2022

UNIVERSITY of GREENWICH

Interview problem and its solution using algorithms and data structures in Python

Algorithms and Data Structures (ADS) - COMP1819

SORTING ALGORITHMS

Table of Contents

Team members:

Member	Name	ID	Lab number
1	Radoslav Gadzhovski	001200583	Suzuki, Atsushi - 02
2	Lirdi Ramku	001163418	Suzuki, Atsushi - 02
3	Jasdeep Kaur	001180518	Suzuki, Atsushi - 02
4	Violeta Marin	001136618	Suzuki, Atsushi - 02

Interview Question

Problem name: Sorting function

Write a Python *function*, sort(sequence), that takes a sequence of floating-point numbers, and returns sorted version of that sequence.

Examples

ID	Input	Output
1	[98.0, 35.2, 1.4, 12.6]	[1.4, 12.6, 35.2, 98.0)
2	[-2.2, 45.1, 0.4, 11.7, -9.0]	[-9.0, -2.2, 0.4, 11.7, 45.1]
3	[4]	[4]
4	[]	[]

Constraints

The size of the sequence is less 10 000 and each value is a floating-point number between

-100.0 and 100.0. The Testing program (main.py) takes input from text files stored in directory “input_data” (correctness.txt, float_random_100.txt, float_random_1000.txt, float_random_2500.txt, float_random_5000.txt, float_random_10000.txt) and should output the runtime in directory “output”.

Answers

Answer 1: Radoslav Gadzhovski

Bubble Sort algorithm

To solve the problem, I used sorting algorithm known as “Bubble Sort”, this is one of the most popular and reliable algorithms out there. The worst-case time complexity of the algorithm is $O(n^2)$ and its space complexity O(1). Bubble Sort is mostly used with small lists of integers where it is most effective and efficient. This algorithm is also known as sinking sort, it runs through the array repeatedly, compares adjacent elements and if they are out of order, it swaps them. (Upadhyay, 2022)

Step by step explanation:

Take a list of integers (6 2 5 3 9), and sort it from lowest number to greatest number. In each step, elements written in bold are being compared. Three passes will be required: (Wikipedia, 2022)

First Pass

(**6 2** 5 3 9) → (**2 6** 5 3 9), Here, algorithm compares the first two elements, and swaps since 6 > 2.

(2 **6 5** 3 9) → (2 **5 6** 3 9), Swap since 6 > 5

(2 5 **6 3** 9) → (2 5 **3 6** 9), Swap since 6 > 3

(2 5 3 **6 9**) → (2 5 3 **6 9**), Now, since these elements are already in order (9 > 6), no swap is needed.

Second Pass

(**2 5** 3 6 9) → (**2 5** 3 6 9), No swap

(2 **5 3** 6 9) → (2 **3 5** 6 9), Swap since 5 > 3

(2 3 **5 6** 9) → (2 3 **5 6** 9), No swap

(2 3 5 **6 9**) → (2 3 5 **6 9**), No swap

Now, the list is sorted, but the algorithm does not know if it is completed. The algorithm needs one additional whole pass without any swap to realise it is sorted.

Third Pass

(**2 3** 5 6 9) → (**2 3** 5 6 9)

(2 **3 5** 6 9) → (2 **3 5** 6 9)

(2 3 **5 6** 9) → (2 3 **5 6** 9)

(2 3 5 **6 9**) → (2 3 5 **6 9**)

Answer 2: Jasdeep Kaur

Insertion Sort Algorithm

Insertion sorting algorithm can be defined as the sorting technique with the help of which a sorted arrangement is erected. The elements of array are compared with each other in a sequence and then assembled concurrently in a particular order. The mechanism is similar to the order of arranging deck of cards. This sort is particularly based on the idea of inserting an element at a specific place and therefore called as Insertion Sort. (Interviewbit, 2022)

Working:

Step 1 - This step includes comparing the element in question with its next element.

Step 2 - If on comparison, it is found that the element can be placed on a specific position, then a place is created for this element by shifting another element one position to the right and then inserting the element at an appropriate position.

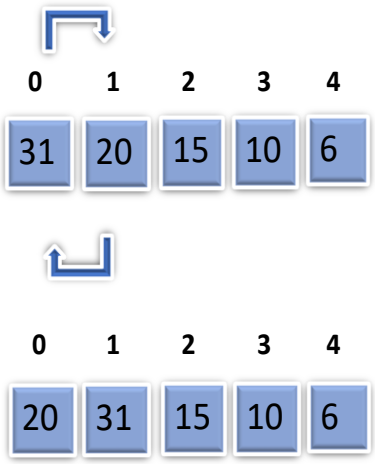
Step 3 - The above-mentioned process is replicated until each element of the array is placed at a suitable position.

Explanation: Let us consider following list of numbers: 31, 20, 15, 10, 6

First Iteration:

Compare 20 with 31, it shows that 20 < 31. Therefore, swap 20 and 31. The array now looks like this: 20, 31, 15, 10, 6

First Iteration



Second Iteration:

Starting with the 2nd element (31), but it was swapped on to the correct position, so we will move ahead to the next element of array.

Now, we will hold on to the 3rd element (15) and compare it with the ones preceding.

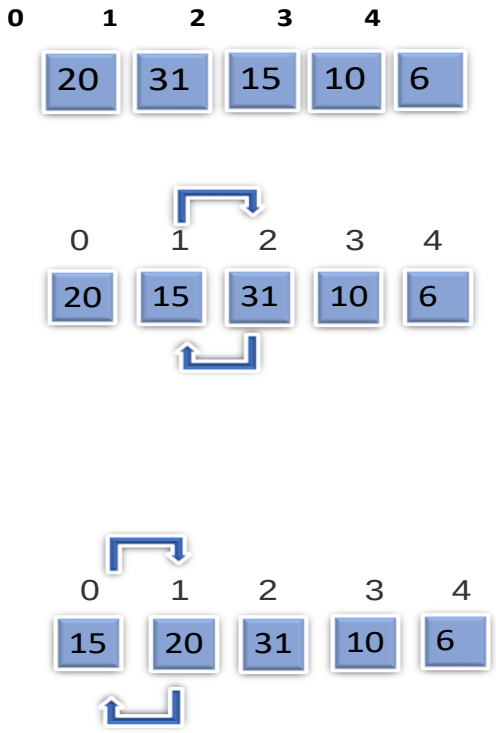
Here, 15 < 31.

So, swap 15 and 31 which means now array will look like:

20, 15, 31, 10, 6.

But we also need to compare it with 20 as well. So, the comparison will take place between 20 and 15 which shows that 15 < 20. Therefore, the array will now look like this: 15, 20, 31, 10, 6

Second Iteration



Third Iteration:

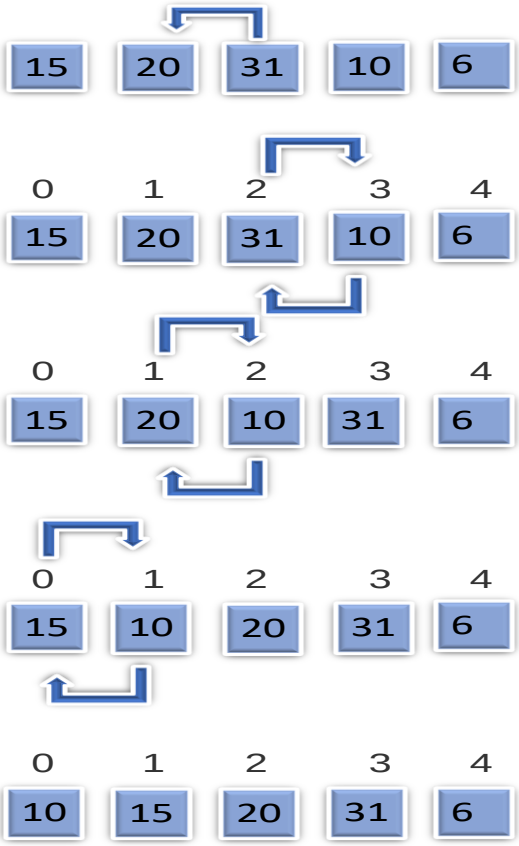
Now we will start the iteration with the 4th element (10) and make the comparison with its preceding elements.

As we can see 10 < 31. Hence, we swap them and this array becomes: 15, 20, 10, 31, 6.

We will now compare 10 with 20. And apparently,10 < 20. We swap two. So, this becomes: 15, 10, 20, 31, 6.

At last, we have to compare 10 with 15 where also 10 < 15. So, we swap both. Therefore, the array will look like: 10, 15, 20, 31, 6.

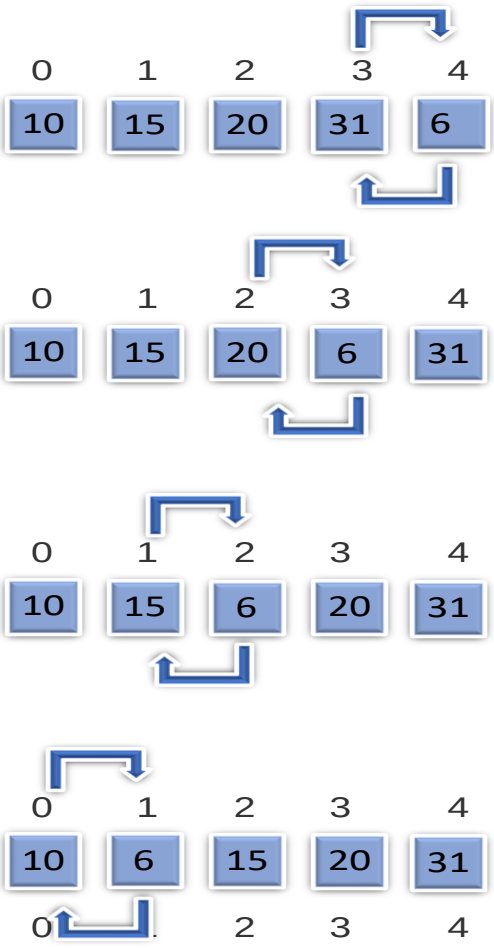
Third Iteration



Fourth Iteration:
The final iteration involves the contrast among last element (6) and preceding elements of this array and get a suitable array with appropriate swapping among elements.
Here, 6 < 31. Thus, swap 6 and 31
Array: 10, 15, 20, 6, 31.
Now, compare 6 with 20 and we know 6 < 20. Swap 6 and 20.
Array: 10, 15, 6, 20, 31.
Compare 6 and 15. Since, 6 < 15. Swap 6 and 15.
Array: 10, 6, 15, 20, 31.
The final comparison for the iteration is between 10 and 6.
Evidently, 6 < 10. Hence, swap 6 and 10.
And finally, array becomes:
6, 10, 15, 20, 31.

Therefore, we get this as a final array of elements after all the corresponding iterations and swapping between elements.

Fourth Iteration



Answer 3: Violeta Marin

Merge Sort

Merge sort is a sorting algorithm based on the Divide and Conquer strategy. It is recursively dividing the array into two equal halves, then sort and combine them.

The Divide and Conquer techniques are completed in three steps: (TechVidvan, 2021)

1**. Divide**: In this step, the array/list divides itself recursively into sub-arrays until the base case is reached.

2**. Recursively solve**: Here, the sub-arrays are sorted using recursion.

3. **Combine**: This step makes use of the merge( ) function to combine the sub-arrays into the final sorted array.

The algorithm for Merge Sort has 5 steps as follows:

In step 1 we need to find the middle index of the array. Middle = 1 + (last – first)/2

For step 2 we divide the array from the middle.

On step 3 we call merge sort for the first half of the array using MergeSort(array, first, middle)

Step 4 we are calling merge sort for the second half of the array. MergeSort(array, middle+1, last)

And on the final step we merge the two sorted halves into a single sorted array.

Applications of Merge Sort

Merge Sort is useful for sorting linked lists in O(nLogn) time, inversion count problem and also is used in External Sorting.

Drawbacks of Merge Sort

Slower comparative to the other sort algorithms for smaller tasks.

Merge sort algorithm requires an additional memory space of 0(n) for the temporary array.

It goes through the whole process even if the array is sorted.

Example of Merge Sort algorithm

Consider an array having the following elements: 2,6,8,5,7,1,3,9. We need to sort this array using merge sort. This array has 8 elements, so the mid is 4, therefore we divide the array into 2 arrays of size 4: 2,6, 8, 5 and 7, 1, 3, 9.

Next we divide this 2 arrays in halves and we have 4 arrays of size 2 as shown: 2,6 8,5 7,1 and 3, 9 .After we divide the arrays again to get 8 arrays of size 1: 2 6 8 5 7 1 3 9 .After this, we have the combining step. We will compare each element with its consecutive elements and arrange them in a sorted manner: 2,6 5,8 1,7 3,9.

In the next iteration, we will compare two arrays and sort them as shown: 2,5,6,8 1,3,7,9. Finally, we will compare the elements of the two arrays each of size 4 and we will get our resultant sorted array as shown: 1,2,3,4,5,6,7,9.

Merge sort is one of the most widely used algorithms in data structures. Although it is not a space-efficient algorithm, its time complexity is of the order O(n logn) which is better than most of the sorting algorithms. (GeeksforGeeks, 2018) (InterviewBit, 2019)

.

Answer 4: Lirdi Ramku

Heap Sort

Heap sort is a type of data structure which is used primarily to continually alter or remove objects from the highest order of priority or the lowest. If insertions are required to be interspersed with a root nodes removal, Heap is also used. Generally, heap is implanted through binary heap which uses a type of binary tree. The highest order of priority and the lowest order of priority is divided into the Minimum Heap and the Maximum Heap.

Maximum Heap: This key is used to signify a root node possesses the greatest key to the children's nodes. This concept is repeated with the lower and all sub trees

Minimum Heap: Alternatively, the Minimum Heap will be used to display the root node as a minimum and the subtrees below being expanded into larger keys. (GeeksforGeeks, 2021)

Algorithm's analysis

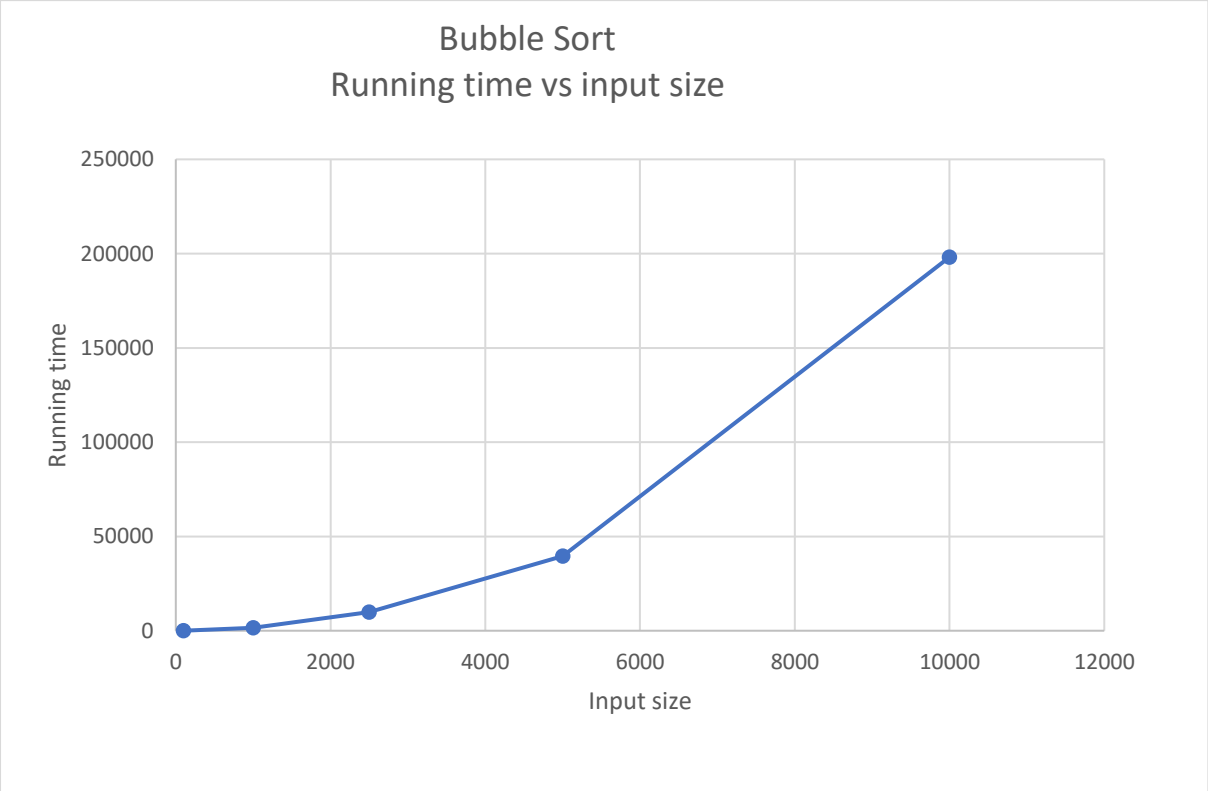
Analysis of Solution 1: Radoslav Gadzhovski

Bubble Sorting Algorithm

Table used to create the chart:

	Input size	Running time
1	100	18.82600132
2	1000	1578.882001
3	2500	9889.58
4	5000	39540.62
5	10000	198172.002

Complexity plot for Bubble Sort:



Analysis of Solution 2: Jasdeep Kaur

Insertion Sorting Algorithm

Table used to create the chart:

	Input size	Running time
1	100	8.527999744
2	1000	90.93799861
3	2500	231.8280004
4	5000	455.4839991
5	10000	915.8200049

Complexity plot for Insertion Sort:

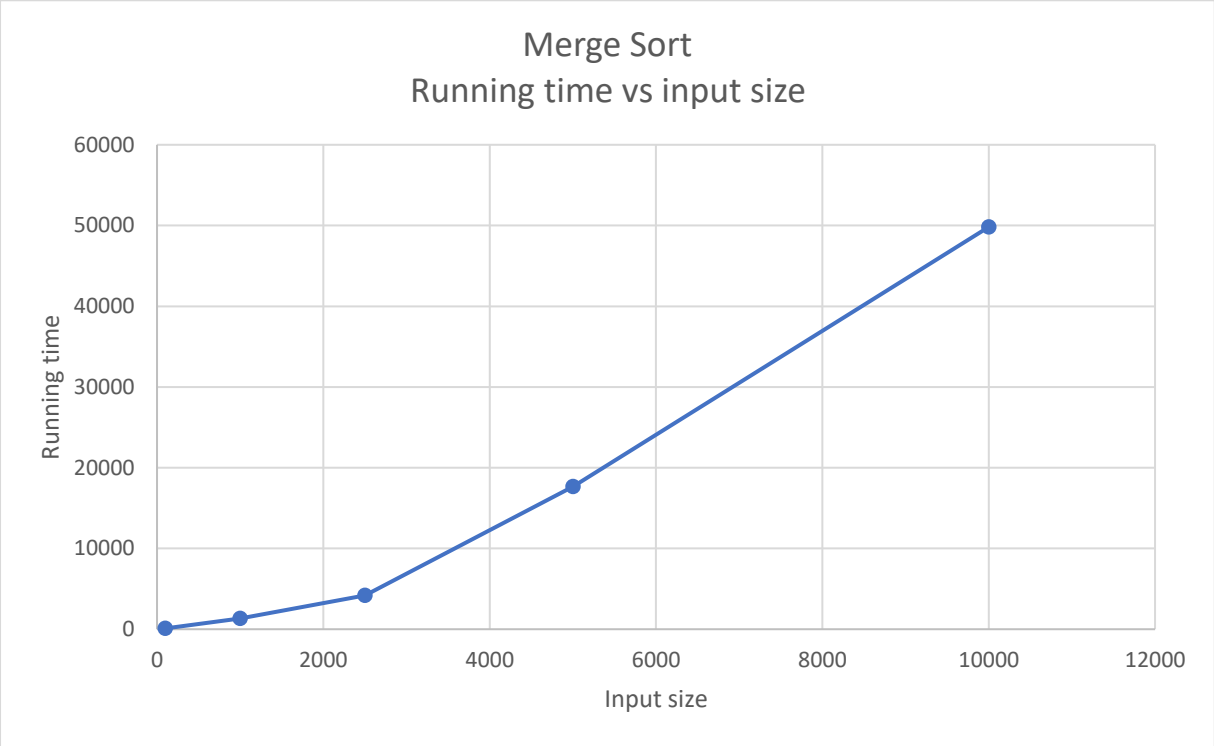
Analysis of Solution 3: Violeta Marin

Merge Sorting Algorithm

Table used to create the chart:

	Input size	Running time
1	100	102.7820003
2	1000	1333.995999
3	2500	4198.777999
4	5000	17650.842
5	10000	49816.112

Complexity plot for Merge Sort:



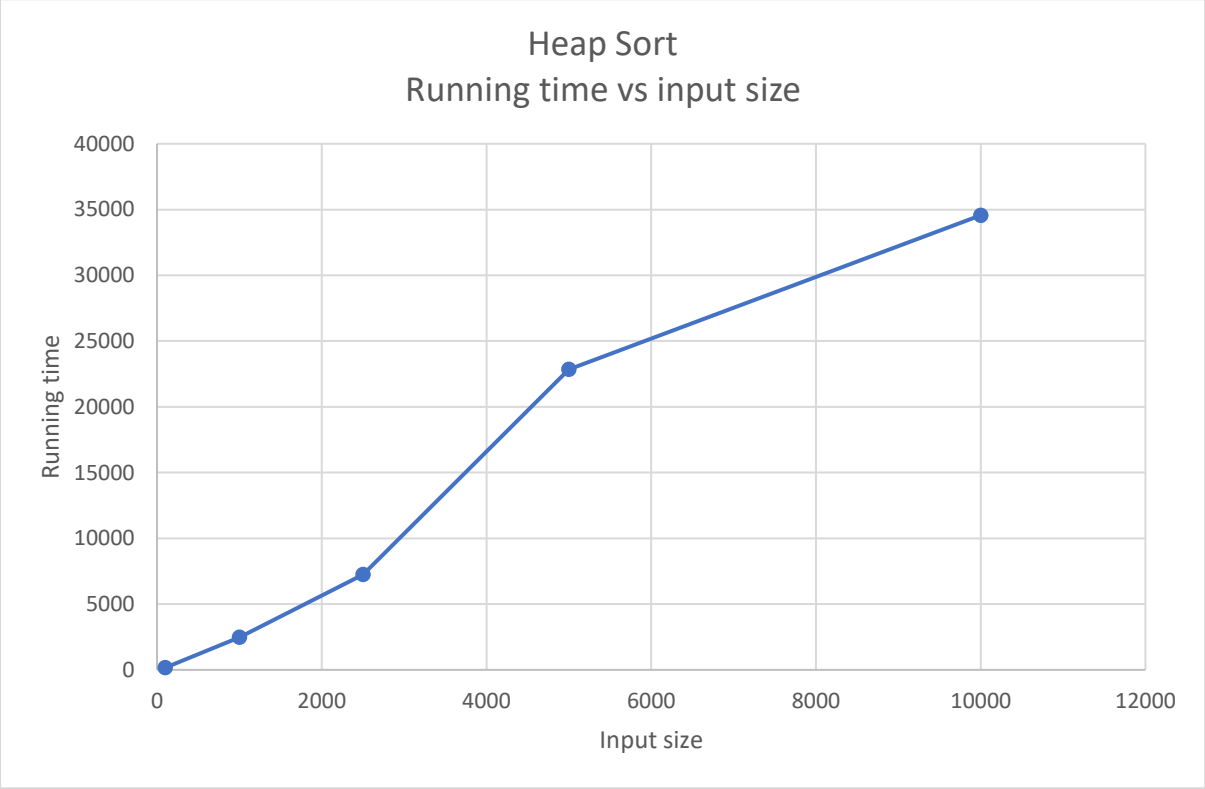
Analysis of Solution 4: Lirdi Ramku

Heap Sorting Algorithm

Table used to create the chart:

	Input size	Running time
1	100	166.8579993
2	1000	2481.314002
3	2500	7235.951999
4	5000	22845.76
5	10000	34562.886

Complexity plot for Heap Sort:

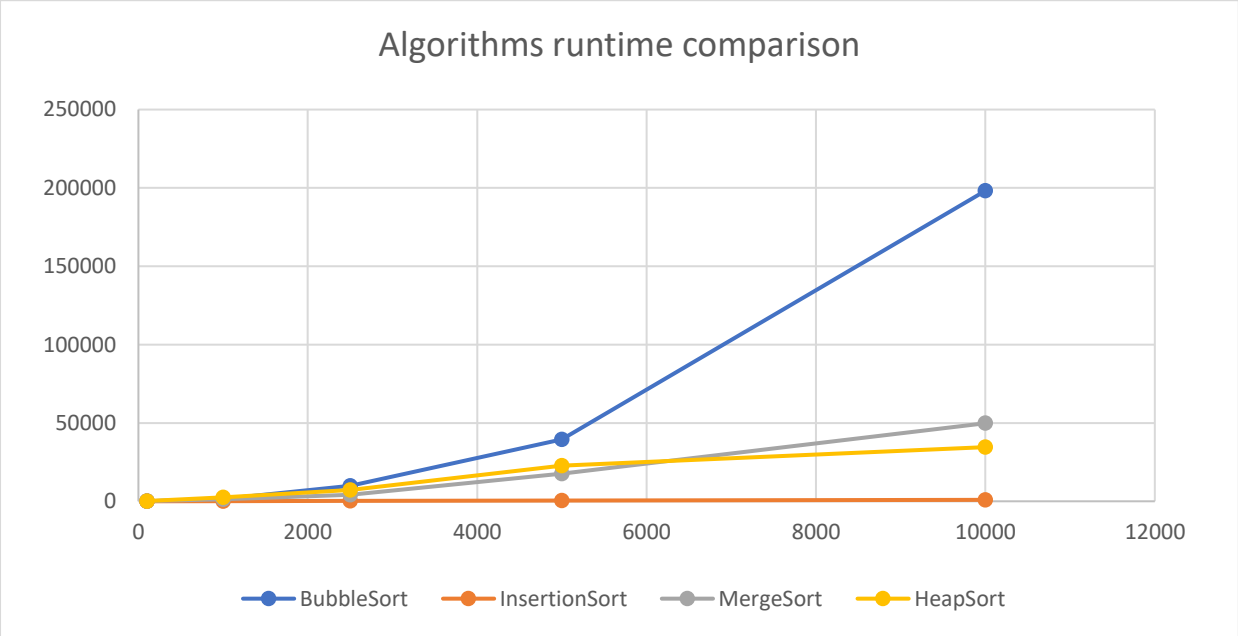


Analysis/Comparison of all Solutions

Table used to create the chart:

	Input size	Running time			
	/////////	Bubble Sort	Insertion Sort	Merge Sort	Heap Sort
1	100	18.82600132	8.527999744	102.7820003	166.858
2	1000	1578.882001	90.93799861	1333.995999	2481.314
3	2500	9889.58	231.8280004	4198.777999	7235.952
4	5000	39540.62	455.4839991	17650.842	22845.76
5	10000	198172.002	915.8200049	49816.112	34562.886

Complexity plot for Bubble Sort, Insertion Sort, Merge Sort and Heap Sort:



Reflection

Each member took an equal part and the contribution is equal.

Name	ID	Individual effort given agreed by team
Radoslav Gadzhovski	001200583	25 % effort
Lirdi Ramku	001163418	25 % effort
Jasdeep Kaur	001180518	25 % effort
Violeta Marin	001136618	25 % effort

References

GeeksforGeeks, 2018. *Merge Sort - GeeksforGeeks.* [Online]
Available at: https://www.geeksforgeeks.org/merge-sort/
[Accessed 17 03 2022].

GeeksforGeeks, 2021. *GeeksforGeeks.* [Online]
Available at: https://www.geeksforgeeks.org/heap-data-structure/
[Accessed 17 03 2022].

InterviewBit, 2019. *Merge Sort Algorithm With Example Program.* [Online]
Available at: https://www.interviewbit.com/tutorial/merge-sort-algorithm/
[Accessed 17 03 2022].

Interviewbit, 2022. *Interviewbit.* [Online]
Available at: https://www.interviewbit.com/tutorial/insertion-sort-algorithm/
[Accessed 16 03 2022].

TechVidvan, 2021. *Merge Sort in Data Structure.* [Online]
Available at: https://techvidvan.com/tutorials/merge-sort/
[Accessed 16 03 2022].

Upadhyay, S., 2022. *Simplilearn.* [Online]
Available at: https://www.simplilearn.com/tutorials/data-structure-tutorial/bubble-sort-algorithm
[Accessed 11 03 2022].

Wikipedia, 2022. *Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/Bubble_sort
[Accessed 11 03 2022].

Appendix

Appendix Main-file – Testing Program(main.py)

We created little Python program which asks the user which file should be taken as input from the directory “Input data” and outputs the runtime analysis in a form of a text file in the directory “output”.

```
1.  # University Of Greenwich
2.  # Algorithms and Data Structures (ADS) - COMP1819
3.  # Design and develop an interview problem and its solution using algorithms and
    data structures in Python.
4.  # -----Testing the efficiency of 4 different algorithms-----
5.
6.
7.  # Import module for measuring execution time
8.  import timeit
9.
10. # Importing all used algorithms
11. from bubble import bubble_sorting
12. from heap import heap_sorting
13. from insertion import insertion_sorting
14. from merge import merge_sorting
15.
16. # Importing the input and output file directory
17. input_dir = "input_data/"
18. output_dir = "output/"
19.
20.
21. # Function to analyze runtime of the algorithms
```

```
def time_analysis(algorithm, data):
    total_time = 0
    for num in range(50):
        begin = timeit.default_timer()
        result = algorithm(data)
        end = timeit.default_timer()
        total_time += end - begin
    return (total_time / 50) * 1000000, result


# Function checking functionality of algorithms with a small sequence of values (50)
def check_algorithms(options, nature, data):
    time_bubble, result_b = time_analysis(bubble_sorting, data)
    time_insert, result_i = time_analysis(insertion_sorting, data)
    time_merge, result_m = time_analysis(merge_sorting, data)
    time_heap, result_h = time_analysis(heap_sorting, data)
    if options == 'functionality':
        out_file = ""
        out_file += f'Output of Bubble Sort is: \n{str(result_b)}\n'
        out_file += f'Output of Insertion Sort: \n{str(result_i)}\n'
        out_file += f'Output of Merge Sort: \n{str(result_m)}\n'
        out_file += f'Output of Heap Sort: \n{str(result_h)}\n'
        open(f'{output_dir}algorithm_check.txt', "w+").write(out_file)
        print(f'To check the algorithm correctness open {output_dir}algorithm_check.txt ')

    out_time = ""
    out_time += f'Algorithm Runtime Analysis\n\n'
    out_time += f'Input Length: {len(data)}\n'
    out_time += f'Bubble sort runtime (in microsecond): {time_bubble}\n'
    out_time += f'Insertion sort runtime (in microsecond): {time_insert}\n'
    out_time += f'Merge sort runtime (in microsecond): {time_merge}\n'
    out_time += f'Heap sort runtime (in microsecond): {time_heap}\n'
    open(f'{output_dir}{options}_runtime.txt', "w+").write(out_time)
    print(f'Runtime analysis can be found in {output_dir}{options}_runtime.txt')


if __name__ == '__main__':
    while True:
        selection = int(input("Choose option:"
                              "\n[1]Check if algorithm is correct\n"
                              "[2]Analysis of runtime of the sorting algorithm \n[3]Exit\n"
                              "Choice:"))
        if selection == 1:
            input_string = open(f"{input_dir}correctness.txt", "r").read().replace(' ', '').split(",")
            input_data = [int(number) for number in input_string]
            check_algorithms('functionality', 'random', input_data)
        elif selection == 2:
            data_size = int(input("Specify one of the data sizes: [100, 1000, 2500, 5000, 10000]:"))
            data_nature = 'float_random'
            input_string = open(f'{input_dir}{data_nature}_{data_size}.txt', "r").read().split(",")
            input_data = [number for number in input_string]
            check_algorithms('analysis', data_nature, input_data)
        elif selection == 3:
            print("Exiting")
            break
        else:
            print("Please choose from option #1, #2 or #3\n")
```

Appendix A.1 – Proposed solution 1

Radoslav Gadzhovski

Bubble Sort algorithm – bubble.py

```
1.  def bubble_sorting(lst):
2.      length_of_lst = len(lst) - 1
3.      ready = False
4.      while not ready:
5.          ready = True
6.          for i in range(length_of_lst):
7.              if lst[i] > lst[i + 1]:
8.                  ready = False
9.                  lst[i], lst[i + 1] = lst[i + 1], lst[i]
10.     return lst
```

Appendix A.2 – Proposed solution 2

Jasdeep Kaur

Insertion Sort algorithm – insertion.py

```
1.  def insertion_sorting(lst):
2.      for ind in range(1, len(lst)):
3.          present = lst[ind]
4.          place = ind
5.          while place > 0 and lst[place-1] > present:
6.              lst[place] = lst[place-1]
7.              place -= 1
8.          lst[place] = present
9.      return lst
```

Appendix A.3 – Proposed solution 3

Violeta Marin

Merge Sort algorithm – merge.py

```
def merge(left_array, right_array):
    new_list = []
    while len(left_array) != 0 and len(right_array) != 0:
        if left_array[0] < right_array[0]:
            new_list.append(left_array[0])
            left_array.remove(left_array[0])
        else:
            new_list.append(right_array[0])
            right_array.remove(right_array[0])
    if len(left_array) == 0:
        new_list += right_array
    else:
        new_list += left_array
    return new_list


def merge_sorting(lst):
    if len(lst) < 2:
        return lst
    else:
        middle = len(lst) // 2
        left_array = merge_sorting(lst[:middle])
        right_array = merge_sorting(lst[middle:])
        return merge(left_array, right_array)
```

Appendix A.4 – Proposed solution 4

Lirdi Ramku

Heap Sort algorithm - heap.py

```
def heap_sorting(array):
    len_arr = len(array) - 1
    least = len_arr // 2
    for i in range(least, -1, -1):
        moving_down(array, i, len_arr)
    for i in range(len_arr, 0, -1):
        if array[0] > array[i]:
            replace(array, 0, i)
            moving_down(array, 0, i - 1)
    return array


def moving_down(array, first, last):
    biggest = 2 * first + 1
    while biggest <= last:
        if (biggest < last) and (array[biggest] < array[biggest + 1]):
            biggest += 1
        if array[biggest] > array[first]:
            replace(array, biggest, first)
            first = biggest
            biggest = 2 * first + 1
```

```
        else:
            return


def replace(arr, z, h):
    temporary = arr[z]
    arr[z] = arr[h]
    arr[h] = temporary
```

Appendix B - Test cases for plotting

ID	Input	Output	Comments
1	50 values from correctness.txt	Sorted list – can be found in: output/algorithm_check.txt	50 values too check the corectness of the algorithms
2	100 values from float_random_100.txt	Runtime analysis in: output/analysis_runtime.txt	Larger sequence to check the runtime
3	1000 values from float_random_1000.txt	Runtime analysis in: output/analysis_runtime.txt	Larger sequence to check the runtime
4	2500 values from float_random_2500.txt	Runtime analysis in: output/analysis_runtime.txt	Larger sequence to check the runtime
5	5000 values from float_random_5000.txt	Runtime analysis in: output/analysis_runtime.txt	Larger sequence to check the runtime
6	10000 values from float_random_100000.txt	Runtime analysis in: output/analysis_runtime.txt	Larger sequence to check the runtime