Week 8 Notes - DBMS

Prof. Partha Pratham Das, IIT KGP

Notes by Adarsh (23f2003570)

L8.1: Algorithms and Data Structures/1: Algorithms & Complexity Analysis (34:12)

Algorithms and Programs

Here's a comparison between **Algorithms** and **Programs** in table form:

Aspect	Algorithm	Program
Definition	A step-by-step procedure or set of rules to solve a problem.	A collection of instructions written in a programming language to perform specific tasks.
Lifecycle	Must Terminate under some condition.	Can be long running and need not terminate.
Nature	Conceptual and abstract.	Concrete and executable.
Purpose	Describes the logic or method to solve a problem.	Implements the algorithm and can be run on a computer.
Representation	Usually represented in pseudocode, flowcharts, or natural language.	Written in programming languages like Python, Java, C++, etc.
Input/Output	Focuses on the problem's input and output without specifying how to achieve it.	Takes inputs, processes them according to the algorithm, and produces outputs.
Execution	Not executable directly; needs to be translated into code.	Can be executed by a computer.
Complexity	Describes the theoretical efficiency (e.g., time or space complexity).	Efficiency depends on implementation, hardware, and other factors.
Scope	Describes a general solution or approach to a problem.	A specific solution to a problem, including all implementation details.
Flexibility	Can be applied to different scenarios with modifications.	Fixed to the specific problem it's written for.
Example	Sorting a list, finding the shortest path in a graph.	A Python program that sorts a list of numbers.

Analysis of Algorithms: Explanation

The **analysis of algorithms** involves evaluating the efficiency and effectiveness of an algorithm in terms of various factors like time and space complexity. It is crucial to understand how an algorithm performs in different scenarios, how scalable it is, and how it compares to other algorithms.

Why Analyze Algorithms?

- **Efficiency:** Helps in determining the best algorithm for a specific problem, especially when dealing with large data sets or complex tasks.
- **Optimization:** Allows identifying potential bottlenecks or areas for improvement in the algorithm's design.
- **Comparison:** Enables comparison between different algorithms to select the most suitable one based on factors like speed, memory usage, and scalability.
- **Predict Performance:** By analyzing an algorithm, we can predict how it will perform under different conditions without actually running it on all possible inputs.

What is Algorithm Analysis?

- **Algorithm analysis** refers to the process of evaluating an algorithm's performance, primarily focusing on its **time complexity** (how long it takes to run) and **space complexity** (how much memory it requires).
- It involves studying the **asymptotic behavior** of the algorithm, often by using **Big O notation**, which expresses the upper bound of the runtime in terms of the size of the input.

Key points of analysis include:

- **Time Complexity:** Determines how the runtime grows as the input size increases.
- **Space Complexity:** Determines how the memory requirements grow as the input size increases.

How to Analyze Algorithms?

1. **Identify the Input:** Define the type and size of input the algorithm will process.
2. **Determine the Operations:** List the steps the algorithm takes to solve the problem and identify which operations are the most time-consuming (e.g., loops, recursions).
3. **Count Basic Operations:** Count the number of basic operations (such as comparisons, additions) the algorithm performs.
4. **Use Big O Notation:** Express the worst-case time and space complexity using Big O notation (e.g., O(n), O(log n), $O(n^2)$).
5. **Best, Worst, and Average Cases:** Analyze the algorithm for different cases:
    - **Best Case:** The scenario where the algorithm performs the least work.
    - **Worst Case:** The scenario where the algorithm performs the most work.
    - **Average Case:** The expected performance for random inputs.

6. **Consider Recursion (if any):** For recursive algorithms, use recurrence relations to analyze performance.
7. **Space Complexity:** Evaluate how much memory the algorithm needs relative to the input size, including both auxiliary space and input space.

Where is Algorithm Analysis Used?

- **Software Development:** To choose the most efficient algorithm for a given problem or dataset.
- **Machine Learning:** In selecting algorithms for tasks like sorting, searching, and classification, where performance can significantly impact results.
- **Data Structures:** To assess and compare the efficiency of data structures (like trees, graphs, and hash tables) based on the operations they support.
- **Database Systems:** Helps in query optimization, indexing, and other performance-sensitive tasks.
- **Computer Networks:** Used in routing protocols, data transmission, and optimization problems in networking.

When Should Algorithm Analysis be Performed?

- **Before Implementation:** During the design phase to compare algorithms theoretically and choose the most efficient one based on expected input sizes.
- **During Testing:** After implementing the algorithm, testing it with various input sizes to verify whether the theoretical analysis holds up in practice.
- **When Scaling:** When the input size grows or when handling large datasets, algorithm analysis can help assess whether the algorithm still performs efficiently.
- **After Optimization:** After making optimizations to the algorithm, analysis should be done to verify improvements.

Summary of Algorithm Analysis:

- **Why:** To optimize performance, compare algorithms, and predict performance.
- **What:** The process of evaluating time and space complexity, often using Big O notation.
- **How:** By counting operations, analyzing best, worst, and average cases, and using Big O notation.
- **Where:** In software development, machine learning, data structures, databases, and networks.
- **When:** During design, testing, scaling, and optimization of algorithms.

Example of Analysis of Bubble Sort

Example of Time and Space Complexity Analysis: **Bubble Sort Algorithm**

Let's take the **Bubble Sort** algorithm as an example to understand both **time complexity** and **space complexity** analysis.

Bubble Sort Algorithm

```
For i = 1 to n-1:
    For j = 0 to n-i-1:
        If arr[j] > arr[j+1]:
            Swap arr[j] and arr[j+1]
```

Time Complexity Analysis:

1. Best Case: The **best-case scenario** occurs when the input array is already sorted. In this case, no swaps are needed, but the algorithm still needs to check each pair of adjacent elements.

- The outer loop runs `n-1` times (for `i` from 1 to `n-1`).
- The inner loop runs `n-i-1` times for each value of `i`.

In the best case (when the list is sorted), we still need to check the comparisons, but **no swaps** are needed. Bubble sort can be optimized by checking if any swaps were made in the inner loop, and if none were made, we can exit early.

For a sorted list:

- The number of operations in the best case is approximately **n-1** comparisons in the outer loop, and the inner loop doesn't make any swaps.

Thus, the best case **time complexity** is **O(n)**, since the algorithm only needs to go through the list once, checking each pair.

2. Worst Case: The **worst-case scenario** occurs when the array is sorted in reverse order. In this case, every comparison will result in a swap.

- The outer loop runs `n-1` times (for `i` from 1 to `n-1`).
- The inner loop runs `n-i-1` times for each value of `i`.

In the worst case (when the list is sorted in reverse), the algorithm performs the maximum number of swaps and comparisons. Every time the inner loop runs, each element is compared with the next one, resulting in a complete traversal of the list.

The total number of comparisons in the worst case:

- For `i = 1`, the inner loop runs `n-1` times.
- For `i = 2`, the inner loop runs `n-2` times.
- And so on...

The total comparisons can be calculated as the sum of the first `n-1` integers:

- $(n-1) + (n-2) + (n-3) + ... + 1 = \frac{n(n-1)}{2}$

Thus, the worst-case **time complexity** is **$O(n^2)$** because the total number of operations grows quadratically with the size of the input.

3. Average Case: In the average case, the input is a random permutation of elements. On average, half of the elements are out of order, and the algorithm performs a mix of comparisons

and swaps.

The number of comparisons is similar to the worst case because we still need to perform comparisons for each pair. The total number of operations is still in the order of $O(n^2)$ in the average case, since the inner loop runs for each element, and the worst-case behavior (where most of the elements are out of order) is common.

Thus, the average-case **time complexity** is also **$O(n^2)$**.

Space Complexity Analysis:

Space Complexity refers to the amount of memory required by the algorithm relative to the input size.

- **Bubble Sort** is an **in-place** sorting algorithm, meaning it doesn't require extra memory for another array or data structure besides the input list. The only additional space used is for variables like the loop counters `i`, `j`, and a temporary variable used for swapping elements.
- As the space needed does not depend on the size of the input and is constant, the **space complexity** is **O(1)**.

Summary of Time and Space Complexity for Bubble Sort:

Case	Time Complexity	Space Complexity
Best Case	O(n)	O(1)
Worst Case	$O(n^2)$	O(1)
Average Case	$O(n^2)$	O(1)

Explanation of Analysis:

1. **Time Complexity:**
    - In the best case (sorted input), Bubble Sort still requires `n-1` comparisons to check if the array is already sorted, which leads to a **linear time complexity** of **O(n)**.
    - In the worst case (reverse sorted input), Bubble Sort performs **quadratic time complexity** of **$O(n^2)$** because every element has to be compared and swapped in each iteration.
    - The average case is similar to the worst case, where the algorithm requires roughly **$O(n^2)$** operations in total.
2. **Space Complexity:**
    - Bubble Sort is an **in-place** algorithm, meaning it doesn't require additional space proportional to the size of the input.
    - Therefore, its **space complexity** is **O(1)**, which is constant space. The extra memory used does not depend on the size of the input list.

Algorithm Analysis (Sum of n numbers)

Algorithm Analysis: **Sum of n Numbers**

```
sum = 0
For i = 1 to n:
    sum = sum + i
Return sum
```

Time Complexity Analysis:

1. **Best Case:**

- In the case of this algorithm, the best, worst, and average cases are essentially the same because the algorithm always goes through the entire loop from 1 to n.
- The **best-case time complexity** is still **O(n)** because the loop runs exactly n times regardless of the input.

2. **Worst Case:**

- The worst case for this algorithm also involves iterating over all the numbers from 1 to n and performing an addition operation each time.
- The **worst-case time complexity** is **O(n)** because the loop runs n times and performs one addition operation per iteration.

3. **Average Case:**

- Since the algorithm always iterates through the entire range of numbers from 1 to n, the **average-case time complexity** is also **O(n)**.
- There is no variation in performance based on the input, as the algorithm always processes all n numbers in a linear sequence.

Space Complexity Analysis:

- The **space complexity** of this algorithm depends on how much additional memory is used beyond the input. The only extra space used is:
  - A variable `sum` to store the cumulative sum, which takes up constant space.
  - A variable `i` for the loop counter, which also takes up constant space.

Since there is no additional space needed that grows with the size of the input (n), the **space complexity** is **O(2)** (constant space).

Case	Time Complexity	Space Complexity
Best Case	O(n)	O(2)
Worst Case	O(n)	O(2)
Average Case	O(n)	O(2)

Algorithm Analysis Find a character in a string in C++

```
int find_char(const string& str, char target) {
    for (int i = 0; i < str.length(); ++i) {
        if (str[i] == target) {
            return i;
        }
    }
    return -1;
}
```

Time Complexity

1. n comparisons using if
2. n times you call str.length()
    1. n times you iterate each character in the string
    2. therefore it becomes $n \times n$ or n^2

Time Complexity = $n + n^2 \approx n^2$

Space Complexity is 3

1. space used by
    1. `str`
    2. `target`
    3. `i`

Models use in Algorithmic Analysis

1. Counting Model

The **Counting Model** is a basic yet fundamental technique in algorithm analysis, especially in analyzing **recursive algorithms** and **loop-based algorithms**. This model focuses on counting the number of basic operations or **steps** an algorithm performs.

Key Points:

- **Basic operations**: In any algorithm, some operations are repeated multiple times. For example, in sorting, comparing two elements counts as an operation.
- **Goal**: The aim of the counting model is to determine how many of these operations are executed as the size of the input grows. This helps to understand the **time complexity** of the algorithm.
- **Used in**: Algorithm analysis (e.g., counting how many comparisons a sorting algorithm like QuickSort or MergeSort makes).

Example (Counting Model):

Consider the simple algorithm that sums all numbers from 1 to n:

```
int sum(int n) {
    int total = 0;
    for (int i = 1; i <= n; i++) {
        total += i;
    }
    return total;
}
```

- **Counting**: The loop runs n times, so the number of operations is **O(n)**.
- **Complexity**: In this case, counting operations shows that the algorithm runs in **O(n)** time.

---

2. **Asymptotic Model**

The **Asymptotic Model** is used to describe the **behavior** of an algorithm's performance in the limit, as the input size grows **towards infinity**. This model provides a way to classify algorithms according to their growth rates (i.e., how the runtime or space complexity increases as the size of the input grows).

Key Concepts:

- **Big-O (O)**: Describes the upper bound of the complexity. It indicates the **worst-case** growth rate of an algorithm.
- **Big-Ω (Ω)**: Describes the lower bound. It indicates the **best-case** performance of an algorithm.
- **Big-Θ (Θ)**: Describes the **tight bound**, meaning the algorithm's performance is bounded both from above and below.

Example:

For a simple linear search algorithm:

- Time complexity: **O(n)** (it grows linearly with the size of the input).

For a binary search algorithm:

- Time complexity: **O(log n)** (it grows logarithmically with the size of the input).

In the **asymptotic model**, we use Big-O, Big-Ω, and Big-Θ notation to describe how an algorithm behaves as the input size increases.

---

3. **Generating Functions**

Generating functions are a powerful mathematical tool used to analyze algorithms, particularly those involving **recurrence relations**. In the context of algorithm analysis, generating functions can help derive exact solutions for the time complexity of recursive algorithms.

Key Points:

- **Definition**: A generating function is a formal power series whose coefficients represent the sequence of numbers we're interested in, often related to the number of operations in a recursive algorithm.

  For example, if an algorithm has a recurrence relation like:

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

  We can use a generating function to solve this recurrence and find the closed-form solution for $T(n)$.

- **Use**: Generating functions are commonly used in combinatorial analysis and for solving recurrences in divide-and-conquer algorithms.

Example:

For the recurrence relation of **MergeSort**:

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

Generating functions can be used to solve this recurrence relation and prove that the time complexity is **O(n log n)**.

---

4. Master Theorem

The **Master Theorem** provides a straightforward method for analyzing the time complexity of divide-and-conquer algorithms, which have a specific recurrence relation structure.

Key Points:

The Master Theorem applies to recurrences of the form:

$$T(n) = aT(n/b) + O(n^d)$$

Where:

- a is the number of subproblems,
- b is the factor by which the problem size is divided in each recursive call,
- n^d is the cost of dividing the problem and combining the results.

The Master Theorem provides **three cases** to determine the time complexity of the recurrence:

1. **Case 1 (When $a > b^d$)**:

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

   (The time complexity is dominated by the recursive calls.)

2. **Case 2 (When $a = b^d$)**:

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

   (The time complexity is a mix of recursion and combining.)

3. **Case 3 (When $a < b^d$)**:

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

(The time complexity is dominated by the division and combining work.)

Example:

For the recurrence:

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

We apply the Master Theorem:

- $a = 2$, $b = 2$, and $d = 1$.
- Since $a = b^d$, we are in **Case 2**, and the time complexity is:

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

Thus, MergeSort has a time complexity of **O(n log n)**.

Model	Key Idea	Used For	Example
Counting Model	Count the number of operations in the algorithm.	Analyzing algorithm steps (loops, operations).	Counting operations in a sorting algorithm.
Asymptotic Model	Describes the performance as the input size grows (Big-O, Big-Ω, Big-Θ).	Characterizing time and space complexity.	Analyzing time complexity of QuickSort.
Generating Functions	Mathematical tool for **solving recurrences** and counting problems.	Solving recurrences in divide-and-conquer algorithms. Used in Recursive Functions.	Using generating functions for MergeSort.
Master Theorem	Provides a direct way to **solve recurrences** of divide-and-conquer algorithms. Used in Recursive Functions.	Analyzing divide-and-conquer algorithms.	Finding the time complexity of MergeSort.

Algorithm Analysis (One more Sample)

```
int count = 0;
for (int i = 0; i < N; i++) {
    for (int j = i + 1; j < N; j++) {
        if (a[i] + a[j] == 0)
            count++;
    }
}
```

Find the growth (or function approximation or Big-O) when `i` increases from 0 to `N`

Operation	Frequency	Approximation
Variable Declaration	1 time for count, 1 time for i, `N` times for j $\therefore N + 2$	$\approx N$
Initialization Statement	count=0 **1 time** i = 0 **1 time** j = i+1 **N times** $\therefore N + 2$	$\approx N$
< than compare	Let N = 5 i = 0 **1 times**, j={1,2,3,4} **4 times** i = 1 **1 times**, j={2,3,4} **3 times** i = 2 **1 times**, j={3,4} **2 times** i = 3 **1 times**, j={4} **1 times** i = 4 **1 times**, j={0} **0 times** $(1+4)+(1+3)+(1+2)+(1+1)+(1+0) = 15$ $\therefore \frac{N \times (N+1)}{2}$	$\approx \frac{N^2}{2}$
= compare	$\frac{N \times (N-1)}{2}$	$\approx \frac{N^2}{2}$
Array Access	$2 \times \frac{N \times (N-1)}{2}$	$\approx \frac{N^2}{2}$
Increment	Increment of i, j, count for i, j $\frac{N \times (N+1)}{2}$ for count $\frac{N \times (N+1)}{2}$ if all are zeros $\therefore \frac{N \times (N+1)}{2}$ to $N(N+1)$ times	$\approx \frac{N^2}{2}$ to $\approx N^2$

Performance of Algorithms

Common order of growth of complexity for various algorithms. This table compares their **time complexity** in terms of Big-O notation, which expresses the upper bound of their time complexity as a function of input size n.

Algorithm	Time Complexity (Best)	Time Complexity (Worst)	Time Complexity (Average)	Space Complexity
Linear Search	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$

Algorithm	Time Complexity (Best)	Time Complexity (Worst)	Time Complexity (Average)	Space Complexity
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(\log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n + k)$
Dijkstra's Algorithm (using a priority queue)	$O(V \log V + E)$	$O(V \log V + E)$	$O(V \log V + E)$	$O(V + E)$
Floyd-Warshall Algorithm	$O(n^3)$	$O(n^3)$	$O(n^3)$	$O(n^2)$
Bellman-Ford Algorithm	$O(V + E)$	$O(V + E)$	$O(V + E)$	$O(V)$
Breadth-First Search (BFS)	$O(V + E)$	$O(V + E)$	$O(V + E)$	$O(V)$
Depth-First Search (DFS)	$O(V + E)$	$O(V + E)$	$O(V + E)$	$O(V)$
Knapsack Problem (Dynamic Programming)	$O(nW)$	$O(nW)$	$O(nW)$	$O(nW)$
Matrix Multiplication (Naive)	$O(n^3)$	$O(n^3)$	$O(n^3)$	$O(n^2)$
Strassen's Matrix Multiplication	$O(n^{\log_2 7})$	$O(n^{\log_2 7})$	$O(n^{\log_2 7})$	$O(n^2)$

Explanation of Table Columns:

1. **Time Complexity (Best)**: The best-case scenario time complexity, which occurs under ideal conditions (e.g., already sorted for sorting algorithms).
2. **Time Complexity (Worst)**: The worst-case scenario time complexity, which occurs under the most challenging input conditions (e.g., reverse order for sorting algorithms).
3. **Time Complexity (Average)**: The average-case time complexity, which represents the expected performance for random inputs.
4. **Space Complexity**: The amount of memory or space the algorithm uses relative to the input size.

Explanation

- **Sorting Algorithms**: Common sorting algorithms like Bubble Sort, Selection Sort, and Insertion Sort have quadratic time complexity $O(n^2)$ in the worst case, whereas more efficient algorithms like Merge Sort and Quick Sort typically perform in $O(n \log n)$.
- **Search Algorithms**: Linear Search has a worst-case time complexity of $O(n)$, while Binary Search, which requires a sorted array, performs much faster with a time complexity of $O(\log n)$.
- **Graph Algorithms**: BFS, DFS, and Dijkstra's Algorithm all have time complexities related to the number of vertices V and edges E. Dijkstra's time complexity is reduced using a priority queue, while Floyd-Warshall has cubic complexity $O(n^3)$.
- **Dynamic Programming**: Algorithms like the Knapsack Problem are often used for optimization problems, and they have polynomial time complexities depending on both the number of items n and the total weight W.

Complexity of Common Data Structure Operations

	Data Structure	Time Complexity								Space Complexity
		Average				Worst				Worst
		Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Linear Data Structures	Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
	Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
	Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
	Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
	Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Non-Linear Data Structures	Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
	Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
	Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
	Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
	B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
	Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
	Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
	AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
	KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Source: Know Thy Complexities! (06-Apr-2021)