4- Sort Colors

NAME	ID
محمد فوقى عبد العاطي	20201872
مصطفى هشام السيد	20201883
احمد ناجح عبد الرازق	20201813
عزالدين عصام سيد	20201843
احمد عماد ابراهيم	20201807

Introduction

Given an array of integers with n objects colored red, white, or blue, sort them in-place so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively.

So, we implemented this problem using five sorting algorithms.

- Insertion Sort
- Merge Sort
- Selection Sort
- Bubble Sort
- Quick Sort

At first, the user runs the application (main.exe), and then he is asked to enter the size of his array which ranges from 1 to 300 integers.

After that he is asked to fill the array according to the size he entered before (integers 0, 1, 2 only accepted, separated by space or next line).

Then he is asked to choose which algorithm he wants to sort this array (each algorithm is assigned to a number from 1 to 5 respectively).

Then there’s the output with the sorted array visible to the user.

Insertion Sort

Pseudocode:

```
a[n] // is an array of numbers with size n
ALGORITHM InsertionSort(array, n):
    for i ← 1 to n do // complexity O(n)
    key ← a[i]
    index ← i
    j ← i - 1
      while(j >= 0 & key < a[j]) //complexity O(n)
         a[j + 1] = a[j]
         index = j
    a[index] = key
```

Analysis:

Insertion sort is a non-recursive algorithm so its complexity will be analyzed in a sequential way by simulating a division in the array into two parts, first is sorted and the second is not sorted.

Insertion sort algorithm takes the first element in the right loop which will be the second element in the array which is called key element and the first element in another loop which is the inner and in case of the key is greater than the last element in the sorted array it will be shifted till reach the suitable index according to its value.

This algorithm is stable as it keeps the equivalent items in their origin sort.

Best Case:

In best case which is the array is already sorted the function execute the outer loop till the end of the array.

So the complexity of the outer loop will be O(n).

Also it has inner loop but it will be executed as the key is forever larger than the element before it so this loop has complexity O(1).

Worst Case:

In worst case array is not sorted so each loop will be executed.

First loop will go through each element so its complexity is O(n).

Inner loop will although loop from the element before the key till first element in worst case so its complexity O(n).

As both are nested loops so their complexity will be multiplied.

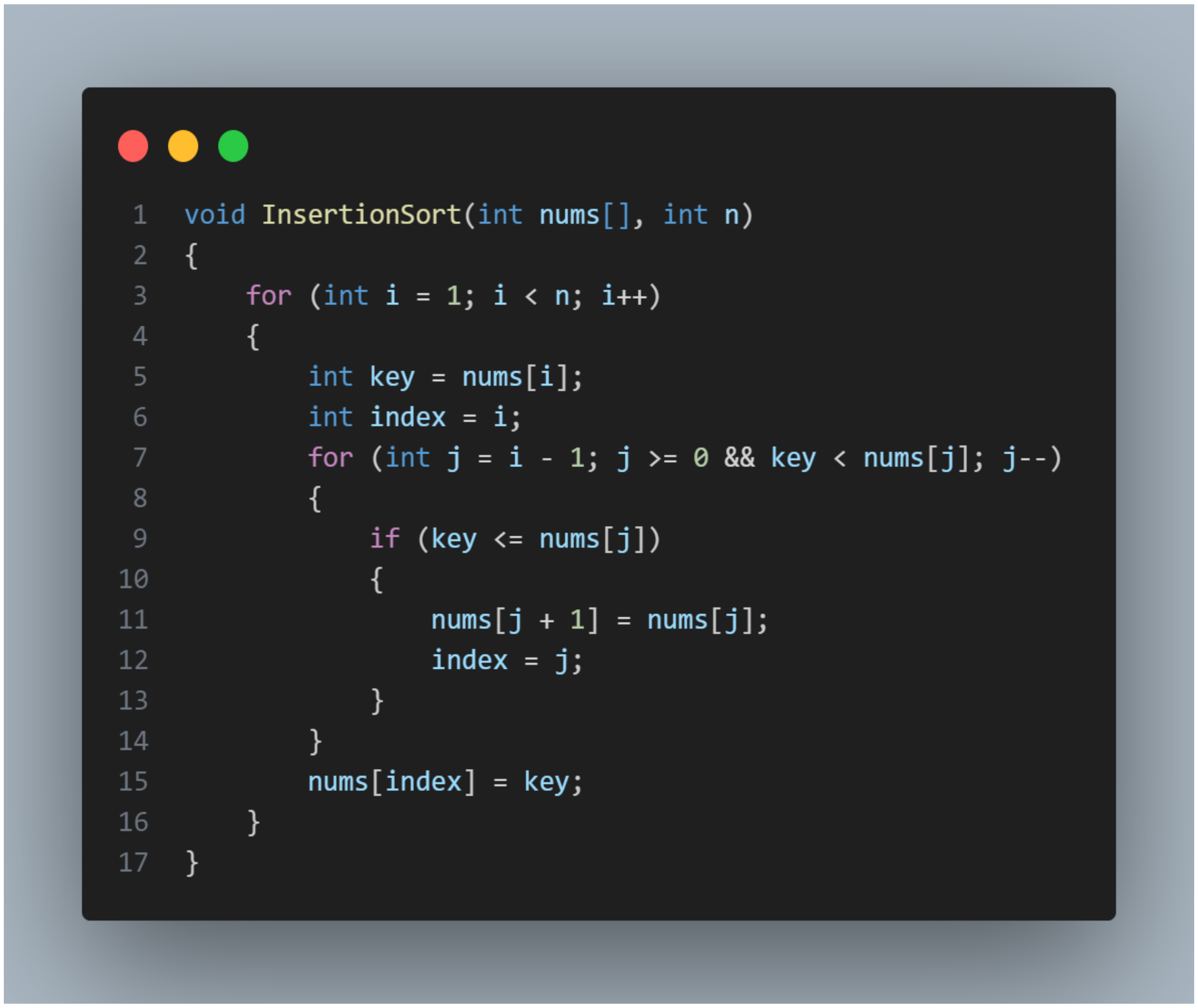
Complexity:

Time Complexity:

Best case complexity: $\omega(n) \leftarrow \omega(n * 1)$

Worst case complexity: $O(n^2) \leftarrow O(n * n)$

Space Complexity: O(1)



```
void InsertionSort(int nums[], int n)
{
    for (int i = 1; i < n; i++)
    {
        int key = nums[i];
        int index = i;
        for (int j = i - 1; j >= 0 && key < nums[j]; j--)
        {
            if (key <= nums[j])
            {
                nums[j + 1] = nums[j];
                index = j;
            }
        }
        nums[index] = key;
    }
}
```

Merge Sort

Pseudocode:

```
ALGORITHM MergeSort(array, start, end):
    if (start < end) then:
        middle ← (start + end) / 2
        MergeSort(array, start, middle)
        MergeSort(array, middle + 1, end)
        Merge(array, start, middle, end)

ALGORITHM Merge(array, start, middle, end)
leftArraySize ← (middle - start) + 1
rightArraySize ← end – middle

//Declare right and l Array [n]
L[leftArraySize + 1]
R[rightArraySize + 1]

for i ← 0 to rightArraySize -1 do
    R[i] = array[middle + i + 1]
R[rightArraySize] = ∞

for i ← 0 to leftArraySize -1 do
```

```
    L[i] = array[start + i]
L[leftArraySize] = ∞
riTracer ← 0
liTracer ← 0
while (start <= end) do:
    if (R[riTracer] <= L[liTracer]) then:
        array[start++] = R[riTracer++]
    else then:
        array[start++] = L[liTracer++]
        array[start++] = L[riTracer++]
```

Analysis:

This algorithm is a recursive algorithm follows divide and conquer approach which is depends on dividing the array into smaller arrays and work on each small array as a unit and solve it then combine them again.

As in the pseudo code there are two recursive calls and the parameters are the halves of the incoming array in the parameter then a function merge as the name implies.

This algorithm is stable as it keeps the equivalent items in their origin sort.

Computing the complexity of merge function first.

It has two loops each will fill the two arrays, both are sequential so the sum of them will be the result O(n) ← O(½ n)

Then a loop for merging both arrays in the origin array will also be O(n)

So the result : O(n) ← ½ n + ½n + n

Then two recursive calls each has a half of the origin array.

As the result for this will be

$T(n) = 2T(\frac{1}{2}n) + n$

$T(\frac{1}{2}n) = 2T(\frac{1}{4}n) + n$

$T(\frac{1}{4}n) = 2T(1/8n) + n$

$T(1/8n) = 2T(1/16n) + n$

…

$T(n) = 2T(1/16n) + 4n$

By tracing this we will discover a pattern in this form

$T(n) = 2T(\frac{1}{2}^k n) + kn$

When length of array = 1 so the $\frac{1}{2}^k(n) = 1$

$2^k = n$

To get k value take log for base 2.

$\log_2 n = k$

So

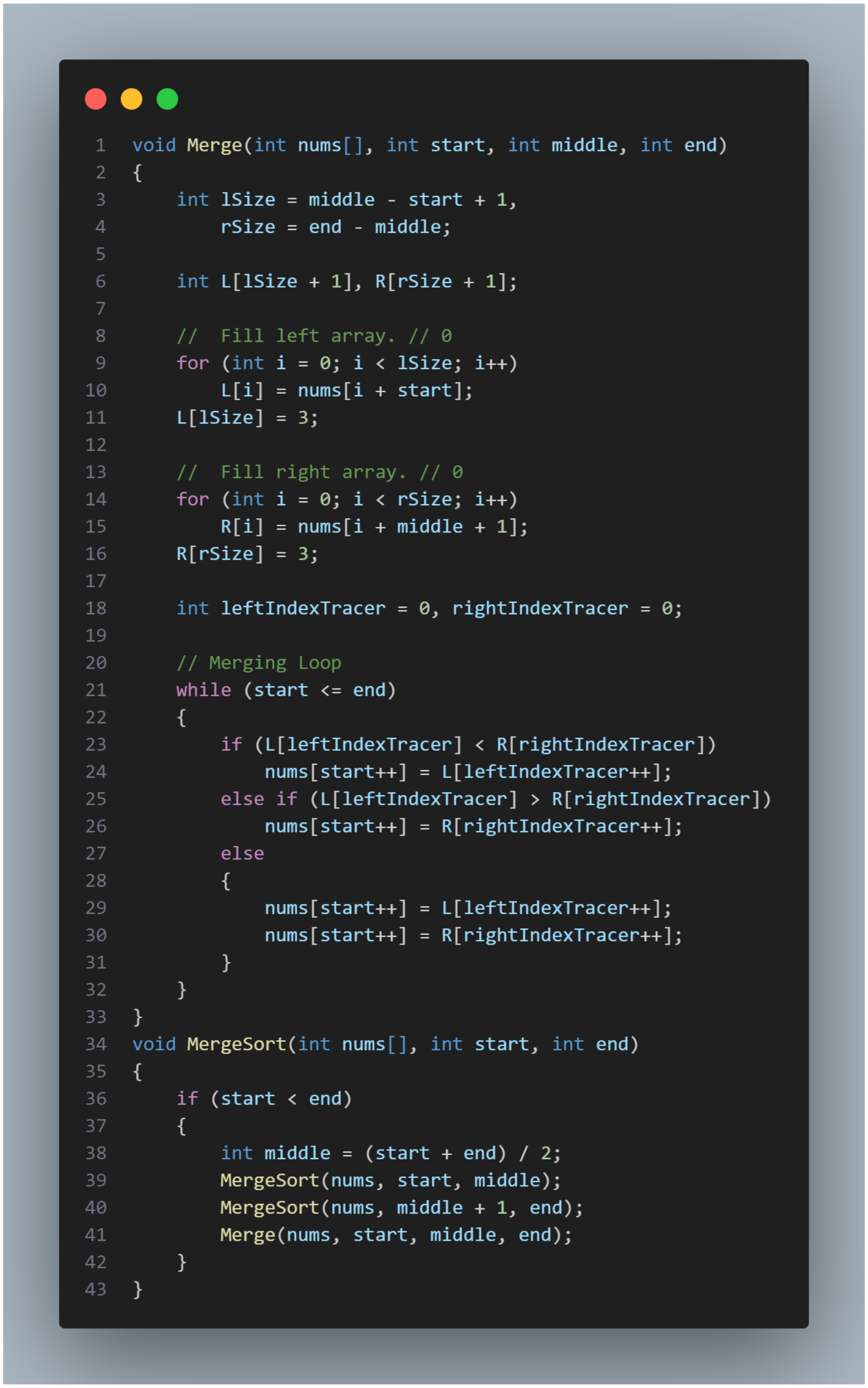
$T(n) = 2T(1) + \log_2 n * n$

$T(n) = n\log_2 n$

Complexity:

Time Complexity: $\Theta(n\log_2 n)$

Space Complexity: $O(n)$



```
void Merge(int nums[], int start, int middle, int end)
{
    int lSize = middle - start + 1,
        rSize = end - middle;

    int L[lSize + 1], R[rSize + 1];

    //  Fill left array. // 0
    for (int i = 0; i < lSize; i++)
        L[i] = nums[i + start];
    L[lSize] = 3;

    //  Fill right array. // 0
    for (int i = 0; i < rSize; i++)
        R[i] = nums[i + middle + 1];
    R[rSize] = 3;

    int leftIndexTracer = 0, rightIndexTracer = 0;

    // Merging Loop
    while (start <= end)
    {
        if (L[leftIndexTracer] < R[rightIndexTracer])
            nums[start++] = L[leftIndexTracer++];
        else if (L[leftIndexTracer] > R[rightIndexTracer])
            nums[start++] = R[rightIndexTracer++];
        else
        {
            nums[start++] = L[leftIndexTracer++];
            nums[start++] = R[rightIndexTracer++];
        }
    }
}
void MergeSort(int nums[], int start, int end)
{
    if (start < end)
    {
        int middle = (start + end) / 2;
        MergeSort(nums, start, middle);
        MergeSort(nums, middle + 1, end);
        Merge(nums, start, middle, end);
    }
}
```

Selection Sort

Pseudocode:

```
SelectionSort(nums[], n)
    for i = 0 to n - 2
        min_position ← i
        for j = i + 1 to n - 1
            if nums[j] < nums[min_position]
                min_position ← j
        if min_position ! ← i
            swap(nums[min_position], nums[i])
```

Analysis:

Selection sort is a simple sorting algorithm that works by repeatedly finding the minimum element from the unsorted part of the list and swapping it with the first element of the unsorted part.

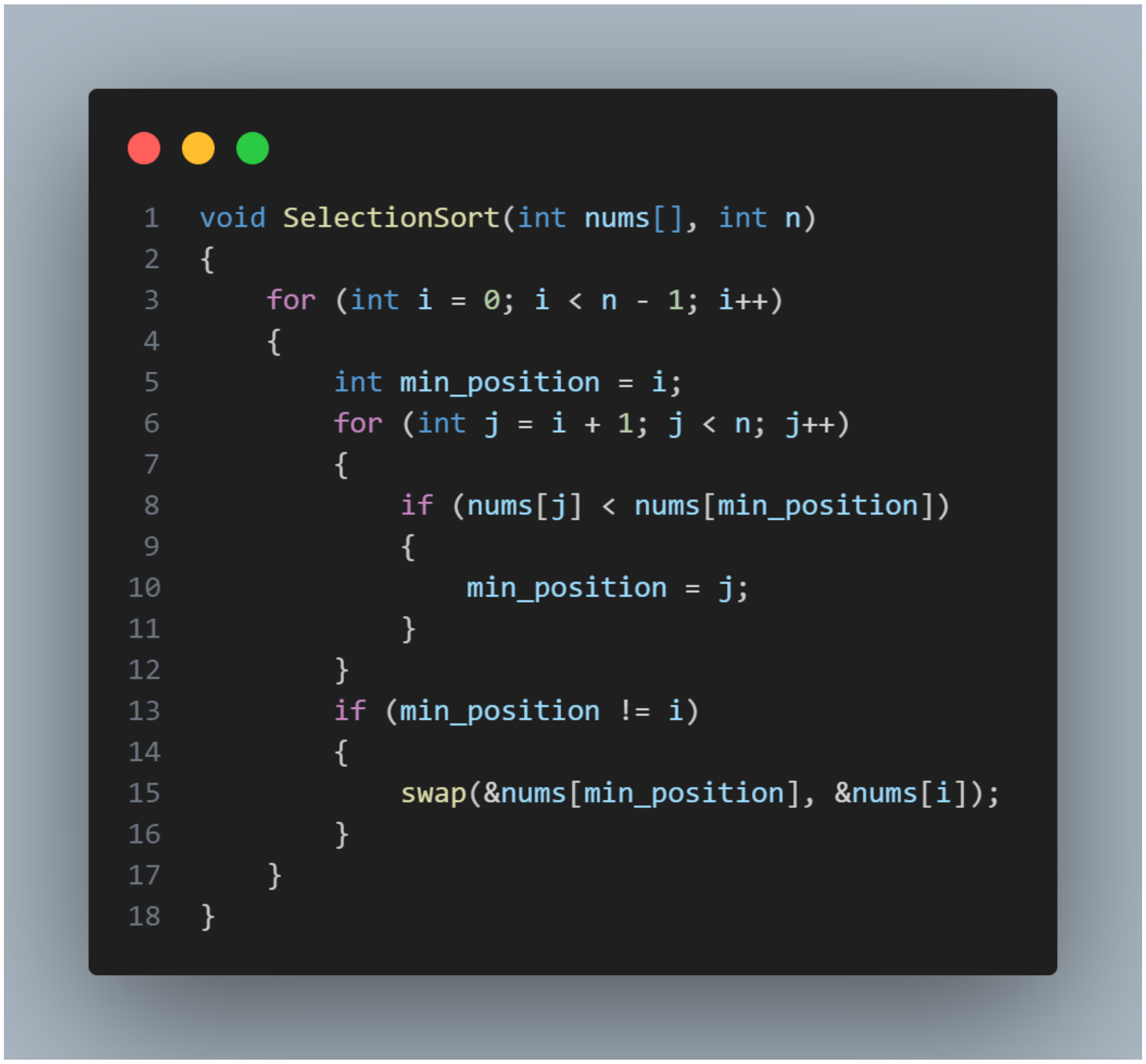
It uses nested loops to iterate over the array so, it's not a recursive algorithm.

It's not a stable sorting algorithm, as it may swap non-adjacent elements with the same value, thus changing their relative order.

Complexity:

Time Complexity: The outer loop runs n-1 times, and the inner loop runs (n-1) + (n-2) + ... + 1 = n(n-1)/2 times in the worst case. Therefore, the time complexity of the Selection Sort algorithm is **O(n^2)**.

Space Complexity: The algorithm performs swapping in place, using only a constant amount of extra space. Therefore, the space complexity is **O(1)**.



```
void SelectionSort(int nums[], int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        int min_position = i;
        for (int j = i + 1; j < n; j++)
        {
            if (nums[j] < nums[min_position])
            {
                min_position = j;
            }
        }
        if (min_position != i)
        {
            swap(&nums[min_position], &nums[i]);
        }
    }
}
```

Bubble Sort

Pseudocode:

```
procedure BubbleSort( list : array of items ,count of List)
  for i = 0 to count do:
    swapped = false
    for j = 0 to count-i-1 do:
      if list[j] > list[j+1] then
        swap( list[j], list[j+1] )
        swapped = true
      end if
    end for
    if(not swapped) then
      break
    end if
  end for
end procedure return list
```

Analysis:

Assume you're attempting to arrange the elements in array of 5 elements in ascending order.

An array contains five elements. That means you must perform four comparisons for the most significant (greatest) element to bubble to the top of the array.

Why do you have four comparisons?

N = The number of elements in an array

N-1 = The number of time comparisons that occur

Therefore: 5 - 1 = 4

First Pass

- Compare the first and second elements, starting with the first index.
- They are swapped if the first element is greater than the second.
- Compare the second and third elements now. If they are not in the correct order, swap them.
- The preceding procedure is repeated until it reaches the final element.

And so on until the array is sorted.

Recursion: Non Recursive

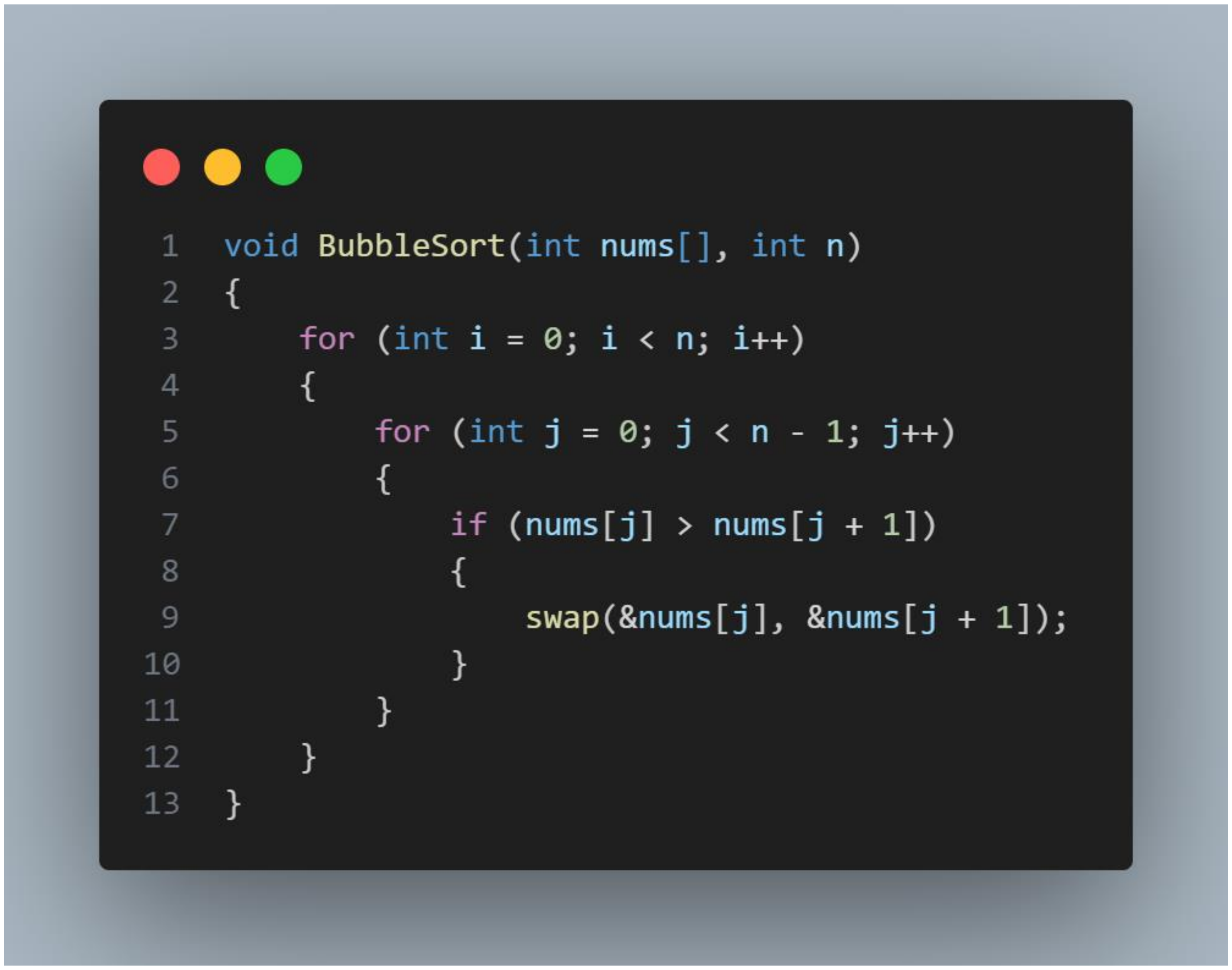
Stability: Stable

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Complexity:

Time Complexity: Best Case → O(N)

Space Complexity: O(1)



```
void BubbleSort(int nums[], int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n - 1; j++)
        {
            if (nums[j] > nums[j + 1])
            {
                swap(&nums[j], &nums[j + 1]);
            }
        }
    }
}
```

Quick Sort

Pseudocode:

Input: the array of numbers and the beginning, end indexes of the sub-list

```
Function Quicksort (int nums[], int low, int high)
if low < high
   {
      Set pivot TO low;
      Set L TO low;
      Set R TO high;
      while L < R
         Start Look in the array nums from the left side and check if nums[L] <= nums[pivot] and L < high
         Then Move the pointer to the right-side L++

         Start Look in the array nums from the Right side and check if nums[R] > nums[pivot]
         Then Move the pointer to the left-side R--
         If L < R
         Then swap the two pointers L, R values

      swap the value from the right pointer to pivot
```

```
        Call the function Quicksort (nums, low, R - 1);
        Call the function Quicksort (nums, R + 1, high);
    }
```

Analysis:

It is a divide and conquer algorithm, sort an array of elements based on some steps: -

1- Pick an element from an array, call it as pivot element.

2- Divide an unsorted array element into two arrays.

3- If the value (the value of the array element) less than pivot element come under first sub array, the remaining elements with value greater than pivot come in second sub array.

So, if I have an array <u>A [....]</u> with length <u>N</u>

- First Get pivot **P** will be the first element in the array

-We have two pointers in the left side of the array **L** and the right side of the array **R**

-①Check if the right pointer **R** is not the pivot in a loop

-②If then check if the pivot **P** is greater than the right **R**, **P** > **R** then swap the elements in **P** and **R** and break the loop, else Decrement **R**

-③Check if the left pointer **L** is not the pivot in a loop

-④if then check if the pivot **P** is less than the left side **L, P** < **L** then swap the elements in **P** and **L** and break the loop, else Increment **L**

-now the pivot is fixed position, and all the left elements are less than the pivot and the right elements are greater the pivot.

-now Divide the array **A** into two sub arrays left part and right part.

-take left, right parts and apply all ①②③④ on it again until the left pointer is greater than or equal to the right pointer.

Stability: Unstable.

Recursion: Recursive

Complexity:

Time Complexity:

O (n log n)

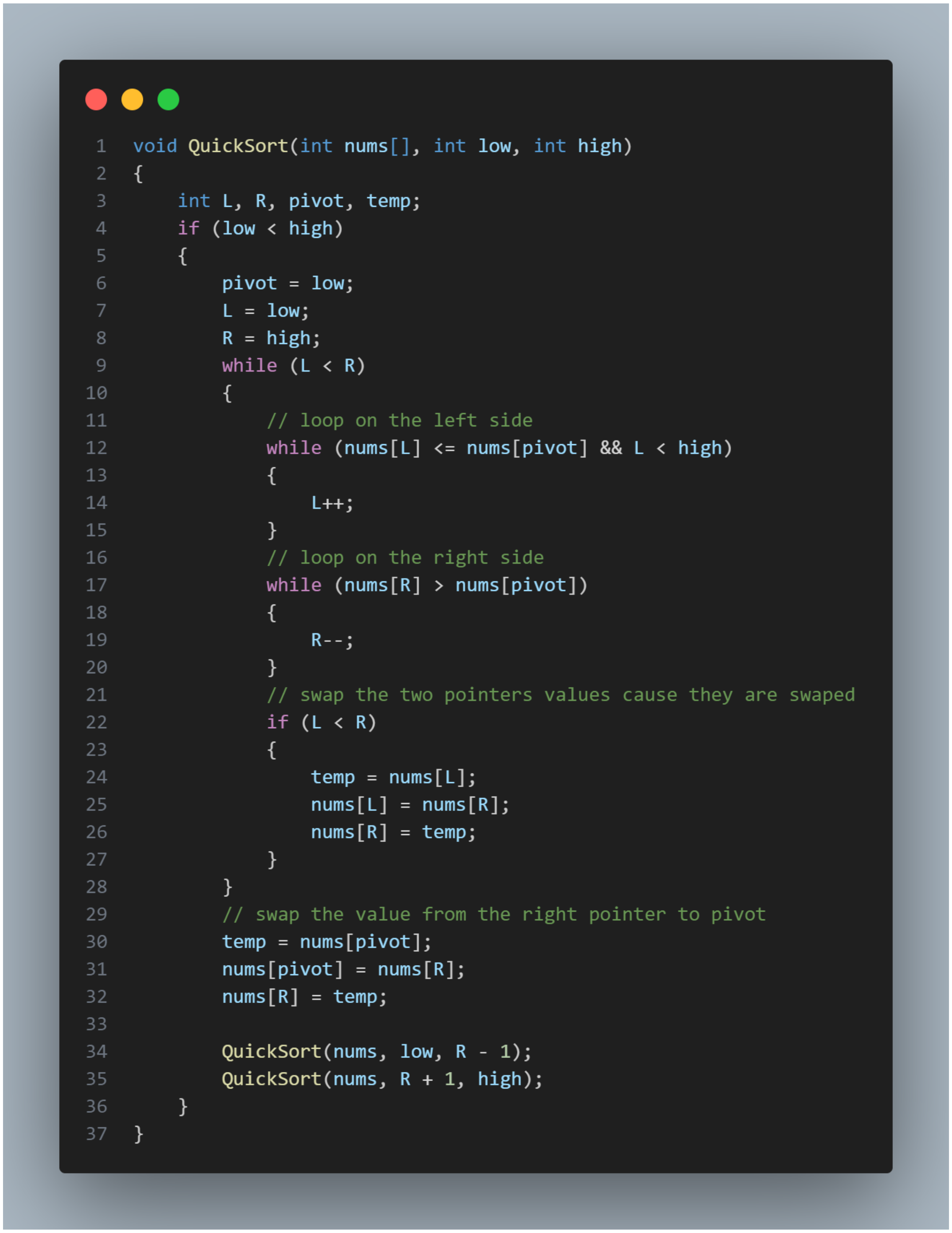
Ω (n log n)

ϴ (n log n) -> T(n) = 2T(n/2) + n log n not quite sure though

Space Complexity:

O (log n)

Recursion: Recursive



```c
void QuickSort(int nums[], int low, int high)
{
    int L, R, pivot, temp;
    if (low < high)
    {
        pivot = low;
        L = low;
        R = high;
        while (L < R)
        {
            // loop on the left side
            while (nums[L] <= nums[pivot] && L < high)
            {
                L++;
            }
            // loop on the right side
            while (nums[R] > nums[pivot])
            {
                R--;
            }
            // swap the two pointers values cause they are swaped
            if (L < R)
            {
                temp = nums[L];
                nums[L] = nums[R];
                nums[R] = temp;
            }
        }
        // swap the value from the right pointer to pivot
        temp = nums[pivot];
        nums[pivot] = nums[R];
        nums[R] = temp;

        QuickSort(nums, low, R - 1);
        QuickSort(nums, R + 1, high);
    }
}
```

Comparison

Algorithm	*Space Complexity*	*Time Complexity*		
		Best Case	**Average Case**	**Worst Case**
Insertion Sort	O(1)	O(n)	O(n^2)	O(n^2)
Merge Sort	O(n)	$\Theta(n\log_2 n)$	$\Theta(n\log_2 n)$	$\Theta(n\log_2 n)$
Selection Sort	O(1)	O(n^2)	O(n^2)	O(n^2)
Bubble Sort	O(1)	O(n)	O(n^2)	O(n^2)
Quick Sort	O (log n)	$\Theta(n\log_2 n)$	$\Theta(n\log_2 n)$	$\Theta(n\log_2 n)$

Conclusion:

No specific algorithm is the best in all cases but varies according to some requirements (**Time & Memory**).

For time priority, it's recommended to use recursive algorithms like **Merge** or **Quick** sorting algorithms.

And for less memory consumption, it's recommended to use non-recursive algorithms like **Selection**, **Bubble,** and **Insertion** Sorting algorithms.