Microprocessor Systems

BLG 212E

Homework 2 Report

Nurefşan Altın
150210053
altinn21@itu.edu.tr

1. C++ Code for Bubble Sort Algorithm Implementation

I have provided the c++ code that I have used in my implementation.

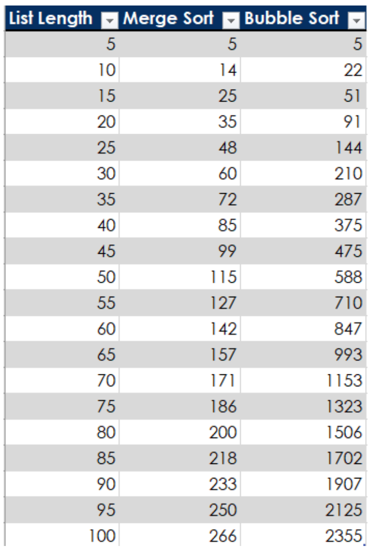
```
class Node
{
public:
    int data;
    Node *next;
    Node(int data)
    {
        this->data = data;
        this->next = NULL;
    }
};

int getLength(Node *head)
{
    Node *current = head;
    int length = 0;
    if (!current)
    {
        return length;
    }
    length++;
    while (current->next)
    {
        length++;
        current = current->next;
    }
    return length;
}
```

```c
void ft_lstsort_asm(Node **head)
{
    int length = getLength(*head);
    int index = 0;
    while (index < length)
    {
        Node *current = *head;
        Node *previous = *head;
        while (current->next)
        {
            Node *next_node = current->next;
            if (current->data > next_node->data)
            {
                current->next = next_node->next;
                next_node->next = current;
                if (current == *head)
                {
                    previous = next_node;
                    *head = next_node;
                }
                else
                {
                    previous->next = next_node;
                    previous = next_node;
                }
                continue;
            }
            previous = current;
            current = current->next;
        }
        index++;
    }
}
```

2. Execution Time Comparison of Bubble Sort and Merge Sort

2.1. Graph and Table for Results



List Length	Merge Sort	Bubble Sort
5	5	5
10	14	22
15	25	51
20	35	91
25	48	144
30	60	210
35	72	287
40	85	375
45	99	475
50	115	588
55	127	710
60	142	847
65	157	993
70	171	1153
75	186	1323
80	200	1506
85	218	1702
90	233	1907
95	250	2125
100	266	2355

Figure 1: Table of calculated ticks values of Bubble Sort and Merge Sort for different data sizes

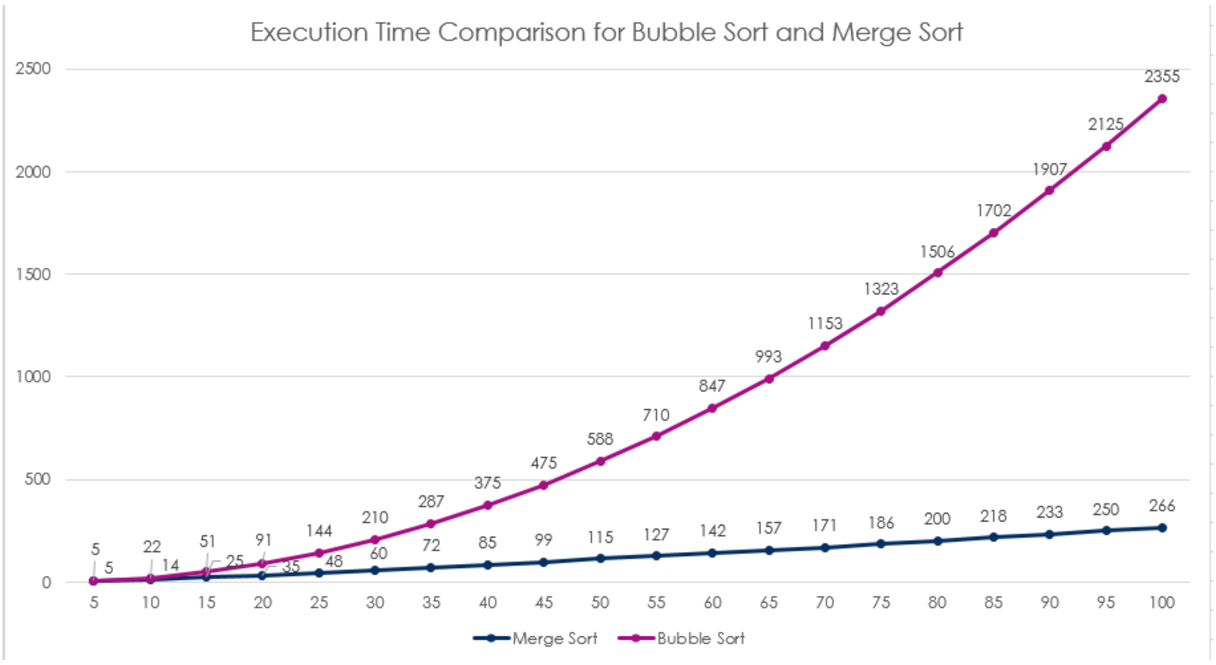


Figure 2: Graph of calculated ticks values of Bubble Sort and Merge Sort for different data sizes

2.2. Analysis of Execution Times of Bubble Sort and Merge Sort

I have implemented the SysTick_Start_asm, SysTick_Handler and SysTick_Stop_asm functions and measured the execution times for both algorithms using these functions. The results are provided in the table and graph.

Bubble sort is a comparison based algorithm that is used to sort a list of items. This algorithm works by comparing adjacent nodes' data and swapping accordingly. In this implementation, the process is more complex because we update the list by swapping the nodes instead of just swapping the data inside the nodes. On the other hand, merge sort is an algorithm that uses divide and conquer approach to sort given items. It can be implemented recursively by splitting the array in two parts as left and right until there is no item to divide. After sorting each portion separately these two are merged and the process continues until all recursive calls are ended.

When we examine the results obtained from datasets of different sizes, it is clear that as the size of the dataset increases, the execution time of the bubble sort algorithm increases quadratically. For example, the list of 20 nodes sorted in $910\,\mu$s while 40 nodes sorted in $3750\,\mu$s, which means if the dataset size doubles, the execution time approximately quadruples. This behavior indicates that the efficiency of the algorithm drops significantly for large datasets and causes considerable time loss. On the other hand, the increase in merge sort algorithm's execution time is considerably low when it is compared to bubble sort and is directly proportional to the increase in dataset size. For example, the list of 30 nodes sorted in $600\,\mu$s while 60 nodes sorted in $1420\,\mu$s, which means if the dataset size doubles, the execution time also doubles. These results indicates that, merge sort preserves its consistency and performs well as the dataset size increases.

Time Complexity Analysis of Bubble Sort

- **Best Case (O(n))**: If data is already sorted, each element is checked once, and the complexity becomes $O(n)$.
- **Average Case (O(n^2))**: If the list is randomly sorted, each pair of elements needs to be compared, so the complexity becomes $O(n^2)$.
- **Worst Case (O(n^2))**: If the list is sorted in descending order, every adjacent pair is swapped, and the complexity becomes $O(n^2)$.

Time Complexity Analysis of Merge Sort

- **Best, Average, and Worst Case (O(n log n))**: Merge Sort always divides the array in two parts and merges them back; therefore, it has a consistent time complexity of O(n log n).

In conclusion, table and the graph shows that, as the dataset size increases bubble sort algorithm operates inefficiently and performs significantly slower than merge sort. Therefore for large datasets, bubble sort is not preferable due to it's poor-performance. However, merge sort is preferable even for large datasets thanks to its consistency and gives efficient outcomes in terms of performance.

3. Formulas Used in Implementation

The reload value used in `SysTick_Start_asm` implementation is calculated as follows:

General formula :

$$\text{Interval} = \text{(ReloadValue + 1)} \times \text{Source\_Clock\_Period}$$

$$\text{Interval} = 10\,\mu s, \quad \text{SystemCoreClock} = 25,000,000$$

$$\text{Interval} = \frac{\text{ReloadValue} + 1}{\text{SystemCoreClock}}$$

$$\text{ReloadValue} + 1 = \text{SystemCoreClock} \times \text{Interval}$$

$$\text{ReloadValue} + 1 = \frac{25,000,000}{100,000}$$

$$\text{ReloadValue} + 1 = 250$$

$$\text{ReloadValue} = 249$$