Theorem

For T(n) = aT(n/b) + f(n) where a ≥ 1, b > 1:

Case 1: If f(n) = O(n^(log_b(a) - ε)) for ε > 0 → T(n) = Θ(n^log_b(a))

Case 2: If f(n) = Θ(n^log_b(a)) → T(n) = Θ(n^log_b(a) · log n)

Case 3: If f(n) = Ω(n^(log_b(a) + ε)) for ε > 0 → T(n) = Θ(f(n))

Common Examples

- **Binary Search**: $T(n) = T(n/2) + O(1) \rightarrow O(\log n)$
- **Merge Sort**: $T(n) = 2T(n/2) + O(n) \rightarrow O(n \log n)$

Sorting Algorithms

Comparison-Based Sorting

- **Lower bound**: $\Omega(n \log n)$ for comparison-based algorithms
- **Bubble Sort**: $O(n^2)$ - Iterative, stable, in-place
- **Selection Sort**: $O(n^2)$ - Iterative, not stable, in-place
- **Insertion Sort**: $O(n^2)$ - Iterative, stable, in-place
- **Merge Sort**: $O(n \log n)$ - Divide & conquer, stable, not in-place
- **Quick Sort**: $O(n^2)$ - Divide & conquer, not stable, in-place

Linear Sorting

- **Counting Sort**: $O(n + k)$ where k is range - Iterative, stable, not in-place
- **Radix Sort**: $O(d(n + k))$ where d is digits - Iterative, stable, uses counting sort