

# Algorithms Illuminated: Comprehensive Revision Notes

Includes Advanced Examples & Exam Problems

Guided Summary based on Tim Roughgarden's Textbooks

## Contents

<b>1</b>	<b>Part 1: Asymptotic Analysis &amp; Divide and Conquer</b>	<b>2</b>
1.1	1. Asymptotic Notation (The Language of Algorithms) . . . . .	2
1.2	2. The Master Method . . . . .	2
1.3	3. QuickSort Analysis . . . . .	2
<b>2</b>	<b>Part 2: Graph Algorithms</b>	<b>2</b>
2.1	4. Graph Search (BFS & DFS) . . . . .	2
2.2	5. Dijkstra's Algorithm . . . . .	3
<b>3</b>	<b>Advanced Exam Questions (ETH / Stanford Style)</b>	<b>3</b>
3.1	Question 1: The "Unimodal" Maximum (Divide & Conquer) . . . . .	3
3.2	Question 2: Bipartite Checking (Graph Search) . . . . .	3
3.3	Question 3: The "Bottleneck" Path (Modified Dijkstra) . . . . .	4
3.4	Question 4: True/False "Gotchas" . . . . .	4
<b>4</b>	<b>Essential Proofs to Memorize</b>	<b>4</b>
4.1	Correctness of Dijkstra . . . . .	4
4.2	Master Method Case 2 (Intuition) . . . . .	5

# 1 Part 1: Asymptotic Analysis & Divide and Conquer

## 1.1 1. Asymptotic Notation (The Language of Algorithms)

We analyze algorithms by bounding their running time  $T(n)$  as input size  $n \rightarrow \infty$ .

- **Big-O** ( $T(n) = \mathcal{O}(f(n))$ ): Upper bound.  $T(n) \leq c \cdot f(n)$  for large  $n$ .
- **Big-Omega** ( $T(n) = \Omega(f(n))$ ): Lower bound.  $T(n) \geq c \cdot f(n)$ .
- **Big-Theta** ( $T(n) = \Theta(f(n))$ ): Tight bound. Limits exist on both sides.

**Example 1** (Ranking Functions). Rank the following from slowest to fastest growth:

$$n^2, \quad n \log n, \quad n!, \quad 2^n, \quad \sqrt{n}, \quad n^{1.5}$$

**Order:**  $\sqrt{n} < n \log n < n^{1.5} < n^2 < 2^n < n!$ .

## 1.2 2. The Master Method

Used for recurrences  $T(n) = aT(n/b) + \mathcal{O}(n^d)$ . Compare  $a$  (subproblem proliferation) vs.  $b^d$  (work reduction rate).

Case	Condition	Result
1	$a = b^d$	$T(n) = \mathcal{O}(n^d \log n)$
2	$a < b^d$	$T(n) = \mathcal{O}(n^d)$ (Work at root dominates)
3	$a > b^d$	$T(n) = \mathcal{O}(n^{\log_b a})$ (Work at leaves dominates)

**Example 2** (Strassen's Matrix Multiplication). Recurrence:  $T(n) = 7T(n/2) + \mathcal{O}(n^2)$ . Here  $a = 7, b = 2, d = 2$ . Since  $7 > 2^2 = 4$ , we are in **Case 3**.

$$T(n) = \mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2.81})$$

This beats the naive  $\mathcal{O}(n^3)$  algorithm.

## 1.3 3. QuickSort Analysis

- **Randomized Pivot:** Guarantees  $\mathcal{O}(n \log n)$  *expected* time.
- **Key Insight:** The running time is proportional to the number of comparisons. Two elements  $z_i$  and  $z_j$  are compared iff one of them is chosen as a pivot while they are still in the same sub-array.
- **Probability:**  $P(\text{compare } z_i, z_j) = \frac{2}{j-i+1}$ .

# 2 Part 2: Graph Algorithms

## 2.1 4. Graph Search (BFS & DFS)

- **BFS (Layers):** Finds shortest paths in unweighted graphs. Computes Connected Components in  $\mathcal{O}(m + n)$ .
- **DFS (Backtracking):** Computes Topological Sort and Strongly Connected Components (SCCs).

## 2.2 5. Dijkstra's Algorithm

Finds shortest paths from  $s$  with **non-negative edge lengths**  $l_e \geq 0$ .

- **Greedy Criterion:** Maintain processed set  $X$ . Always extract  $v \notin X$  minimizing:

$$\text{Score}(v) = \min_{u \in X, (u,v) \in E} \{A[u] + l_{uv}\}$$

- **Implementation:** Use a Heap. Store vertices with keys = Dijkstra Score.
- **Runtime:**  $\mathcal{O}(m \log n)$  with binary heap.

## 3 Advanced Exam Questions (ETH / Stanford Style)

*These questions test algorithmic design and deep conceptual understanding rather than rote application.*

### 3.1 Question 1: The "Unimodal" Maximum (Divide & Conquer)

**Problem:** You are given an array  $A$  of  $n$  distinct integers. The array is "unimodal": it increases up to a maximum and then decreases. (e.g.,  $[1, 3, 8, 12, 9, 4, 2]$ ). Design an  $\mathcal{O}(\log n)$  algorithm to find the maximum element.

**Solution 1.** We cannot scan the array ( $\mathcal{O}(n)$ ). We must use a Binary Search variation.

1. Pick the middle element  $m$  at index  $n/2$ .
2. Look at its neighbors  $m - 1$  and  $m + 1$ .
3. **Case A:** If  $A[m - 1] < A[m] < A[m + 1]$ , the peak is to the **right**. Recurse on right half.
4. **Case B:** If  $A[m - 1] > A[m] > A[m + 1]$ , the peak is to the **left**. Recurse on left half.
5. **Case C:** If  $A[m - 1] < A[m]$  and  $A[m] > A[m + 1]$ , then  $A[m]$  is the peak. Return it.

**Time:**  $T(n) = T(n/2) + \mathcal{O}(1) \implies \mathcal{O}(\log n)$ .

### 3.2 Question 2: Bipartite Checking (Graph Search)

**Problem:** A graph is bipartite if its vertices can be split into two sets  $V_1, V_2$  such that every edge connects a node in  $V_1$  to one in  $V_2$  (i.e., 2-colorable). Design an  $\mathcal{O}(m + n)$  algorithm to determine if a connected undirected graph is bipartite.

**Solution 2.** Use BFS (Breadth-First Search).

1. Run BFS starting from arbitrary node  $s$ .
2. Assign  $s$  to "Layer 0". All neighbors of  $s$  are "Layer 1", their neighbors "Layer 2", etc.
3. **Coloring Rule:** Nodes in even layers get Color A. Nodes in odd layers get Color B.
4. **Check:** Iterate through all edges  $(u, v)$ . If  $u$  and  $v$  have the *same* color (i.e., belong to the same layer parity), the graph is **not** bipartite.
5. If the check passes for all edges, it is bipartite.

**Proof Intuition:** A graph is bipartite iff it contains no odd cycles. BFS layers detect odd cycles effectively.

### 3.3 Question 3: The "Bottleneck" Path (Modified Dijkstra)

**Problem:** Instead of minimizing the *sum* of edge weights, we want to maximize the *capacity* of the path. The capacity of a path is defined as the **minimum** edge weight along that path. Find the path from  $s$  to  $t$  that maximizes this bottleneck capacity.

**Solution 3.** Modify Dijkstra's Algorithm.

- **Score Definition:** Instead of  $A[v] = \text{dist}(s, v)$ , let  $W[v]$  be the max-capacity to reach  $v$ .
- **Initialization:**  $W[s] = \infty$ , all others  $-\infty$ .
- **Greedy Step:** Pick  $v \notin X$  with the **maximum**  $W[v]$ .
- **Relaxation:** When considering edge  $(u, v)$ , the candidate capacity is  $\min(W[u], \text{weight}_{uv})$ .
- **Update:** If  $\min(W[u], \text{weight}_{uv}) > W[v]$ , update  $W[v]$ .

This is effectively Prim's algorithm for Maximum Spanning Tree adapted for single-path queries.

### 3.4 Question 4: True/False "Gotchas"

1. **Statement:** "DFS always finds the shortest path in an unweighted graph."

**Answer: False.** DFS goes deep. It might find a path of length 10 before finding a neighbor of length 1. BFS is required for shortest paths in unweighted graphs.

2. **Statement:** "If we square every edge weight ( $l_e^2$ ), the shortest path remains the same."

**Answer: False.** Squaring penalizes large weights disproportionately.

*Ex:* Path A edges: 2, 2 (Sum 4). Path B edge: 3 (Sum 3). B is shorter.

Squared: A becomes  $4 + 4 = 8$ . B becomes 9. A is now shorter.

3. **Statement:** "In a DAG (Directed Acyclic Graph), we can find shortest paths even with negative edge weights in  $\mathcal{O}(m + n)$ ."

**Answer: True.** We can process vertices in **Topological Order**. Since there are no cycles, we relax edges in one linear pass, avoiding the infinite loops that negative cycles cause in general graphs.

## 4 Essential Proofs to Memorize

### 4.1 Correctness of Dijkstra

**Theorem:** Dijkstra's algorithm correctly computes shortest paths if  $l_e \geq 0$ .

*Proof.* By induction on the size of set  $X$ .

- Base case:  $|X| = 1$  ( $s$  is correct).
- Inductive step: Suppose all  $u \in X$  have correct distances  $A[u]$ . Let  $v$  be the next node added via edge  $(u^*, v)$ .
- Any other path to  $v$  must leave  $X$  via some other edge  $(y, z)$ .
- Length of other path  $\geq A[y] + l_{yz}$ .
- By the greedy choice,  $A[u^*] + l_{u^*v} \leq A[y] + l_{yz}$ .
- Since edge weights are non-negative, the path cannot get shorter after leaving  $X$ . Thus, the greedy path is optimal.

□

## 4.2 Master Method Case 2 (Intuition)

Why is  $T(n) = \mathcal{O}(n^d \log n)$  when  $a = b^d$ ?

*Proof.* The work at depth  $j$  is  $a^j \times c(\frac{n}{b^j})^d$ . Substituting  $a = b^d$ , the terms cancel out:

$$\text{Work}_j = (b^d)^j \cdot c \frac{n^d}{(b^j)^d} = b^{dj} \cdot c \frac{n^d}{b^{dj}} = c \cdot n^d$$

The work at **every level** is the same ( $cn^d$ ). Since there are  $\log_b n$  levels, total work is  $\mathcal{O}(n^d \log n)$ .  $\square$