A Comprehensive Guide to Data Structures and Algorithms

A Student Handbook

October 8, 2025

Contents

Part I

Foundations of Algorithm Analysis

Chapter 1

Asymptotic Notation

Asymptotic notation is used to describe the limiting behavior of a function when the argument tends towards a particular value or infinity. In computer science, it's used to analyze the time and space complexity of algorithms, focusing on their performance as the input size grows large.

1.1 Big-O Notation (O)

- **Definition:** $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.
- **Meaning:** Provides an **asymptotic upper bound**. It describes the worst-case scenario, indicating that the algorithm's resource usage grows no faster than $g(n)$.
- **Example:** If an algorithm's running time is $T(n) = 3n^2 + 5n + 10$, it is $O(n^2)$. For large n, the n^2 term dominates the growth.

1.2 Omega Notation (Ω)

- **Definition:** $f(n) = \Omega(g(n))$ if there exist positive constants c and n_0 such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0$.
- **Meaning:** Provides an **asymptotic lower bound**. It describes the best-case scenario, indicating that the algorithm's resource usage grows at least as fast as $g(n)$.
- **Example:** The algorithm with $T(n) = 3n^2 + 5n + 10$ is also $\Omega(n^2)$ because its growth is at least as fast as n^2.

1.3 Theta Notation (Θ)

- **Definition:** $f(n) = \Theta(g(n))$ if there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$.
- **Meaning:** Provides a **tight bound**. It means $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$, indicating the algorithm's growth rate is precisely $g(n)$.

- **Example:** The algorithm with $T(n) = 3n^2 + 5n + 10$ is $\Theta(n^2)$ as it is bounded both above and below by n^2.

1.4 Little-o and Little-omega Notations

- **Little-o (o):** A strict upper bound. $f(n) = o(g(n))$ means $f(n)$ grows strictly slower than $g(n)$. Example: $5n = o(n^2)$.
- **Little-omega (ω):** A strict lower bound. $f(n) = \omega(g(n))$ means $f(n)$ grows strictly faster than $g(n)$. Example: $n^2 = \omega(n)$.

1.5 Hierarchy of Complexity Classes

Functions are commonly ordered by their growth rate. For a large input size n, the following hierarchy holds:

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

- $O(1)$: Constant time (e.g., accessing an array element).
- $O(\log n)$: Logarithmic time (e.g., binary search).
- $O(n)$: Linear time (e.g., traversing a list).
- $O(n \log n)$: Log-linear time (e.g., efficient sorting algorithms like Merge Sort).
- $O(n^2)$: Quadratic time (e.g., simple sorting algorithms like Bubble Sort).
- $O(2^n)$: Exponential time (e.g., solving traveling salesman problem via brute force).
- $O(n!)$: Factorial time (e.g., generating all permutations of a list).

Part II

Linear Data Structures

Chapter 2

Stacks

A stack is a linear data structure that follows the **Last-In, First-Out (LIFO)** principle. The last element added to the stack is the first one to be removed. It can be visualized as a stack of plates.

2.1 Abstract Data Type (ADT) Definition

- **Objects:** An ordered list of elements with a single access point called the `top`.
- **Operations:**
  - `Create(S)`: Creates an empty stack S.
  - `Push(S, x)`: Inserts element `x` at the `top` of S.
  - `Pop(S)`: Removes and returns the element from the `top` of S.
  - `Peek(S)`: Returns the element at the `top` without removing it.
  - `IsEmpty(S)`: Returns true if S is empty, otherwise false.
  - `IsFull(S)`: Returns true if S is at maximum capacity (for array-based stacks).

2.2 Array-Based Implementation

A stack can be implemented using a fixed-size array and an integer variable `top` that tracks the index of the top element.

```
// Global variables for the stack
// MAX_SIZE is the maximum capacity of the stack
// S: array of size MAX_SIZE
// top: integer, initialized to -1

procedure Create(S):
    top = -1

procedure IsEmpty():
    if top == -1:
        return TRUE
    else:
        return FALSE

```

```
procedure IsFull():
    if top == MAX_SIZE - 1:
        return TRUE
    else:
        return FALSE

procedure Push(S, x):
    if IsFull():
        print "Stack Overflow"
    else:
        top = top + 1
        S[top] = x

procedure Pop(S):
    if IsEmpty():
        print "Stack Underflow"
        return NULL
    else:
        item = S[top]
        top = top - 1
        return item

procedure Peek(S):
    if IsEmpty():
        print "Stack is empty"
        return NULL
    else:
        return S[top]
```

Listing 2.1: Array-Based Stack Operations

All primary stack operations (`Push`, `Pop`, `Peek`) have a time complexity of $O(1)$.

Chapter 3

Queues

A queue is a linear data structure that follows the **First-In, First-Out (FIFO)** principle. The first element added to the queue is the first one to be removed, like a line of people waiting for a service.

3.1 Abstract Data Type (ADT) Definition

- **Objects:** An ordered list with two ends: a `front` for removal and a `rear` for insertion.
- **Operations:**
  - `Create(Q)`: Creates an empty queue Q.
  - `Enqueue(Q, x)`: Inserts element `x` at the `rear` of Q.
  - `Dequeue(Q)`: Removes and returns the element from the `front` of Q.
  - `Peek(Q)`: Returns the `front` element without removing it.
  - `IsEmpty(Q)`: Returns true if Q is empty.
  - `IsFull(Q)`: Returns true if Q is full (for array-based queues).

3.2 Circular Queue (Array-Based Implementation)

A simple linear queue implemented with an array is inefficient because space at the front cannot be reused after dequeuing. A **circular queue** overcomes this limitation by allowing the `rear` pointer to wrap around to the beginning of the array.

We use two pointers, `front` and `rear`, initialized to `-1` for an empty queue.

```
// Global variables for the queue
// maxSize: maximum capacity of the queue
// queue[]: an array to hold elements
// front: index of the first element (initially -1)
// rear: index of the last element (initially -1)

procedure IsEmpty():
    return front == -1

procedure IsFull():
    // Condition for a full queue
```

```
    return (front == 0 AND rear == maxSize - 1) OR ((rear + 1) %
   maxSize == front)

procedure Enqueue(queue, element):
    if IsFull():
        print "Queue Overflow"
        return

    if IsEmpty(): // First element insertion
        front = 0
        rear = 0
    else: // Subsequent insertions
        rear = (rear + 1) % maxSize

    queue[rear] = element
    print element, "inserted into queue"

procedure Dequeue(queue):
    if IsEmpty():
        print "Queue Underflow"
        return NULL

    element = queue[front]

    if front == rear: // Only one element was in the queue
        front = -1
        rear = -1
    else:
        front = (front + 1) % maxSize

    print element, "removed from queue"
    return element

procedure Peek(queue):
    if IsEmpty():
        print "Queue is empty"
        return NULL
    return queue[front]
```

Listing 3.1: Circular Queue Operations

All primary circular queue operations have a time complexity of $O(1)$.

Chapter 4

Linked Lists

A linked list is a linear data structure where elements, called **nodes**, are stored in non-contiguous memory locations. Each node contains data and a pointer (or link) to the next node in the sequence.

4.1 Singly Linked List (SLL)

Each node has two components: `data` and a `next` pointer. Traversal is only possible in the forward direction.

```
Structure Node:
    data
    next_pointer

// --- Insertion at the beginning (O(1)) ---
procedure InsertBegin(head, x):
    newNode = allocate Node(x)
    newNode.next = head
    head = newNode
    return head

// --- Insertion at the end (O(n)) ---
procedure InsertEnd(head, x):
    newNode = allocate Node(x)
    if head == NULL:
        return newNode // The new node is the head

    temp = head
    while temp.next != NULL:
        temp = temp.next
    temp.next = newNode
    return head

// --- Deletion from the beginning (O(1)) ---
procedure DeleteBegin(head):
    if head == NULL:
        print "Underflow"
        return NULL

    temp = head
    head = head.next
    free(temp)
```

```
    return head

// --- Traversal (O(n)) ---
procedure Traverse(head):
    temp = head
    while temp != NULL:
        print(temp.data)
        temp = temp.next
```

Listing 4.1: Singly Linked List Node and Basic Operations

4.2 Doubly Linked List (DLL)

Each node has three components: `data`, a `next` pointer, and a `prev` pointer. This allows for traversal in both forward and backward directions.

```
Structure Node:
    prev_pointer
    data
    next_pointer

// --- Insertion at the beginning (O(1)) ---
procedure InsertBegin(head, x):
    newNode = allocate Node(x)
    newNode.next = head
    if head != NULL:
        head.prev = newNode
    head = newNode
    return head

// --- Traversal Forward (O(n)) ---
procedure TraverseForward(head):
    temp = head
    while temp != NULL:
        print temp.data
        temp = temp.next
```

Listing 4.2: Doubly Linked List Node and Insertion

4.3 Circular Linked Lists

In a circular linked list, the last node's `next` pointer points back to the head node instead of being `NULL`. This creates a continuous loop. This structure is useful for applications like round-robin scheduling.

- **Circular Singly Linked List (CSLL):** Singly linked list where 'tail.next = head'.
- **Circular Doubly Linked List (CDLL):** Doubly linked list where 'tail.next = head' and 'head.prev = tail'.

4.4 Comparison of Linked List Variants

* Assumes a pointer to the `tail` node is maintained.
** Becomes $O(1)$ if a `tail` pointer is maintained.

Table 4.1: Linked List Feature Comparison

Feature	**SLL**	**DLL**	**CSLL**	**CDLL**
Structure	data, next	prev, data, next	data, next	prev, data, next
Traversal	Forward	Forward & Back	Circular	Circular & Bidir.
Memory/Node	Data + 1 ptr	Data + 2 ptrs	Data + 1 ptr	Data + 2 ptrs
Insert at Begin	$O(1)$	$O(1)$	$O(1)$*	$O(1)$*
Insert at End	$O(n)$**	$O(n)$**	$O(1)$*	$O(1)$*
Delete at End	$O(n)$	$O(n)$**	$O(n)$	$O(1)$*

Part III

Non-Linear Data Structures

Chapter 5

Trees

A tree is a hierarchical data structure consisting of nodes connected by edges. It is non-linear and used to represent hierarchical relationships.

5.1 Binary Trees

A binary tree is a tree in which each node has at most two children, referred to as the **left child** and the **right child**.

5.1.1 Properties

- The maximum number of nodes at level i (root is at level 0) is 2^i.
- A binary tree of height h has at most $2^{h+1} - 1$ nodes.

5.1.2 Types of Binary Trees

- **Full Binary Tree:** Every node has either 0 or 2 children.
- **Complete Binary Tree:** All levels are completely filled except possibly the last, which is filled from left to right.
- **Perfect Binary Tree:** All internal nodes have 2 children, and all leaf nodes are at the same level.
- **Balanced Binary Tree:** The height difference between the left and right subtrees of any node is at most 1.
- **Degenerate (or Skewed) Tree:** Each parent node has only one child, resembling a linked list.

5.1.3 Tree Traversal Algorithms

Traversal is the process of visiting each node in a tree exactly once.

- **In-order (Left, Root, Right):** Visits the left subtree, then the root, then the right subtree. For a BST, this produces a sorted sequence.
- **Pre-order (Root, Left, Right):** Visits the root, then the left subtree, then the right subtree. Useful for creating a copy of the tree.
- **Post-order (Left, Right, Root):** Visits the left subtree, then the right subtree, then the root. Useful for deleting nodes.
- **Level-order:** Visits nodes level by level, from left to right, using a queue.

```
// In-order Traversal
procedure Inorder(node):
    if node == NULL: return
    Inorder(node.left)
    print(node.data)
    Inorder(node.right)

// Pre-order Traversal
procedure Preorder(node):
    if node == NULL: return
    print(node.data)
    Preorder(node.left)
    Preorder(node.right)

// Post-order Traversal
procedure Postorder(node):
    if node == NULL: return
    Postorder(node.left)
    Postorder(node.right)
    print(node.data)
```

Listing 5.1: Tree Traversal Pseudocode

5.2 Binary Search Trees (BST)

A Binary Search Tree (BST) is a binary tree with a specific ordering property:

1. For any given node, all values in its **left subtree** are **less than** the node's value.
2. For any given node, all values in its **right subtree** are **greater than** the node's value.
3. Both the left and right subtrees must also be binary search trees.

This property allows for efficient searching, insertion, and deletion operations, with an average-case time complexity of $O(\log n)$. The worst-case is $O(n)$ if the tree becomes degenerate.

```
// Search for a key 'x' in a BST rooted at 'root'
procedure Search(root, x):
    if root == NULL or root.key == x:
        return root // Found or not found

    if x < root.key:
        return Search(root.left, x)
    else:
        return Search(root.right, x)

// Insert a key 'x' into a BST rooted at 'root'
procedure Insert(root, x):
    if root == NULL:
        return new Node(x) // Create a new node

    if x < root.key:
        root.left = Insert(root.left, x)
    else if x > root.key:
        root.right = Insert(root.right, x)

    return root // Return the (possibly modified) root pointer
```

Listing 5.2: BST Search and Insert Operations

Deletion in a BST is more complex and involves three cases: deleting a leaf node (0 children), a node with one child, or a node with two children. In the third case, the node is typically replaced by its in-order successor (the smallest node in its right subtree).

5.3 Self-Balancing Trees: AVL Trees

An AVL (Adelson-Velsky and Landis) tree is a self-balancing binary search tree. For every node, the height difference between its left and right subtrees (the **balance factor**) is at most 1 (i.e., it can only be -1, 0, or 1). This property ensures the tree's height remains $O(\log n)$, guaranteeing efficient operations.

When an insertion or deletion violates the balance property, the tree is rebalanced using operations called **rotations**.

- **Left Rotation and Right Rotation:** These are simple rotations to fix imbalances.
- **The Four Cases of Imbalance:**
    1. **Left-Left (LL):** A node is inserted into the left subtree of the left child. Fixed by a single Right Rotation.
    2. **Right-Right (RR):** A node is inserted into the right subtree of the right child. Fixed by a single Left Rotation.
    3. **Left-Right (LR):** A node is inserted into the right subtree of the left child. Fixed by a Left Rotation followed by a Right Rotation.
    4. **Right-Left (RL):** A node is inserted into the left subtree of the right child. Fixed by a Right Rotation followed by a Left Rotation.

All operations (search, insert, delete) in an AVL tree have a worst-case time complexity of $O(\log n)$.

Chapter 6

Heaps

A heap is a specialized tree-based data structure that is a **complete binary tree** and satisfies the **heap property**. It is commonly used for implementing priority queues.

6.1 Properties and Types

- **Shape Property:** The tree must be a complete binary tree. This allows it to be efficiently stored in an array.
- **Heap Property:**
  - **Max-Heap:** The value of each parent node is greater than or equal to the values of its children. The root holds the maximum element.
  - **Min-Heap:** The value of each parent node is less than or equal to the values of its children. The root holds the minimum element.

6.2 Array Representation

For a node at index ‘i‘ in a 0-indexed array:

- **Parent:** ‘floor((i - 1) / 2)‘
- **Left Child:** ‘2 * i + 1‘
- **Right Child:** ‘2 * i + 2‘

6.3 Core Heap Operations

The main operations rely on restoring the heap property.

- **Heapify-Down (or Max-Heapify):** Given a node ‘i‘ that might violate the heap property, this procedure lets the value at ‘i‘ ”float down” the tree until the heap property is restored for the subtree rooted at ‘i‘. This is the core of ‘Extract-Max‘ and ‘Build-Heap‘.
- **Heapify-Up:** Used during insertion. A new element is added to the end of the heap, and it ”bubbles up” by swapping with its parent until the heap property is restored.

```
// Restores the heap property for the subtree rooted at index i
// n is the size of the heap
procedure MaxHeapify(A, n, i):
    largest = i
    left = 2*i + 1
    right = 2*i + 2

```

```
    // Check if left child is larger than root
    if left < n AND A[left] > A[largest]:
        largest = left

    // Check if right child is larger than current largest
    if right < n AND A[right] > A[largest]:
        largest = right

    // If largest is not the root, swap and recurse
    if largest != i:
        swap(A[i], A[largest])
        MaxHeapify(A, n, largest)
```

Listing 6.1: Heapify-Down (for a Max-Heap)

6.3.1 Complexity of Heap Operations

Table 6.1: Time Complexity of Binary Heap Operations

Operation	**Time Complexity**
Insert	$O(\log n)$
Extract-Max/Min	$O(\log n)$
Peek (Get-Max/Min)	$O(1)$
Build-Heap	$O(n)$

Chapter 7

Hash Tables

A hash table is a data structure that maps **keys** to **values** for highly efficient lookup. It uses a **hash function** to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found.

7.1 Key Components

- **Hash Function:** A function that takes a key and returns an index in the hash table. A good hash function should be fast and distribute keys uniformly across the table to minimize collisions.
- **Hash Table:** An array that stores the data.
- **Collision:** Occurs when two different keys hash to the same index.

7.2 Collision Resolution Techniques

Since collisions are practically unavoidable, strategies are needed to handle them.

7.2.1 Separate Chaining

Each slot in the hash table stores a pointer to another data structure, typically a linked list, which holds all the key-value pairs that hashed to that index.

- **Insert/Search/Delete:** The hash function is used to find the correct linked list. The operation is then performed on that list.
- **Performance:** The average time complexity is $O(1 + \alpha)$, where α (the load factor) is the ratio of stored elements (n) to table slots (m). If α is a small constant, the complexity is $O(1)$.

7.2.2 Open Addressing

All key-value pairs are stored directly in the hash table array. When a collision occurs, the algorithm probes for the next available slot.

- **Linear Probing:** If slot 'h(k)' is occupied, try 'h(k)+1', 'h(k)+2', etc. (with wrap-around). This method is simple but suffers from **primary clustering**, where long runs of occupied slots build up, degrading performance.
- **Quadratic Probing:** If slot 'h(k)' is occupied, try 'h(k)+1^2', '$h(k)+2^2$', $etc. This helps alleviate primary clustering$
- **Double Hashing:** Uses a second hash function to determine the probe step size. This is one of the most effective methods for reducing clustering.

The time complexity for open addressing operations is highly dependent on the load factor α, which must be kept below 1.

Part IV

Sorting Algorithms

Chapter 8

Comparison-Based Sorting

These algorithms sort a list by comparing pairs of elements. Their performance is fundamentally limited by a lower bound of $\Omega(n \log n)$.

8.1 Bubble Sort

- **Idea:** Repeatedly step through the list, compare adjacent elements, and swap them if they are in the wrong order. The largest elements "bubble" to the end.
- **Time Complexity:** $O(n^2)$ in average and worst cases. $O(n)$ in the best case (if the array is already sorted and an optimization is used).
- **Space Complexity:** $O(1)$ (in-place).
- **Stability:** Stable.

8.2 Selection Sort

- **Idea:** Repeatedly find the minimum element from the unsorted part of the array and put it at the beginning.
- **Time Complexity:** $O(n^2)$ in all cases (best, average, worst).
- **Space Complexity:** $O(1)$ (in-place).
- **Stability:** Not stable.

8.3 Insertion Sort

- **Idea:** Build the final sorted array one item at a time. It iterates through the input elements and inserts each element into its correct position in the sorted part of the array.
- **Time Complexity:** $O(n^2)$ in average and worst cases. $O(n)$ in the best case. Efficient for small or nearly sorted datasets.
- **Space Complexity:** $O(1)$ (in-place).
- **Stability:** Stable.

8.4 Merge Sort

- **Idea:** A classic **Divide and Conquer** algorithm.
    1. **Divide:** The array is divided into two halves.
    2. **Conquer:** Each half is recursively sorted.

3. **Combine:** The two sorted halves are merged into a single sorted array.

- **Time Complexity:** $O(n \log n)$ in all cases.
- **Space Complexity:** $O(n)$ (requires temporary arrays for merging).
- **Stability:** Stable.

8.5 Quicksort

- **Idea:** Another **Divide and Conquer** algorithm.
  1. **Pivot:** An element is chosen as a pivot.
  2. **Partition:** The array is reordered so that all elements smaller than the pivot come before it, and all elements greater come after it.
  3. **Recurse:** The algorithm is recursively applied to the sub-arrays.
- **Time Complexity:** $O(n \log n)$ in average and best cases. The worst case is $O(n^2)$, which occurs with a poor pivot choice (e.g., on an already sorted array). Using a randomized pivot makes the worst case highly unlikely.
- **Space Complexity:** $O(\log n)$ on average (for the recursion stack). $O(n)$ in the worst case.
- **Stability:** Not stable.

8.6 Heapsort

- **Idea:** Uses a binary heap data structure.
  1. **Build-Heap:** First, convert the input array into a max-heap.
  2. **Sort:** The largest element is at the root. Swap it with the last element of the heap, reduce the heap size by one, and call ‘MaxHeapify‘ on the root to restore the heap property. Repeat until the heap is empty.
- **Time Complexity:** $O(n \log n)$ in all cases.
- **Space Complexity:** $O(1)$ (in-place).
- **Stability:** Not stable.

Chapter 9

Non-Comparison-Based Sorting

These algorithms sort elements without comparing them, often by using properties of the elements themselves (like their integer value). They can achieve better than $O(n \log n)$ time complexity but are restricted to specific types of input.

9.1 Counting Sort

- **Idea:** Works by counting the number of occurrences of each distinct element in the input array. This count information is then used to determine the correct position of each element in the sorted output array.
- **Restrictions:** Requires the input to be integers within a known, relatively small range $[0, k]$.
- **Time Complexity:** $O(n+k)$, where n is the number of elements and k is the range of the input. It is linear if $k = O(n)$.
- **Space Complexity:** $O(n + k)$.
- **Stability:** Stable.

9.2 Radix Sort

- **Idea:** Sorts integers by processing individual digits. It sorts the numbers digit by digit, from the least significant digit (LSD) to the most significant digit (MSD). It uses a stable sorting algorithm (like Counting Sort) as a subroutine for each digit.
- **Time Complexity:** $O(d \cdot (n + k))$, where d is the number of digits in the largest number, n is the number of elements, and k is the base (e.g., 10 for decimal).
- **Space Complexity:** $O(n + k)$.
- **Stability:** Stable.

Part V

Advanced Topics

Chapter 10

Disjoint Set Union (DSU)

Also known as Union-Find, this data structure manages a collection of disjoint (non-overlapping) sets. It is highly efficient and crucial for algorithms like Kruskal's for finding minimum spanning trees.

10.1 Core Operations

1. **MakeSet(x):** Creates a new set containing only the element 'x'.
2. **Find(x):** Returns the representative (or root) of the set containing 'x'. This is used to check if two elements belong to the same set.
3. **Union(x, y):** Merges the two sets containing 'x' and 'y'.

10.2 Implementation and Optimizations

The sets are typically represented as rooted trees stored in a parent array. Without optimizations, the trees can become very tall, leading to $O(n)$ per operation. Two key optimizations reduce this cost dramatically:

1. **Union by Rank/Size:** When merging two trees, always attach the root of the smaller tree to the root of the larger tree. This keeps the trees shallow.
2. **Path Compression:** During a 'Find(x)' operation, make every node on the path from 'x' to the root point directly to the root. This flattens the tree structure over time.

With both optimizations, a sequence of m operations on n elements takes $O(m \cdot \alpha(n))$ time, where $\alpha(n)$ is the extremely slow-growing inverse Ackermann function. For all practical purposes, this makes the amortized time per operation nearly constant.

Chapter 11

Selection Algorithms

The selection problem is to find the k-th smallest element in an unsorted list. A common case is finding the median ($k = n/2$).

11.1 Quickselect

This algorithm is related to Quicksort. It partitions the array around a pivot and then recursively operates on only one of the two partitions—the one that contains the k-th element.

- **Time Complexity:** Average case is $O(n)$. Worst case is $O(n^2)$, but like Quicksort, this can be avoided by using a random pivot.

11.2 Median of Medians Algorithm

This is a deterministic selection algorithm that guarantees a worst-case $O(n)$ time complexity by cleverly choosing a pivot that is guaranteed to be "good."

1. Divide the array into groups of 5 elements.
2. Find the median of each group.
3. Recursively find the median of these medians. This element is used as the pivot.
4. Partition the array around this pivot and recurse into the correct partition.

This ensures the pivot is not too close to the minimum or maximum, leading to balanced partitions and a linear time guarantee.

Appendix A

The Master Theorem

The Master Theorem provides a "cookbook" method for solving recurrence relations of the form commonly found in divide-and-conquer algorithms:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ is the number of subproblems, $b > 1$ is the factor by which the input size is reduced, and $f(n)$ is the cost of the work done outside the recursive calls (dividing and combining).

To solve the recurrence, we compare $f(n)$ with $n^{\log_b a}$.

1. **Case 1:** If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$. *(The cost is dominated by the leaves of the recursion tree.)*
2. **Case 2:** If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$. *(The cost is spread evenly across the levels of the recursion tree.)*
3. **Case 3:** If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$ and sufficiently large n (the regularity condition), then $T(n) = \Theta(f(n))$. *(The cost is dominated by the work at the root.)*

A.1 Examples

- **Merge Sort:** $T(n) = 2T(n/2) + O(n)$. Here, $a = 2, b = 2$, so $\log_b a = \log_2 2 = 1$. $f(n) = O(n) = \Theta(n^1)$. This is **Case 2**. Thus, $T(n) = \Theta(n^{\log_2 2} \log n) = \Theta(n \log n)$.
- **Binary Search:** $T(n) = T(n/2) + O(1)$. Here, $a = 1, b = 2$, so $\log_b a = \log_2 1 = 0$. $f(n) = O(1) = O(n^{0-\epsilon})$ for $\epsilon = 1$. This is **Case 1**. Thus, $T(n) = \Theta(n^{\log_2 1}) = \Theta(n^0) = \Theta(1)$. Wait, this is not right. The theorem solves for the cost, but the recurrence for binary search is on the depth. Let's re-evaluate: $f(n) = \Theta(n^0)$, so $k = 0$ in the more general form $f(n) = \Theta(n^{\log_b a} \log^k n)$. Here $k = 0$, so $T(n) = \Theta(n^{\log_b a} \log^{k+1} n) = \Theta(n^0 \log^1 n) = \Theta(\log n)$.