COMPUTER THINKING WITH ALGORITHMS

FINAL PROJECT

STUDENT: GEORGE ALEXANDRU CRACIUN

G00398303

Table of Contents

1. INTRODUCTION

Sorting algorithms represent a set of instructions, designed to perform a sorting function over an array input. The most commonly ordering of items used in sorting algorithms is numerical order (ascending i.e., 0-9 or descending i.e., 9-0). Numerical ordering it's not the only ordering type sorting algorithms can perform. Among few of other sorting algorithms, the following can be mentioned:

- Lexicographical or alphabetical sorting algorithm.
- Ordering of custom objects (types).

Sorting algorithms are of paramount importance in computer science. It is believed that around 25% of the CPU time spent worldwide is consumed on sorting tasks. For these reasons, studying of sorting algorithms and knowledge of each sorting algorithms strength and weaknesses is very important for any information technology professional.

TIME AND SPACE COMPLEXITY

The subject of time and space complexity of sorting algorithms is very important as it can affect greatly the overall performance of the computer programme which is using any specific sorting algorithm.

The below table shows the characteristics of all the sorting algorithms used in the final project.

Algorithm	Best Case	Worst Case	Average Case	Space Complexity	Stable (Yes/No)
Selection Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	No
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	Yes
Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(n+k)	Yes
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	No
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes

When each two sorting algorithms are compared between each other, few characteristics are considered. Each of these characteristics shows the appropriateness of using one sorting algorithm for a specific application over the other:

- Comparator function: The comparator function in a sorting algorithm represents the number of times the sorting algorithm compares items from the input array to decide the order for each element in the output sorted array. Using Big-O notation all the sorting algorithms require at least O ($n\ logn$) comparisons in the best case and O (n^2) comparisons in the worst case.
- In-place sorting: A sorting algorithm is said about to be an “In-place sorting” if they require a constant O(1) additional space for sorting operation. This characteristic is important for applications where any additional space requirements for performing the sorting operation is critical (additional overhead load for memory usage).
- Stable sort: It is said that a sorting algorithm is stable if after the sorting operation is done, the items which are equal (as defined in in the comparator function) are arranged in the same position for the sorted array in respect with one another as in the input array. If the positions of any two items, which are equal (by the comparator function) are swapped after sorting, it said the sorting algorithm is not stable.

- Comparison-based algorithms: Comparison-based algorithms are performing the sorting operation by comparing each two items in the input array and deciding which is greater than the other by using the operators; greater than, less than and equal to (>, <, =). For any comparison-based sorting algorithm the best complexity is O (n logn) which can be proved mathematically.
- Non-comparison-based algorithms: These types of algorithms are not comparing each item with other items in the input array but rely on arithmetical keys. These types of algorithms are usually faster in terms of time complexity than the comparison-based ones for certain types of input arrays.

2. SORTING ALGORITHMS DIAGRAMS

2.1. SELECTION SORT

Selection sort is a comparison-based algorithm. Time complexity of selection sort is shown in the following table:

Algorithm	Best Case	Worst Case	Average Case
Selection Sort	O(n)	$O(n^2)$	$O(n^2)$

Time complexity: $O(n^2)$ in the worst and average cases as there are two nested arrays.

The good thing about selection sort is that it doesn't take more than O(n) swaps and it's useful when memory write is a costly operation.

SELECTION SORT DIAGRAM

INITIAL INPUT ARRAY: [64, 25, 12, 22 ,11]

1. 64 25 12 22 **11** / min = 11
2. 11 25 **12** 22 64 / min = 12
3. 11 12 25 **22** 64 / min = 22
4. 11 12 22 **25** 64 / min = 25
5. **11 12 22 25 64** / sorted array

Sequence Of Operations:

arr[] = 64, 25, 12, 22, 11

//Find the minimum element in arr[0…4] and place it at the beginning of the array (index i=0)

11 25 12 22 64

//Find the minimum element in arr[1…4] and place it at the index i=1

11 12 25 22 64

//Find the minimum element in arr[2…4] and place it at the index i=2

11 12 22 25 64

//Find the minimum element in arr[3…4] and place it at the index i=3

11 12 22 25 64

2.2. INSERTION SORT

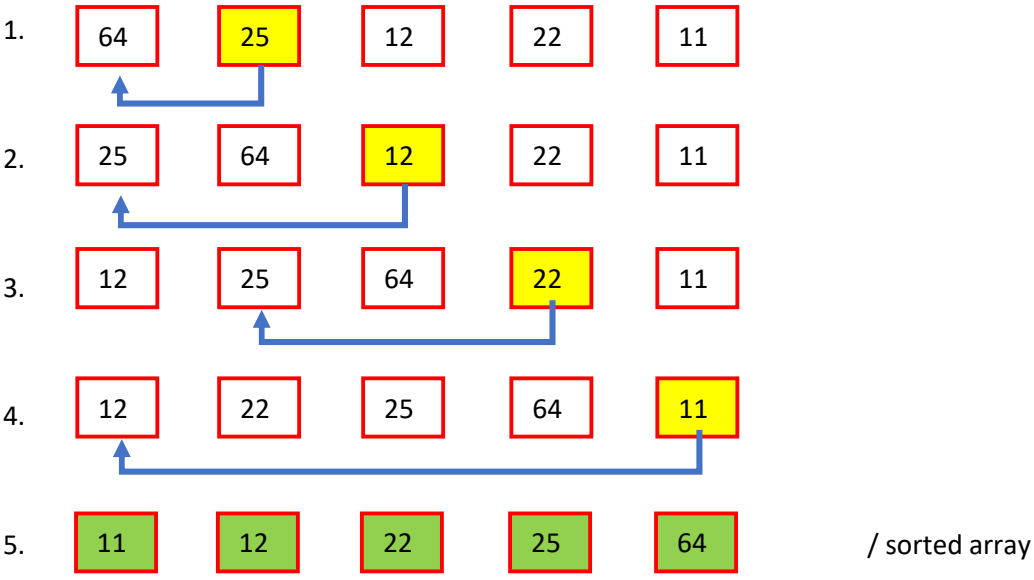
Insertion sort is a comparison-based sorting algorithm which is a simple sorting algorithm and intuitive as it's similar with how a person would sort a playing card deck.

Time complexity: $O(n^2)$

In-place sorting: Yes

Insertion sort is best suited when input array is almost sorted. In this case is offers a good performance. It is alright to use it with a small size array. It offers a bad performance for large input arrays.

INITIAL INPUT ARRAY: [64, 25, 12, 22 ,11]



Sequence Of Operations:

arr[] = 64, 25, 12, 22, 11

// Loop over the array's elements from index i=1 to index i=4

// i=1 -> Since 25 is smaller than 64, move 64 and insert 25 before 64
25, 64, 12, 22, 11

// i=2 -> Since 12 is smaller than 64 and 12, move 12 at index i=0 and all the other smaller indexes will move one position from their current position.
12, 25, 64, 22, 11

// i=3 -> Since 25 and 64 are both greater than 22, move 22 between 12 and 25 (index i = 1). Elements from indexes I = 1 and i = 2 will move to the new positions (one index above current position).
12, 22, 25, 64, 11

// i=4 -> Since 11 is smaller than all the other elements in the array, 11 will move at the beginning of the array and all other elements will move one position ahead from their current position.
11, 12, 22, 25, 64 //Sorted array

2.3. COUNTING SORT

Counting sort is a non- comparison-based algorithm which is the sorting the elements in an input array not by comparing the elements between them, but by the help of hashing elements distinct keys. Counting sort is effective when range is not greater than number of elements to be sorted.

Sequence Of Operations:

INITIAL INPUT ARRAY: arr[] = [5, 4, 8, 7 , 2]

1. Find max (biggest) element in the array -> 8
2. Initialize array of length max + 1 with all elements' value set to 0. This array will be used to store the count of the elements (cumulative sum).

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

3. Initialize output array of length arr.lenght + 1 = 6
4. Store the count of each element from the initial array at their corresponding index in the count array.

INPUT ARRAY

5	4	8	7	2

COUNT ARRAY

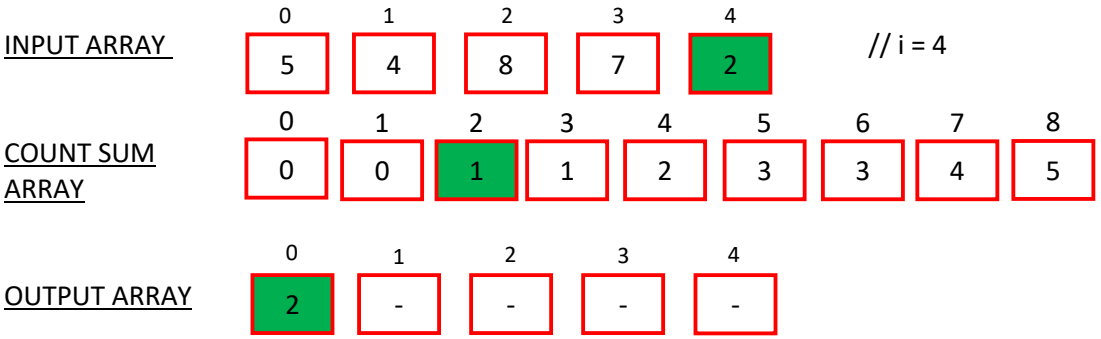
0	1	2	3	4	5	6	7	8
0	0	1	0	1	1	0	1	1

If there is no element in the input array for corresponding index in the count array the value in the count array will be set to 0.

5. Store the cumulative sum of count array elements.

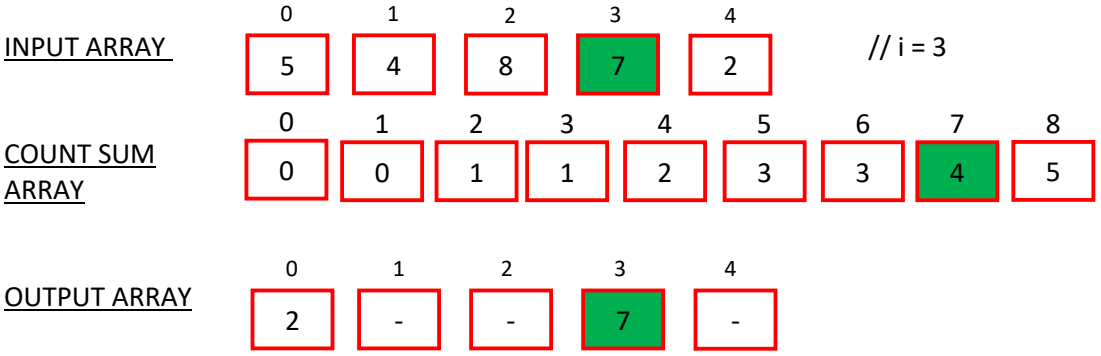
	INDEX POSITION								
Step	i = 0	i = 1	i = 2	i = 3	i = 4	i = 5	i = 6	i = 7	i = 8
5.1. (Initialize i = 1)	0	0	1	0	1	1	0	1	1
	0 + 0 = 0								
5.2. (i = 2)	0	0	1	0	1	1	0	1	1
		0 + 1 = 1							
5.3. (i = 3)	0	0	1	0	1	1	0	1	1
			1 + 0 = 1						
5.4. (i = 4)	0	0	1	1	1	1	0	1	1
				1 + 1 = 2					
5.5. (i = 5)	0	0	1	1	2	1	0	1	1
					2 + 1 = 3				
5.6. (i = 6)	0	0	1	1	2	3	0	1	1
						3 + 0 = 3			
5.7. (i = 7)	0	0	1	1	2	3	3	1	1
							3 + 1 = 4		
5.8. (i = 8)	0	0	1	1	2	3	3	4	1
								4 + 1 = 5	
5.9.	0	0	1	1	2	3	3	4	5
	CUMULATIVE COUNT ARRAY								

6. The following step is done iterative for all elements in the input array. For simplification purpose, only the last two elements sorting process are shown (i = 4 and i = 3).



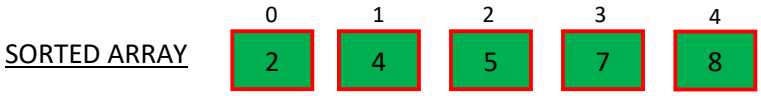
This final step to sort each element in the input array consists in the following operations:

- The value for each element in the input array is read **arr[i=4] = 2**
- The value corresponding to the index A from count array is taken and 1 is subtracted from it. **COUNT_ARRAY[2] = 1 – 1 = 0**
- Now the sorted position of the element at the index i = 4 is set to 0.



This final step to sort each element in the input array consists in the following operations:

- The value for each element in the input array is read **arr[i=3] = 7**
- The value corresponding to the index A from count array is taken and 1 is subtracted from it. **COUNT_ARRAY[7] = 4 – 1 = 3**
- Now the sorted position of the element at the index i = 3 is set to 3.



2.4. HEAP SORT

Heapsort is an in-place algorithm for which the typical implementation is not stable.

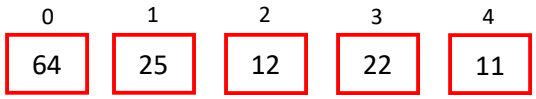
Time complexity: The time complexity of heap sort is O (n logn).

This sorting algorithm is very efficient. This is a comparison-based algorithm based on a binary heap data structure. It is in a way similar with selection sort in that we find a max element in turn (greatest for each iteration) and move it at the end of the input array.
Memory usage of this algorithm is minimal as it doesn't require any additional memory space to perform the sorting operations apart from the space required to host the input array.

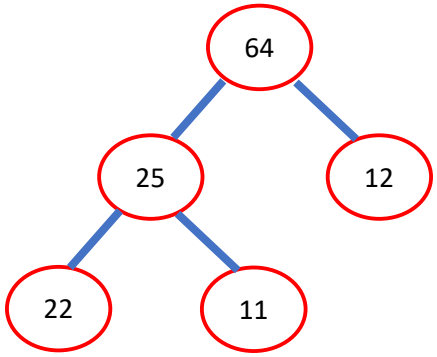
Sequence Of Operations:

arr[] = 64, 25, 12, 22, 11

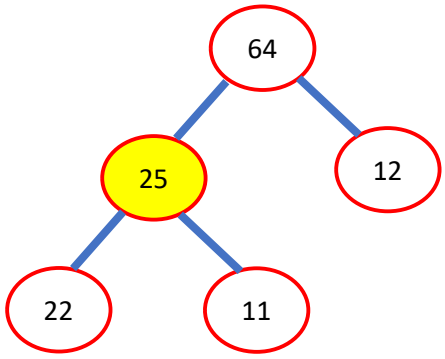
1. INPUT ARRAY



Building a heap data structure from the initial array



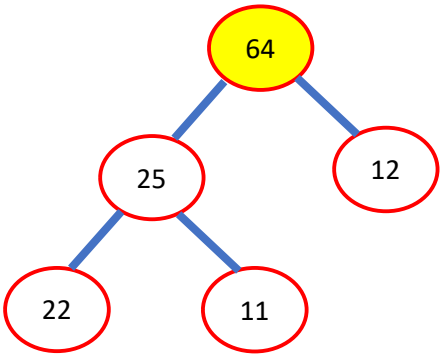
2. Transform the heap data structure to a max heap.
3. The (heapify) operation starts from index i = 1.



//arr[l] < arr[largest]
//arr[r] < arr[largest]

Max heap step is complete at index i=1

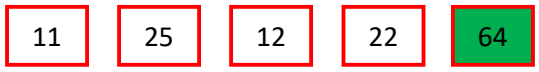
4. The (heapify) operation continues at index i=0.



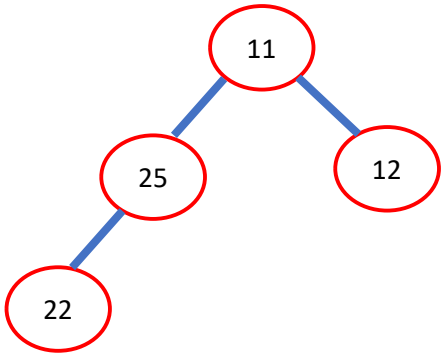
//arr[l] < arr[largest]
//arr[r] < arr[largest]

Max heap step is complete at index i=0

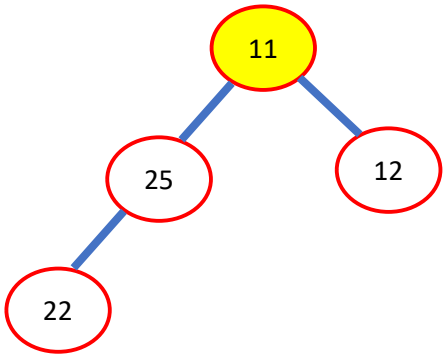
5. At this stage the largest element is stored at the root of the array. Next step is to swap the root with the element at the last index in the array. After swapping the heap data structure will look as shown below.



//Final index of the element with value 64 (now the element is popped from the heap data structure)



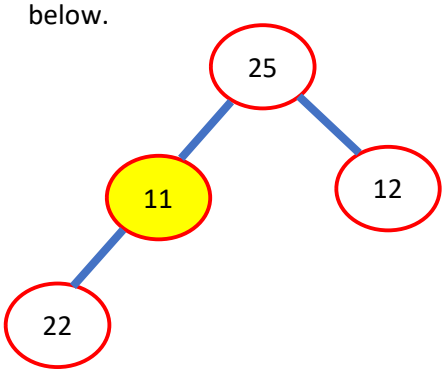
6. The (heapify) operation is recursively called to rearrange again the elements in the heap (index i=0).



arr[l] > arr[i=0]

Heap data needs to be rearranged.

7. The (heapify) operation continues from the element at index i=1 as shown in the diagram below.



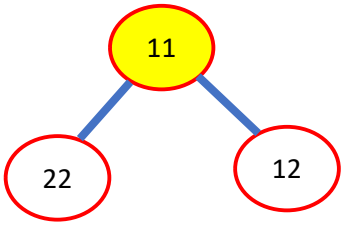
arr[l] > arr[i=1]

Heap data needs to be rearranged.

8. Now the largest element is stored at the root index of the heap. Heap element from index i=0 is swapped with element from index i=3.

//Last two elements are sorted (now element 25 is popped from the heap data structure)

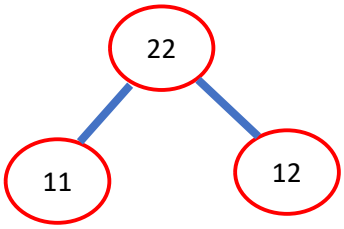
9. After the last step, the heap data structure looks as shown below. The (heapify) operation starts again from index i=0.



arr[l] > arr[i=0]

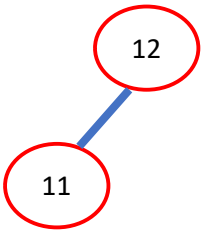
Heap data needs to be rearranged.

10. Step 8 is repeated and now sorted subarray has 3 elements in their final sorted positions.



//Last three elements are sorted (now element 22 is popped from the heap data structure)

11. There are now only 2 elements in the heap data structure. They are already forming a max heap and are ready to be moved in the final output array positions.



OUTPUT ARRAY (Sorted Array)

2.5. MERGE SORT

Merge sort is a very commonly used sorting algorithm based on the "divide conquer paradigm". Merge sort is called a "divide and conquer paradigm" as the first thing in the sorting operation is to divide the input array into subarrays. The division of the arrays happen up to the point when there are only subarrays with just one element. At that point in the sorting operation, the merging of the subarrays starts to take place. The implementation of merge sort is done through recursion.

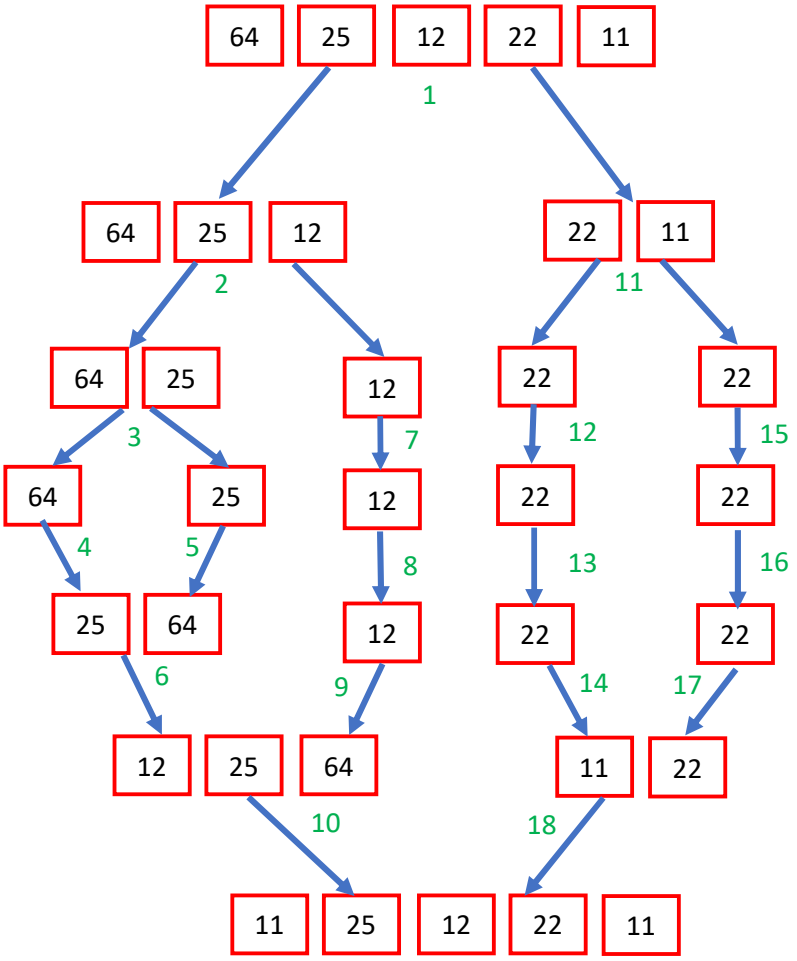
The high-level process of merge sort can be defined as:

- Divide: The input array is divided into two subarrays considering the pivot point at the middle of the input array. This step is carried out recursively until there are no half arrays to be divided.
- Conquer: This step is consisting in the sorting and merging of the already divided arrays. At the last call of this recursive function, we get the sorted output array.

The following diagram shows the recursive nature of merge sort implementation.

Input array is divided into subarrays up to the point when there are only subarrays of size 1. After this, the merging and sorting operation start.

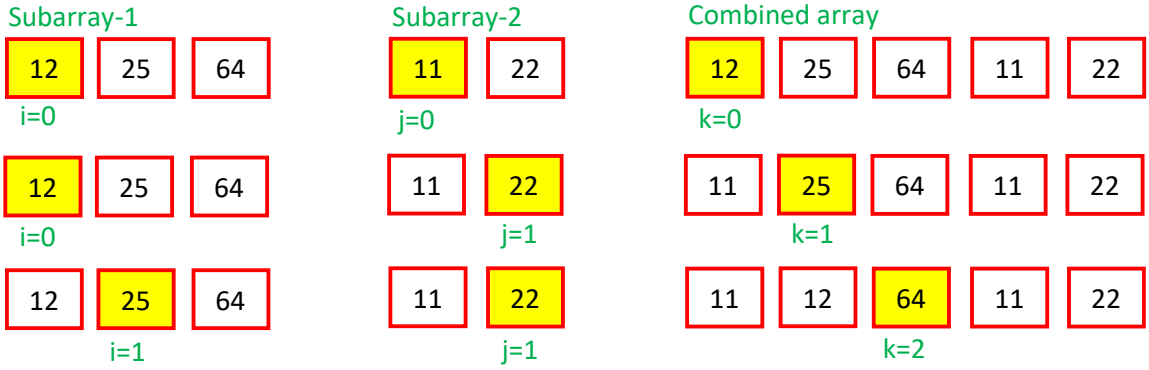
INITIAL INPUT ARRAY: arr[] = [5, 4, 8, 7 , 2]

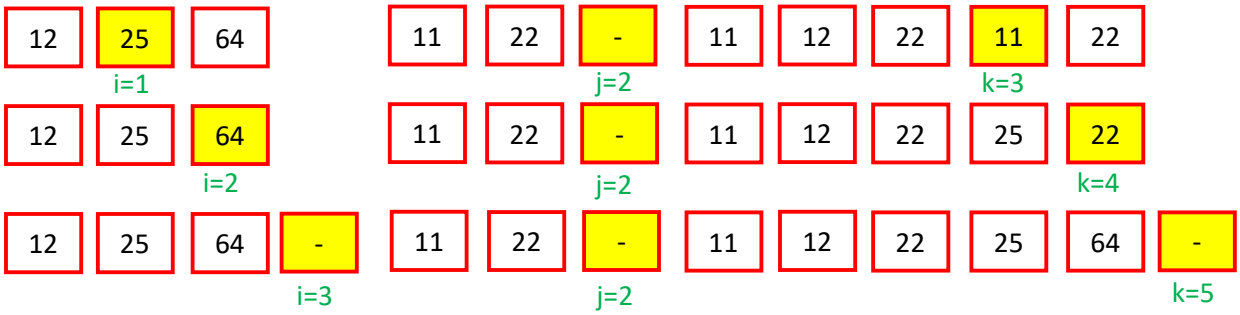


//Green colour numbers represent the order of function stack calls

Time complexity: O(n logn)
Space complexity: O(n) as temporary arrays are created in every recursive call

The most important step in the merge sorting process is the merge step as in this step sorting and merging of the subarrays takes place. Considering the main merge sort function call stack diagram shown above, I will analyse how the merge operation is done at the stack calls number 10 and 18.





OUTPUT ARRAY (Sorted Array)

3. IMPLEMENTATION AND BENCHMARKING

3.1. BENCHMARKING

The results of running the benchmarking sorting algorithms command-line program are presented in Table 1 and Figure 1 below:

Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000	20000
Selection Sort	0.000	0.200	0.100	0.000	0.000	0.900	2.400	8.500	11.300	16.300	23.700	37.700	42.100	157.400
Insertion Sort	0.000	0.000	0.200	0.000	0.000	0.000	0.000	1.000	2.100	3.000	5.200	6.500	9.900	42.300
Counting Sort	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.200	0.000	0.000
Merge Sort	0.000	0.000	0.000	0.000	0.000	0.000	0.400	1.000	1.000	1.500	1.500	1.800	1.200	4.000
Heap Sort	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.100	0.000	0.400	1.000	0.000	0.100	2.100

Table 1: Sample result table – all values are in milliseconds, and are the average of 10 repeated runs

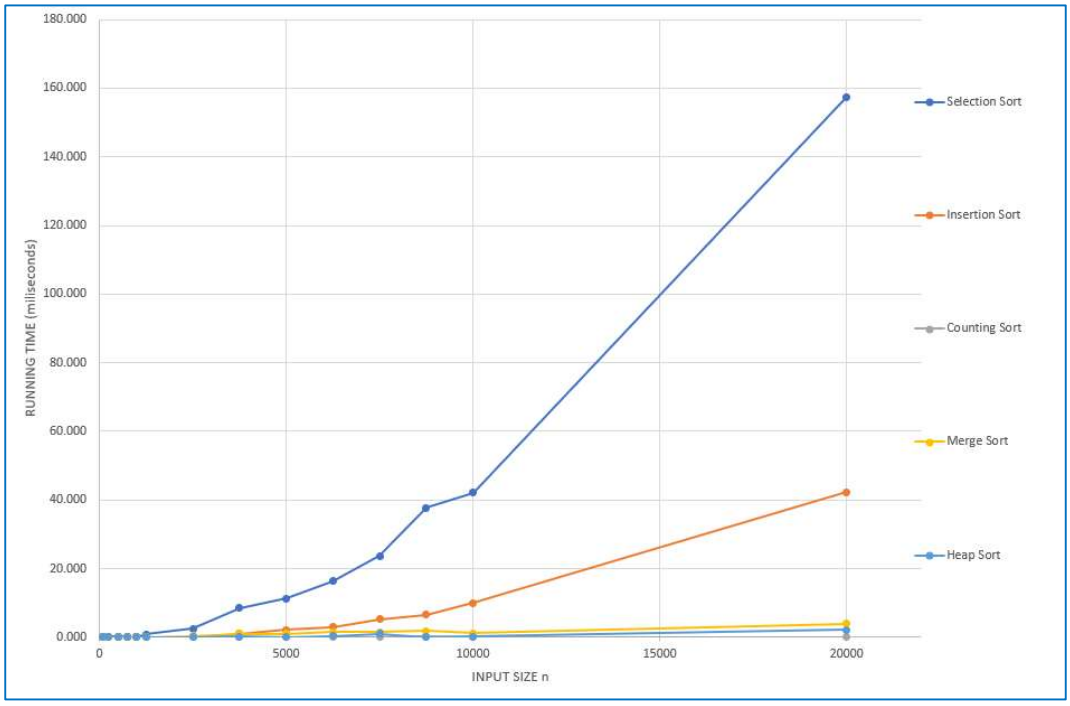


Figure 1: Sample graph (representative for the data shown in Table 1)

3.2. SUMMARY

The following comments can be done regarding the expected results of running the program:

- The best performing algorithms among those analysed is Counting Sort. The results for this algorithm were expected as input array data (integers array) is most suited for usage with this algorithm. Also, a contributing factor which makes this algorithm performance best in comparison with the others is that the range of int's in input array (100) is smaller than the input size.
- The running time complexity for Insertion Sort and Selection sort are both as expected $O(n^2)$. These are not suited to be used in any real-life application which needs to sort input array of these sizes.
- The obtained results for Heap Sort and Merge sort are as expected O(n log n). These represents common good practice to be used in any application to sort input numerical arrays.

3.3. BIBLIOGRAPHY

Insertion Sort - GeeksforGeeks
https://www.geeksforgeeks.org/insertion-sort/ [Accessed on 14.05.2022].

Selection Sort - GeeksforGeeks
https://www.geeksforgeeks.org/selection-sort/ [Accessed on 14.05.2022].

Merge Sort in Java – Baeldung
https://www.baeldung.com/java-merge-sort [Accessed on 14.05.2022].

Counting Sort Algorithm – javaTpoint
https://www.baeldung.com/java-merge-sort [Accessed on 14.05.2022].

Heap Sort – GeeksforGeeks
https://www.geeksforgeeks.org/heap-sort/ [Accessed on 14.05.2022].

Heap Sort – Programiz
https://www.programiz.com/dsa/merge-sort [Accessed on 14.05.2022].