# Algorithms Summary

Mathematical Notes

October 19, 2025

## Contents

# 1 Algorithm Analysis

## 1.1 Time Complexity

**Definition 1.1.** **Time complexity** measures how the running time of an algorithm grows as the input size increases.

## 1.2 Space Complexity

**Definition 1.2.** **Space complexity** measures how much memory an algorithm uses as the input size increases.

## 1.3 Big-O Notation

**Definition 1.3.** $f(n) = O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

## 1.4 Common Complexity Classes

- **Constant**: $O(1)$ - Time doesn't depend on input size
- **Logarithmic**: $O(\log n)$ - Binary search
- **Linear**: $O(n)$ - Linear search
- **Linearithmic**: $O(n \log n)$ - Merge sort, heap sort
- **Quadratic**: $O(n^2)$ - Bubble sort, selection sort
- **Cubic**: $O(n^3)$ - Matrix multiplication
- **Exponential**: $O(2^n)$ - Brute force solutions
- **Factorial**: $O(n!)$ - Permutation generation

## 1.5 Other Notations

- **Big-Omega**: $f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$
- **Big-Theta**: $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- **Little-o**: $f(n) = o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

# 2 Sorting Algorithms

## 2.1 Comparison-Based Sorting

**Theorem 2.1.** Any comparison-based sorting algorithm has a lower bound of $\Omega(n \log n)$ comparisons in the worst case.

### 2.2 Bubble Sort

- **Time Complexity**: $O(n^2)$ worst case, $O(n)$ best case
- **Space Complexity**: $O(1)$
- **Stable**: Yes
- **In-place**: Yes

**Algorithm 1** Bubble Sort

```
procedure BubbleSort(A[1..n])
    for i = 1 to n − 1 do
        for j = 1 to n − i do
            if A[j] > A[j + 1] then
                Swap(A[j], A[j + 1])
            end if
        end for
    end for
end procedure
```

### 2.3 Selection Sort

- **Time Complexity**: $O(n^2)$
- **Space Complexity**: $O(1)$
- **Stable**: No
- **In-place**: Yes

### 2.4 Insertion Sort

- **Time Complexity**: $O(n^2)$ worst case, $O(n)$ best case
- **Space Complexity**: $O(1)$
- **Stable**: Yes
- **In-place**: Yes

### 2.5 Merge Sort

- **Time Complexity**: $O(n \log n)$
- **Space Complexity**: $O(n)$
- **Stable**: Yes
- **In-place**: No

**Algorithm 2** Merge Sort

```
procedure MergeSort(A[1..n])
    if n ≤ 1 then
        return A
    end if
    mid ← ⌊n/2⌋
    left ← MergeSort(A[1..mid])
    right ← MergeSort(A[mid + 1..n])
    return Merge(left, right)
end procedure
```

### 2.6 Quick Sort

- **Time Complexity**: $O(n \log n)$ average, $O(n^2)$ worst case
- **Space Complexity**: $O(\log n)$ average, $O(n)$ worst case
- **Stable**: No
- **In-place**: Yes (with proper implementation)

### 2.7 Heap Sort

- **Time Complexity**: $O(n \log n)$
- **Space Complexity**: $O(1)$
- **Stable**: No
- **In-place**: Yes

### 2.8 Counting Sort

- **Time Complexity**: $O(n + k)$ where $k$ is the range of input
- **Space Complexity**: $O(k)$
- **Stable**: Yes
- **In-place**: No

### 2.9 Radix Sort

- **Time Complexity**: $O(d(n + k))$ where $d$ is the number of digits
- **Space Complexity**: $O(n + k)$
- **Stable**: Yes
- **In-place**: No

# 3 Searching Algorithms

## 3.1 Linear Search

- **Time Complexity**: $O(n)$
- **Space Complexity**: $O(1)$

## 3.2 Binary Search

- **Time Complexity**: $O(\log n)$
- **Space Complexity**: $O(1)$ iterative, $O(\log n)$ recursive
- **Requirement**: Array must be sorted

**Algorithm 3** Binary Search

```
procedure BinarySearch(A[1..n], target)
    left ← 1, right ← n
    while left ≤ right do
        mid ← ⌊(left + right)/2⌋
        if A[mid] = target then
            return mid
        else if A[mid] < target then
            left ← mid + 1
        else
            right ← mid − 1
        end if
    end while
    return −1                                ▷ Not found
end procedure
```

## 3.3 Interpolation Search

- **Time Complexity**: $O(\log \log n)$ average, $O(n)$ worst case
- **Space Complexity**: $O(1)$
- **Requirement**: Uniformly distributed sorted array

# 4 Graph Algorithms

## 4.1 Graph Representation

- **Adjacency Matrix**: $O(V^2)$ space, $O(1)$ edge lookup
- **Adjacency List**: $O(V + E)$ space, $O(V)$ edge lookup

## 4.2 Breadth-First Search (BFS)

- **Time Complexity**: $O(V + E)$
- **Space Complexity**: $O(V)$
- **Uses**: Shortest path in unweighted graphs, level-order traversal

**Algorithm 4** Breadth-First Search

```
procedure BFS(G, start)
    queue ← empty queue
    visited ← empty set
    Enqueue(queue, start)
    Add(visited, start)
    while queue is not empty do
        vertex ← Dequeue(queue)
        Process(vertex)
        for each neighbor v of vertex do
            if v not in visited then
                Add(visited, v)
                Enqueue(queue, v)
            end if
        end for
    end while
end procedure
```

## 4.3 Depth-First Search (DFS)

- **Time Complexity**: $O(V + E)$
- **Space Complexity**: $O(V)$
- **Uses**: Topological sorting, cycle detection, path finding

## 4.4 Dijkstra's Algorithm

- **Time Complexity**: $O((V + E) \log V)$ with binary heap
- **Space Complexity**: $O(V)$
- **Uses**: Shortest path in weighted graphs with non-negative weights

## 4.5 Bellman-Ford Algorithm

- **Time Complexity**: $O(VE)$
- **Space Complexity**: $O(V)$
- **Uses**: Shortest path with negative weights, negative cycle detection

### 4.6 Floyd-Warshall Algorithm

- **Time Complexity**: $O(V^3)$
- **Space Complexity**: $O(V^2)$
- **Uses**: All-pairs shortest paths

### 4.7 Minimum Spanning Tree

#### 4.7.1 Kruskal's Algorithm

- **Time Complexity**: $O(E \log E)$
- **Space Complexity**: $O(V)$
- **Uses**: Union-Find data structure

#### 4.7.2 Prim's Algorithm

- **Time Complexity**: $O(E \log V)$ with binary heap
- **Space Complexity**: $O(V)$

## 5 Dynamic Programming

### 5.1 Principle of Optimality

**Definition 5.1.** A problem has the **principle of optimality** if an optimal solution contains optimal solutions to subproblems.

### 5.2 Fibonacci Sequence

- **Naive Recursion**: $O(2^n)$
- **Memoization**: $O(n)$ time, $O(n)$ space
- **Tabulation**: $O(n)$ time, $O(n)$ space
- **Space-Optimized**: $O(n)$ time, $O(1)$ space

### 5.3 Longest Common Subsequence (LCS)

- **Time Complexity**: $O(mn)$
- **Space Complexity**: $O(mn)$
- **Space-Optimized**: $O(\min(m, n))$

### 5.4 Longest Increasing Subsequence (LIS)

- **Naive DP**: $O(n^2)$
- **Binary Search**: $O(n \log n)$

### 5.5 Edit Distance (Levenshtein)

- **Time Complexity**: $O(mn)$
- **Space Complexity**: $O(mn)$
- **Space-Optimized**: $O(\min(m, n))$

### 5.6 Knapsack Problem

#### 5.6.1 0/1 Knapsack

- **Time Complexity**: $O(nW)$ where $W$ is capacity
- **Space Complexity**: $O(nW)$

#### 5.6.2 Unbounded Knapsack

- **Time Complexity**: $O(nW)$
- **Space Complexity**: $O(W)$

### 5.7 Matrix Chain Multiplication

- **Time Complexity**: $O(n^3)$
- **Space Complexity**: $O(n^2)$

## 6 Greedy Algorithms

### 6.1 Greedy Choice Property

**Definition 6.1.** A **greedy choice property** means that a locally optimal choice leads to a globally optimal solution.

### 6.2 Activity Selection Problem

- **Time Complexity**: $O(n \log n)$ (due to sorting)
- **Space Complexity**: $O(1)$

### 6.3 Huffman Coding

- **Time Complexity**: $O(n \log n)$
- **Space Complexity**: $O(n)$

### 6.4 Fractional Knapsack

- **Time Complexity**: $O(n \log n)$ (due to sorting)
- **Space Complexity**: $O(1)$

# 7 Divide and Conquer

## 7.1 Master Theorem

**Theorem 7.1.** For recurrence relations of the form $T(n) = aT(n/b) + f(n)$ where $a \geq 1$, $b > 1$:

- If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
- If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$
- If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ and $af(n/b) \leq cf(n)$ for some $c < 1$, then $T(n) = \Theta(f(n))$

## 7.2 Binary Search

- **Recurrence**: $T(n) = T(n/2) + O(1)$
- **Solution**: $T(n) = O(\log n)$

## 7.3 Merge Sort

- **Recurrence**: $T(n) = 2T(n/2) + O(n)$
- **Solution**: $T(n) = O(n \log n)$

## 7.4 Quick Sort

- **Average Case**: $T(n) = 2T(n/2) + O(n) = O(n \log n)$
- **Worst Case**: $T(n) = T(n-1) + O(n) = O(n^2)$

## 7.5 Strassen's Matrix Multiplication

- **Time Complexity**: $O(n^{\log_2 7}) \approx O(n^{2.81})$
- **Space Complexity**: $O(n^2)$

# 8 String Algorithms

## 8.1 Naive String Matching

- **Time Complexity**: $O((n - m + 1)m)$ where $n$ is text length, $m$ is pattern length
- **Space Complexity**: $O(1)$

## 8.2 KMP (Knuth-Morris-Pratt) Algorithm

- **Time Complexity**: $O(n + m)$
- **Space Complexity**: $O(m)$

### 8.3 Rabin-Karp Algorithm

- **Average Time Complexity**: $O(n + m)$
- **Worst Time Complexity**: $O(nm)$
- **Space Complexity**: $O(1)$

### 8.4 Boyer-Moore Algorithm

- **Best Case Time Complexity**: $O(n/m)$
- **Worst Case Time Complexity**: $O(nm)$
- **Space Complexity**: $O(k)$ where $k$ is alphabet size

## 9 Tree Algorithms

### 9.1 Tree Traversals

- **Preorder**: Root, Left, Right
- **Inorder**: Left, Root, Right (gives sorted order for BST)
- **Postorder**: Left, Right, Root
- **Level-order**: Breadth-first traversal

### 9.2 Binary Search Tree Operations

- **Search**: $O(\log n)$ average, $O(n)$ worst case
- **Insert**: $O(\log n)$ average, $O(n)$ worst case
- **Delete**: $O(\log n)$ average, $O(n)$ worst case

### 9.3 AVL Tree

- **Height**: $O(\log n)$
- **All Operations**: $O(\log n)$
- **Balance Factor**: Height difference between left and right subtrees

### 9.4 Red-Black Tree

- **Height**: $O(\log n)$
- **All Operations**: $O(\log n)$
- **Properties**: Root is black, red nodes have black children, all paths have same number of black nodes

# 10 Hash Tables

## 10.1 Hash Functions

- **Division Method**: $h(k) = k \bmod m$
- **Multiplication Method**: $h(k) = \lfloor m(kA \bmod 1) \rfloor$ where $0 < A < 1$

## 10.2 Collision Resolution

### 10.2.1 Chaining

- **Average Search Time**: $O(1 + \alpha)$ where $\alpha$ is load factor
- **Worst Case Search Time**: $O(n)$

### 10.2.2 Open Addressing

- **Linear Probing**: $h(k, i) = (h'(k) + i) \bmod m$
- **Quadratic Probing**: $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
- **Double Hashing**: $h(k, i) = (h_1(k) + i h_2(k)) \bmod m$

# 11 Advanced Data Structures

## 11.1 Segment Tree

- **Space Complexity**: $O(n)$
- **Query Time**: $O(\log n)$
- **Update Time**: $O(\log n)$
- **Uses**: Range queries and updates

## 11.2 Fenwick Tree (Binary Indexed Tree)

- **Space Complexity**: $O(n)$
- **Query Time**: $O(\log n)$
- **Update Time**: $O(\log n)$
- **Uses**: Prefix sums, range sum queries

## 11.3 Disjoint Set Union (Union-Find)

- **Union by Rank**: $O(\log n)$ per operation
- **Path Compression**: $O(\alpha(n))$ per operation where $\alpha$ is inverse Ackermann
- **Both Optimizations**: $O(\alpha(n))$ per operation

### 11.4 Trie (Prefix Tree)

- **Space Complexity**: $O(ALPHABET\_SIZE \times N \times M)$ where $N$ is number of strings, $M$ is average length
- **Search Time**: $O(m)$ where $m$ is length of string
- **Insert Time**: $O(m)$

## 12 Computational Complexity

### 12.1 P vs NP

**Definition 12.1.** **P** is the class of decision problems solvable in polynomial time by a deterministic Turing machine.

**Definition 12.2.** **NP** is the class of decision problems solvable in polynomial time by a non-deterministic Turing machine, or equivalently, problems for which a solution can be verified in polynomial time.

### 12.2 NP-Complete Problems

- Boolean Satisfiability (SAT)
- 3-SAT
- Clique Problem
- Vertex Cover
- Hamiltonian Cycle
- Traveling Salesman Problem
- Subset Sum
- Knapsack Problem

### 12.3 NP-Hard Problems

- Halting Problem
- Optimization versions of NP-Complete problems
- Integer Linear Programming

## 13 Approximation Algorithms

### 13.1 Approximation Ratio

**Definition 13.1.** An algorithm has **approximation ratio** $\rho(n)$ if for any input of size $n$, the cost $C$ of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost $C^*$ of an optimal solution: $\max(C/C^*, C^*/C) \leq \rho(n)$.

### 13.2 Vertex Cover

- **Greedy Algorithm**: 2-approximation
- **LP Rounding**: 2-approximation

### 13.3 Set Cover

- **Greedy Algorithm**: $O(\log n)$-approximation

### 13.4 Traveling Salesman Problem

- **Metric TSP**: 2-approximation (double tree algorithm)
- **Christofides Algorithm**: 1.5-approximation for metric TSP

## 14 Randomized Algorithms

### 14.1 Las Vegas Algorithms

**Definition 14.1.** **Las Vegas algorithms** always produce the correct result, but running time is random.

### 14.2 Monte Carlo Algorithms

**Definition 14.2.** **Monte Carlo algorithms** have deterministic running time but may produce incorrect results with some probability.

### 14.3 Quick Sort (Randomized)

- **Expected Time Complexity**: $O(n \log n)$
- **Worst Case Time Complexity**: $O(n^2)$
- **Space Complexity**: $O(\log n)$

### 14.4 Randomized Selection

- **Expected Time Complexity**: $O(n)$
- **Worst Case Time Complexity**: $O(n^2)$
- **Space Complexity**: $O(1)$

## 15 Online Algorithms

### 15.1 Competitive Ratio

**Definition 15.1.** An online algorithm has **competitive ratio** $c$ if for any input sequence, the cost of the algorithm is at most $c$ times the cost of an optimal offline algorithm.

### 15.2 Paging Problem

- **LRU**: $k$-competitive where $k$ is cache size
- **FIFO**: $k$-competitive
- **Optimal Offline**: MIN algorithm

### 15.3 Ski Rental Problem

- **Deterministic**: 2-competitive
- **Randomized**: $e/(e-1) \approx 1.58$-competitive