Frequency Count Method for Time and Space Complexity

The **frequency count method** is used to analyze **time complexity** and **space complexity** by counting how many times key operations (e.g., loops, assignments, function calls) are executed relative to the input size.

Time Complexity Using Frequency Count Method

Definition:

Time complexity measures how the runtime of an algorithm grows with the input size. The frequency count method counts how many times the basic operations (like comparisons or assignments) occur based on input size.

Example:

Consider the following code:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        cout << i * j;  // Basic operation
    }
}
```

- The outer loop runs `n` times.
- For each iteration of the outer loop, the inner loop also runs `n` times.
- Therefore, the basic operation (`cout << i * j;`) runs `n * n` = `n^2` times.

Time Complexity: O(n^2)

Space Complexity Using Frequency Count Method

Definition:

Space complexity measures how the memory usage of an algorithm grows with the input size. The frequency count method counts how many new variables or data structures are created relative to the input size.

Example:

Consider the following code:

```
int arr[n];  // Array of size n
for (int i = 0; i < n; i++) {
    arr[i] = i * i;
}
```

- The array `arr` requires space for `n` elements.
- The space used by `arr` is proportional to `n`.

Space Complexity: O(n)

Time Complexity and Space Complexity

Time Complexity:

- **Definition**: Time complexity measures the amount of time an algorithm takes to complete relative to the input size. It is expressed as a function of the input size, typically denoted as `n`.
- **Goal**: The goal is to understand how the execution time increases as the input size grows.
- **Common Big-O Notations**:
  - **O(1)**: Constant time – The operation takes the same amount of time regardless of input size.
  - **O(n)**: Linear time – The time increases linearly with the input size.
  - **O(n^2)**: Quadratic time – The time increases quadratically with the input size, commonly seen in algorithms with nested loops.
  - **O(log n)**: Logarithmic time – The time grows logarithmically, often seen in divide-and-conquer algorithms like binary search.
  - **O(n log n)**: Log-linear time – Common in efficient sorting algorithms like Merge Sort and Quick Sort.

Space Complexity:

- **Definition**: Space complexity measures the amount of memory space an algorithm needs as a function of the input size.
- **Goal**: To evaluate how the memory requirement of an algorithm grows as the input size increases.
- **Common Big-O Notations**:
  - **O(1)**: Constant space – The algorithm uses a fixed amount of space regardless of the input size.
  - **O(n)**: Linear space – The space requirement grows linearly with the input size, such as when storing an array of size `n`.

Primitive Data Types in C and C++

Definition:

Primitive data types are basic data types provided by the language to store simple values. These types are predefined and do not require any user-defined structures.

C and C++ Primitive Data Types:

1. **int**:
    - Represents an integer (whole number).
    - Typically 4 bytes in both C and C++.
    - Range: -32,768 to 32,767 (for 2-byte int) or higher for larger systems.
2. **char**:
    - Represents a single character.
    - Typically 1 byte.
    - Range: -128 to 127 (signed) or 0 to 255 (unsigned).
3. **float**:
    - Represents a floating-point number (decimal number).
    - Typically 4 bytes in both C and C++.
    - Precision: 6-7 decimal places.
4. **double**:
    - Represents a double-precision floating-point number.
    - Typically 8 bytes.
    - Precision: 15-16 decimal places.
5. **void**:
    - Represents an absence of data.
    - Used for functions that do not return a value.
6. **bool** (in C++ only):
    - Represents a boolean value: `true` or `false`.
    - Typically 1 byte.

Size of Primitive Types:

- `int`: Typically 4 bytes.
- `char`: Typically 1 byte.
- `float`: Typically 4 bytes.
- `double`: Typically 8 bytes.
- `bool`: Typically 1 byte (though implementation-dependent).

Non-Primitive Data Types in C and C++

Definition:

Non-primitive (or **derived**) data types are more complex data types derived from primitive types. They are used to store collections or structures of data.

C and C++ Non-Primitive Data Types:

1. **Arrays**:
    - A collection of elements of the same type, indexed by position.
    - Can be one-dimensional (1D) or multi-dimensional (2D, 3D, etc.).
2. **Structures (struct)**:
    - A user-defined data type that groups variables of different types together.
    - Useful for representing more complex entities with different attributes.

```
struct Student {
    int age;
    char name[50];
    float grade;
};
```

3. **Unions**:
    - Similar to structures but with a key difference: all members of a union share the same memory location.
    - Only one member can hold a value at any given time.

```
union Data {
    int intValue;
    float floatValue;
};
```

4. **Classes (C++ only)**:
    - A class is a blueprint for creating objects (instances).
    - Can contain variables (data members) and functions (member functions) to operate on the data.

```
class Car {
public:
    string model;
    int year;
    void start() {
        cout << "Starting the car..." << endl;
    }
};
```

5. **Pointers**:
    - A pointer is a variable that holds the memory address of another variable.
    - In C and C++, pointers are heavily used for dynamic memory allocation and manipulation.
6. **Strings (C++ only)**:

- A string is a sequence of characters. In C, strings are arrays of `char`, while in C++ there is a dedicated `string` class.

Comparison of Primitive and Non-Primitive Types:

- **Primitive Types**:
  - Simple, predefined, and built into the language.
  - Store single values (e.g., integers, characters).
  - Have a fixed size.
- **Non-Primitive Types**:
  - More complex, user-defined, and allow storing multiple values or more advanced structures.
  - Often have dynamic memory requirements (e.g., pointers, arrays).
  - Can store collections of values (e.g., arrays, structures).

Asymptotic Notation

Asymptotic notation is used to describe the behavior of an algorithm in terms of its performance (time or space complexity) as the input size grows toward infinity. It helps in evaluating the efficiency of an algorithm and allows comparisons between different algorithms.

There are three primary types of asymptotic notation:

1. **Big O Notation (O)**
2. **Theta Notation (Θ)**
3. **Omega Notation (Ω)**

---

1. Big O Notation (O)

- **Definition**: Big O notation describes the **upper bound** of the complexity of an algorithm. It provides an **upper limit** on the growth rate of an algorithm's time or space complexity. It represents the **worst-case** scenario for an algorithm's performance, ensuring that the algorithm will never take longer than a certain amount of time.
- **Mathematical Form**:

  ```
  f(n) = O(g(n)) if there exist constants C > 0 and n₀ ≥ 0 such that for all n ≥ n₀, |f(n)| ≤ C * |g(n)|
  ```

  Where `f(n)` is the function representing the algorithm's complexity, and `g(n)` is the comparison function, usually a simple polynomial.
- **Example**: For an algorithm that performs 3n + 2 operations, the time complexity is O(n) because, as `n` grows, the constant factors become negligible, and the dominant term is linear in nature.

```
// Example: O(n)
for (int i = 0; i < n; i++) {
    // constant time operation
}
```

- **Common Big O complexities**:
  - O(1): Constant time
  - O(log n): Logarithmic time
  - O(n): Linear time
  - O(n log n): Log-linear time
  - $O(n^2)$: Quadratic time

---

2. Theta Notation (Θ)

- **Definition**: Theta notation describes the **tight bound** of an algorithm's complexity. It provides both the **upper** and **lower bounds** of the complexity, meaning it describes the **exact** growth rate of an algorithm. When an algorithm has a time complexity of Θ(f(n)), it means the algorithm's complexity grows exactly as f(n), both in the worst and best case.
- **Mathematical Form**:

$$f(n) = \Theta(g(n)) \text{ if there exist constants } C_1 > 0,\ C_2 > 0,\ \text{and } n_0 \geq 0 \text{ such that for all } n \geq n_0,\ C_1 * g(n) \leq f(n) \leq C_2 * g(n)$$

- **Example**: If an algorithm has a time complexity of $2n + 3$, it has a time complexity of Θ(n), because both the upper and lower bounds grow linearly with n.

```
// Example: Θ(n)
for (int i = 0; i < n; i++) {
    // constant time operation
}
```

---

3. Omega Notation (Ω)

- **Definition**: Omega notation describes the **lower bound** of an algorithm's complexity. It provides a **guarantee** that the algorithm will take at least a certain amount of time or space in the best-case scenario. It represents the **best-case** complexity of an algorithm, which means that the algorithm will never take less than a certain amount of time.
- **Mathematical Form**:

```
f(n) = Ω(g(n)) if there exist constants C > 0 and n₀ ≥ 0 such that for all n ≥ n₀, f(n) ≥ C * g(n)
```

- **Example**: For an algorithm with a best-case complexity of $2n$, the time complexity is Ω(n) because, in the best case, the algorithm requires at least linear time to execute.

```
// Example: Ω(n)
for (int i = 0; i < n; i++) {
    // constant time operation
}
```

Summary:

- **Big O (O)**: Upper bound, worst-case time complexity.
- **Theta (Θ)**: Tight bound, exact time complexity.
- **Omega (Ω)**: Lower bound, best-case time complexity.

Arrays in C and C++

An **array** is a collection of elements of the same data type stored in contiguous memory locations. Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

Ways to Enter Data in Arrays

1. **Static Initialization**: The values of the array elements are specified at the time of declaration.

```
int arr[5] = {1, 2, 3, 4, 5};
```

2. **Dynamic Initialization**: The values of the array elements are assigned at runtime using loops or user input.

```
int arr[5];
for (int i = 0; i < 5; i++) {
    cin >> arr[i];
}
```

3. **Partial Initialization**: Not all elements need to be initialized; the remaining elements will be set to zero.

```
int arr[5] = {1, 2};  // Other elements are initialized to 0
```

Types of Arrays

1. Single-Dimensional Arrays

A **single-dimensional array** (1D array) is a simple list of elements stored in a contiguous block of memory.

- **Example**:

```
int arr[5] = {1, 2, 3, 4, 5};  // A 1D array with 5 elements
```

2. Multi-Dimensional Arrays

A **multi-dimensional array** is an array of arrays, where each element can be another array. They are useful for representing more complex structures such as matrices, grids, etc.

- **2D Array Example**:

```
int arr[3][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };  // 2D array (3x3
matrix)
```

- **3D Array Example**:

```
int arr[2][3][4] = { {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}},
                     {{13, 14, 15, 16}, {17, 18, 19, 20}, {21, 22, 23, 24}}
};  // 3D array (2x3x4)
```

Address Calculation of Array Elements

Arrays are stored in contiguous memory locations, and the address of each element can be calculated using a formula. The formula depends on the type of array (1D, 2D, or 3D) and the **storage order** (Row-Major Order or Column-Major Order).

1. Row-Major Order (RMO)

In **Row-Major Order**, the elements of the array are stored row by row. The entire first row is stored first, followed by the second row, and so on.

- **Formula for 1D Array**: For a 1D array, the address of an element can be calculated as:

```
Address of arr[i] = Base address + (i * Size of each element)
```

- **Formula for 2D Array**: For a 2D array, the address of an element at row `i` and column `j` is given by:

```
Address of arr[i][j] = Base address + ((i * number of columns + j) * Size of each element)
```

- **Formula for 3D Array**: For a 3D array, the address of an element at position `i, j, k` is:

```
Address of arr[i][j][k] = Base address + ((i * number of rows * number of columns) + (j * number of columns) + k) * Size of each element
```

2. Column-Major Order (CMO)

In **Column-Major Order**, the elements of the array are stored column by column. The entire first column is stored first, followed by the second column, and so on.

- **Formula for 1D Array**: The formula remains the same as in Row-Major Order for 1D arrays:

```
Address of arr[i] = Base address + (i * Size of each element)
```

- **Formula for 2D Array**: For a 2D array, the address of an element at row `i` and column `j` in Column-Major Order is:

```
Address of arr[i][j] = Base address + ((j * number of rows + i) * Size of each element)
```

- **Formula for 3D Array**: For a 3D array in Column-Major Order, the address of an element at position `i, j, k` is:

```
Address of arr[i][j][k] = Base address + ((k * number of rows * number of columns) + (j * number of rows) + i) * Size of each element
```

Example of Address Calculation

Consider the following example:

```
int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

- **Row-Major Order**:

    - The elements are stored as:

```
{1, 2, 3, 4, 5, 6}
```

    - To find the address of `arr[1][2]`:

```
Address of arr[1][2] = Base address + ((1 * 3 + 2) * Size of each
element)
Address of arr[1][2] = Base address + (5 * Size of each element)
```

- **Column-Major Order**:

    - The elements are stored as:

```
{1, 4, 2, 5, 3, 6}
```

    - To find the address of `arr[1][2]`:

```
Address of arr[1][2] = Base address + ((2 * 2 + 1) * Size of each
element)
Address of arr[1][2] = Base address + (5 * Size of each element)
```

Summary:

- **Single-Dimensional Arrays** store elements linearly in a contiguous block of memory.
- **Multi-Dimensional Arrays** store elements in multiple rows/columns, and the elements can be accessed using multiple indices.
- **Row-Major Order** stores elements row by row, while **Column-Major Order** stores elements column by column.
- Address calculation in arrays depends on the number of rows, columns, and the chosen order (RMO or CMO).

Sorting Algorithms

1. **Bubble Sort**:

- **Definition**: Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

- **Time Complexity**:
  - Best Case: O(n)
  - Average Case: O(n^2)
  - Worst Case: O(n^2)
- **Space Complexity**: O(1)

Example Code:

```
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(arr[j], arr[j+1]);
            }
        }
    }
}
```

---

2. **Selection Sort**:

- **Definition**: Selection Sort is an in-place comparison-based sorting algorithm. It works by dividing the input into two parts: the sorted part and the unsorted part. The algorithm repeatedly selects the smallest (or largest) element from the unsorted part and swaps it with the leftmost unsorted element.
- **Time Complexity**:
  - Best Case: O(n^2)
  - Average Case: O(n^2)
  - Worst Case: O(n^2)
- **Space Complexity**: O(1)

Example Code:

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        int minIdx = i;
        for (int j = i+1; j < n; j++) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
            }
        }
        swap(arr[i], arr[minIdx]);
    }
}
```

3. **Insertion Sort**:

- **Definition**: Insertion Sort works similarly to how we sort playing cards. It picks elements one by one and places them in their correct position in the sorted part of the array.
- **Time Complexity**:
  - Best Case: O(n)
  - Average Case: $O(n^2)$
  - Worst Case: $O(n^2)$
- **Space Complexity**: O(1)

Example Code:

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

4. **Merge Sort**:

- **Definition**: Merge Sort is a divide-and-conquer algorithm. It splits the array into two halves, recursively sorts them, and merges them back together. The merging process ensures the array is sorted.
- **Time Complexity**:
  - Best Case: O(n log n)
  - Average Case: O(n log n)
  - Worst Case: O(n log n)
- **Space Complexity**: O(n)

Example Code:

```
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int left[n1], right[n2];
```

```
    for (int i = 0; i < n1; i++)
        left[i] = arr[l + i];
    for (int i = 0; i < n2; i++)
        right[i] = arr[m + 1 + i];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (left[i] <= right[j]) {
            arr[k] = left[i];
            i++;
        } else {
            arr[k] = right[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = left[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = right[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

---

5. **Quick Sort**:

- **Definition**: Quick Sort is also a divide-and-conquer algorithm. It picks an element as a pivot and partitions the array into two sub-arrays, one with elements smaller than the pivot and the other with elements greater. It then recursively sorts the sub-arrays.
- **Time Complexity**:
  - Best Case: O(n log n)
  - Average Case: O(n log n)
  - Worst Case: $O(n^2)$
- **Space Complexity**: O(log n)

Example Code:

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Searching Algorithms

1. **Linear Search**:

- **Definition**: Linear Search checks every element in the array sequentially until the desired element is found or the end of the array is reached.
- **Time Complexity**: O(n)
- **Space Complexity**: O(1)

Example Code:

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

---

2. **Binary Search**:

- **Definition**: Binary Search works on sorted arrays by repeatedly dividing the search interval in half. If the value is less than the middle element, it narrows the search to the left half; otherwise, to the right half.
- **Time Complexity**: O(log n)
- **Space Complexity**: O(1)

Example Code:

```
int binarySearch(int arr[], int n, int x) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == x)
            return mid; // Element found
        else if (arr[mid] < x)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1; // Element not found
}
```

Array Operations

1. **Insert at Any Position**:

- **Definition**: This operation inserts an element at a specific position in an array. It requires shifting the elements from the position onward to the right to make space for the new element.
- **Example Code**:

```
void insertAtPos(int pos, int value, int arr[], int& size) {
    if (size >= 10 || pos < 0 || pos > size) {
        cout << "Insertion not possible." << endl;
        return;
    }
    for (int i = size; i > pos; i--) {
        arr[i] = arr[i - 1];
    }
    arr[pos] = value;
    size++;
}
```

---

2. **Delete from Any Position**:

- **Definition**: This operation deletes an element at a specific position in the array. It requires shifting the elements from the position onward to the left to fill the gap created by the deletion.
- **Example Code**:

```
void deleteAtPos(int pos, int arr[], int& size) {
    if (pos < 0 || pos >= size) {
        cout << "Deletion not possible." << endl;
        return;
    }
    for (int i = pos; i < size - 1; i++) {
        arr[i] = arr[i + 1];
    }
    size--;
}
```

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

// Bubble Sort
void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

// Selection Sort
void selectionSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < size; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        swap(arr[i], arr[minIndex]);
    }
}

// Insertion Sort
void insertionSort(int arr[], int size) {
    for (int i = 1; i < size; i++) {
        int key = arr[i];
```

```
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

// Merge Sort (Helper Function)
void merge(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        merge(arr, left, mid);
        merge(arr, mid + 1, right);

        // Merging two halves
        int n1 = mid - left + 1;
        int n2 = right - mid;

        int leftArr[n1], rightArr[n2];
        for (int i = 0; i < n1; i++) leftArr[i] = arr[left + i];
        for (int i = 0; i < n2; i++) rightArr[i] = arr[mid + 1 + i];

        int i = 0, j = 0, k = left;
        while (i < n1 && j < n2) {
            if (leftArr[i] <= rightArr[j]) {
                arr[k++] = leftArr[i++];
            } else {
                arr[k++] = rightArr[j++];
            }
        }
        while (i < n1) arr[k++] = leftArr[i++];
        while (j < n2) arr[k++] = rightArr[j++];
    }
}

// Quick Sort (Helper Function)
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);
```

```
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

// Linear Search
int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}

// Binary Search
int binarySearch(int arr[], int size, int target) {
    int low = 0, high = size - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1;
}

// Insert at any position
void insertAtPos(int pos, int value, int arr[], int &size) {
    if (size >= 10 || pos < 0 || pos > size) {
        cout << "Insertion not possible." << endl;
        return;
    }

    for (int i = size; i > pos; i--) {
        arr[i] = arr[i - 1];
    }
    arr[pos] = value;
    size++;
}

// Delete at any position
void deleteAtPos(int pos, int arr[], int &size) {
    if (pos < 0 || pos >= size) {
        cout << "Deletion not possible." << endl;
        return;
    }

    for (int i = pos; i < size - 1; i++) {
        arr[i] = arr[i + 1];
```

```
        }
        size--;
    }

    // Print array
    void printArray(int arr[], int size) {
        for (int i = 0; i < size; i++) {
            cout << arr[i] << " ";
        }
        cout << endl;
    }

    int main() {
        int size;
        cout << "Enter the number of elements: ";
        cin >> size;

        int arr[size];
        cout << "Enter the elements: ";
        for (int i = 0; i < size; i++) {
            cin >> arr[i];
        }

        // Display original array
        cout << "Original array: ";
        printArray(arr, size);

        // Sorting Algorithms
        cout << "Bubble Sort:" << endl;
        bubbleSort(arr, size);
        printArray(arr, size);

        cout << "Selection Sort:" << endl;
        selectionSort(arr, size);
        printArray(arr, size);

        cout << "Insertion Sort:" << endl;
        insertionSort(arr, size);
        printArray(arr, size);

        cout << "Merge Sort:" << endl;
        merge(arr, 0, size - 1);
        printArray(arr, size);

        cout << "Quick Sort:" << endl;
        quickSort(arr, 0, size - 1);
        printArray(arr, size);

        // Searching Algorithms
        int target;
        cout << "Enter the target element for search: ";
        cin >> target;

        int result = linearSearch(arr, size, target);
```

```
    cout << "Linear Search: Element " << target << " found at index " << result <<
endl;

    result = binarySearch(arr, size, target);
    cout << "Binary Search: Element " << target << " found at index " << result <<
endl;

    // Insertion and Deletion operations
    int pos, value;
    cout << "Enter position and value to insert at position: ";
    cin >> pos >> value;
    insertAtPos(pos, value, arr, size);
    cout << "After insertion: ";
    printArray(arr, size);

    cout << "Enter position to delete from: ";
    cin >> pos;
    deleteAtPos(pos, arr, size);
    cout << "After deletion: ";
    printArray(arr, size);

    return 0;
}
```

```
#include <stdio.h>

// Bubble Sort
void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// Selection Sort
void selectionSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < size; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        int temp = arr[i];
        arr[i] = arr[minIndex];
```

```
            arr[minIndex] = temp;
        }
    }

    // Insertion Sort
    void insertionSort(int arr[], int size) {
        for (int i = 1; i < size; i++) {
            int key = arr[i];
            int j = i - 1;
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
        }
    }

    // Merge Sort (Helper Function)
    void merge(int arr[], int left, int right) {
        if (left < right) {
            int mid = left + (right - left) / 2;
            merge(arr, left, mid);
            merge(arr, mid + 1, right);

            // Merging two halves
            int n1 = mid - left + 1;
            int n2 = right - mid;

            int leftArr[n1], rightArr[n2];
            for (int i = 0; i < n1; i++) leftArr[i] = arr[left + i];
            for (int i = 0; i < n2; i++) rightArr[i] = arr[mid + 1 + i];

            int i = 0, j = 0, k = left;
            while (i < n1 && j < n2) {
                if (leftArr[i] <= rightArr[j]) {
                    arr[k++] = leftArr[i++];
                } else {
                    arr[k++] = rightArr[j++];
                }
            }
            while (i < n1) arr[k++] = leftArr[i++];
            while (j < n2) arr[k++] = rightArr[j++];
        }
    }

    // Quick Sort (Helper Function)
    int partition(int arr[], int low, int high) {
        int pivot = arr[high];
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
```

```
            arr[j] = temp;
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return i + 1;
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

// Linear Search
int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}

// Binary Search
int binarySearch(int arr[], int size, int target) {
    int low = 0, high = size - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1;
}

// Insert at any position
void insertAtPos(int pos, int value, int arr[], int *size) {
    if (*size >= 10 || pos < 0 || pos > *size) {
        printf("Insertion not possible.\n");
        return;
    }

    for (int i = *size; i > pos; i--) {
        arr[i] = arr[i - 1];
    }
    arr[pos] = value;
```

```
    (*size)++;
}

// Delete at any position
void deleteAtPos(int pos, int arr[], int *size) {
    if (pos < 0 || pos >= *size) {
        printf("Deletion not possible.\n");
        return;
    }

    for (int i = pos; i < *size - 1; i++) {
        arr[i] = arr[i + 1];
    }
    (*size)--;
}

// Print array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int size;
    printf("Enter the number of elements: ");
    scanf("%d", &size);

    int arr[size];
    printf("Enter the elements: ");
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }

    // Display original array
    printf("Original array: ");
    printArray(arr, size);

    // Sorting Algorithms
    printf("Bubble Sort:\n");
    bubbleSort(arr, size);
    printArray(arr, size);

    printf("Selection Sort:\n");
    selectionSort(arr, size);
    printArray(arr, size);

    printf("Insertion Sort:\n");
    insertionSort(arr, size);
    printArray(arr, size);

    printf("Merge Sort:\n");
    merge(arr, 0, size - 1);
```

```
    printArray(arr, size);

    printf("Quick Sort:\n");
    quickSort(arr, 0, size - 1);
    printArray(arr, size);

    // Searching Algorithms
    int target;
    printf("Enter the target element for search: ");
    scanf("%d", &target);

    int result = linearSearch(arr, size, target);
    printf("Linear Search: Element %d found at index %d\n", target, result);

    result = binarySearch(arr, size, target);
    printf("Binary Search: Element %d found at index %d\n", target, result);

    // Insertion and Deletion operations
    int pos, value;
    printf("Enter position and value to insert at position: ");
    scanf("%d %d", &pos, &value);
    insertAtPos(pos, value, arr, &size);
    printf("After insertion: ");
    printArray(arr, size);

    printf("Enter position to delete from: ");
    scanf("%d", &pos);
    deleteAtPos(pos, arr, &size);
    printf("After deletion: ");
    printArray(arr, size);

    return 0;
}
```