Chapter One

Introduction to Data Structures & Algorithms

Introduction

- Data Structures and Algorithms (DSA) are fundamental concepts in computer science that help in organizing, managing, and processing data efficiently.
- A program is written in order to solve a problem. A solution to a problem actually consists of two things:
  - A way to organize the data
  - Sequence of steps to solve the problem
- The way data are organized in a computer memory is said to be **Data Structure** and the sequence of computational steps to solve a problem is said to be an **algorithm**.
- Therefore, a program is nothing but data structures plus algorithms.
- A **data structure** is a way of storing organizing data in a computer so that it can be used efficiently.

Abstract data Types

- An entity with the properties just described is called an abstract data type (ADT).
- Abstract Data Types Consists of data to be stored and operations supported on them.
- The ADT is a theoretical model for a data structure that specifies:
  - What can be stored in the Abstract Data Type
  - What operations can be done on/by the Abstract Data Type.
  - It does not specify how to store or how to implement the operation. It is also independent of any programming language
- Example: ADT employees of an organization:
  - This ADT stores employees with their **relevant attributes** and discarding irrelevant attributes.
  - Relevant:- Name, ID, Sex, Age, Salary, Dept, Address
  - Non-Relevant:- weight, color, height
  - This ADT support **operations** like hiring, firing, retiring.

Abstraction

- **Abstraction** is the process of hiding the implementation details of a system and exposing only the functionality to the user.
- **Abstraction**
  - A process of classifying characteristics as relevant and irrelevant for the particular purpose at hand and ignoring the irrelevant ones.
  - Focus on high level details: removes the complexity by providing only the relevant information needed to understand or use the system.
- Example: A Stack ADT provides operations like push() and pop(), but the user doesn't need to know whether the stack is implemented using an array or a linked list.

Advantages of Abstraction

01

Simplicity - Reduces complexity by hiding unnecessary details.

02

Reusability - Abstract systems can be reused across different implementations.

03

Modularity - Improves code organization and maintenance by separating functionality.

04

Focus on Problem-Solving - Allows developers to concentrate on the functionality without being stacked by implementation specifics.

Algorithm

- **An algorithm** is a finite, well-defined sequence of instructions designed to solve a specific problem or perform a computation.
- **An algorithm** is a well-defined computational procedure that takes some value or a set of values as input and produces some value or a set of values as output.
- The **purpose of an algorithm:**
  - Accept input values
  - Change a value hold by a data structure
  - Re-organize the data structure itself (e.g. sorting)
  - Display the content of the data structure, and so on

Properties of Algorithm

- **Finiteness**: Algorithm must complete after a finite number of steps.
- **Definiteness**: Each step must be clearly defined, having one and only one interpretation. At each point in computation, one should be able to tell exactly what happens next.
- **Correctness**: It must compute correct answer for all possible legal inputs.
- **Language Independence**: It must not depend on any programming language.
- **Completeness**: It must solve the problem completely.
- **Effectiveness**: It must be possible to perform each step exactly and in a finite amount of time.
- **Efficiency**: It must solve with the least amount of computational resources such as time and space.
- **Generality**: Algorithm should be valid on all possible inputs.

Algorithm Analysis

- **Algorithm analysis** refers to the process of determining the amount of computing time and storage space required by different algorithms.
  - it's a process of predicting the resource requirement of algorithms in a given environment.
  - It helps determine the suitability of an algorithm for a specific problem, especially as input sizes grow.
- To classify some data structures and algorithms as good, we need precise ways of analyzing them in terms of resource requirement. The main resources are:
  - Running Time
  - Memory Usage
  - Communication Bandwidth

Complexity Analysis

- **Complexity Analysis** is the systematic study of the cost of computation, measured either in time units or in operations performed, or in the amount of storage space required.
  - Helps estimate runtime and memory requirements.
  - Allows selection of the most efficient algorithm for a problem.
  - Ensures better utilization of computational resources.
  - Assesses how well the algorithm performs with increasing input sizes (Scalability).
- The **goal** is to have a meaningful measure that permits comparison of algorithms independent of operating platform.

Types of Complexity Analysis

- **Time Complexity -** measures the amount of time an algorithm takes to complete as a function of the input size n.
  - Determines how well an algorithm performs for large input sizes.
  - Helps choose the most efficient algorithm for a problem.
  - Guides improvements in algorithmic design to minimize runtime.
- Basic components of time complexity analysis:
  - **Basic Operations** - The fundamental steps of an algorithm, such as addition, multiplication, comparisons, and memory access.
  - **Input Size (*nn*) -** The number of elements in the input dataset, which directly impacts the number of operations required.
  - **Growth Rate** - How the number of operations increases as n grows.

Types of Complexity Analysis

- **Space Complexity** - refers to the amount of memory an algorithm requires to solve a problem as a function of the input size n.
  - Determines whether an algorithm can run on systems with limited memory.
  - Assesses how well the algorithm handles increasing input sizes.
  - Balances between time and space usage to optimize overall performance.
- It includes memory used by:
  - Input data.
  - Auxiliary data structures (e.g., arrays, stacks).
  - Function calls and recursion stacks.
- Basic components of time complexity analysis:
  - **Fixed Part -** Memory required for constants and variables.
  - **Variable Part -** Memory required for dynamic structures (e.g., arrays, lists) and recursive calls.
  - **Auxiliary Space-** Extra memory used by the algorithm apart from the input.

Complexity Analysis Rules

- **First -** Identify the Basic Operations, Count operations that contribute significantly to runtime(time) or memory usage.
  - ❑Execution of one of the following operations takes time 1:
    - Assignment Operation
    - Single Input/Output Operation
    - Single Boolean Operations
    - Single Arithmetic Operations
    - Function Return
  - ❑Running time of a **Selection statement** (if, switch) is the time for the condition evaluation + the maximum of the running times for the individual clauses in the selection.
    - Time for condition evaluation + the maximum time of its clauses

Complexity Analysis Rules

- **Loops:** Running time for a loop is equal to the running time for the statements inside the loop * number of iterations.
  - The total running time of a statement inside a group of nested loops is the running time of the statements multiplied by the product of the sizes of all the loops.
  - For nested loops, analyze inside out.
    - Always assume that the loop executes the maximum number of iterations possible.
- **Function Call:** The running time of a function call combines the setup time, the time for evaluating parameters, and the execution time of the function body.

Complexity Analysis Rules

```cpp
#include <iostream>
using namespace std;

int main() {
    int n = 5;

    for (int i = 0; i < n; i++) {
        if (i % 2 == 0) {  // Check if 'i' is even
            cout << i << " is even." << endl;
        } else {
            cout << i << " is odd." << endl;
        }
    }

    return 0;
}
```

- **Loop**: The loop runs n times, so its complexity is $O(n)$.Conditional
- **Check**: The condition i % 2 == 0 takes $O(1)$ time for each iteration.
- **Total Complexity**: The loop runs n times, and in each iteration, we perform constant-time operations (condition check and printing), so the total complexity is:

$$O(n) \times O(1) = O(n)$$

Complexity Analysis Rules

```cpp
#include <iostream>
using namespace std;

int main() {
    int n = 6;

    for (int i = 0; i < n; i++) {
        if (i == 0) {
            cout << "Zero" << endl;
        } else if (i == 1) {
            cout << "One" << endl;
        } else if (i == 2) {
            cout << "Two" << endl;
        } else {
            cout << "Other number" << endl;
        }
    }
    return 0;
}
```

- **Loop**: The loop runs n times, so its complexity is $O(n)$.
- **Conditional Check**: Each condition in the if-else ladder is checked in constant time $O(1)$, so for each iteration, the total time for the condition evaluation is $O(1)$.
- **Total Complexity**: The loop runs n times, and in each iteration, we perform constant-time operations (condition check and printing), so the total complexity is:

$$O(n) \times O(1) = O(n)$$

Complexity Analysis Rules

```cpp
#include <iostream>
using namespace std;
int main() {
    int n = 3;
    int m = 2;

    for (int i = 0; i < n; i++) {  // Outer loop
        for (int j = 0; j < m; j++) {  // Inner loop
            if ((i + j) % 2 == 0) {  // Check sum of i and j is even
                cout << "Even sum (" << i << ", " << j << ")" << endl;
            } else {
                cout << "Odd sum (" << i << ", " << j << ")" << endl;
            }
        }
    }
    return 0;
}
```

- **Outer Loop**: Runs n times.
- **Inner Loop**: Runs mm times for each outer loop iteration.
- **Condition Check:** Each conditional check takes $O(1)$O(1), and it happens for every iteration of the inner loop.
- **Total Complexity**: The total complexity is:

$O(n)\times O(m)\times O(1)$=O(n×m)

This means the total complexity depends on both the outer loop and inner loop iterations.

Complexity Analysis Rules

```cpp
void funcA(int n) {
    for (int i = 0; i < n; i++) {  // O(n)
        cout << i << " ";
    }
}

void funcB(int n) {
    for (int i = 0; i < n * 2; i++) {  // O(2n)
        cout << i << " ";
    }
}

void funcC(int n) {
    funcA(n);  // O(n)
    funcB(n);  // O(2n)
}

int main() {
    funcC(5);  // O(3n) = O(n)
}
```

- **Function A**: Runs n times. $O(n)$
- **Function B**: Runs n times for each outer loop iteration. $O(2n)$
- **Function C**: Add the complexities of the two function calls:

  $O(n)+O(2n)=\mathrm{O}(3\mathrm{n})$

Total Complexity: The dominant operation is in funcC, which calls funcA and funcB.

Therefore Time Complexity is:
$\mathrm{O}(3\mathrm{n}) = \mathrm{O}(\mathrm{n})$

Complexity Analysis

- **Best case:** Represents the scenario where the algorithm performs the least number of operations.
  - Minimum time required for program execution.
  - Determines the minimum time complexity for the algorithms.
- Example:

  For linear search, if the target element is the first element in the array:

  Time Complexity: O(1)This is the best-case scenario since the algorithm finds the result immediately.

Complexity Analysis

- **Average case:** Represents the expected number of operations an algorithm performs on average across all possible inputs.
  - Average time required for program execution.
  - It considers all inputs and their likelihood of occurrence, providing a balanced view.
- Example:

  For linear search, the target element might, on average, be somewhere in the middle of the array:

  Time Complexity: $O(n/2)=O(n)$O(n/2)=O(n).

Complexity Analysis

- **Worst case:** Represents the scenario where the algorithm performs the maximum number of operations.
  - Maximum time required for program execution.
  - Determines the maximum time complexity for the algorithm, crucial for ensuring system performance under heavy loads.
- Example:

  For linear search, if the target element is not in the array, the algorithm will traverse the entire array:

  Time Complexity: $O(n)$O(n).

Complexity Analysis

```python
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Element found
    return -1  # Element not found

# Example Array
arr = [10, 20, 30, 40, 50]
target = 30
result = linear_search(arr, target)
print("Element found at index:", result)
```

- **Best Case**: The target is the first element of the array.
  - **Target:** find 10
  - **Time Complexity**: O(1)
  - Only one comparison is needed.
- **Average Case:** The target is located somewhere in the middle of the array.
  - **Target:** find 30
  - **Time Complexity**: O(n/2)=O(n)
  - On average, half of the array is searched.
- **Worst Case:** The target is the last element of the array, or it does not exist.
  - **Target**: 60 (not in the array)
  - **Time Complexity**: $O(n)$
  - The entire array is searched.

Asymptotic Analysis

- **Asymptotic analysis -** attempts to estimate the resource consumption of an algorithm.
- It allows us to compare the relative costs of two or more algorithms for solving the same problem.
- **Asymptotic analysis -** is concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.
  - Helps understand how the algorithm behaves as the problem size grows.
  - Provides a way to compare algorithms irrespective of hardware or software implementation.
  - Ignores constants and lower-order terms, focusing only on the dominant term that has the greatest impact on growth.

Asymptotic Analysis

Benefits:

- Asymptotic analysis allows to predict how well an algorithm will scale as the problem size increases.
- It helps in understanding which algorithm is more efficient and suitable for large datasets.
- By analyzing different algorithms with respect to their time and space complexities, we can compare their performances for different input sizes.
- Independence from Machine and Implementation which means Asymptotic analysis focuses on the algorithm’s behavior rather than the underlying hardware or programming language details.
- It allows us to compare the relative costs of two or more algorithms for solving the same problem.

Asymptotic Analysis

- **Big O Notation (O) -** Describes the worst-case scenario for an algorithm, i.e., the maximum amount of time or space the algorithm can take.
- **Big Omega Notation (Ω) -** Describes the best-case scenario for an algorithm, i.e., the minimum amount of time or space the algorithm will take.
- **Big Theta Notation (θ)** - Describes the exact growth of an algorithm, i.e., the algorithm's running time is bounded both from above and below by the same function.

Why Data structure for Data science?

- Data Science revolves around extracting knowledge and insights from data.
  - But before you can analyze it, you need efficient ways to organize and manipulate that data. This is where Data Structures and Algorithms (DSA) come into play.
- DSA is the foundation for efficiently managing and processing data, and it plays a crucial role in various aspects of data science.

Why Data structure for Data science?

- Data structure and algorithm is relevant in Data Science for:
  - **Data Organization and Management:** DSA helps you organize large datasets efficiently, allowing for faster retrieval and manipulation of information.
  - **Problem-Solving Approach:** DSA fosters a logical and analytical approach to problem-solving, a valuable asset for tackling complex data science challenges.
  - **Algorithm Implementation:** Many machine learning models rely on algorithms for tasks like classification or clustering. Understanding DSA allows you to implement and optimize these algorithms effectively.
  - **Efficient Data Processing:** By choosing the right data structures and algorithms, you can streamline data processing workflows and extract insights from data faster. This becomes especially critical when dealing with big data.

Thank you!