Chapter 1-4 from textbook

Table of Contents

Chapter 1 Role of Algorithms
1.1 Algorithms
1.2 Algorithms as a technology
Chapter 2 Getting Started
2.1 Insertion sort
2.2 Analyzing algorithms
2.3 Designing algorithms
Chapter 3 Characterizing Running Times
3.1 O-notation, �-notation, and ‚-notation
3.2 Asymptotic notation: formal deûnitions
3.3 Standard notations and common functions
Chapter 4 Divide and Conquer
4.1 Multiplying square matrices
4.2 Strassen's algorithm for matrix multiplication
4.3 The substitution method for solving recurrences
4.4 The recursion-tree method for solving recurrences
4.5 The master method for solving recurrences
? 4.6 Proof of the continuous master theorem
? 4.7 Akra-Bazzi recurrences
Chapter 5 Probabilistic Analysis and Randomized Algorithms
5.1 The hiring problem
5.2 Indicator random variables
5.3 Randomized algorithms
? 5.4 Probabilistic analysis and further uses of indicator random variables

Chapter 1

1.1 Algorithms

An **algorithm** is any well-defined computational procedure that takes some value, or set of value as **input** and produces some value, or set of values, as **output** in a finite amount of time. An algorithm is thus a sequence of computational steps that transform the input into the output.

They are used to solve specified **computation problems**

One example is a **sorting problem**.

$$\text{Input: a sequence of numbers} \langle a'_1, a'_2 . \dots a'_n \rangle$$

Output: A permutation (reordering) $\langle a'_1, a'_2 \ldots a'_n \rangle$of the input sequence such that $a'_1 \leq a'_2 \leq \ldots a'_n$

An instance is an example.

An algorithm for a computational problem is correct if, for every problem instance provided as input, it halts finishes its computing in finite time and outputs the correct solution to the problem instance. A correct algorithm solves the given computational problem. An incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer

Data Structures

A **data structure** is a way to store and organize data in order to facilitate access and modifications.

This is important to part of algorithm design and highly dependent and changes for each problem

Examples:

1. **Arrays**: Fixed-size collections of elements, ideal for random access but inefficient for insertions and deletions.
2. **Linked Lists**: Dynamic collections where each element points to the next, enabling efficient insertions and deletions but slower access.
3. **Graphs**: Collections of vertices and edges, used to represent relationships (e.g., networks).
4. **Hash Tables**: Provide near-instant lookup using hash functions, but require efficient collision resolution.

Each data structure has strengths and weaknesses depending on the problem requirements.

1.2 Algorithms as a Technology

If computers were infinitely fast and memory were free, would you still need to study algorithms? The answer is yes. Without algorithms, you cannot guarantee:

1. That your solution halts (completes).
2. That your solution provides the correct result.

Since computing resources are finite, studying algorithms ensures optimal use of:

- **Time**: How long the computation takes.
- **Space**: How much memory the computation uses.

Efficiency

Algorithms designed to solve the same problem can vary greatly in efficiency. Efficiency is typically measured in:

1. **Time Complexity**: How the running time scales with input size.
2. **Space Complexity**: How memory usage scales with input size.

Example: Sorting Algorithms

- **Insertion Sort**: Time complexity .
- **Merge Sort**: Time complexity .

For large inputs, merge sort significantly outperforms insertion sort due to its slower growth rate.

Concrete Example:

- Suppose Computer A runs insertion sort at 10 billion instructions/second, and Computer B runs merge sort at 10 million instructions/second.
- Sorting 10 million numbers:
  - Computer A takes seconds.
  - Computer B takes seconds.

Despite being slower in raw computing power, Computer B's efficient algorithm (merge sort) dramatically outperforms Computer A for large inputs.

Algorithms in Modern Technology

Algorithms are fundamental to many modern technologies:

- **Networking**: Routing algorithms ensure efficient data transmission.
- **Cryptography**: Algorithms secure online communication (e.g., RSA).
- **Machine Learning**: Algorithms drive pattern recognition and decision-making.

While machine learning automates some tasks, explicitly designed algorithms remain indispensable for well-understood problems.

Chapter 2 Getting Started

2.1 Insertion Sort

Our first algorithm, **insertion sort**, solves the sorting problem introduced in Chapter 1.

Problem Statement

- **Input**: A sequence of numbers .
- **Output**: A permutation (reordering) such that .

Insertion sort builds the sorted array one element at a time, maintaining a sorted subarray at each step. It works similarly to sorting playing cards in your hand:

1. Start with the first card as the sorted subarray.
2. Pick the next card and insert it into its correct position in the sorted subarray.
3. Repeat until all cards are sorted.

Key Concepts in Insertion Sort

- **Keys**: The elements being sorted are referred to as keys. For instance, in a student database, keys could represent roll numbers, with additional information (satellite data) accompanying them.
- **Satellite Data**: Information associated with keys that move along with them during sorting. For example, if sorting students by ID, their grades or names are satellite data that stay with the IDs.
- **Loop Invariant**: A formal assertion that remains true throughout specific points in a program, particularly at the start of each iteration of a loop. In insertion sort:
  - Subarray is always sorted before processing the -th element.

Pseudocode

```
INSERTION-SORT(A):
  for i = 2 to length(A):
    key = A[i]
    j = i - 1
    while j > 0 and A[j] > key:
      A[j + 1] = A[j]
      j = j - 1
    A[j + 1] = key
```

Pseudocode Conventions

1. **Indentation**: Indicates block structure, making the code easier to read and understand.
2. **Loop Constructs**: Similar to those in common programming languages (e.g., C, Python):
   - `for` loops iterate a fixed number of times.
   - `while` loops iterate until a condition is no longer true.
3. **Array Indexing**: Arrays are generally 1-indexed unless explicitly stated. For example, `A[i]` refers to the -th element of the array.
4. **Comments**: Lines starting with `//` explain the code, clarifying logic or purpose.

5. **Variables**: Local to the given procedure unless otherwise noted. Global variables are avoided.
6. **Return Values**: Procedures explicitly return a value, or multiple values, to the caller.

Loop Invariant Explanation

To prove correctness, consider the loop invariant at the start of each iteration:

1. **Initialization**: Before the first iteration, the subarray (a single element) is trivially sorted.
2. **Maintenance**: If the subarray is sorted at the start of the iteration, inserting into its correct position ensures that is sorted at the end.
3. **Termination**: When the loop ends (), the entire array is sorted.

Example Walkthrough

Given the input :

1. **Initial Array**: .
2. Insert 2 into the sorted subarray : .
3. Insert 4: .
4. Insert 6: .
5. Insert 1: .
6. Insert 3: .

Analyzing Time Complexity

- **Best Case**: Input is already sorted. Only comparisons are made, so the time complexity is .
- **Worst Case**: Input is sorted in reverse order. In this case, comparisons are made, resulting in time complexity.
- **Average Case**: Input elements are randomly ordered. On average, insertion sort performs comparisons.

Insertion sort is efficient for small inputs or nearly sorted data due to its simplicity and low overhead.

2.2 Analyzing Algorithms

Analyzing an **algorithm** involves predicting the computational resources required for its execution. These resources include **time complexity** (how long it takes to run) and **space complexity** (how much memory it requires). Most often, the primary focus is on computational time.

Importance of Analyzing Algorithms

By analyzing different algorithms, we can:

- Identify the most efficient algorithm for a given problem.
- Compare multiple candidate algorithms based on their performance.
- Rule out inferior algorithms that take excessive time or space.
- Predict performance across different input sizes.

Before analyzing an algorithm, we need a **computational model** to express the costs associated with its execution. The primary model used in this book is the **random-access machine (RAM) model**.

The RAM Model

The **RAM model** assumes that a program executes instructions sequentially, with each instruction taking a constant amount of time. There are no concurrent operations, and each basic operation (such as addition, multiplication, data access, or control flow) has an equal cost.

Assumptions of the RAM Model:

- **Each basic operation (e.g., arithmetic, data access) takes constant time.**
- **Memory access time is uniform.** Retrieving an element from an array or storing a value in memory has the same cost, regardless of position.
- **No parallel processing.** The model assumes a single processor executing one instruction at a time.
- **Realistic instruction set.** The RAM model includes common operations found in real-world computers (addition, multiplication, conditional branching, etc.), but excludes unrealistic operations (e.g., a single instruction that sorts an array instantly).

This model provides a useful abstraction for algorithm analysis because it reflects how algorithms would be implemented in practice.

Analysis of Insertion Sort

One way to determine the running time of an algorithm is to implement it in a programming language and measure its execution time. However, this approach has limitations:

- The timing results depend on the hardware, compiler, and system configuration.
- The same algorithm may run differently on different computers.
- Different implementations of the same algorithm may produce varying results.

Instead, we analyze an algorithm **theoretically** by:

1. Counting the number of basic operations it performs.
2. Expressing this count as a function of the input size .

This approach allows us to compare algorithms independent of machine-specific details.

Running Time and Input Size

The running time of an algorithm depends on the **input size**, which varies based on the problem:

- **Sorting problem:** The input size is the number of elements to be sorted.
- **Graph algorithms:** The input size may be measured by the number of vertices and edges.
- **Matrix operations:** The input size could be the number of rows and columns.

For insertion sort, the input size is the number of elements in the array.

Order of Growth and Asymptotic Notation

To simplify analysis, we focus on the **order of growth**, which describes how the running time increases as input size grows.

1. **Θ (Theta) Notation:** Provides an asymptotically tight bound. If an algorithm is , then it has both an upper and lower bound of .
2. **O (Big-O) Notation:** Provides an upper bound on the running time, representing the worst-case growth rate.
3. **Ω (Omega) Notation:** Provides a lower bound, representing the best-case growth rate.

Usage in Practice:

- Big-O notation is commonly used to describe the worst-case performance, ensuring the algorithm will not perform worse than the given bound.
- For algorithms like insertion sort, the choice between best, average, and worst cases depends on the expected input distribution.

Counting Operations in Insertion Sort

We analyze **Insertion Sort** by examining how many times each line of pseudocode executes:

```
INSERTION-SORT(A, n)
  for i = 2 to n:              # Executes (n - 1) times
    key = A[i]                # Executes (n - 1) times
    j = i - 1                 # Executes (n - 1) times
    while j > 0 and A[j] > key:  # Varies based on input
      A[j + 1] = A[j]         # Executes at most (i - 1) times per i
      j = j - 1               # Executes at most (i - 1) times per i
    A[j + 1] = key            # Executes (n - 1) times
```

The **while loop** is the key to determining the running time. In the best case, the while loop runs **once per iteration**, while in the worst case, it runs **times**.

The total number of operations follows:

1. **Best Case (Already Sorted Input):**
   - The `while` loop condition `j > 0 and A[j] > key` fails immediately for every iteration of the outer `for` loop. Hence, only **one comparison** is made per iteration.
   - Total comparisons:

     $$n-1$$

   - Total assignments:

     $$n-1$$

   - **Total operations:**

     $$2(n-1)$$

     , which is

     $$O(n)$$

     .
2. **Worst Case (Reverse Sorted Input):**
   - The `while` loop runs for

     $$i-1$$

     iterations for each

     $$i$$

     , shifting all elements greater than `key`.
   - Total comparisons:

     $$\sum_{i=2}^{n}(i-1)=\sum_{i=1}^{n-1}i=\frac{n(n-1)}{2}$$

- Total assignments (shifting + placing `key`):

$$\sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2}$$

- **Total operations:**

$$n(n-1) = O(n^2)$$

.

3. **Average Case (Random Input):**
   - On average, the `while` loop runs half as many iterations as in the worst case.
   - Total comparisons:

$$\frac{1}{2}\sum_{i=2}^{n}(i-1) = \frac{1}{2}\cdot\frac{n(n-1)}{2} = \frac{n^2}{4}$$

   - Total assignments:

$$\frac{1}{2}\sum_{i=2}^{n}(i-1) = \frac{n^2}{4}$$

   - **Total operations:**

$$\frac{n^2}{2} = O(n^2)$$

.

These calculations illustrate how insertion sort performs across different input scenarios, using summation notation to capture the number of operations. The worst-case scenario highlights its inefficiency for large inputs, while its simplicity makes it suitable for small or nearly sorted datasets.

Key Takeaways

- **Insertion Sort** has a worst-case time complexity of , making it inefficient for large inputs.
- It performs well for **small** or **nearly sorted** inputs.
- Algorithm analysis helps in selecting the best approach by focusing on asymptotic performance rather than raw execution time.

2.3 Designing Algorithms

The Incremental and Divide-and-Conquer Methods

Algorithms can be designed using various strategies. Two common approaches are:

1. **Incremental Method** - Elements are processed one by one and inserted into their correct position within a growing sorted subarray. Example: **Insertion Sort**.
2. **Divide-and-Conquer Method** - The problem is divided into smaller subproblems, solved recursively, and then combined to produce the final result. Example: **Merge Sort**.

2.3.1 The Divide-and-Conquer Method

The **divide-and-conquer** strategy breaks down a problem into smaller instances of itself, solves them recursively, and combines the results to form the final solution. It follows these three steps:

1. **Divide** - Split the problem into smaller subproblems.
2. **Conquer** - Solve the subproblems recursively.
3. **Combine** - Merge the solutions of subproblems to obtain the final solution.

One example of a divide-and-conquer algorithm is **Merge Sort**, which sorts an array by recursively dividing it into halves, sorting each half, and merging them.

Merge Sort Algorithm

```
MERGE-SORT(A, p, r)
1. if p ≥ r
2.    return  // base case: array of one element is already sorted
3. q = ⌊(p + r) / 2⌋  // midpoint of A[p:r]
4. MERGE-SORT(A, p, q)  // recursively sort left half
5. MERGE-SORT(A, q + 1, r)  // recursively sort right half
6. MERGE(A, p, q, r)  // merge sorted halves
```

Merge Procedure
MERGE(A, p, q, r)

1. n_L = q - p + 1 // length of left subarray
2. n_R = r - q // length of right subarray
3. let L[0:n_L - 1] and R[0:n_R - 1] be new arrays
4. for i = 0 to n_L - 1
5. L[i] = A[p + i] // copy left subarray
6. for j = 0 to n_R - 1
7. R[j] = A[q + 1 + j] // copy right subarray
8. i = 0, j = 0, k = p // initialize pointers
9. while i < n_L and j < n_R

```
if L[i] ≤ R[j]
    A[k] = L[i]
    i = i + 1
else
    A[k] = R[j]
    j = j + 1
k = k + 1
while i < n_L // copy remaining L elements
A[k] = L[i]
i = i + 1
k = k + 1
while j < n_R // copy remaining R elements
A[k] = R[j]
j = j + 1
k = k + 1
```

The running time of **Merge Sort** follows the recurrence relation:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n \leq 1 \\ 2T(n/2) + \Theta(n), & \text{otherwise} \end{cases}$$

Steps in the Recursion Tree: 1. **Divide Step**: Finding the midpoint of the array takes **Θ(1)** time. 2. **Conquer Step**: Recursively solving two subproblems, each of size **n/2**. 3. **Combine Step**: Merging two sorted subarrays takes **Θ(n)** time. #### Expanding the Recursion Tree: - **Level 0**: The top level involves dividing the array and merging the results: **Θ(n)**. - **Level 1**: Two subproblems of size **n/2**, each contributing **Θ(n/2)**. Total: **2 × Θ(n/2) = Θ(n)**. - **Level 2**: Four subproblems of size **n/4**, each contributing **Θ(n/4)**. Total: **4 × Θ(n/4) = Θ(n)**. - **Level (\log n)**: At the last level, there are (2^{\log n} = n) subproblems, each of size 1. Total: **n × Θ(1) = Θ(n)**. #### Total Work: Summing across all levels of the recursion tree:

$$T(n) = \sum_{i=0}^{\log n} \Theta(n) = \Theta(n \log n)$$

Summary: - **Divide Step**: (\Theta(1)) - **Conquer Step**: (2T(n/2)) - **Combine Step**: (

\Theta(n)) Thus, the total time complexity of Merge Sort is:

$$T(n) = \Theta(n \log n)$$

Key Takeaway: Merge Sort achieves an efficient worst-case time complexity of **Θ(n \log n)**, making it more suitable for large inputs than quadratic algorithms such as **Insertion Sort**.

2.3.2 Analyzing Divide-and-Conquer Algorithms

A **recurrence equation** describes the running time of recursive algorithms. For merge sort, the recurrence is:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n \leq 1 \\ 2T(n/2) + \Theta(n), & \text{otherwise} \end{cases}$$

Using the Recursion Tree Method

To analyze this recurrence, we expand it into a **recursion tree**:

1. At the top level:
   - The cost of dividing the array and merging the results is **Θ(n)**.
2. At the second level:
   - Two subproblems of size **n/2** are solved, each contributing **Θ(n/2)**.
   - Total work at this level: **2 × Θ(n/2) = Θ(n)**.
3. At the third level:
   - Four subproblems of size **n/4** are solved, each contributing **Θ(n/4)**.
   - Total work at this level: **4 × Θ(n/4) = Θ(n)**.

Generalization Across All Levels

- At each level of the tree, the total work is **Θ(n)**.
- The recursion continues for **log(n) + 1** levels (since the problem size halves at each level).
- Total work across all levels:

$$T(n) = \sum_{i=0}^{\log n} \Theta(n) = \Theta(n \log n)$$

Key Insights

- The **divide** step contributes **Θ(1)**.
- The **conquer** step involves solving two subproblems of size **n/2**, each contributing **T(n/2)**.
- The **combine** step (merging) contributes **Θ(n)**.

Thus, the total running time is:

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$$

Key Takeaways

- **Divide-and-conquer** algorithms are powerful tools for solving recursive problems.
- **Merge Sort** achieves a worst-case time complexity of **Θ(n \log n)**, making it much more efficient than quadratic algorithms like **Insertion Sort** for large input sizes.
- Recurrence relations and the recursion tree method provide systematic ways to analyze these algorithms.

Chapter 3: Characterizing Running Times

Introduction to Characterizing Running Times

The **order of growth** of an algorithm is a simplified way to describe how its running time increases as input size grows. This chapter explores **asymptotic notations**, which provide a mathematical framework for analyzing and comparing algorithm efficiency.

Key concepts covered include:

- The **upper bounds** (Big-O notation),
- **Lower bounds** (Omega notation), and
- **Tight bounds** (Theta notation).

3.1 O-notation, Ω-notation, and Θ-notation

O-notation

Big-O notation describes an **upper bound** on the running time of an algorithm. If a function (f(n)) is (O(g(n))), it grows no faster than (g(n)), up to a constant factor, as (n \to \infty).

Formally:

$$f(n) = O(g(n)) \quad \text{if there exist constants } c > 0 \text{ and } n_0 > 0 \text{ such that } 0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0.$$

Example:

- (3n^2 + 5n + 7 = O(n^2)), because higher-order terms dominate growth as (n) becomes large.

Ω-notation

Omega notation provides a **lower bound**. If (f(n)) is (\Omega(g(n))), (f(n)) grows at least as fast as (g(n)) for large (n).

Formally:

$$f(n) = \Omega(g(n)) \quad \text{if there exist constants } c > 0 \text{ and } n_0 > 0 \text{ such that } 0 \leq c \cdot g(n) \leq f(n), \forall n \geq n_0.$$

Example:

- (3n^2 + 5n + 7 = \Omega(n^2)), because (3n^2) dominates growth as (n) becomes large.

Θ-notation

Theta notation provides a **tight bound**. If (f(n)) is (\Theta(g(n))), then (f(n)) grows **exactly** at the rate of (g(n)) for large (n).

Formally:

$$f(n) = \Theta(g(n)) \quad \text{if there exist constants } c_1, c_2 > 0 \text{ and } n_0 > 0 \text{ such that } c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n),$$

Example:

- (3n^2 + 5n + 7 = \Theta(n^2)), because (3n^2) bounds the growth from above and below.

Example: Insertion Sort

Revisiting **Insertion Sort**, we analyze its running time using asymptotic notation:

```
INSERTION-SORT(A, n):
  for i = 2 to n:
    key = A[i]
    j = i - 1
    while j > 0 and A[j] > key:
      A[j + 1] = A[j]
      j = j - 1
    A[j + 1] = key
```

Best Case Analysis of Insertion Sort

- The **best case** occurs when the input array is already sorted.
- In this scenario, the `while` loop condition `A[j] > key` fails **immediately** in each iteration.
- This means that each iteration of the `for` loop performs **only one** comparison and **one** assignment.

Counting Operations

1. **Comparisons:**
   - The `while` loop executes **once** per iteration of the `for` loop.
   - Total comparisons:

$$\sum_{i=2}^{n} 1 = n - 1.$$

2. **Assignments:**
   - Each iteration assigns `key` to its position exactly once.
   - Total assignments: (n - 1).
3. **Total Operations:**
   - The total number of operations (comparisons + assignments) is:

$$T(n) = 2(n-1).$$

   - Thus, the best-case time complexity is:

$$\Theta(n).$$

Worst Case Analysis of Insertion Sort

- The **worst case** occurs when the input array is sorted in **reverse** order.
- Each new element has to be **compared and shifted past all previous elements** before being inserted in the correct position.

Counting Operations

1. **Comparisons:**
   - The `while` loop executes **(i - 1)** times for each (i).
   - Total comparisons:

$$\sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2}.$$

2. **Assignments:**

- Each shift operation moves an element forward in the array.
- Since every element moves (i - 1) times, the total assignments follow the same pattern:

$$\sum_{i=2}^{n}(i-1)=\frac{n(n-1)}{2}.$$

3. **Total Operations:**
   - The worst-case total number of operations:

$$T(n)=n(n-1).$$

   - This results in a worst-case time complexity of:

$$\Theta(n^2).$$

Average Case Analysis of Insertion Sort

- The **average case** occurs when the input array is randomly ordered.
- On average, each element **shifts past half** of the previous elements.

Counting Operations

1. **Comparisons:**
   - The `while` loop executes **about half as many** times as in the worst case.
   - Average comparisons:

$$\sum_{i=2}^{n}\frac{i-1}{2}=\frac{1}{2}\cdot\frac{n(n-1)}{2}.$$

2. **Assignments:**
   - The number of shifts follows the same pattern as the comparisons.
   - Total assignments:

$$\sum_{i=2}^{n}\frac{i-1}{2}=\frac{1}{2}\cdot\frac{n(n-1)}{2}.$$

3. **Total Operations:**
   - The average-case total number of operations:

$$T(n)=\Theta(n^2).$$

Formal Definitions of Asymptotic Notation

For a function (f(n)), asymptotic notation is formally defined as:

1. **Big-O Notation (Upper Bound)**:
   - (f(n)) grows **at most** as fast as (g(n)).
   - Formal definition:

$$f(n) = O(g(n)) \quad \Leftrightarrow \quad \exists c, n_0 > 0 \text{ such that } f(n) \leq c \cdot g(n), \forall n \geq n_0.$$

2. **Omega Notation (Lower Bound)**:
   - (f(n)) grows **at least** as fast as (g(n)).
   - Formal definition:

$$f(n) = \Omega(g(n)) \quad \Leftrightarrow \quad \exists c, n_0 > 0 \text{ such that } f(n) \geq c \cdot g(n), \forall n \geq n_0.$$

3. **Theta Notation (Tight Bound)**:
   - (f(n)) grows **exactly** at the rate of (g(n)).
   - Formal definition:

$$f(n) = \Theta(g(n)) \quad \Leftrightarrow \quad \exists c_1, c_2, n_0 > 0 \text{ such that } c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0.$$

Key Takeaways

- **Best-case complexity**: ($\Theta(n)$) (already sorted array).
- **Worst-case complexity**: ($\Theta(n^2)$) (reverse sorted array).
- **Average-case complexity**: ($\Theta(n^2)$) (random order).
- **Asymptotic notation** simplifies algorithm comparison by focusing on growth rates.

3.2 Asymptotic Notation: Formal Definitions

Asymptotic notation provides a mathematical way to describe the growth rate of a function as the input size approaches infinity. It allows us to analyze an algorithm's efficiency without focusing on machine-specific details.

Formal Definitions of Asymptotic Notation

For a function (f(n)), asymptotic notation defines bounds on its growth rate using **Big-O (upper bound), Omega (lower bound), and Theta (tight bound)**.

Big-O Notation (Upper Bound)

Big-O notation describes the **worst-case growth rate** of a function. It provides an upper bound, meaning that for sufficiently large (n), the function does not grow faster than a given rate.

Definition

A function (f(n)) is said to be **(O(g(n)))** if there exist positive constants (c) and (n_0) such that:

$$f(n) \leq c \cdot g(n) \quad \text{for all } n \geq n_0.$$

This means that beyond a certain point (n_0), (f(n)) grows **at most** as fast as (g(n)), up to a constant factor.

Example

If $(f(n) = 3n^2 + 5n + 7)$, we can say:

$$f(n) = O(n^2)$$

since there exists a constant (c) such that (3n^2 + 5n + 7 \leq c \cdot n^2) for sufficiently large (n).

Omega Notation (Lower Bound)

Omega notation provides a **lower bound** on the growth of a function. It ensures that for sufficiently large (n), the function grows **at least** as fast as a given rate.

Definition

A function (f(n)) is said to be **(\Omega(g(n)))** if there exist positive constants (c) and (n_0) such that:

$$f(n) \geq c \cdot g(n) \quad \text{for all } n \geq n_0.$$

This means that beyond a certain point (n_0), (f(n)) grows **at least** as fast as (g(n)), up to a constant factor.

Example

For $(f(n) = 3n^2 + 5n + 7)$, we can say:

$$f(n) = \Omega(n^2)$$

since there exists a constant (c) such that $(3n^2 + 5n + 7 \geq c \cdot n^2)$ for sufficiently large (n).

Theta Notation (Tight Bound)

Theta notation provides a **tight bound**, meaning that the function grows **both at most and at least** as fast as the given rate.

Definition

A function (f(n)) is said to be **(\Theta(g(n)))** if there exist positive constants $(c_1, c_2,)$ and (n_0) such that:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \text{for all } n \geq n_0.$$

This means that beyond a certain point (n_0), (f(n)) grows **at the same rate** as (g(n)), up to constant factors.

Example

For $(f(n) = 3n^2 + 5n + 7)$, we can conclude:

$$f(n) = \Theta(n^2)$$

because (f(n)) is **both** (O(n^2)) and (\Omega(n^2)), meaning it grows **exactly** at the rate of (n^2), ignoring constant factors.

Asymptotic Notation and Running Time

Asymptotic notation is crucial in algorithm analysis since it allows us to describe performance across different input sizes without needing exact execution times.

Order of Growth Hierarchy

Common functions used in asymptotic notation, ordered from fastest-growing to slowest-growing:

- **Factorial**: $(O(n!))$
- **Exponential**: $(O(2^n))$
- **Polynomial**: $(O(n^c))$ for some constant (c)
- **Logarithmic**: $(O(\log n))$
- **Constant**: $(O(1))$

For large (n), algorithms with faster-growing functions will have much worse performance.

Graphical Interpretation

To visualize asymptotic behavior:

- $(O(n))$ means the function grows at most linearly.

- $(O(n^2))$ means it grows quadratically.
- $(O(\log n))$ means it grows very slowly compared to polynomial functions.

Key Takeaways

- **Big-O** provides an **upper bound** on growth.
- **Omega** provides a **lower bound** on growth.
- **Theta** provides a **tight bound**, meaning both an upper and lower bound exist.
- Asymptotic notation helps analyze algorithm **efficiency** without needing machine-specific details.
- Growth rates follow a hierarchy, from **fastest-growing (factorial, exponential)** to **slowest-growing (logarithmic, constant)**.

3.3 Standard Notations and Common Functions

This section reviews standard mathematical notations and common functions, which are essential when analyzing the **growth rate** of algorithms.

Mathematical Properties of Functions

Monotonicity

A function is **monotonically increasing** if:

$$f(x) \leq f(y) \quad \text{for all } x \leq y$$

A function is **monotonically decreasing** if:

$$f(x) \geq f(y) \quad \text{for all } x \leq y$$

If the inequality is **strict**, we say the function is **strictly increasing** or **strictly decreasing**.

Floors and Ceilings

- The **floor function** $\lfloor x \rfloor$ returns the greatest integer less than or equal to x.
- The **ceiling function** $\lceil x \rceil$ returns the smallest integer greater than or equal to x.

For any real number x:

$$\lfloor x \rfloor \le x \le \lceil x \rceil$$

For any integers a, b:

$$\left\lfloor \frac{a}{b} \right\rfloor \le \frac{a}{b} \le \left\lceil \frac{a}{b} \right\rceil$$

Modular Arithmetic

For any integer a and positive integer n, the value $a \mod n$ is the remainder when dividing a by n:

$$a \mod n = a - n\lfloor a/n \rfloor$$

Some useful modular properties:

1. If $a \equiv b \mod n$, then $a + c \equiv b + c \mod n$.
2. If $a \equiv b \mod n$, then $ac \equiv bc \mod n$.
3. If $a \equiv b \mod n$ and $c \equiv d \mod n$, then $a + c \equiv b + d \mod n$.

Polynomials

A **polynomial function** of degree d is defined as:

$$p(n) = a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0$$

where the coefficients $a_0, a_1, \ldots, a_d$ are constants.

- If $a_d > 0$, the function is **monotonically increasing**.
- If $a_d < 0$, the function is **monotonically decreasing**.
- If $d > 0$, the **dominant term** is $a_d n^d$, which determines the asymptotic growth rate.

Thus, for large n:

$$p(n) = \Theta(n^d)$$

Exponentials

For all real numbers x:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

This implies:

1. $e^0 = 1$
2. $e^x e^y = e^{x+y}$
3. $e^{\ln x} = x$

A function that grows **exponentially** has the form:

$$f(n) = O(c^n), \quad \text{for some } c > 1$$

This means exponential functions grow **faster than polynomial functions**.

Logarithms

We define common logarithms:

- $\log_2 x$ (binary logarithm)
- $\ln x = \log_e x$ (natural logarithm)
- $\log_{10} x$ (decimal logarithm)

Using the **change of base formula**:

$$\log_a x = \frac{\log_b x}{\log_b a}$$

For growth rate comparison:

- $O(n)$ grows **faster** than $O(\log n)$.
- $O(n^2)$ grows **faster** than $O(n \log n)$.

For large n, logarithmic functions grow **very slowly**.

Factorials

The **factorial function** is defined as:

$$n! = 1 \cdot 2 \cdot 3 \cdots n$$

Using **Stirling's Approximation** for large n:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

This shows that factorials grow **faster than exponentials**.

Fibonacci Numbers

The Fibonacci sequence is defined as:

$$F_0 = 0, \quad F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \quad \text{for } n \geq 2$$

Using the **golden ratio** ϕ, the Fibonacci numbers can be approximated by:

$$F_n \approx \frac{\phi^n}{\sqrt{5}}, \quad \text{where } \phi = \frac{1+\sqrt{5}}{2}$$

This means Fibonacci numbers grow **exponentially**.

Key Takeaways

- **Polynomials** grow at a rate of $O(n^d)$, with the highest-degree term dominating.
- **Exponentials** ($O(c^n)$) grow faster than any polynomial function.
- **Logarithms** grow slowly and appear in many divide-and-conquer algorithms.
- **Factorials** grow **even faster** than exponentials.
- **Fibonacci numbers** follow an **exponential growth pattern**.

Chapter 4: Divide and Conquer

4.1 Multiplying Square Matrices

Matrix multiplication is a fundamental operation in many **scientific computations**, **graphics processing**, and **machine learning**. The **divide-and-conquer approach** provides a recursive method to perform matrix multiplication more efficiently than the naive method.

Definition of Matrix Multiplication

Given two $n \times n$ matrices A and B, the product C is also an $n \times n$ matrix where each entry $C[i,j]$ is computed as:

$$C[i,j] = \sum_{k=1}^{n} A[i,k] \cdot B[k,j]$$

- The **entry** at row i and column j of C is computed as the **dot product** of the i^{th} row of matrix A and the j^{th} column of matrix B.
- This operation requires $O(n)$ **multiplications** per entry, leading to a total complexity of $O(n^3)$ using the naive method.

Naïve Approach: Iterative Matrix Multiplication

The simplest method to multiply matrices follows a **triple nested loop** structure:

```
Algorithm: Standard Matrix Multiplication

1. MATRIX-MULTIPLY(A, B, C)
2.   for i = 1 to n:
3.      for j = 1 to n:
4.         C[i, j] = 0
5.         for k = 1 to n:
6.            C[i, j] = C[i, j] + A[i, k] * B[k, j]
```

Analysis of Time Complexity

- There are n^2 **elements** in matrix **C**.
- Each element requires n **multiplications** and n **additions**.
- **Total number of operations**:
  $O(n^2) \times O(n) = O(n^3)$
- The naive method is **inefficient for large matrices**.

Divide-and-Conquer Approach

Instead of computing the $n \times n$ matrix product directly, divide-and-conquer recursively splits the problem into smaller subproblems.

1. **Step 1: Divide Matrices**
   Matrices A, B, and C are split into four $n/2 \times n/2$ submatrices:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

   Each quadrant C_{ij} is computed recursively:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

   Each multiplication is now a problem of size $n/2 \times n/2$.
2. **Step 2: Combine Submatrices**
   Combine the results of the recursive multiplications to form C.

Recurrence Relation

For each recursive level:

- 8 recursive subproblems of size $n/2 \times n/2$.
- Combine results in $O(n^2)$ time (matrix addition).

The recurrence is:

$$T(n) = 8T(n/2) + O(n^2)$$

Using the **Master Theorem**:

- $a = 8$ (number of subproblems),
- $b = 2$ (subproblem size divisor),
- $d = 2$ (cost of merging subproblems).

Since $d = \log_2 a = 3$, the total complexity is:

$$T(n) = O(n^3)$$

Thus, the **divide-and-conquer approach** has the same asymptotic complexity as the naive method.

Strassen's Algorithm

Key Improvement

Volker Strassen (1969) reduced the number of recursive multiplications from 8 to 7, improving the asymptotic complexity.

Step 1: Define Auxiliary Matrices

Strassen's algorithm computes 7 auxiliary matrices:

$$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$P_2 = (A_{21} + A_{22})B_{11}$$

$$P_3 = A_{11}(B_{12} - B_{22})$$

$$P_4 = A_{22}(B_{21} - B_{11})$$

$$P_5 = (A_{11} + A_{12})B_{22}$$

$$P_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$P_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

Step 2: Combine Results

Using these 7 products, Strassen reconstructs the quadrants of C:

$$C_{11} = P_1 + P_4 - P_5 + P_7$$
$$C_{12} = P_3 + P_5$$
$$C_{21} = P_2 + P_4$$
$$C_{22} = P_1 - P_2 + P_3 + P_6$$

Recurrence Relation

The recurrence for Strassen's algorithm is:

$$T(n) = 7T(n/2) + O(n^2)$$

Using the **Master Theorem**:

- $a = 7$, $b = 2$, $d = 2$.
- Since $d < \log_2 a \approx 2.81$, the total complexity is:

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

Advantages

- Reduces multiplication count.
- Faster for large matrices.

Disadvantages

- Increased additions and subtractions ($O(n^2)$ per level).
- Potential numerical stability issues.

Comparison of Methods

Method	Multiplications	Complexity
Naive	$O(n^3)$	$O(n^3)$
Divide-and-Conquer	$O(n^3)$	$O(n^3)$
Strassen's Algorithm	$O(n^{2.81})$	$O(n^{2.81})$

Strassen's algorithm is suitable for large matrices where fewer multiplications outweigh the cost of additional additions.

Key Takeaways

- Divide-and-conquer reduces problem size but does not improve naive complexity.
- Strassen’s algorithm achieves $O(n^{2.81})$ complexity by reducing multiplications.
- Practical applications include **machine learning**, **scientific computing**, and **graphics**.

4.2 Strassen’s Algorithm for Matrix Multiplication

Strassen’s algorithm is an advanced **divide-and-conquer** approach that improves upon standard **matrix multiplication** by reducing the number of recursive multiplications.

Key Idea

Instead of performing **8 recursive multiplications** in the divide-and-conquer approach, Strassen's method **reduces it to 7 multiplications**, improving the time complexity.

The standard divide-and-conquer method follows the recurrence:

$$T(n) = 8T(n/2) + O(n^2)$$

which simplifies to:

$$T(n) = O(n^3)$$

Strassen’s method improves this by using **7 multiplications** instead:

$$T(n) = 7T(n/2) + O(n^2)$$

which leads to a final complexity of:

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

Step 1: Matrix Partitioning

Given two $n \times n$ matrices A and B, we divide them into **four** $n/2 \times n/2$ submatrices:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

The result matrix C is also partitioned in the same way:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Step 2: Compute the 7 Auxiliary Matrices

Strassen introduced **7 key matrix multiplications** instead of 8:

$$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$P_2 = (A_{21} + A_{22})B_{11}$$

$$P_3 = A_{11}(B_{12} - B_{22})$$

$$P_4 = A_{22}(B_{21} - B_{11})$$

$$P_5 = (A_{11} + A_{12})B_{22}$$

$$P_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$P_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

Each P_i is computed using **only one recursive multiplication**.

Step 3: Compute the Resulting Matrix

Using these **7 products**, we reconstruct the final result matrix C:

$$C_{11} = P_1 + P_4 - P_5 + P_7$$
$$C_{12} = P_3 + P_5$$
$$C_{21} = P_2 + P_4$$
$$C_{22} = P_1 - P_2 + P_3 + P_6$$

This reduces the number of **recursive multiplications** from **8** to **7**, achieving an improved time complexity.

Analysis of Time Complexity

Strassen's algorithm follows the recurrence relation:

$$T(n) = 7T(n/2) + O(n^2)$$

Applying the **Master Theorem**:

- $a = 7$ (number of recursive calls)
- $b = 2$ (dividing problem size by 2)
- $d = 2$ (cost of merging subproblems)

Since $d < \log_2 7 \approx 2.81$, the recurrence solves to:

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

This is **asymptotically faster** than the standard $O(n^3)$ complexity.

Comparison of Methods

Method	Multiplications	Complexity
Naive Algorithm	$O(n^3)$	$O(n^3)$
Divide-and-Conquer	$O(n^3)$	$O(n^3)$
Strassen's Algorithm	$O(n^{2.81})$	$O(n^{2.81})$

Advantages of Strassen's Algorithm

✅ **Fewer recursive multiplications** (7 instead of 8).
✅ **Improved time complexity** ($O(n^{2.81})$ vs $O(n^3)$).
✅ Useful for **large matrices** where multiplication dominates the computation.

Disadvantages of Strassen's Algorithm

❌ **Increased additions and subtractions** ($O(n^2)$ extra operations per step).
❌ **Numerical instability** (due to precision loss in floating-point arithmetic).
❌ **Not always practical** for small matrices (naive method often faster due to lower overhead).

Key Takeaways

- Strassen's algorithm is an optimized divide-and-conquer method for **matrix multiplication**.
- It **reduces the number of recursive multiplications** from **8 to 7**, improving complexity to $O(n^{2.81})$.
- More **theoretically efficient** than naive multiplication, but **practical trade-offs** exist.

Strassen's approach paved the way for **even faster** matrix multiplication methods used in **scientific computing**, **machine learning**, and **graphics**.

4.3 The Substitution Method for Solving Recurrences

The **substitution method** is a technique for solving recurrences by making an educated guess about the solution and proving it correct via **mathematical induction**.

Key Steps of the Substitution Method

To solve a recurrence relation using the **substitution method**, follow these steps:

1. **Guess the form of the solution** (based on intuition or previous examples).
2. **Use mathematical induction** to prove that the guess satisfies the recurrence.
3. **Refine the guess if necessary**, adjusting constants or asymptotic bounds.

Example 1: Solving Merge Sort's Recurrence

The standard recurrence for **Merge Sort** is:

$$T(n) = 2T(n/2) + O(n)$$

Step 1: Guess the Solution

A common guess for this type of recurrence is:

$$T(n) = O(n \log n)$$

Step 2: Prove the Guess by Induction

We use **inductive hypothesis**: Assume that for some constant c, the recurrence satisfies:

$$T(n) \leq cn \log n$$

for some sufficiently large n.

Base Case

For **small** n, say $n = 1$, we assume $T(1) = O(1)$, which is trivially true.

Inductive Step

Assume the hypothesis holds for $n/2$:

$$T(n/2) \leq c(n/2) \log(n/2)$$

Substituting this into the recurrence:

$$T(n) = 2T(n/2) + O(n)$$

Expanding using the induction hypothesis:

$$T(n) \leq 2\left(c(n/2)\log(n/2)\right) + O(n)$$

Rewriting:

$$T(n) \leq cn\log(n/2) + O(n)$$

Since $\log(n/2) = \log n - \log 2 = \log n - 1$, we get:

$$T(n) \leq cn(\log n - 1) + O(n)$$

Simplifying:

$$T(n) \leq cn\log n - cn + O(n)$$

For large c, we can absorb the $-cn + O(n)$ into the **big-O notation**, proving:

$$T(n) = O(n\log n)$$

Thus, our **guess was correct**, and the recurrence simplifies to:

$$T(n) = \Theta(n\log n)$$

Example 2: Solving Another Recurrence

Consider the recurrence:

$$T(n) = 3T(n/2) + O(n)$$

Step 1: Guess the Solution

From experience, a reasonable guess is:

$$T(n) = O(n^{\log_2 3})$$

Since $\log_2 3 \approx 1.58$, we suspect:

$$T(n) = O(n^{1.58})$$

Step 2: Prove by Induction

Using the **inductive hypothesis**:

$$T(n/2) \leq c(n/2)^{\log_2 3}$$

Substituting into the recurrence:

$$T(n) \leq 3 \cdot c(n/2)^{\log_2 3} + O(n)$$

Expanding:

$$T(n) \leq 3c \cdot n^{\log_2 3}/2^{\log_2 3} + O(n)$$

Since $3/2^{\log_2 3} = 1$, this simplifies to:

$$T(n) \leq cn^{\log_2 3} + O(n)$$

For large enough c, the additional term $O(n)$ is absorbed, proving:

$$T(n) = O(n^{\log_2 3})$$

Thus, our guess was correct.

Common Pitfalls in the Substitution Method

- **Guessing too loosely**: If the guess is too weak (e.g., $O(n)$ instead of $O(n \log n)$), induction may fail.
- **Failing to handle base cases**: Always verify that the **base case** supports the guessed solution.
- **Ignoring constant factors**: Sometimes, choosing an appropriate **constant** c is necessary for induction to work.

Advantages of the Substitution Method

✅ Works well for many **divide-and-conquer recurrences**.
✅ Helps develop **intuition** about the growth rate of functions.
✅ Allows for **precise bounding** of time complexity.

Disadvantages of the Substitution Method

❌ Requires **guessing the solution first** (which may not always be obvious).
❌ Some recurrences are **hard to simplify** without additional techniques.

Key Takeaways

- The **substitution method** involves **guessing a solution** and proving it via **induction**.
- It works well for many **recursive algorithms**, such as **Merge Sort**.
- The method is **not always straightforward**, but it is a **powerful analytical tool**.

For more complex recurrences, **alternative methods** like the **recursion-tree method** or **Master Theorem** may be more effective.

4.4 The Recursion-Tree Method for Solving Recurrences

The **recursion-tree method** is a powerful tool for analyzing the time complexity of divide-and-conquer algorithms. This method provides an intuitive way to visualize how recursive calls **break down the problem** and **accumulate costs** at each level of recursion.

Key Idea of the Recursion-Tree Method

A recurrence describes how a problem of size n is broken into smaller subproblems. The recursion tree helps us determine the **total work done** by summing the work at each level.

Example 1: Merge Sort Recurrence

Consider the recurrence for **Merge Sort**:

$$T(n) = 2T(n/2) + O(n)$$

Step 1: Expand the Recursion Tree

We **expand the recurrence** by substituting it recursively:

1. The first recursive call breaks $T(n)$ into **two subproblems** of size $n/2$ each.
2. Each subproblem is further split into two subproblems of size $n/4$.
3. This continues **until the base case is reached** (i.e., when $n = 1$).

The tree structure looks like this:

```
               T(n)
           /    |    \
       T(n/2) T(n/2) T(n/2)
      /  \      / \      /  \
  T(n/4) T(n/4) T(n/4) T(n/4)
  /  \   /  \   /  \   /  \
...    ...    ...    ...    ...    ...
```

Step 2: Calculate Work at Each Level

- At **level 0** (root level), we do $O(n)$ work.

- At **level 1**, each of the **two** subproblems does $O(n/2)$ work. Total:

$$2 \cdot O(n/2) = O(n)$$

- At **level 2**, each of the **four** subproblems does $O(n/4)$ work. Total:

$$4 \cdot O(n/4) = O(n)$$

- In general, at **level** i, there are 2^i subproblems, each doing $O(n/2^i)$ work, so:

$$2^i \cdot O(n/2^i) = O(n)$$

Step 3: Determine the Number of Levels

- The recursion stops when $n/2^i = 1$, meaning:

$$i = \log_2 n$$

Step 4: Sum the Work Across Levels

Since each level does $O(n)$ work, and there are $\log n$ levels, the total work is:

$$O(n \log n)$$

Thus, we conclude:

$$T(n) = O(n \log n)$$

This confirms the **Merge Sort complexity**.

Example 2: Recurrence with More Splits

Now consider:

$$T(n) = 3T(n/2) + O(n)$$

Step 1: Expand the Recursion Tree

1. **At level 0**: $O(n)$ work.
2. **At level 1**: 3 subproblems of size $n/2$, each doing $O(n/2)$ work.
3. **At level 2**: 9 subproblems of size $n/4$, each doing $O(n/4)$ work.
4. Continuing until **base case**.

Each level does:

$$3^i \cdot O(n/2^i)$$

Total work at **level** i:

$$3^i \cdot O(n/2^i) = O(n(3/2)^i)$$

The recursion stops at $i = \log_2 n$, so:

$$T(n) = O(n \cdot (3/2)^{\log_2 n})$$

Using exponent rules:

$$(3/2)^{\log_2 n} = n^{\log_2 (3/2)}$$

Thus, the recurrence simplifies to:

$$T(n) = O(n^{\log_2 3})$$

Since $\log_2 3 \approx 1.58$, we conclude:

$$T(n) = O(n^{1.58})$$

Advantages of the Recursion-Tree Method

✅ Provides **visual intuition** for how recursive calls contribute to total cost.
✅ Works well for recurrences that **do not fit** standard forms.
✅ Helps analyze **nested recurrences** (e.g., matrix multiplication).

Disadvantages

❌ Can be tedious for **complex recurrences**.
❌ May require **guessing the depth of recursion**.
❌ Sometimes a **formal recurrence theorem** (like the Master Theorem) is easier.

Key Takeaways

- The **recursion-tree method** expands the recurrence **level by level**.
- Each level contributes **some amount of work**; summing these gives the **total complexity**.
- If the work **remains constant per level**, the depth determines complexity.
- If the work **grows per level**, use geometric series approximations.
- This method is useful when **other techniques** (like the Master Theorem) are difficult to apply.

For more **general recurrence solving**, we introduce the **Master Theorem** in Section 4.5.

4.5 The Master Method for Solving Recurrences

The Master Method provides a systematic way to solve divide-and-conquer recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Where:

- a is the number of subproblems.
- b is the factor by which the input size is divided.
- $f(n)$ is the cost of work done outside the recursive calls.

The Master Theorem allows us to classify the recurrence into one of three cases to determine the asymptotic behavior of $T(n)$.

Master Theorem

For the recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, let $p = \log_b a$ (where p is the critical exponent). The solution is determined as follows:

1. **Case 1:** $f(n) = O(n^{p-\epsilon})$ **for some** $\epsilon > 0$
   If $f(n)$ grows asymptotically slower than n^p, the recursion is dominated by the divide step.
   - **Result**: $T(n) = \Theta(n^p)$
2. **Case 2:** $f(n) = \Theta(n^p \log^k n)$ **for** $k \geq 0$
   If $f(n)$ grows at the same rate as n^p, possibly with an additional logarithmic factor, the recursion is balanced.
   - **Result**: $T(n) = \Theta(n^p \log^{k+1} n)$
3. **Case 3:** $f(n) = \Omega(n^{p+\epsilon})$ **for some** $\epsilon > 0$
   If $f(n)$ grows asymptotically faster than n^p and satisfies the regularity condition $af\left(\frac{n}{b}\right) \leq cf(n)$ for some $c < 1$, the recursion is dominated by the cost of $f(n)$.
   - **Result**: $T(n) = \Theta(f(n))$

Steps to Apply the Master Theorem

1. Compute $p = \log_b a$.
2. Compare $f(n)$ to n^p to determine which case applies.
3. Verify the regularity condition if applying Case 3.
4. Conclude the asymptotic bound $T(n)$.

Examples

1. **Example 1**
   Recurrence: $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$
   - $a = 2$, $b = 2$, $p = \log_2 2 = 1$.
   - $f(n) = O(n) = n^1$.
   - Matches Case 2 with $k = 0$.
   - **Result**: $T(n) = \Theta(n \log n)$.
2. **Example 2**
   Recurrence: $T(n) = 4T\left(\frac{n}{2}\right) + O(n^2)$
   - $a = 4$, $b = 2$, $p = \log_2 4 = 2$.
   - $f(n) = O(n^2) = n^2$.
   - Matches Case 2 with $k = 0$.
   - **Result**: $T(n) = \Theta(n^2 \log n)$.
3. **Example 3**
   Recurrence: $T(n) = T\left(\frac{n}{2}\right) + O(n^2)$
   - $a = 1$, $b = 2$, $p = \log_2 1 = 0$.
   - $f(n) = O(n^2)$.
   - Matches Case 3 (regularity condition satisfied).
   - **Result**: $T(n) = \Theta(n^2)$.

When the Master Theorem Doesn’t Apply

The Master Theorem may not apply when:

- $f(n)$ does not fit neatly into the given cases.
- The recurrence involves non-polynomial growth rates.
  In such cases, alternative methods such as the substitution or recursion-tree methods must be used.

Key Takeaways

- The Master Theorem is a powerful tool for solving divide-and-conquer recurrences.
- Always compute $p = \log_b a$ to compare the growth of $f(n)$ with n^p.
- Verify the regularity condition for Case 3 to ensure correctness.

4.6 Proof of the Continuous Master Theorem

The proof of the **Continuous Master Theorem** (Theorem 4.13) provides a more rigorous way to establish asymptotic bounds for recurrence relations of the form:

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

where:

- $a \geq 1$ and $b > 1$ are constants,
- $f(n)$ is a function defined over real numbers.

The proof follows a structured approach, breaking down the recurrence tree into smaller components and analyzing the sum of costs at different levels.

Lemma 4.3: Growth of Summation Terms

Let $a, b > 1$ be constants, and let $f(n)$ be a function defined over real numbers $n \geq 1$. Then the recurrence:

$$T(n) = \begin{cases} \Theta(1), & n \leq c \\ aT(n/b) + f(n), & n > 1 \end{cases}$$

has a solution:

$T(n) = \Theta\left(f(n) + \sum_{i=0}^{\log_b n} a^i f(n/b^i)\right).$

This result describes how work accumulates at each level of recursion.

Breakdown of Recursion Tree Analysis

To understand the behavior of $T(n)$, we analyze the recursion tree structure:

```
T(n)
├── T(n/b)   ── f(n)
│     ├── T(n/b^2)   ── f(n/b)
│     │     ├── T(n/b^3)   ── f(n/b^2)
│     │     ├── ...
│     ├── ...
├── T(n/b)   ── f(n)
│     ├── T(n/b^2)   ── f(n/b)
│     │     ├── T(n/b^3)   ── f(n/b^2)
│     │     ├── ...
│     ├── ...
```

Each level i contributes:

$a^i f(n/b^i)$

This sum captures the cost at each recursive level, summing up to form the total computational work.

Thus, the total cost across all levels of the recursion tree is given by the sum:

$T(n) = \sum_{i=0}^{\log_b n} a^i f(n/b^i)$

This formula accumulates the work done at each level of recursion.

Bounding the Summation

To evaluate this sum, we analyze its growth using asymptotic bounds:

1. **Case 1: Work at leaves dominates**
   If $f(n)$ grows **faster** than the accumulated sum of recursive calls, then the total complexity is determined by $f(n)$:

   $T(n) = \Theta(f(n))$
2. **Case 2: Balanced Growth**
   If the work done at each level remains roughly the same, then we sum over all levels:

   $T(n) = \Theta(f(n) \cdot \log_b n)$
3. **Case 3: Recursive calls dominate**
   If the total contribution of recursive calls **grows faster** than $f(n)$, then:

   $T(n) = \Theta(n^{\log_b a})$

Final Complexity Analysis

The recurrence growth follows three possibilities:

- **If** $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, recursion dominates and $T(n) = \Theta(n^{\log_b a})$.
- **If** $f(n) = \Theta(n^{\log_b a})$, work is balanced, leading to $T(n) = \Theta(n^{\log_b a} \log n)$.
- **If** $f(n) = \Omega(n^{\log_b a + \epsilon})$ and satisfies the smoothness condition, then $T(n) = \Theta(f(n))$.

These cases form the **Continuous Master Theorem**, extending the original Master Theorem to a broader class of recurrence relations.

Final Theorem Statement

For recurrence of the form:

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

The asymptotic behavior follows:

1. **If** $f(n) = O(n^{\log_b a-\epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. **If** $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. **If** $f(n) = \Omega(n^{\log_b a+\epsilon})$ and satisfies the smoothness condition, then $T(n) = \Theta(f(n))$.

Exercises

4.6.1

Prove that:

$\sum_{i=0}^{\log_b n} \Theta(f(n/b^i)) = \Theta(f(n) \cdot \log_b n)$.

4.6.2

Show that **Case 3 remains valid** if:

$f(n) = O(n^{\log_b a-\epsilon})$.

4.7 Akra-Bazzi Recurrences

The **Akra-Bazzi method** provides a more generalized approach for solving divide-and-conquer recurrences, extending beyond the limitations of the **Master Theorem**. It is particularly useful for cases where the recurrence does not fit neatly into the standard form.

General Form of Akra-Bazzi Recurrences

The recurrence is given by:

$T(n) = g(n) + \sum_{i=1}^{k} a_i T(n/b_i + h_i(n))$

where:

- a_i are constants (not necessarily equal),
- b_i are constants greater than 1,
- $h_i(n)$ is a perturbation function,
- $g(n)$ is the additional work done outside recursion.

This method **handles non-uniform subproblem sizes** (where divisions of n are not exact) and **accounts for additive functions** $h_i(n)$ that modify subproblem sizes.

Akra-Bazzi Theorem

If a recurrence is of the form:

$T(n) = g(n) + \sum_{i=1}^{k} a_i T(n/b_i + h_i(n))$

where:

- $h_i(n) = o(n)$ (perturbation function is small),
- $g(n)$ is asymptotically positive,
- The function p satisfies:

  $p(n) = \Theta(n^c) \quad \text{for some } c > 0$

Then, the solution follows:

$T(n) = \Theta\left(n^p \left(1 + \int_1^n \frac{g(u)}{u^{p+1}} du\right)\right).$

This theorem extends the **Master Theorem** to more complex recursive structures.

Applying the Akra-Bazzi Method

The solution involves the following steps:

1. **Find the exponent** p
   Solve for p using the characteristic equation:

   $\sum_{i=1}^{k} a_i b_i^{-p} = 1.$
2. **Compute the integral term**
   Evaluate:

   $I(n) = \int_1^n \frac{g(u)}{u^{p+1}} du.$
3. **Combine terms**
   The final solution takes the form:

   $T(n) = \Theta(n^p(1 + I(n))).$

Example

Consider the recurrence:

$T(n) = 2T(n/2 + \sqrt{n}) + n.$

Step 1: Solve for p

Using:

$\sum a_i b_i^{-p} = 1 \Rightarrow 2 \cdot 2^{-p} = 1.$

Solving for p:

$2^{-p} = \frac{1}{2} \Rightarrow p = 1.$

Step 2: Compute Integral

For $g(n) = n$, evaluate:

$I(n) = \int_1^n \frac{u}{u^2} du = \int_1^n u^{-1} du = \ln n.$

Step 3: Final Solution

Using:

$T(n) = \Theta(n^p(1 + I(n))).$

$T(n) = \Theta(n(1 + \ln n)) = \Theta(n \log n).$

Key Takeaways

✓ Handles non-uniform subproblem sizes
✓ Accounts for additive perturbations $h_i(n)$
✓ Extends the Master Theorem to more complex cases

This **Akra-Bazzi method** is essential for analyzing recursive algorithms **where the Master Theorem fails**.

Exercises

4.7.1

Solve the recurrence:

$T(n) = 3T(n/3 + \sqrt{n}) + n^2.$

4.7.2

Prove that **Case 3** of the Master Theorem is **overly restrictive** and fails to handle:

$T(n) = T(n/2) + n/\log n.$