Module 2 Fundamentals of Algorithm Analysis

1. Recursion vs Iteration

Concepts:

Recursion: A process where a function calls itself directly or indirectly.

Example: Calculating factorial using recursion.

Analogy: Think of opening nested Russian dolls. Each doll represents a function call until you reach the smallest doll (base case).

Iteration: Uses repetition structures like loops to execute a set of instructions repeatedly.

Example: Calculating factorial using iteration.

Analogy: Imagine climbing stairs one step at a time until you reach the top (end condition).

C++ Examples:

```
// Recursive Factorial
int factorialRecursive(int n) {
    if (n == 0) return 1; // Base case
    return n * factorialRecursive(n - 1);
}

// Iterative Factorial
int factorialIterative(int n) {
    int result = 1;
    for (int i = 1; i <= n; ++i) {
        result *= i;
    }
    return result;
}
```

Types of Recursion

Direct Recursion:

Function calls itself directly

```
void directRecursive() {
    // Some Code
    directRecursive();
    // Some Code
}
```

Indirect Recursion:

Function fun calls another function fun2, which in turn calls fun.

```
void fun2(); // Forward declaration

void fun() {
    // Some Code
    fun2();
    // Some Code
}

void fun2() {
    // Some Code
    fun();
    // Some Code
}
```

Tail Recursion:

The recursive call is the last operation in the function

```
void tailRecursive(int n) {
    if (n == 0) return;
    std::cout << n << " ";
    tailRecursive(n - 1); // Last operation
}
```

Non-Tail Recursion:

There are operations after the recursive call.

```
void nonTailRecursive(int n) {
    if (n == 0) return;
    nonTailRecursive(n - 1);
    std::cout << n << " "; // Operation after recursive call
}
```

Analysis Framework

Efficiency Metrics:

Time Complexity: Indicates how fast an algorithm runs.
Space Complexity: Amount of memory required by the algorithm.

Example:

Consider finding the sum of all elements in an array.

```
// Time Complexity: O(n)
// Space Complexity: O(1)
int sumArray(int arr[], int n) {
    int sum = 0;
    for (int i = 0; i < n; ++i) {
        sum += arr[i];
    }
    return sum;
}
```

Analogy: Think of a library cataloging books. The time it takes depends on the number of books (input size), and the space needed might include a small notepad to keep track..

Asymptotic Notations

Big-O Notation (O):

Represents the upper bound of an algorithm's running time.

Example: If an algorithm has a complexity of $O(n^2)$, it means the running time grows quadratically with input size.

Big-Omega Notation (Ω):

Represents the lower bound of an algorithm's running time.

Example: Ω(n) implies that the algorithm will take at least linear time.

Theta Notation (Θ):

Represents both upper and lower bounds.

Example: Θ(n log n) indicates that the algorithm's running time is tightly bound by n log n.

```
// O(1) - Constant Time
int getFirstElement(int arr[]) {
    return arr[0];
}

// O(n) - Linear Time
int findElement(int arr[], int n, int key) {
    for (int i = 0; i < n; ++i) {
        if (arr[i] == key) return i;
    }
    return -1;
}

// O(n^2) - Quadratic Time
void printPairs(int arr[], int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            std::cout << "(" << arr[i] << ", " << arr[j] << ")" << std::endl;
        }
    }
```

```
}
```

Best, Worst, and Average Case Analysis
Sequential Search Example:

```
int sequentialSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; ++i) {
        if (arr[i] == key) return i;
    }
    return -1;
}
```

Best Case: O(1) – The key is found at the first position.

Worst Case: O(n) – The key is either at the last position or not present.

Average Case: O(n) – Assuming uniform distribution of the key's position.

Analogy: Searching for a book in a disorganized shelf:

Best Case: The book is right at the front.

Worst Case: The book is at the back or missing.

Average Case: On average, you might need to check halfway through the shelf.