**VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY**

**UNIVERSITY OF SCIENCE**

**FACULTY OF INFORMATION TECHNOLOGY**



**DATA STRUCTURES & ALGORITHMS**

**SORTING ALGORITHMS PROJECT**

**Group: 10**

**Instructors:**

Mr. Bùi Huy Thông

Mrs. Trần Thị Thảo Nhi

**Group members:**

20127169 – Phạm Huy Hoàng

20127458 – Đặng Tiến Đạt

20127566 – Hoàng Quốc Nam

20127637 – Võ Lê Anh Thông

# **INTRODUCTION**

First of all, Group 10 would like to send our sincere thanks to the University of Science have brought Data Structures & Algorithms to the education program. Specially, we would like to thank deeply our practice lecturers & instructors *– Mr. Bùi Huy Thông* and *Mrs. Trần Thị Thảo Nhi*, who have taught and imparted valuable knowledge to us during school time. During studying times, we have improved ourselves with many useful skills, serious and effective – learning spirit. This will definitely be valuable knowledge, plant the seed for us to go out into the wider world.

After this mid – term project, we, Group 10, have improved a lot about communication skills, time management skills and problem – solving. So precious that we are learnt how to work like a team, listen to our partner, ask for helps and share difficulties with each other. Besides, acquiring knowledge by search on the web is one of the necessary skills needed in the Faculty of Information Technology.

Data Structures & Algorithms is an interesting subject, very useful with high practicality. Guaranteed to provide enough knowledge, associated with the actual demand of students. However, because of our limited knowledge and receptive ability, even we have tried our best but certainly, our project is hard to avoid deficiencies and mistakes, we are hope that our report is considered and feedback to make our project more complete.

Sincerely yours.

**II. INFORMATION**

After many days of studying and absorbing knowledge from the lecturer, group 10 understood the application and convenience of the sort technique in programming.

In this project, we were asked to implement all algorithm (for ascending order only), besides we chose Set 2 (11 algorithms):

* Selection Sort.
* Insertion Sort.
* Bubble Sort.
* Shaker Sort.
* Shell Sort.
* Heap Sort.
* Merge Sort.
* Quick Sort.
* Counting Sort.
* Radix Sort.
* Flash Sort.
* **Data size**
* Data size of input array: n.
* Array: a (begins at a[0] and ends at a[n-1]).
* The element at i position of array a: a[i] (∀i ∈ [0; n-1]).
* Data size of input array: 10000 ≤ n ≤ 500000.
* Data size of each element: 0 ≤ a[i] ≤ n.
* **Charts and tables**
* Line graph showing the running time of each function.
* The time unit in the graphs is *millisecond* (ms).
* Bar chart showing the comparisons of each function.
* **Compiling**
* To compile the program, we use the compile command:

**g++ main.cpp DataGenerator.cpp -o <file\_name>.exe**

**III. ALGORITHM PRESENTATION:**

1. **Selection Sort**
2. **Ideas**
3. **Step-by-step descriptions**
4. **Complexity evaluations**
5. **Variants/improvements**
6. **Insertion Sort**
7. **Ideas**

* Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part (in this case, it will be placed at the position where the element a[i] larger than a[i-1] and smaller than a[i+1]).

1. **Step-by-step descriptions**

* To sort an array of size n in ascending order:
* 1: Iterate from a[1] to a[n] over the array.
* 2: Compare the current element (key) to its predecessor.
* 3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element. And all elements which from its their index to position 0 are larger than the key, swap them with the top.
* Ex: There is an unsorted array:

**a = {12, 11, 13, 5, 6}**

|  |  |  |
| --- | --- | --- |
| Stage | Array | Explanation |
| 0 | 12, 11, 13, 5, 6 | In the beginning, the array doesn’t have no change, start at a[1] = 11. |
| 1 | 11, 12, 13, 5, 6 | Now, we compare and see that a[0] = 12 > a[1], so we will swap them to place at suitable positions (11 < 12) |
| 2 | 11, 12, 13, 5, 6 | Then, a[1] < a[2] (12 < 13), so we continue the loop. |
| 3 | 5, 11, 12, 13, 6 | We can see a[3] < a[2], so we keep compare with a[1], a[0]. All of them is larger than a[3], so we place 5 at a[0]. |
| 4 | 5, 6, 11, 12, 13 | At a[5] = 6, we still compare with a[4],a[3],…  But we can see that, 5 < 6 < 11, so we will place 6 at position between them. |
| 5 | 5, 6, 11, 12, 13 | The algorithm finishes here. We will have a sorted array. |

1. **Complexity evaluations**

* **Time complexity:**
* Average complexity: O(n2) → Randomized data.
* Best Case: O(n) → Sorted Data (+ Nearly Sorted Data).
* Worst Case: O(n2) → Reversed Data.
* **Space Complexity**: O(1)
* Running time in milisecond table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data size | 10000 | 30000 | 50000 | 100000 | 300000 | 500000 |
| Randomized | 48 | 427 | 1221 | 4869 | 46944 | 134285 |
| Nearly Sorted | 0 | 1 | 1 | 1 | 1 | 1 |
| Sorted | 0 | ~ 0 | ~ 1 | 1 | 1 | 1 |
| Reversed | 95 | 866 | 2379 | 9728 | 95357 | 271759 |

1. **Variants / Improvements**

* Shell Sort.
* Binary Insertion Sort.

1. **Counting Sort**
2. **Ideas**

* Counting sort is a sorting technique based on numbers between a specific range. It works by counting the number of elements having distinct key values. Then doing some arithmetic to calculate the position of each object in the output sequence.

1. **Step-by-step descriptions**

* To sort an array of size n in ascending order:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **a** | 1 | 4 | 1 | 2 | 7 | 5 | 2 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** |

* 1: Find out the maximum and minimun elements (let then be *max, min*) from the given array. In the instance, max = 7, min = 1).
* 2: Initialize an array (count) of length max + 1 with all elements 0. This array is used for storing the count of the elements in the array.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **count** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |

* 3: Store the count of each element at their respective index in count array.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **count** | 0 | 2 | 2 | 0 | 1 | 1 | 0 | 1 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |

* 4: Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **count** | 0 | 2 | 4 | 4 | 5 | 6 | 6 | 7 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |

* 5: Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated as shown in figure below.
* 6: After placing each element at its correct position, decrease its count by one.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **a** | **1** | 4 | 1 | 2 | **7** | 5 | 2 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **count** | 0 | **2** | 4 | 4 | 5 | 6 | 6 | **7** |

2 - 1

7 - 1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **output** | 1 | **1** | 2 | 2 | 4 | 5 | **7** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **a** | 1 | 4 | 1 | 2 | 7 | 5 | 2 |

1. **Complexity evaluations**

* There are mainly four main loops.

|  |  |
| --- | --- |
| **Loop** | **Complexity** |
| 1: Generate Count array | n |
| 2: Store cumulative sum of the elements of the count array | Count.size = max – min + 1 = k |
| 3: Find the index of each element of the original array in the count array | n |
| 4: Copy the output array to the origin array | n |

* **Time complexity:** Because there isn’t any compare, so:
* Average complexity: O(n + k)
* Best Case: O(n + k)
* Worst Case: O(n + k)

( k is the range of input )

* **Space Complexity**: O(n + k).
* Running time in milisecond table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data size | 10000 | 30000 | 50000 | 100000 | 300000 | 500000 |
| Randomized | 1 | 1 | 4 | 4 | 11 | 18 |
| Nearly Sorted | 1 | 1 | 3 | 5 | 14 | 21 |
| Sorted | 0.1 | 1 | 2 | 7 | 14 | 23 |
| Reversed | 1 | 3 | 2 | 5 | 14 | 20 |

1. **Flash Sort**
2. **Ideas**

* The idea of the algorithm is to divide the elements into 𝑚 different segments, the element 𝑎[𝑖] will be in the segment: 1 + **(m = 0.43 \* n)**
* After fission is complete, the algorithm will arrange the elements in the same segment by the insertion sort algorithm.

1. **Step-by-step descriptions**

a = {4, 1, 0, 3, 2} ; n = 5 → m = 3

→ a = {1, 0 | 3, 2 | 4}

|  |  |
| --- | --- |
| Element | Segment |
| 4 | 3 |
| 1 | 1 |
| 0 | 1 |
| 2 | 2 |
| 3 | 2 |

* In the next step, we will use insertion sort to sort each segment:

→ a = {1, 0 | 3, 2 | 4}

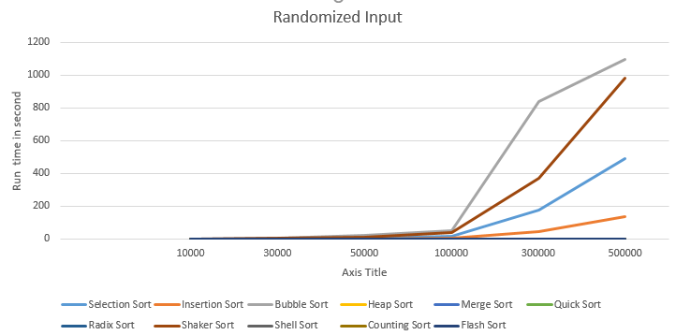
* At this point, the algorithm stops.

1. **Complexity evaluations**

* **Time complexity:**
* Average complexity: O(n + m)
* Best Case: O(n + m)
* Worst Case: O(n2)
* **Space Complexity**: O(m).

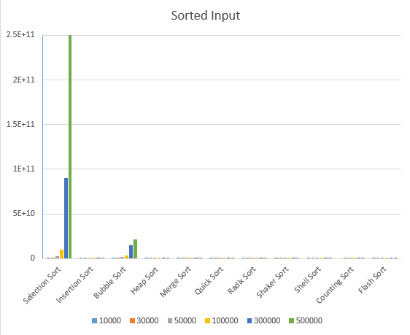
# **IV. Experimental results and comments**

1. Randomized data



2. Nearly sorted data

3. Sorted data



4. Reverse sorted data

**V. References**