C++ Programming For Absolute Beginners & Question Set IV

Sahas Talasila

Contents

Introduction

In this document, we will explore some fundamental **searching** and **sorting** algorithms in C++. We will look at:

- Two common searching techniques:
  - **Linear Search**
  - **Binary Search**
- Three popular sorting algorithms:
  - **Merge Sort**
  - **Bubble Sort**
  - **Selection Sort**

We will explain each algorithm in simple terms, show its **pseudocode**, and provide a **C++** code example. Then, you will find **practice questions** to test your understanding, including multiple choice, short-form, and open-ended coding tasks with solutions.

2.1 Linear Search

Linear search (also known as *sequential search*) is the simplest searching algorithm. It checks each element in the array one by one until the target value is found or until the end of the array is reached.

Intuitive Explanation

Imagine you have a list of names on a sheet of paper, and you want to find whether a specific name is on that list. You would start from the top of the list and keep checking each name until you find the one you're looking for or run out of names.

Pseudocode (Linear Search)

```
procedure LinearSearch(array A, value x)
    for i from 0 to A.length - 1
        if A[i] == x
            return i  // index where x is found
    end for
    return -1         // x not found in the array
end procedure
```

C++ Example (Linear Search)

```cpp
#include <iostream>


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

int main() {
    int arr[] = {4, 2, 5, 1, 9};
    int size = sizeof(arr)/sizeof(arr[0]);
    int target = 5;

    int result = linearSearch(arr, size, target);
    if(result != -1) {
        std::cout << "Element found at index: " << result << std::endl;
    } else {
        std::cout << "Element not found." << std::endl;
    }

    return 0;
}
```

2.2 Binary Search

Binary search is a more efficient searching algorithm but it requires the array (or list) to be *sorted*. It works by repeatedly dividing the search space in half.

Intuitive Explanation

If you have a dictionary and you want to look up a word, you do not read each word from the beginning. Instead, you open the dictionary around the middle, compare the word you're looking for with the one on the page, and decide whether to move left or right depending on alphabetical order.

Pseudocode (Binary Search)

```
procedure BinarySearch(sortedArray A, value x)
    left = 0
    right = A.length - 1

    while left <= right
        mid = (left + right) / 2
        if A[mid] == x
            return mid      // x found at index mid
        else if A[mid] < x
            left = mid + 1  // x is in the right half
        else
            right = mid - 1 // x is in the left half
    end while

    return -1               // x not found
end procedure
```

C++ Example (Binary Search)

```cpp
#include <iostream>

int binarySearch(int arr[], int size, int target) {
    int left = 0;
    int right = size - 1;

    while(left <= right) {
        int mid = (left + right) / 2;
        if(arr[mid] == target) {
            return mid;
        } else if(arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}

int main() {
    // Sorted array is required
    int arr[] = {1, 2, 4, 5, 9};
    int size = sizeof(arr)/sizeof(arr[0]);
    int target = 5;

    int result = binarySearch(arr, size, target);
    if(result != -1) {
        std::cout << "Element found at index: " << result << std::endl;
    } else {
        std::cout << "Element not found." << std::endl;
    }

```

```
33      return 0;
34  }
```

3.1 Bubble Sort

Bubble sort is a simple sorting method that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This pass-through is repeated until the list is sorted.

Intuitive Explanation

Think of bubbles rising to the top of water: the largest (heaviest) element keeps moving (or "bubbling") to the end of the array with each pass.

Pseudocode (Bubble Sort)

```
procedure BubbleSort(array A)
    n = A.length
    repeat
        swapped = false
        for i from 0 to n - 2
            if A[i] > A[i+1]
                swap A[i] and A[i+1]
                swapped = true
            end if
        end for
    until not swapped
end procedure
```

C++ Example (Bubble Sort)

```cpp
#include <iostream>


void bubbleSort(int arr[], int n) {
    bool swapped;
    do {
        swapped = false;
        for(int i = 0; i < n - 1; i++) {
            if(arr[i] > arr[i + 1]) {
                // Swap
                int temp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = temp;
                swapped = true;
            }
        }
        // Keep doing passes until no swaps in a pass
    } while(swapped);
}

int main() {
    int arr[] = {5, 1, 4, 2, 8};
    int n = sizeof(arr)/sizeof(arr[0]);

    bubbleSort(arr, n);

    std::cout << "Sorted array (Bubble Sort): ";
    for(int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
```

```
    }
    std::cout << std::endl;
    return 0;
}
```

3.2 Selection Sort

Selection sort divides the array into a sorted and unsorted region. It repeatedly finds the *minimum* element from the unsorted region and places it at the beginning of the unsorted region.

Intuitive Explanation

Imagine if you had to line up your friends by height. You look for the shortest person and move them to the front of the line. Then you look for the second shortest among the remaining people, place them next, and so on.

Pseudocode (Selection Sort)

```
procedure SelectionSort(array A)
    n = A.length
    for i from 0 to n - 2
        minIndex = i
        for j from i+1 to n - 1
            if A[j] < A[minIndex]
                minIndex = j
            end if
        end for
        if minIndex != i
            swap A[i] and A[minIndex]
        end if
    end for
end procedure
```

C++ Example (Selection Sort)

```cpp
#include <iostream>


void selectionSort(int arr[], int n) {
    for(int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for(int j = i + 1; j < n; j++) {
            if(arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // Swap
        if(minIndex != i) {
            int temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        }
    }
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
```

```
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);

    std::cout << "Sorted array (Selection Sort): ";
    for(int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

3.3 Merge Sort

Merge sort is a *divide-and-conquer* algorithm. It divides the array into halves, sorts each half recursively, and then merges the two sorted halves.

Intuitive Explanation

If you have a bunch of small sorted lists, you can merge them into a single large sorted list by repeatedly taking the smallest (or next smallest) from the front of each list.

Pseudocode (Merge Sort)

```
procedure MergeSort(array A)
    if A.length > 1
        mid = A.length / 2
        leftArray = A[0..mid-1]
        rightArray = A[mid..A.length-1]

        MergeSort(leftArray)
        MergeSort(rightArray)

        A = merge(leftArray, rightArray)
    end if
end procedure

procedure merge(leftArray, rightArray)
    i = 0, j = 0
    mergedArray = empty

    while i < leftArray.length and j < rightArray.length
        if leftArray[i] <= rightArray[j]
            mergedArray.append(leftArray[i])
            i = i + 1
        else
            mergedArray.append(rightArray[j])
            j = j + 1
        end if
    end while

    // Append any remaining elements from leftArray or rightArray
    while i < leftArray.length
        mergedArray.append(leftArray[i])
        i++
```

```
    end while

    while j < rightArray.length
        mergedArray.append(rightArray[j])
        j++
    end while

    return mergedArray
end procedure
```

C++ Example (Merge Sort)

```cpp
#include <iostream>


void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Temp arrays
    int *L = new int[n1];
    int *R = new int[n2];

    // Copy data
    for(int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for(int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    // Merge temp arrays back into arr[left..right]
    int i = 0, j = 0, k = left;
    while(i < n1 && j < n2) {
        if(L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy remaining elements
    while(i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while(j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }

    // Cleanup
    delete[] L;
    delete[] R;
}

void mergeSort(int arr[], int left, int right) {
```

```
    if(left < right) {
        int mid = (left + right) / 2;

        // Sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    mergeSort(arr, 0, n-1);

    std::cout << "Sorted array (Merge Sort): ";
    for(int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

4.1 1. Multiple Choice Questions (10 Questions)

1. Which searching algorithm operates by sequentially checking each element in the list?
   (a) (A) Linear Search
   (b) (B) Binary Search
   (c) (C) Merge Search
   (d) (D) Quick Search
2. In which scenario is **binary search** *not* suitable?
   (a) (A) Array is sorted in ascending order
   (b) (B) Array is sorted in descending order
   (c) (C) Array is not sorted at all
   (d) (D) Array has negative values
3. Which sorting algorithm works by repeatedly swapping adjacent elements if they are in the wrong order?
   (a) (A) Bubble Sort
   (b) (B) Selection Sort
   (c) (C) Merge Sort
   (d) (D) Insertion Sort
4. **Selection sort** finds the minimum element and places it at:
   (a) (A) The end of the array
   (b) (B) The beginning of the array
   (c) (C) The middle of the array
   (d) (D) Random position in the array
5. **Merge sort** is based on which algorithmic technique?
   (a) (A) Greedy
   (b) (B) Divide and Conquer
   (c) (C) Dynamic Programming
   (d) (D) Brute Force
6. In the worst case, how many comparisons does **binary search** make in an array of n elements?
   (a) (A) n
   (b) (B) n^2
   (c) (C) $\log_2 n$
   (d) (D) $n \log_2 n$
7. Which algorithm **always** has a time complexity of $O(n^2)$ in the worst case among the following?
   (a) (A) Merge Sort

(b) (B) Bubble Sort

(c) (C) Quick Sort (average case)

(d) (D) Binary Search

8. Which of the following algorithms **requires** the list to be sorted before it can be used?

   (a) (A) Linear Search

   (b) (B) Selection Sort

   (c) (C) Binary Search

   (d) (D) Bubble Sort

9. During **merge sort**, the process of dividing continues until:

   (a) (A) The array has been divided into two equal parts

   (b) (B) Each sub-array has one or zero elements

   (c) (C) We find the pivot

   (d) (D) The first and last elements are the same

10. In **bubble sort**, in one pass, the largest element tends to:

    (a) (A) Bubble up to its correct position at the end

    (b) (B) Settle at the front

    (c) (C) Get placed in the middle of the array

    (d) (D) Vanish from the array

4.2 2. Short Answer Questions (10 Questions)

Provide concise answers (1–3 sentences) for each.

1. What is the main difference between **linear search** and **binary search**?
2. Why must an array be **sorted** before applying binary search?
3. Explain in simple terms how **bubble sort** works.
4. When is **selection sort** preferred over bubble sort?
5. What does **merge sort** do after it divides the array into halves?
6. Give the worst-case time complexity of **bubble sort**.
7. Can **binary search** be used on a linked list efficiently? Why or why not?
8. What is the best-case scenario for **bubble sort**?
9. Name the technique used by **merge sort** and briefly explain it.
10. How many passes does **selection sort** need to place all elements in the correct position?

4.3 3. Open-Ended Coding Questions (5 Questions)

1. **Write a C++ program to perform a linear search** on an array of integers. Prompt the user for the size of the array, the elements, and the target number. Print the result (index or not found).
2. **Implement binary search in C++** such that you use a recursive function `binarySearchRecursive()`. Test it with a sorted list of integers.
3. **Write a bubble sort function** that sorts strings instead of integers. Demonstrate it with an array of names (e.g., `"Alice", "Bob", "Charlie"`).
4. **Implement selection sort in C++** but write it in a way that sorts in descending order rather than ascending order.
5. **Create a program that uses merge sort** to sort an array of floating-point numbers. Compare the sorted result with the output of another built-in sort function (e.g., `std::sort`) to ensure correctness.

4.4 4. Spot the Error Questions

1. Spot the error in this linear search implementation:

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

2. Spot the error in this binary search implementation:

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

3. Spot the error in this bubble sort implementation:

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

4. Spot the error in this selection sort implementation:

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

```
                min = j;
        }
        arr[i] = arr[min];
        arr[min] = arr[i];
    }
}
```

5. Spot the error in this merge sort split:

```
void mergeSort(int arr[], int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2;
        mergeSort(arr, low, mid);
        mergeSort(arr, mid, high);
        merge(arr, low, mid, high);
    }
}
```

6. Spot the error in the merge function used in merge sort:

```
void merge(int arr[], int left, int mid, int right) {
    int i = left, j = mid, k = 0;
    int temp[right-left+1];

    while (i < mid && j <= right) {
        if (arr[i] < arr[j])
            temp[k++] = arr[i++];
        else
            temp[k++] = arr[j++];
    }

    while (i < mid)
        temp[k++] = arr[i++];
    while (j <= right)
        temp[k++] = arr[j++];

    for (int l = left; l <= right; l++)
        arr[l] = temp[l];
}
```

7. Spot the error in this optimized bubble sort:

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        bool swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j+1]) {
                std::swap(arr[j], arr[j+1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}
```

8. Spot the error in this linear search that handles strings:

```
int linearSearch(std::string arr[], int n, std::string key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key)
            return i;
    }
    return;
}
```

9. Spot the error in binary search using recursion:

```
int binarySearch(int arr[], int low, int high, int key) {
    if (low > high)
        return -1;

    int mid = (low + high) / 2;

    if (arr[mid] == key)
        return mid;
    else if (arr[mid] < key)
        return binarySearch(arr, mid + 1, high, key);
    else
        binarySearch(arr, low, mid - 1, key);
}
```

10. Spot the error in this descending 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(arr[i], arr[j]);
        }
    }
}
```

Conclusion

In this document, we covered:

- **Searching Algorithms:** Linear Search and Binary Search
- **Sorting Algorithms:** Bubble Sort, Selection Sort, and Merge Sort

Each algorithm has different use cases and performance characteristics. By understanding their strengths and weaknesses, you can choose the most suitable algorithm for a given problem. Practice implementing these algorithms, and experiment with different data sets to get a feel for their performance.