C Programming For Absolute Beginners & Question Set IV

Sahas Talasila

Contents

Data Structures and Algorithms Introduction

In this section I will cover the introduction to pseudocode and understanding how algorithms and data structures form.

1.1 Pseudocode

This subsection is entirely **OPTIONAL**

Pseudocode is a way of structuring your ideas, similar to flowcharts. It is completely optional, but I like to visualise what the code would look like in simpler terms. I decided to introduce it now as the search/sort algorithms can be tricky to code, so you can understand it better. I have also explained it without pseudocode.

Algorithm 1 How to Read Pseudocode

procedure ReadPseudocode
    Step 1: Identify the structure of the algorithm
        Algorithms in pseudocode typically consist of procedures, loops, conditionals, and variables.
    Step 2: Understand the procedure declaration
        A procedure is often declared using `\Procedure` or `Function`.
        It defines the actions of the algorithm, with input parameters and output results.
    Step 3: Recognize loops and conditionals
        `For`, `While`, and `Repeat` represent loops in pseudocode.
        `If`, `ElseIf`, `Else` represent decision-making (conditionals).
    Step 4: Understand variable assignments
        Assignment operations are typically written as `variable ← value`.
    Step 5: Follow the flow of execution
        Pseudocode is executed line by line, following the logic defined by the algorithm.
        After understanding a loop or conditional, track how values change or decisions affect execution.
    Step 6: Consider edge cases and assumptions
        Think about what happens in the algorithm for special cases (e.g., empty lists or boundary values).
        Some pseudocode may omit details, so use your knowledge of the algorithm to fill in gaps.
    Example of pseudocode:
        `For each element in a list`
        `If element is equal to target, return the index.`
        `Else, continue to next element.`
end procedure

We will look at **5** different search/sort algorithms, along with an example of each algorithm and how they work. First we will look at two search algorithms: **Linear Search** and **Binary Search**. After that, we will look at 3 sorting algorithms: **Selection Sort, Bubble Sort** and **Merge Sort**.

1.2 Linear Search

Linear search checks each element in an array sequentially until the desired element is found or the array ends, returning an error message. In the `main()` function, we have a function call, error handling and size checking script: `sizeof(arr)` gives the total size of the array in bytes. `sizeof(arr[0])` gives the size of one element (in this case, int is typically 4 bytes). Dividing the total size by the size of one element gives the number of elements. We haven't used this in-built function before.

Let's look at the actual search function.

Function Header: `int linearSearch(int arr[], int n, int target);`.
arr[]: The array to search in.
n: The size of the array.
target: The value to search for.
Returns the index of the target, or -1 if not found.

Algorithm 2 Linear Search

procedure LINEARSEARCH(array, target)
    for $i = 0$ **to** length(array) - 1 **do**
        if array[i] = target **then**
            Return i ▷ Target found at index i
        end if
    end for
    Return -1 ▷ Target not found
end procedure

```c
#include <stdio.h>
// Create a function that iterates through an array

int linearSearch(int arr[], int size, int target) { // arr[] -> what we will search
    for (int i = 0; i < size; i++) {                // int size -> How large our array is
        if (arr[i] == target) {                     // Target -> The number we are looking for
            return i; // Return the index of the target
        }
    }
    return -1; // Return -1 if the target is not found
}

int main() {
    int arr[] = {3, 5, 7, 9, 11};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 7;

    int result = linearSearch(arr, size, target);
    if (result != -1) {
        printf("Element found at index %d\n", result);
    } else {
        printf("Element not found\n");
    }

    return 0;
}
```

1.3 Binary Search

Binary search works by splitting a **sorted array** (Hint: You might need to use this algorithm with a sorting algorithm in your exam) into **two halves** and checks if the target is in either half. It keeps repeating this process until we cannot halve any more or we reach the target value.
The array `[arr]` **must** be sorted for binary search to work.
Target Value: We search for 10 in this array.
Function Call: Calls `binarySearch` with the **array**, its **size**, and the **target**.

Let's look at the actual binary search function. We start by initialising two 'pointers'. In this case we will think of them as magnifying glasses, compared to actual pointers, which we learnt before:

low = 0; high = n - 1; **low** starts at the first index. **High starts at the last index**.

Midpoint Calculation:

`mid = low + (high - low) / 2; This avoids memory overflow compared to (low + high) / 2.`

Compare Midpoint:

If arr[mid] == target, the target is found, and the index is returned.
If arr[mid] < target, update low = mid + 1 to search the right half.
If arr[mid] > target, update high = mid - 1 to search the left half.

Repeat Until Found or Exhausted:

The loop continues until low > high. Pseudocode below for a better structure.

Algorithm 3 Binary Search

```
procedure BinarySearch(array, target)
    low ← 0
    high ← length(array) - 1
    while low ≤ high do
        mid ← floor((low + high) / 2)
        if array[mid] = target then
            Return mid                          ▷ Target found at index [mid]
        else if array[mid] ¡ target then
            low ← mid + 1
        else
            high ← mid - 1
        end if
    end while
    Return -1                                   ▷ Target not found
end procedure
```

C code on the next page.

```c
/* Binary Search:
Concept:
Binary search is a faster searching technique that works on sorted arrays. It
    repeatedly divides the search interval in half, until the TARGET has been found.
*/

// Example
#include <stdio.h>

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

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid; // Element found
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1; // Element not found
}

int main() {
    int arr[] = {2, 4, 6, 8, 10, 12};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 8;

    int result = binarySearch(arr, size, target);
    if (result != -1) {
        printf("Element found at index %d\n", result);
    } else {
        printf("Element not found\n");
    }

    return 0;
}
```

1.4 Selection Sort

Start with the outer loop, and then look at the inner loop.

Selection Sort Function:

Outer Loop (Passes):

`for (int i = 0; i < n - 1; i++)`

Iterates through the unsorted portion of the array.

we then find the Minimum Element:

`int minIndex = i;`

Assume the first unsorted element is the smallest. The inner loop finds the smallest element in the unsorted portion.

Swapping: Swap the smallest element with the first unsorted element:

Now let's look at the main() function, which is much simpler:

Print original array. Then sort the array using selectionSort and print out the sorted array. I've also included the execution flow or trace, to show what it looks like:

Execution Flow: For the array [29, 10, 14, 37, 13]:

Pass 1: Minimum is 10, swap with 29 → [10, 29, 14, 37, 13]. Pass 2: Minimum is 13, swap with 29 → [10, 13, 14, 37, 29]. Repeat until fully sorted: [10, 13, 14, 29, 37].

```c
/* Concept - Selection sort repeatedly selects the smallest element from the unsorted
   portion and places it in the sorted portion. */

// Example
#include <stdio.h>

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

int main() {
    int arr[] = {29, 10, 14, 37, 13};
    int size = sizeof(arr) / sizeof(arr[0]);

    selectionSort(arr, size);

    printf("Sorted array: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

1.5 Bubble Sort

Let's try and break it down, starting with the selection sort function.

Start with the outer loop (iterates through the array, one-by-one):

```
for (int i = 0; i < n - 1; i++)
```

Each pass ensures the largest element in the unsorted portion "bubbles up" to its correct position. So the total number of passes is (n - 1) loops.

Inner Loop (Comparisons):

```
for (int j = 0; j < n - i - 1; j++)
```

Compares adjacent elements within the unsorted portion. It then decreases in size as the largest elements are sorted.

Swapping Elements:

```
if (arr[j] > arr[j + 1])
```

Swaps the elements if the left element is greater than the right.
Temporary variable `temp` is used to facilitate swapping.

Now, let's look at the main() function.

we start with an Array Declaration: `int arr[] = {64, 34, 25, 12, 22, 11, 90};`

(Better if it is an array with unsorted integers.)

Next we can calculate size:

```
int size = sizeof(arr) / sizeof(arr[0]);
```

it is the same logic as in previous examples.

We then print out the original array:

1. Calls printArray to display the unsorted array.
2. Sort the Array:
3. Calls bubbleSort to sort the array.
4. Print Sorted Array:

Displays the sorted array after sorting.

Algorithm 4 Bubble Sort

```
procedure BubbleSort(array)
    for i = 0 to length(array) - 1 do
        for j = 0 to length(array) - i - 2 do
            if array[j] ¿ array[j + 1] then
                Swap array[j] and array[j + 1]
            end if
        end for
    end for
end procedure
```

```c
// Bubble Sort

/* Concept
Bubble sort repeatedly steps through the list, compares adjacent elements, and swaps
    them if they are in the wrong order.
*/

// Example
#include <stdio.h>

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;
            }
        }
    }
}

int main() {
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int size = sizeof(arr) / sizeof(arr[0]);

    bubbleSort(arr, size);

    printf("Sorted array: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

1.6 Merge Sort

We start with an array declaration:

int arr[] = 38, 27, 43, 3, 9, 82, 10; An unsorted array has been declared.

next we find Array Size:

`int size = sizeof(arr) / sizeof(arr[0]);`

The number of elements in the array is calculated. Then, we have a function call to Merge Sort:

mergeSort(arr, 0, n - 1);
Passes the array, the leftmost index (0), and the rightmost index (n-1).
Print Sorted Array:

Calls printArray to display the array after sorting.

Pseudocode on the next page:

Algorithm 5 Merge Sort

```
procedure MergeSort(array)
    if length(array) > 1 then
        mid ← floor(length(array) / 2)
        left ← array[0 ... mid]
        right ← array[mid + 1 ... length(array) - 1]
        MergeSort(left)
        MergeSort(right)
        Merge(array, left, right)
    end if
end procedure
procedure Merge(array, left, right)
    i ← 0, j ← 0, k ← 0
    while i ¡ length(left) and j ¡ length(right) do
        if left[i] ¡ right[j] then
            array[k] ← left[i]
            i ← i + 1
        else
            array[k] ← right[j]
            j ← j + 1
        end if
        k ← k + 1
    end while
    while i ¡ length(left) do
        array[k] ← left[i]
        i ← i + 1
        k ← k + 1
    end while
    while j ¡ length(right) do
        array[k] ← right[j]
        j ← j + 1
        k ← k + 1
    end while
end procedure
```

```c
#include <stdio.h>

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

    int L[n1], R[n2];
    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];
    }

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

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

int main() {
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int size = sizeof(arr) / sizeof(arr[0]);

    mergeSort(arr, 0, size - 1);

    printf("Sorted array: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Questions

I have included **3 types of questions**. First, **Multiple Choice**, next **Short Form** and **Open Ended Questions**.

1. Which of the following is the **worst-case time complexity** of **linear search**?
   (a) $\mathcal{O}(1)$
   (b) $\mathcal{O}(\log n)$
   (c) $\mathcal{O}(n)$
   (d) $\mathcal{O}(n \log n)$
2. What is the **average-case time complexity** of **binary search** on a sorted array?
   (a) $\mathcal{O}(1)$
   (b) $\mathcal{O}(n)$
   (c) $\mathcal{O}(\log n)$
   (d) $\mathcal{O}(n^2)$
3. Which of the following sorting algorithms has an **average-case time complexity** of $\mathcal{O}(n^2)$?
   (a) Merge Sort
   (b) Bubble Sort
   (c) Quick Sort
   (d) Heap Sort
4. In **merge sort**, which strategy is used to break down the array into subproblems?
   (a) Dynamic Programming
   (b) Greedy Approach
   (c) Divide and Conquer
   (d) Backtracking
5. Which sorting algorithm **repeatedly finds the minimum element** from the unsorted part and places it at the beginning?
   (a) Bubble Sort
   (b) Selection Sort
   (c) Merge Sort
   (d) Insertion Sort
6. Which condition is required for **binary search** to work correctly?
   (a) The array must be unsorted.
   (b) The array must be sorted.
   (c) The array must have all distinct elements.
   (d) The array can contain only positive numbers.
7. How many **passes** does **bubble sort** need in the worst case for an array of size n?

(a) 1

(b) $n - 1$

(c) $\log n$

(d) n^2

8. Which sorting algorithm is typically **not stable** by default?

   (a) Bubble Sort

   (b) Merge Sort

   (c) Selection Sort

   (d) Insertion Sort

9. What is the **primary advantage** of **linear search**?

   (a) It only works on sorted arrays.

   (b) It has constant time complexity.

   (c) It requires no extra space and works on unsorted data.

   (d) It is more efficient than binary search for large n.

10. During the **merge step** of merge sort, what happens?

    (a) A pivot is selected to split the array.

    (b) Adjacent elements are swapped if they are out of order.

    (c) Two sorted sublists are combined into a single sorted list.

    (d) The minimum element is placed at the beginning of the array.

Short-Answer Questions (10)

1. Name one **key difference** between binary search and linear search in terms of requirements on the data set.
2. Which sorting algorithm among Bubble Sort, Merge Sort, and Selection Sort generally has the **lowest worst-case time complexity**?
3. Define a **stable sorting algorithm** in one sentence.
4. In **bubble sort**, how many elements are guaranteed to be in their final position after the i-th pass?
5. Explain **why binary search is more efficient** than linear search on large, sorted arrays.
6. Which searching algorithm, **binary or linear**, can handle *unsorted* data directly?
7. In **merge sort**, what size are the sublists before the algorithm starts merging them back?
8. Which sorting algorithm **repeatedly swaps adjacent elements** if they are in the wrong order?

Long-Form (Coding) Questions (5)

(Hint 1: For all of these questions, write down your thought process and write down how the algorithm works properly.)

(Hint 2: Be careful when choosing the array size values, a smaller array will make debugging easier (maximum 5 ˜10 elements) and remember that it is **zero-based** index)

1. **Implement Linear Search:** Write a C function `int linearSearch(int arr[], int n, int target)` that returns the index of `target` in the array (or -1 if not found). In `main`, read the array size `n`, fill the array, read a `target` value, call `linearSearch`, then print the result.

2. **Implement Binary Search:** Write a C function `int binarySearch(int arr[], int n, int target)` assuming the array is sorted in ascending order. Test it in `main` by reading a sorted array and a `target` value, then printing whether `target` was found.

3. **Implement Bubble Sort:** Write a function `void bubbleSort(int arr[], int n)` that sorts an array in ascending order using the bubble sort algorithm. In `main`, read an array from the user, call `bubbleSort`, and then display the sorted result.

4. **Implement Selection Sort:** Write a function `void selectionSort(int arr[], int n)` that sorts an array in ascending order by repeatedly finding the minimum element from the unsorted part. In `main`, demonstrate this by reading an array, calling `selectionSort`, and printing the final sorted array.

5. **Implement Merge Sort:** Write a function `void mergeSort(int arr[], int left, int right)` that uses the divide-and-conquer approach to sort an array in ascending order. Implement a separate `merge` function to combine two sorted subarrays. Show how you'd call `mergeSort` from `main`.