Introduction to Algorithms

What is an Algorithm?

An algorithm is a step-by-step procedure or set of rules for solving a problem. Think of it as a recipe for baking a cake: you follow specific instructions to achieve the desired output (a delicious cake).

Analogy:

Imagine you're giving directions to a friend to reach your house. You break down the journey into steps like "Turn left at the traffic light," "Go straight for 2 km," etc. Similarly, an algorithm breaks down a computational problem into smaller, executable steps.

Definition:
An algorithm must satisfy the following criteria:

- Input: Takes zero or more inputs.
- Output: Produces one or more outputs.
- Definiteness: Each step is clear and unambiguous.
- Finiteness: Terminates after a finite number of steps.
- Effectiveness: Each operation is basic and executable.

Characteristics of Algorithms

- Unique Name: Every algorithm should have a distinct name to identify its purpose.
- Well-Defined Inputs/Outputs: Inputs and outputs are explicitly defined.
- Unambiguous Operations: Each step is precise and leaves no room for misinterpretation.
- Finite Execution: The algorithm halts after completing its task.

Example

```
// Example: Algorithm to find the sum of two numbers
#include <iostream>
using namespace std;

int main() {
   int a, b, sum;
```

```
    cout << "Enter two numbers: ";
    cin >> a >> b;
    sum = a + b; // Unambiguous operation
    cout << "Sum: " << sum;
    return 0; // Finite execution
}
```

Steps in Problem Solving

- Problem Definition: Understand the problem clearly. For example, if the task is sorting, define what "sorted" means.
- Model Development: Create a conceptual model of the solution.
- Algorithm Specification: Define the algorithm's inputs, outputs, and steps.
- Design and Testing: Write the algorithm and verify its correctness.
- Analysis: Evaluate the algorithm's efficiency (time and space complexity).
- Implementation: Translate the algorithm into code.
- Testing and Documentation: Test the program and document its functionality.

Algorithm Design and Analysis

Design Principles:

- Divide and Conquer: Break the problem into smaller subproblems.
- Greedy Approach: Make locally optimal choices at each step.
- Dynamic Programming: Solve overlapping subproblems efficiently.
- Analysis:
- Time Complexity: Measures how runtime grows with input size.
- Space Complexity: Measures memory usage.

Example: Linear Search

```
// Linear search algorithm
#include <iostream>
using namespace std;
```

```
int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            return i; // Return index if found
        }
    }
    return -1; // Return -1 if not found
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 30;
    int result = linearSearch(arr, n, key);
    if (result != -1) {
        cout << "Element found at index: " << result;
    } else {
        cout << "Element not found";
    }
    return 0;
```

Applications of Algorithms

Sorting

Rearranges elements in ascending or descending order.

Common algorithms:

Bubble Sort
Insertion Sort
Selection Sort

Example: Bubble Sort

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap elements
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
```

```
        }
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22};
    int n = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, n);
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Searching

Finds a specific element in a dataset. Common algorithms:

- Linear Search
- Binary Search

Example: Binary Search

```
int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == key) {
            return mid;
        } else if (arr[mid] < key) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1;
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 30;
    int result = binarySearch(arr, n, key);
```

```
    if (result != -1) {
        cout << "Element found at index: " << result;
    } else {
        cout << "Element not found";
    }
    return 0;
}
```

Fundamental Data Structures

Arrays

A collection of elements stored in contiguous memory locations.

Example:

```
int arr[5] = {1, 2, 3, 4, 5};
cout << "Third element: " << arr[2];
```

Linked Lists

A sequence of nodes where each node contains data and a pointer to the next node.

Stacks and Queues

Stacks: LIFO (Last In, First Out)
Queues: FIFO (First In, First Out)

Example: Stack Implementation

```
#define MAX 100
int stack[MAX], top = -1;

void push(int value) {
    if (top >= MAX - 1) {
        cout << "Stack Overflow";
    } else {
        stack[++top] = value;
    }
}

int pop() {
    if (top < 0) {
        cout << "Stack Underflow";
```

```
        return -1;
    } else {
        return stack[top--];
    }
}
```

Examples and Code Snippets

Euclid's Algorithm for GCD

```
int gcd(int m, int n) {
    while (n != 0) {
        int r = m % n;
        m = n;
        n = r;
    }
    return m;
}

int main() {
    int m = 60, n = 24;
    cout << "GCD: " << gcd(m, n);
    return 0;
}
```