C Programming For Absolute Beginners & Question Set IV

Sahas Talasila

Contents

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 is like looking for your friend in a crowded cafe by walking through each table one by one until you find them. It's the most straightforward searching method.

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.

- Start at the beginning of the list
- Check each item one by one
- If you find what you're looking for, stop and return the position
- If you reach the end without finding it, the item isn't there

Example 1: Finding a Song in Your Playlist

```
Position: 0    1    2    3    4    5    6
Songs:   "A"  "B"  "C"  "D"  "E"  "F"  "G"

Looking for "E":
Check 0: "A" != "E"
Check 1: "B" != "E"
Check 2: "C" != "E"
Check 3: "D" != "E"
Check 4: "E" = "E" -> Found at position 4
```

Example 2: Finding a Missing Item

```
Looking for "Z":
Check all positions 0 through 6
Never find "Z" -> Return "not found"
```

Let's look at the actual search functionon the next page (pseudocode and C code)

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 is like finding a word in a dictionary. You open to the middle and decide whether to go left or right based on alphabetical order. It only works on **sorted data**.

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**.

- Look at the middle item of the sorted list
- If it's what you want, you're done!
- If your target is smaller, search the left half
- If your target is larger, search the right half
- Repeat until found or no more items to check

Example: Finding Your Test Score

```
Scores: [45, 52, 67, 73, 81, 89, 92, 95, 98]
Looking for 89:

Step 1: Check position 4 -> 81 < 89 -> Search right half
Step 2: Check position 6 -> 92 > 89 -> Search left half
Step 3: Check position 5 -> 89 = 89 -> Found!
```

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 and C code on the next two pages for a better structure.

Algorithm 3 Binary Search

```
1: procedure BinarySearch(array, target)
2:     low ← 0
3:     high ← length(array) - 1
4:     while low ≤ high do
5:         mid ← floor((low + high) / 2)
6:         if array[mid] = target then
7:             Return mid                          ▷ Target found at index [mid]
8:         else if array[mid] ¡ target then
9:             low ← mid + 1
10:        else
11:            high ← mid - 1
12:        end if
13:    end while
14:    Return -1                                   ▷ Target not found
15: end procedure
```

```c
#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

Selection sort is like organising your bookshelf by repeatedly finding the shortest book and placing it at the beginning.

- Find the smallest item in the list
- Swap it with the first item
- Repeat for the remaining list

Example: Sorting Scores

```
Initial: [64, 25, 12, 22, 11]

Before: [64, 25, 12, 22, 11]
Pass 1: Find 11 -> Swap with 64 -> [11, 25, 12, 22, 64]

Before: [64, 25, 12, 22, 11]
Pass 2: Find 12 -> Swap with 25 -> [11, 12, 25, 22, 64]

Before: [64, 25, 12, 22, 11]
Pass 3: Find 22 -> Swap with 25 -> [11, 12, 22, 25, 64]

Before: [64, 25, 12, 22, 11]
Pass 4: Already sorted -> [11, 12, 22, 25, 64]
```

Algorithm 4 Selection Sort

Require: Array A of length n
Ensure: A is sorted in ascending order

```
for i = 0 to n − 2 do
    minIndex ← i
    for j = i + 1 to n − 1 do
        if A[j] < A[minIndex] then
            minIndex ← j
        end if
    end for
    if minIndex ≠ i then
        Swap A[i] and A[minIndex]
    end if
end for
```

1.4.1 Code Walkthrough

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

Bubble sort is like bubbles rising to the surface. Larger elements "bubble up" to the end by swapping adjacent elements.

- Compare adjacent elements
- Swap if out of order
- Repeat until no swaps needed

1.5.1 Example: Sorting Your Friends by Height

Let's sort these heights (in cm) from shortest to tallest:

Initial: [170, 155, 180, 160, 165]

```
Pass 1:
Compare 170 & 155: 170 > 155, swap → [155, 170, 180, 160, 165]
Compare 170 & 180: 170 < 180, no swap → [155, 170, 180, 160, 165]
Compare 180 & 160: 180 > 160, swap → [155, 170, 160, 180, 165]
Compare 180 & 165: 180 > 165, swap → [155, 170, 160, 165, 180]

Pass 2:
Compare 155 & 170: no swap → [155, 170, 160, 165, 180]
Compare 170 & 160: 170 > 160, swap → [155, 160, 170, 165, 180]
Compare 170 & 165: 170 > 165, swap → [155, 160, 165, 170, 180]
Compare 170 & 180: no swap → [155, 160, 165, 170, 180]

Pass 3:
Compare 155 & 160: no swap
Compare 160 & 165: no swap
Compare 165 & 170: no swap
Compare 170 & 180: no swap
No swaps needed - we're done!
```

We will walkthrough the C code and look at the pseudocode on the next page

1.5.2 Code Walkthrough

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 5 Bubble Sort

```
1: procedure BubbleSort(array)
2:     for i = 0 to length(array) - 1 do
3:         for j = 0 to length(array) - i - 2 do
4:             if array[j] ¿ array[j + 1] then
5:                 Swap array[j] and array[j + 1]
6:             end if
7:         end for
8:     end for
9: 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

Merge sort uses a divide-and-conquer strategy. Divide the list, sort each half, then merge.

- Divide the list into halves
- Recursively sort each half
- Merge the sorted halves

1.6.1 Example: Sorting Your Spotify Playlist by Release Year

```
Let's sort these songs by release year:

Initial: [1995, 2010, 1980, 2020, 1990, 2005, 2015, 1985]

Step 1: Divide into halves
Left:  [1995, 2010, 1980, 2020]
Right: [1990, 2005, 2015, 1985]

Step 2: Keep dividing
Left side:
  [1995, 2010] and [1980, 2020]
  [1995] [2010] and [1980] [2020]

Right side:
  [1990, 2005] and [2015, 1985]
  [1990] [2005] and [2015] [1985]

Step 3: Merge back together (sorting as we go)
Left side:
  [1995] + [2010] → [1995, 2010]
  [1980] + [2020] → [1980, 2020]
  [1995, 2010] + [1980, 2020] → [1980, 1995, 2010, 2020]

Right side:
  [1990] + [2005] → [1990, 2005]
  [2015] + [1985] → [1985, 2015]
  [1990, 2005] + [1985, 2015] → [1985, 1990, 2005, 2015]

Step 4: Final merge
[1980, 1995, 2010, 2020] + [1985, 1990, 2005, 2015]
→ [1980, 1985, 1990, 1995, 2005, 2010, 2015, 2020]
```

1.6.2 The Merging Process Explained

1. When merging two sorted lists, we compare the first elements of each:
2. Take the smaller one.
3. Move to the next element in that list.
4. Repeat until one list is empty.
5. Add remaining elements from the other list.

1.6.3 Code Walkthrough

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).

We then print Sorted Array: Calls printArray to display the array after sorting.

Algorithm 6 Merge Sort

```
1: procedure MergeSort(array)
2:     if length(array) > 1 then
3:         mid ← floor(length(array) / 2)
4:         left ← array[0 ... mid]
5:         right ← array[mid + 1 ... length(array) - 1]
6:         MergeSort(left)
7:         MergeSort(right)
8:         Merge(array, left, right)
9:     end if
10: end procedure
11: procedure Merge(array, left, right)
12:     i ← 0, j ← 0, k ← 0
13:     while i ¡ length(left) and j ¡ length(right) do
14:         if left[i] ¡ right[j] then
15:             array[k] ← left[i]
16:             i ← i + 1
17:         else
18:             array[k] ← right[j]
19:             j ← j + 1
20:         end if
21:         k ← k + 1
22:     end while
23:     while i ¡ length(left) do
24:         array[k] ← left[i]
25:         i ← i + 1
26:         k ← k + 1
27:     end while
28:     while j ¡ length(right) do
29:         array[k] ← right[j]
30:         j ← j + 1
31:         k ← k + 1
32:     end while
33: end procedure
```

C code on the next page:

```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;
}
```

1.7 OPTIONAL – Insertion Sort

Insertion Sort is a simple algorithm that builds the sorted array one element at a time, just like how you might sort a hand of playing cards.

- Start with the second element.
- Compare it with elements before it.
- Shift any larger elements to the right.
- Insert the current element into the correct spot.
- Repeat for the rest of the array.

$i = 1$ | 7 | 3 | 10 | 5 | 8 |

$i = 1$ | 7 | 3 | 10 | 5 | 8 |

$j = 1,\ key = 3$ | 7 | | 10 | 5 | 8 | $3 < 7$? Yes

$j = 0,\ key = 3$ | | 7 | 10 | 5 | 8 | $j > 0$? No

$i = 2$ | 3 | 7 | 10 | 5 | 8 |

$j = 2,\ key = 10$ | 3 | 7 | | 5 | 8 | $10 < 7$? No

$i = 3$ | 3 | 7 | 10 | 5 | 8 |

$j = 3,\ key = 5$ | 3 | 7 | 10 | | 8 |

$j = 2, 1,\ key = 5$ | 3 | | 7 | 10 | 8 |

$i = 4$ | 3 | 5 | 7 | 10 | 8 |

$j = 4,\ key = 8$ | 3 | 5 | 7 | 10 | |

$j = 3,\ key = 8$ | 3 | 5 | 7 | | 10 |

Here is our final, sorted array, which used **Insertion Sort**

| 3 | 5 | 7 | 8 | 10 |

1.7.1 Pseudocode

Algorithm 7 Insertion Sort

```
procedure InsertionSort(array)
    for i = 1 to length(array) − 1 do
        key ← array[i]
        j ← i − 1
        while j ≥ 0 and array[j] > key do
            array[j + 1] ← array[j]
            j ← j − 1
        end while
        array[j + 1] ← key
    end for
end procedure
```

1.7.2 Performance, OPTIONAL

- Best Case: $O(n)$ (already sorted)
- Worst Case: $O(n^2)$ (reverse sorted)
- Space: $O(1)$ (in-place)

I have the code implementation below

```c
#include <stdio.h>

void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        // Shift elements
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

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

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

    insertionSort(arr, n);
    printf("Sorted array (Insertion Sort):\n");
    printArray(arr, n);
    return 0;
}
```

1.8 OPTIONAL – Quick Sort

Quick Sort is a divide-and-conquer algorithm that works by selecting a pivot element and partitioning the array around the pivot.

- Choose a pivot.
- Move elements smaller than pivot to the left, larger to the right.
- Recursively sort the left and right subarrays.

Algorithm 8 QuickSort

```
1: procedure QuickSort(array, low, high)
2:     if low < high then
3:         pi ← Partition(array, low, high)
4:         QuickSort(array, low, pi - 1)
5:         QuickSort(array, pi + 1, high)
6:     end if
7: end procedure
```

Algorithm 9 Partition

```
1: procedure Partition(array, low, high)
2:     pivot ← array[high]
3:     i ← low − 1
4:     for j = low to high − 1 do
5:         if array[j] < pivot then
6:             i ← i + 1
7:             swap array[i] ↔ array[j]
8:         end if
9:     end for
10:    swap array[i + 1] ↔ array[high]
11:    return i + 1
12: end procedure
```

- Best/Average: $O(n \log n)$
- Worst: $O(n^2)$ (poor pivot choices)
- Space: $O(\log n)$ (recursive calls)

```c
#include <stdio.h>

void swap(int* a, int* b) {
    int t = *a; *a = *b; *b = t;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // pivot
    int i = low - 1;        // index of smaller element

    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);
    }
}

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

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    quickSort(arr, 0, n - 1);
    printf("Sorted array (Quick Sort):\n");
    printArray(arr, n);
    return 0;
}
```

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

2.1 Multiple Choice 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.

2.2 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?

2.3 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 ~~1~~0 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`.

2.4 Spot The Error Questions

1. Spot the error in the linear search implementation below:

```
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 the 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 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];
    }
}
```

4. Spot the error in this bubble sort logic:

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

5. Spot the error in the merge step of merge sort:

```
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l;
    int n2 = r - m;

    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = j = 0;
    k = l;
    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++];
}
```

6. Spot the error in this recursive merge sort structure:

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

7. Spot the error in this binary search base case:

```
int binarySearch(int arr[], int l, int r, int key) {
    if (l >= r)
        return -1;
    int mid = (l + r) / 2;
    if (arr[mid] == key)
        return mid;
    if (arr[mid] > key)
        return binarySearch(arr, l, mid - 1, key);
    return binarySearch(arr, mid + 1, r, key);
}
```

8. Spot the off-by-one error in this linear search:

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

9. Spot the mistake in this selection sort swap:

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

10. Spot the incorrect loop range in this bubble sort:

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