CSE 5311: DESIGN AND ANALYSIS OF ALGORITHMS PROJECT

Sorting Algorithms

Sai Rohith Pasala
1001873156

Spring 2021

CONTENTS

Implement and compare the following search algorithms:

- **Merge Sort**
- **Heap Sort**
- **Quick Sort regular**
- **Quick Sort 3-median**
- **Insertion Sort**
- **Selection Sort**
- **Bubble Sort**

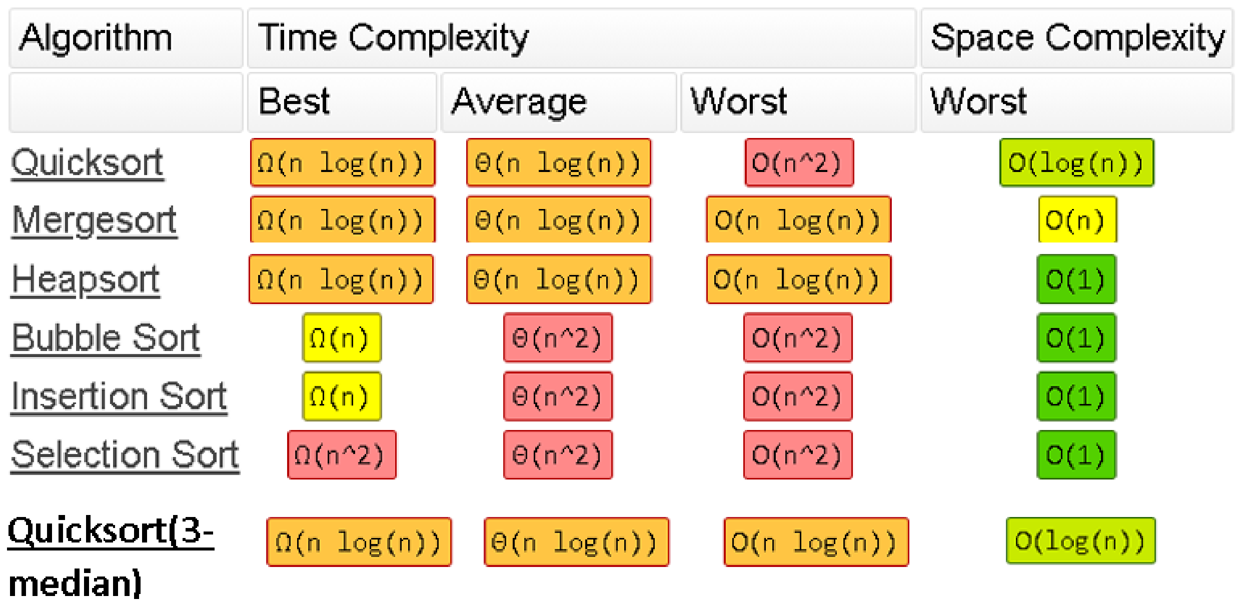
1. Time and Space Complexities:

Time Complexity: Time Complexity is defined as the number of times a particular instruction set is executed rather than the total time is taken. It is because the total time taken also depends on some external factors like the compiler used, processor's speed, programming language used, Operating System, amount of RAM, Cache, other Human Factors etc.

Space Complexity: Space Complexity is the total memory space required by the program for its execution.

Despite these parameters, the efficiency of an algorithm also depends upon **nature** and **size of** the **input**.

Following is the Complexities of different sorting algorithms used in this project:



Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Quicksort(3-median)	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(log(n))

2. Hardware and Software Specifications used:

Hardware Specifications:

❖ Processor	:	Intel i5 7th gen (4 cores)
❖ Hard Disk	:	256 GB.
❖ RAM	:	8 GB.

Software Specifications:

❖ Operating System	:	Windows 10.
❖ Programming Language	:	Python(v. 3.7).
❖ Software IDE	:	Anaconda – Jupyter Notebook.
❖ Text Editor	:	Notepad++.

In this Project, Other than these we have used:

- Python’s Data Structures such as Lists
- Python’s timeit Library and its inbuilt functions
- Python’s assignment operators, user defined functions, Conditional Statements and ‘for’ loops

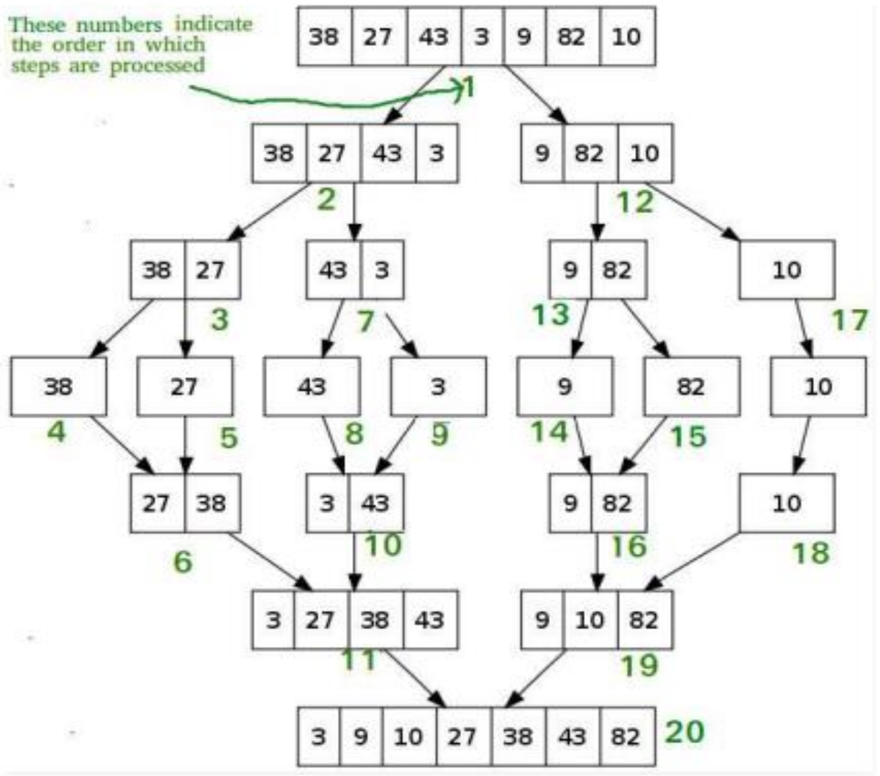
3. SORTING ALGORITHMS

3.1 Merge Sort:

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sub list consists of a single element and merging those sub lists in a manner that results into a sorted list.

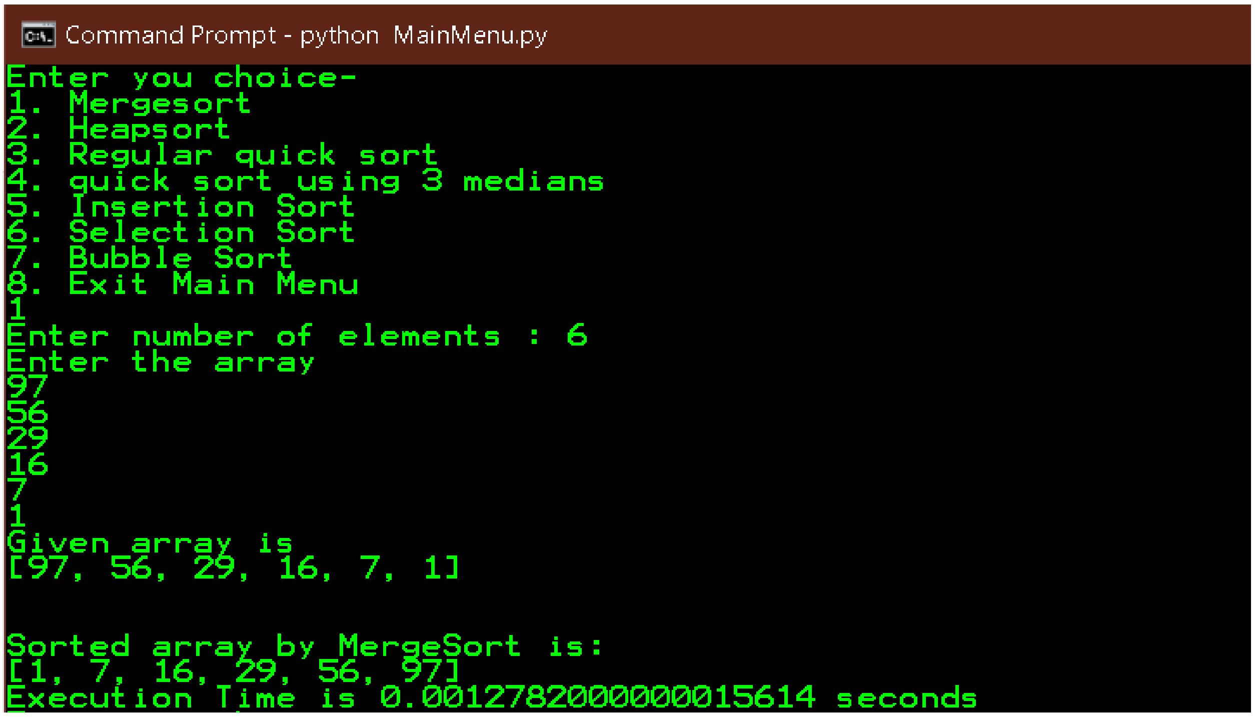
Approach:

- Divide the unsorted list into 2 sub lists each time.
- Do this until each sub list has only 1 element left.
- Take the adjacent pairs of two single element lists and merge the, to form a list of 2 elements. N will now convert into N/2 lists of size 2.
- Repeat the process till a final single sorted list is obtained.



Time Complexity: Worst case – O(n log n) **Space Complexity**: Worst Case – O(n)

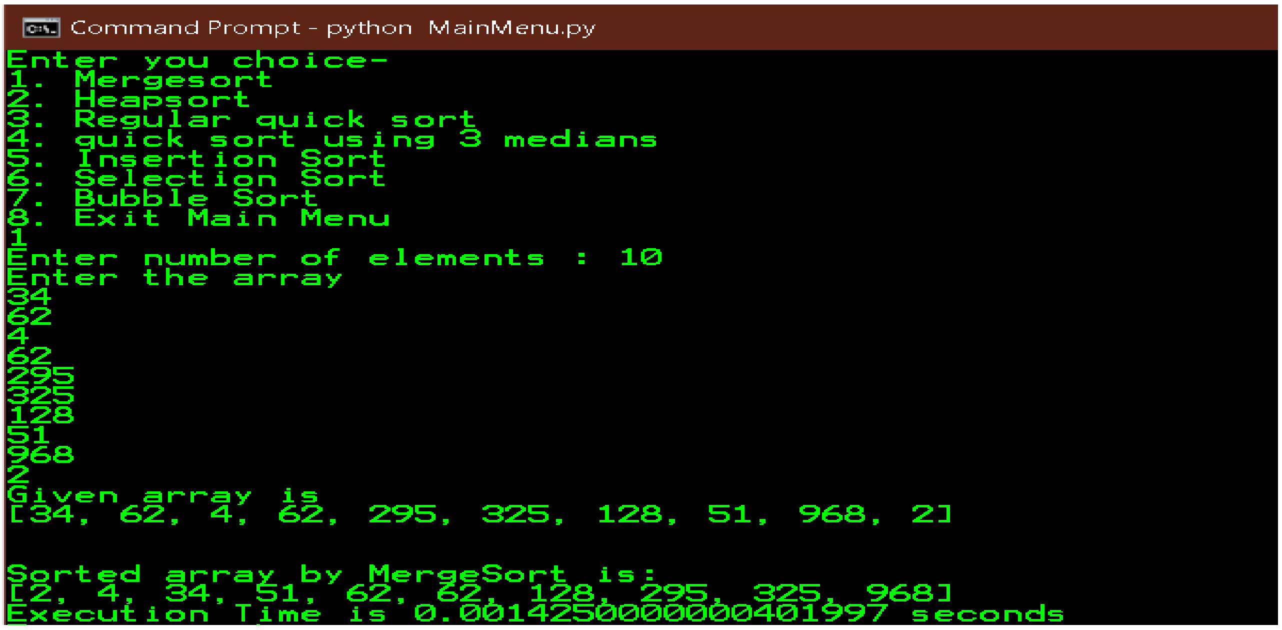
Input 1:



```
Command Prompt - python  MainMenu.py
Enter you choice-
1. Mergesort
2. Heapsort
3. Regular quick sort
4. quick sort using 3 medians
5. Insertion Sort
6. Selection Sort
7. Bubble Sort
8. Exit Main Menu
1
Enter number of elements : 6
Enter the array
97
56
29
16
7
1
Given array is
[97, 56, 29, 16, 7, 1]


Sorted array by MergeSort is:
[1, 7, 16, 29, 56, 97]
Execution Time is 0.0012782000000015614 seconds
```

Input array is [97, 56, 29, 16, 7, 1], execution time: 0.00127820 seconds.

Input 2:



```
Command Prompt - python  MainMenu.py
Enter you choice-
1. Mergesort
2. Heapsort
3. Regular quick sort
4. quick sort using 3 medians
5. Insertion Sort
6. Selection Sort
7. Bubble Sort
8. Exit Main Menu
1
Enter number of elements : 10
Enter the array
34
62
4
62
295
325
128
51
968
2
Given array is
[34, 62, 4, 62, 295, 325, 128, 51, 968, 2]


Sorted array by MergeSort is:
[2, 4, 34, 51, 62, 62, 128, 295, 325, 968]
Execution Time is 0.0014250000000401997 seconds
```

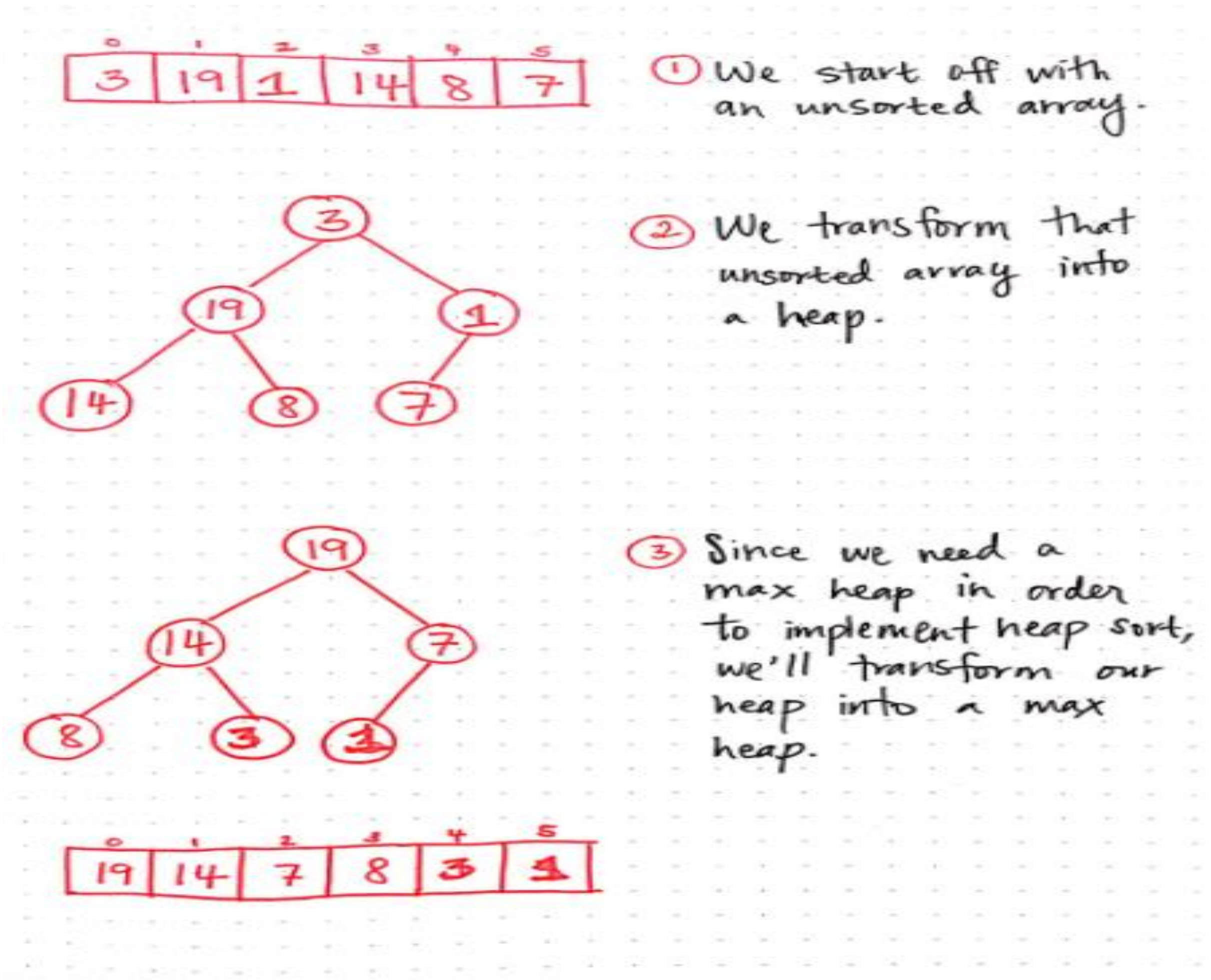
Input array is [34, 62, 4, 62, 295, 325, 128, 51, 968, 2], execution time: 0.00142500 seconds.

3.2 Heap Sort

Heap sort is a comparison-based sorting technique based on binary heap data structure. This Algorithm involves preparing the list to be sorted by first turning it into a max heap. It is like selection sort where we first find the minimum element and place the minimum element at the beginning,

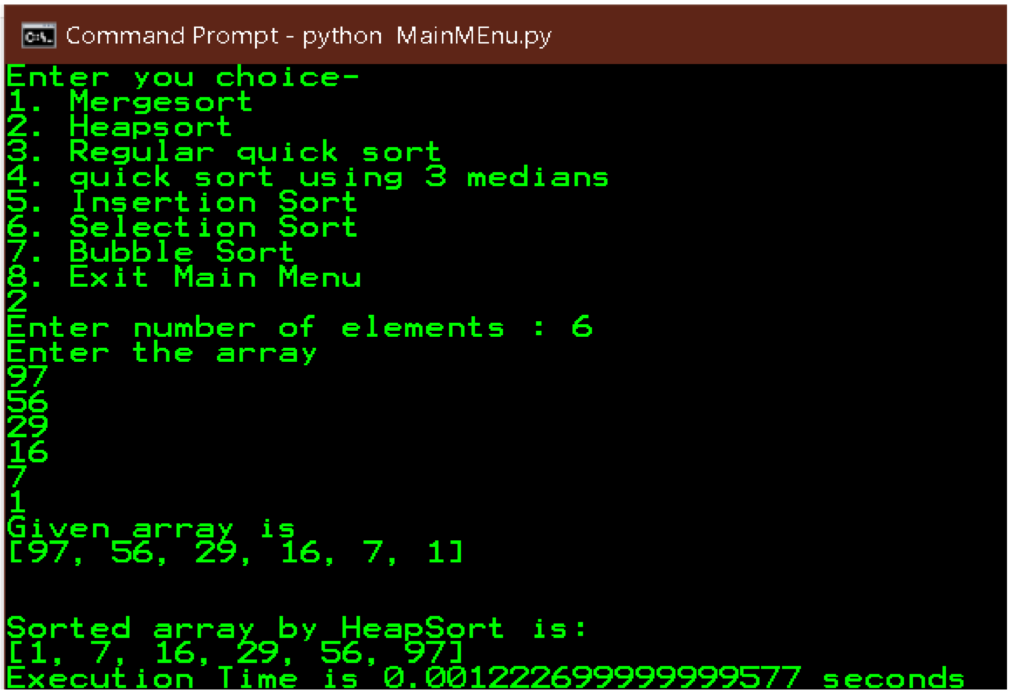
Approach:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of the tree.
3. Repeat above steps while size of heap is greater than 1.


0 1 2 3 4 5
3 19 1 14 8 7
① We start off with an unsorted array.
3 19 1 14 8 7
② We transform that unsorted array into a heap.
19 14 7 8 3 1
③ Since we need a max heap in order to implement heap sort, we'll transform our heap into a max heap.
0 1 2 3 4 5
19 14 7 8 3 1

Time Complexity: O (n log n) for all cases **Space Complexity**: O (1)

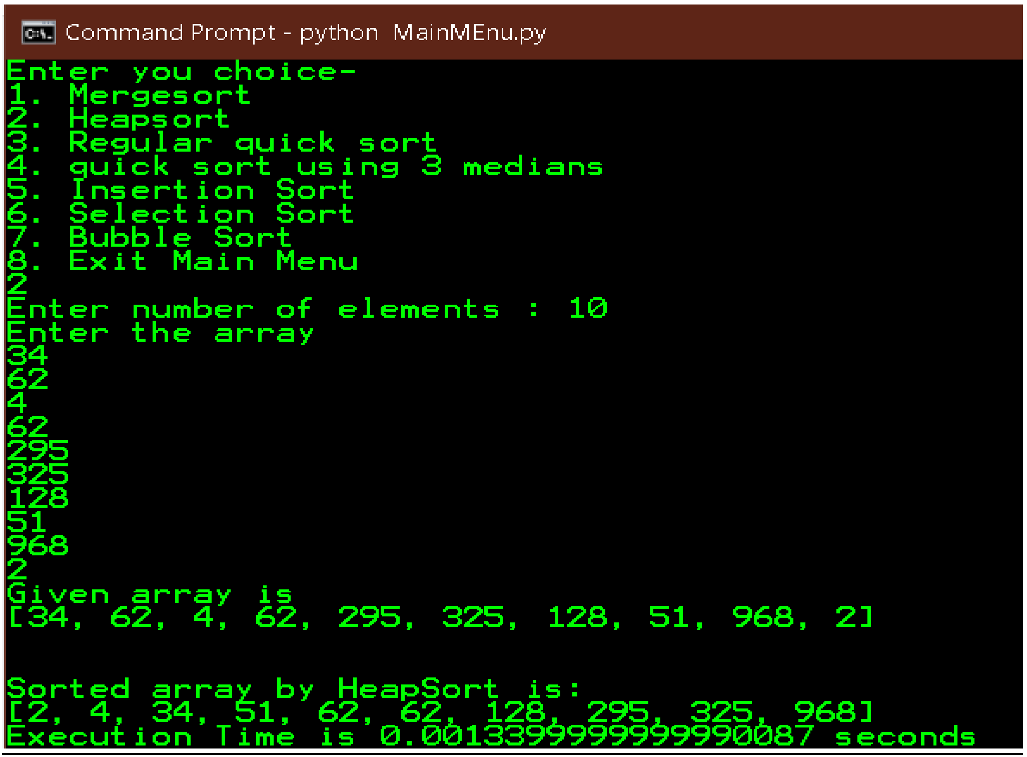
Input 1:



```
Enter you choice-
1. Mergesort
2. Heapsort
3. Regular quick sort
4. quick sort using 3 medians
5. Insertion Sort
6. Selection Sort
7. Bubble Sort
8. Exit Main Menu
2
Enter number of elements : 6
Enter the array
97
56
29
16
7
1
Given array is
[97, 56, 29, 16, 7, 1]


Sorted array by HeapSort is:
[1, 7, 16, 29, 56, 97]
Execution Time is 0.0012226999999999577 seconds
```

Input array is [97, 56, 29, 16, 7, 1], execution time: 0.001222699 seconds.

Input 2:



```
Enter you choice-
1. Mergesort
2. Heapsort
3. Regular quick sort
4. quick sort using 3 medians
5. Insertion Sort
6. Selection Sort
7. Bubble Sort
8. Exit Main Menu
2
Enter number of elements : 10
Enter the array
34
62
4
62
295
325
128
51
968
2
Given array is
[34, 62, 4, 62, 295, 325, 128, 51, 968, 2]


Sorted array by HeapSort is:
[2, 4, 34, 51, 62, 62, 128, 295, 325, 968]
Execution Time is 0.0013399999999990087 seconds
```

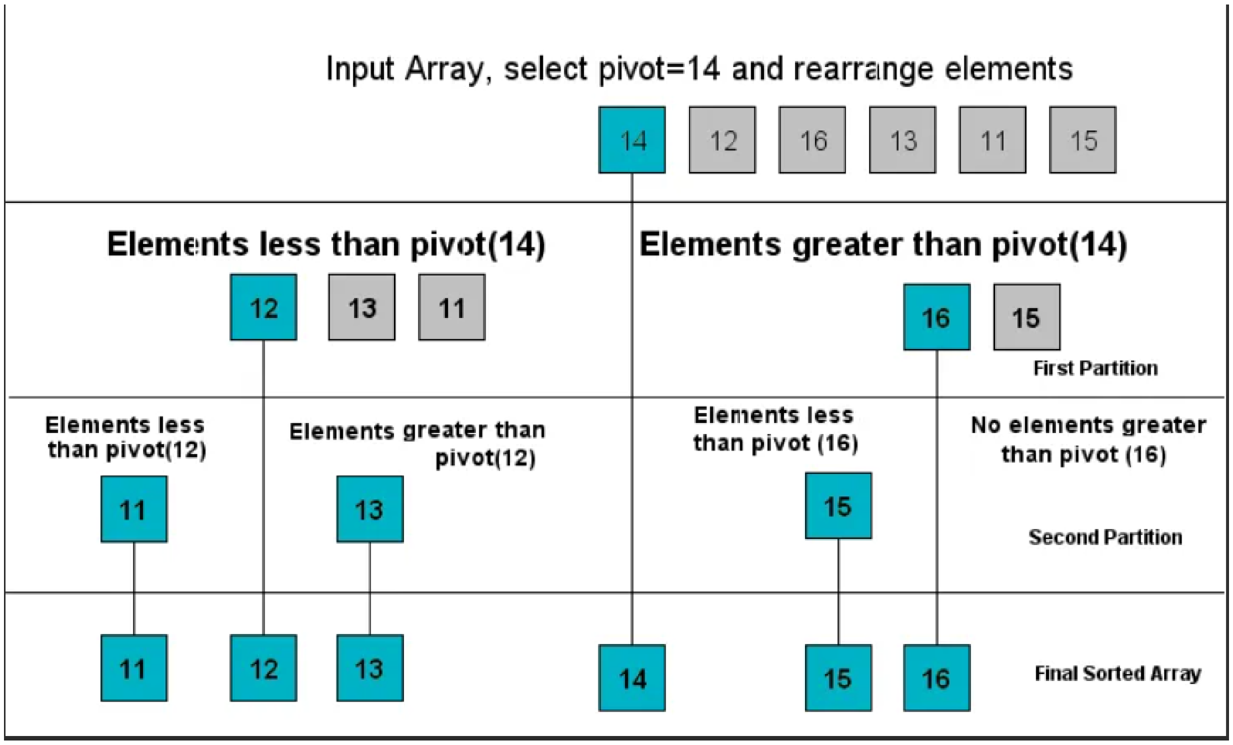
Input array is [34, 62, 4, 62, 295, 325, 128, 51, 968, 2], execution time: 0.00133999 seconds.

3.3 Quick Sort (Regular)

Just like Merge Sort, Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quicksort that pick pivot in different ways.

- Always pick first element as pivot.
- Always pick last element as pivot.
- Pick a random element as pivot.
- Pick a median as pivot.

The key process in quicksort is partition (). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

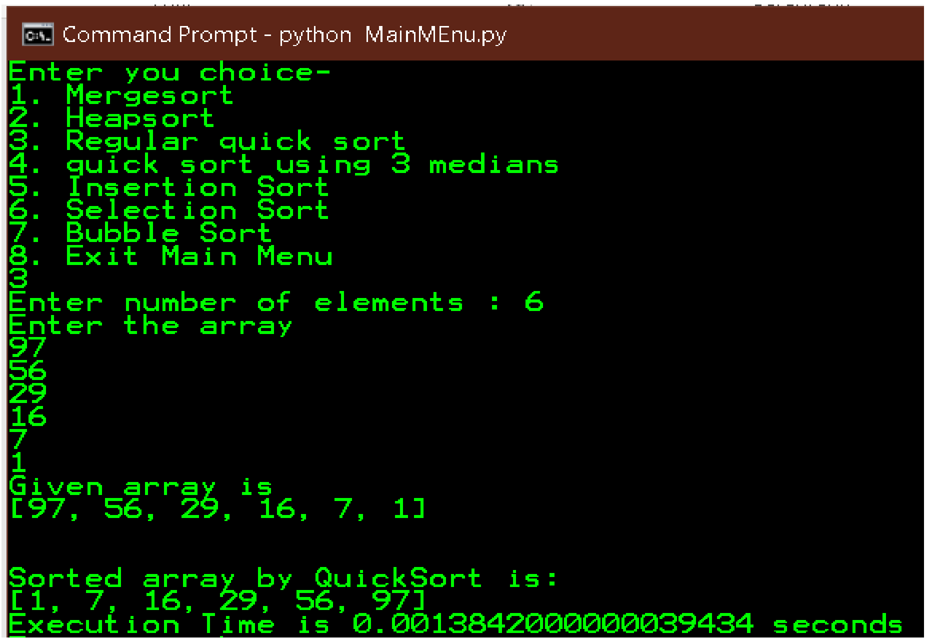

Input Array, select pivot=14 and rearrange elements
14
12
16
13
11
15
Elements less than pivot(14)
Elements greater than pivot(14)
12
13
11
16
15
First Partition
Elements less than pivot(12)
Elements greater than pivot(12)
Elements less than pivot (16)
No elements greater than pivot (16)
11
13
15
Second Partition
11
12
13
14
15
16
Final Sorted Array

In this project we have taken First element as pivot and implemented the quicksort

<u>Time Complexity</u>: O(n log n) for best and average case and for worst case O(n^2)

<u>Space Complexity</u>: O(log n)

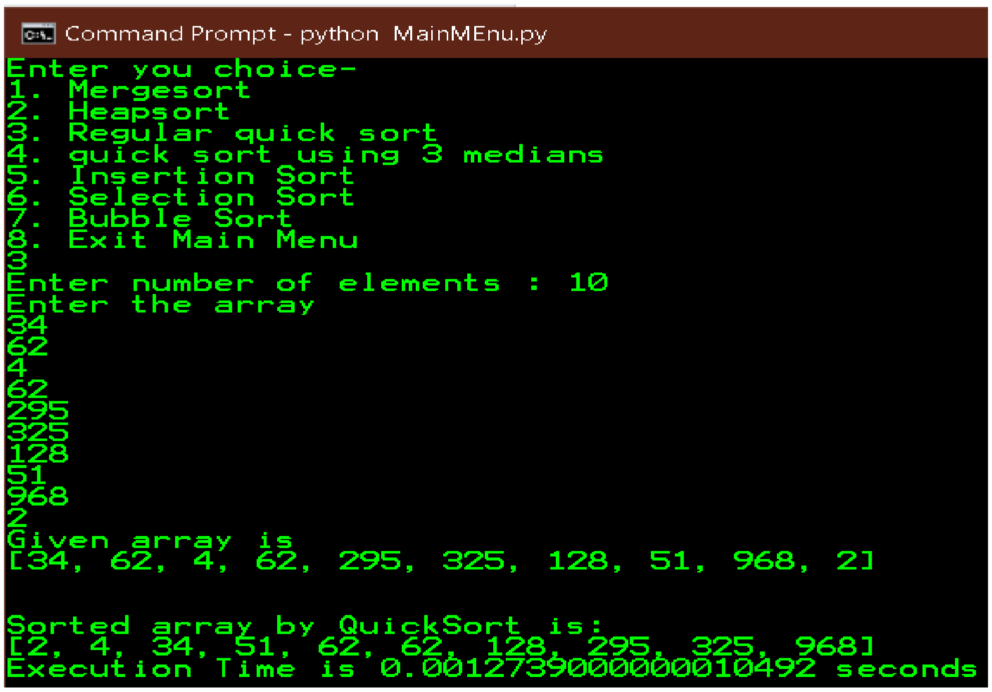
Input 1:



```
Command Prompt - python  MainMEnu.py
Enter you choice-
1. Mergesort
2. Heapsort
3. Regular quick sort
4. quick sort using 3 medians
5. Insertion Sort
6. Selection Sort
7. Bubble Sort
8. Exit Main Menu
3
Enter number of elements : 6
Enter the array
97
56
29
16
7
1
Given array is
[97, 56, 29, 16, 7, 1]


Sorted array by QuickSort is:
[1, 7, 16, 29, 56, 97]
Execution Time is 0.0013842000000039434 seconds
```

Input array is [97, 56, 29, 16, 7, 1], execution time: 0.001384200 seconds.

Input 2:



```
Command Prompt - python  MainMEnu.py
Enter you choice-
1. Mergesort
2. Heapsort
3. Regular quick sort
4. quick sort using 3 medians
5. Insertion Sort
6. Selection Sort
7. Bubble Sort
8. Exit Main Menu
3
Enter number of elements : 10
Enter the array
34
62
4
62
295
325
128
51
968
2
Given array is
[34, 62, 4, 62, 295, 325, 128, 51, 968, 2]

Sorted array by QuickSort is:
[2, 4, 34, 51, 62, 62, 128, 295, 325, 968]
Execution Time is 0.0012739000000010492 seconds
```

Input array is [34, 62, 4, 62, 295, 325, 128, 51, 968, 2], execution time: 0.00127390 seconds.

3.4 Quick Sort(3 -median)

Just like Merge Sort, Quick Sort is a Divide and Conquer algorithm. Only difference between regular quicksort and this is it gives more accurate sorting by considering all 3 pivots instead of suing only one pivot like regular quicksort.

<u>Approach:</u>

Choose the median of the first, middle, and last entries in the list. “This will usually give a better approximation of the actual median” and improves the efficiency of partitioning. In addition, we can:

• move the smallest entry to the first entry

• move the largest entry to the middle entry

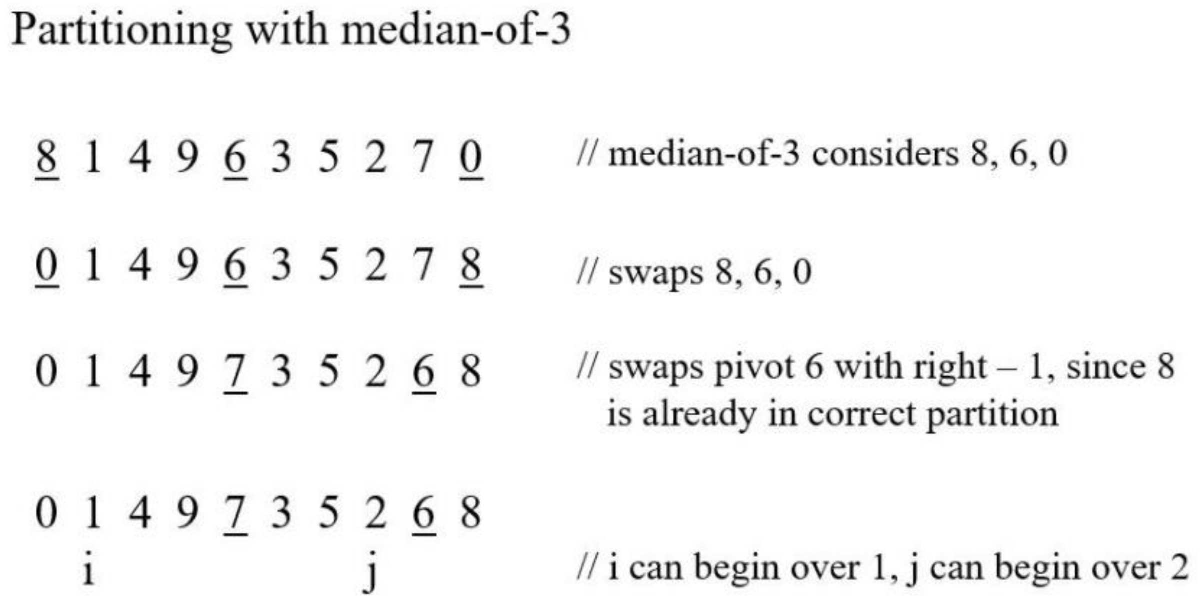
Next, our goal is to partition all remaining elements based on whether they are smaller than or greater than the pivot. We will find two entries:

◦ One larger than the pivot (staring from the front)

◦ One smaller than the pivot (starting from the back)

Which are out of order and then swap them. Continue doing so until the appropriate entries you find are actually in order. Then we move the largest median to the end of the array and move pivot to its place.

Next step is to begin calling quick sort recursively on the first half and it sorts the array in ascending order.



Partitioning with median-of-3

$\underline{8}\ 1\ 4\ 9\ \underline{6}\ 3\ 5\ 2\ 7\ \underline{0}$ // median-of-3 considers 8, 6, 0

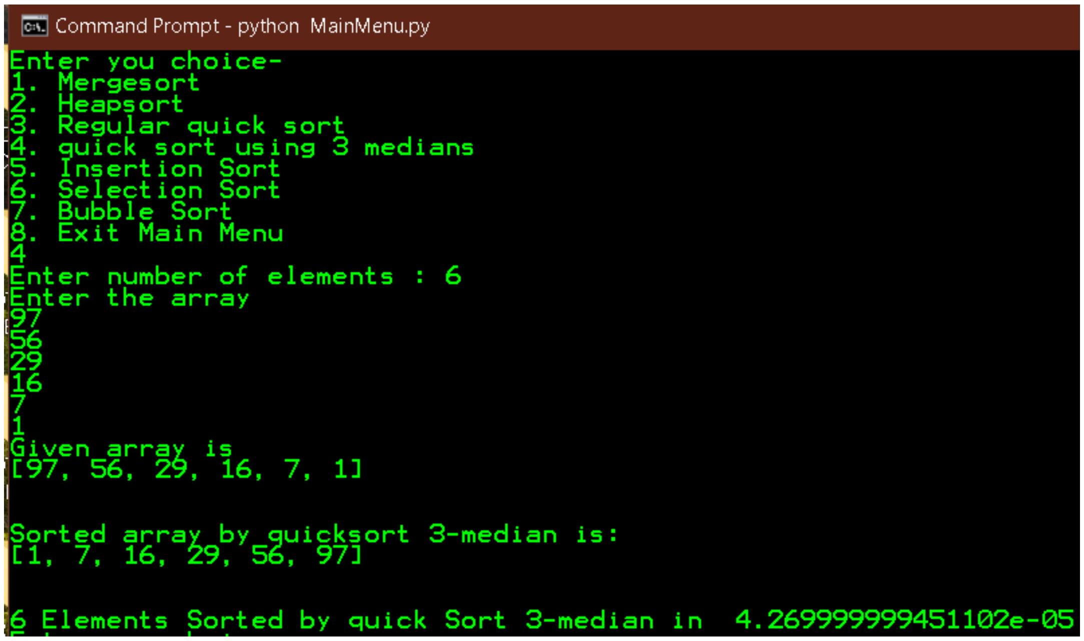
$\underline{0}\ 1\ 4\ 9\ \underline{6}\ 3\ 5\ 2\ 7\ \underline{8}$ // swaps 8, 6, 0

$0\ 1\ 4\ 9\ \underline{7}\ 3\ 5\ 2\ \underline{6}\ 8$ // swaps pivot 6 with right – 1, since 8 is already in correct partition

$0\ \overset{}{\underset{i}{1}}\ 4\ 9\ \underline{7}\ 3\ 5\ \underset{j}{2}\ \underline{6}\ 8$ // i can begin over 1, j can begin over 2

<u>Time complexity:</u> O(n log n) for all cases **<u>Space Complexity:</u>** O(log n)

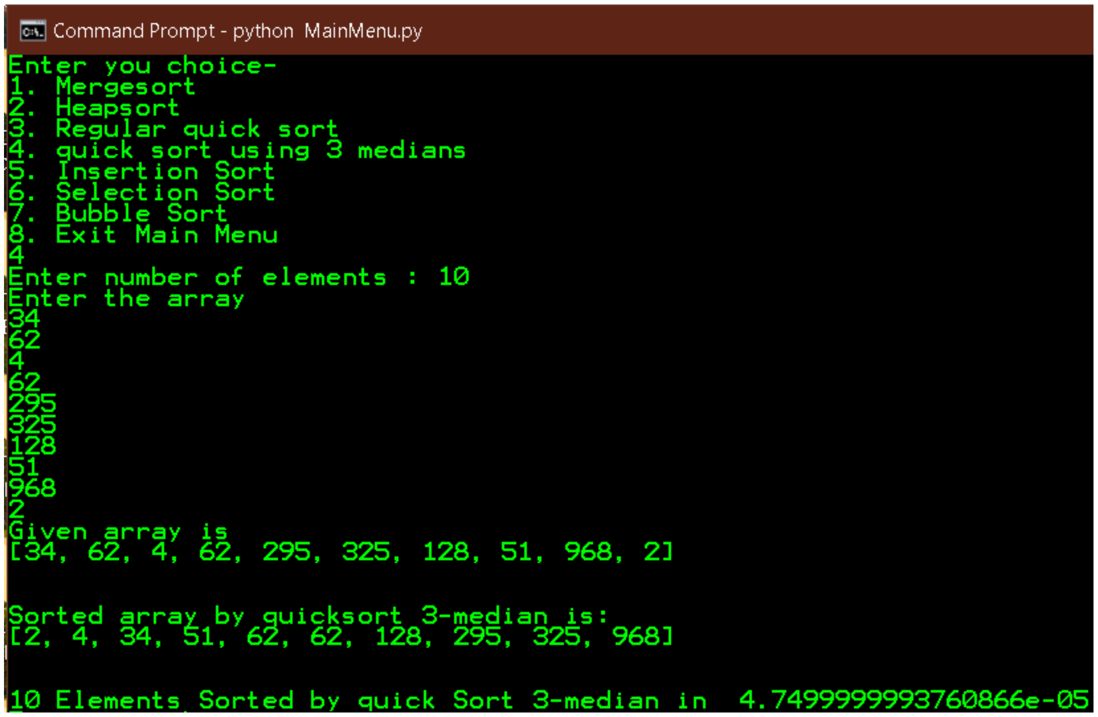
Input 1:



```
Command Prompt - python  MainMenu.py
Enter you choice-
1. Mergesort
2. Heapsort
3. Regular quick sort
4. quick sort using 3 medians
5. Insertion Sort
6. Selection Sort
7. Bubble Sort
8. Exit Main Menu
4
Enter number of elements : 6
Enter the array
97
56
29
16
7
1
Given array is
[97, 56, 29, 16, 7, 1]


Sorted array by quicksort 3-median is:
[1, 7, 16, 29, 56, 97]


6 Elements Sorted by quick Sort 3-median in  4.269999999451102e-05
```

Input array is [97, 56, 29, 16, 7, 1], execution time: 4.269*e^-5 = 0.0012873 seconds

Input 2:



```
Command Prompt - python  MainMenu.py
Enter you choice-
1. Mergesort
2. Heapsort
3. Regular quick sort
4. quick sort using 3 medians
5. Insertion Sort
6. Selection Sort
7. Bubble Sort
8. Exit Main Menu
4
Enter number of elements : 10
Enter the array
34
62
4
62
295
325
128
51
968
2
Given array is
[34, 62, 4, 62, 295, 325, 128, 51, 968, 2]


Sorted array by quicksort 3-median is:
[2, 4, 34, 51, 62, 62, 128, 295, 325, 968]


10 Elements Sorted by quick Sort 3-median in  4.7499999993760866e-05
```

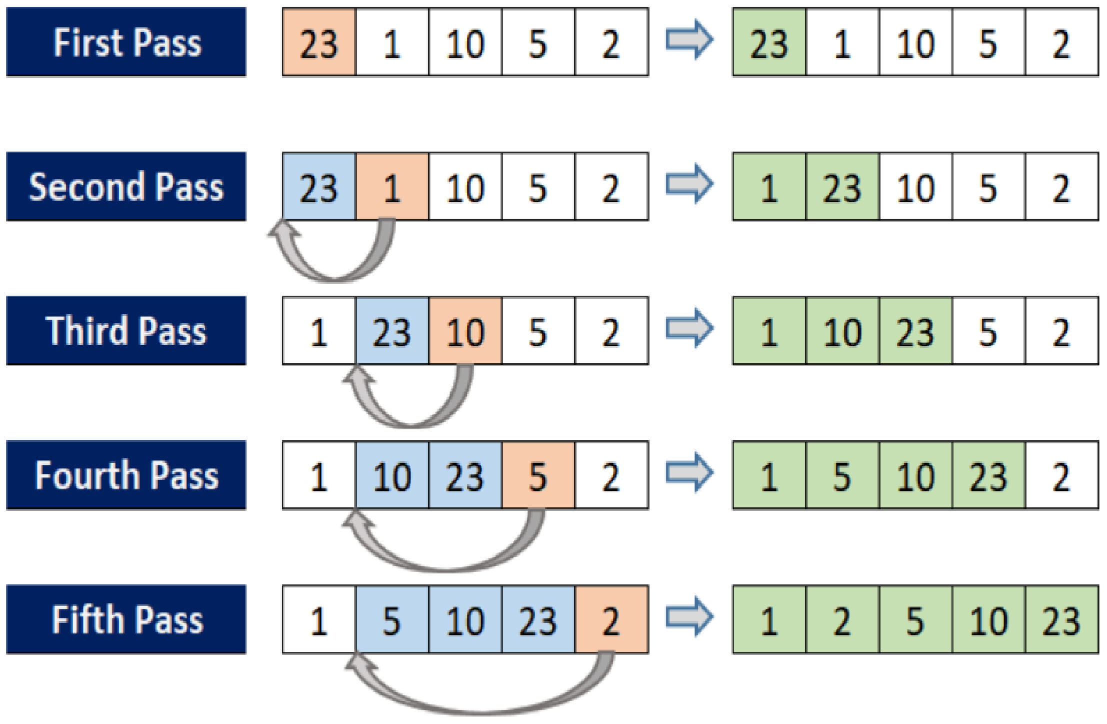
Input array is [34, 62, 4, 62, 295, 325, 128, 51, 968, 2], execution time: 4.749*e^-5 = 0.001316 seconds.

3.5 Insertion Sort

Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time. It is based on the idea that one element from input elements is consumed in each iteration to find its correct position.

Approach:

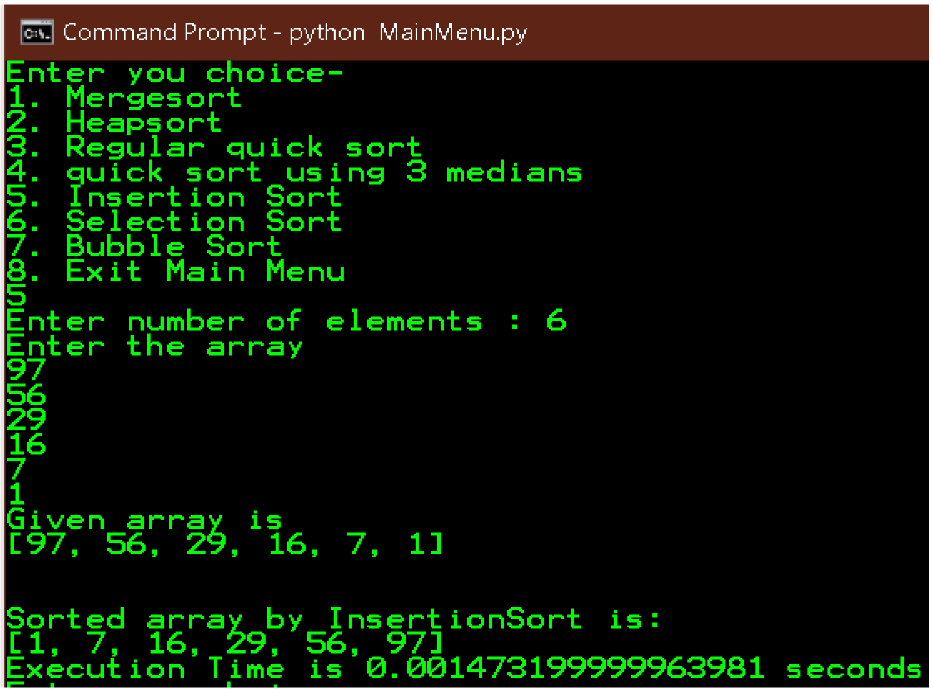
- The input is an unsorted array.
- Find the correct position of single element in the list by comparing it to all other elements.
- Move on to other elements after allotting one single element in its correct position and repeat this procedure till all elements in array are sorted.



Time Complexity: O(n) for best case, O(n^2) for average and worst case.

Space Complexity: O(1)

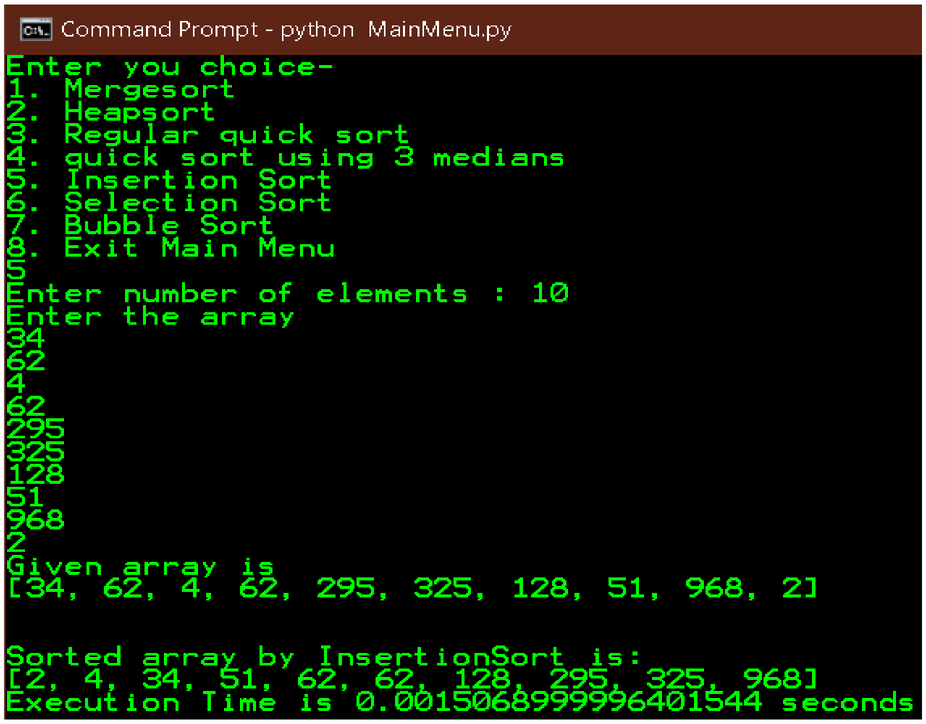
<u>Input 1:</u>



```
Enter you choice-
1. Mergesort
2. Heapsort
3. Regular quick sort
4. quick sort using 3 medians
5. Insertion Sort
6. Selection Sort
7. Bubble Sort
8. Exit Main Menu
5
Enter number of elements : 6
Enter the array
97
56
29
16
7
1
Given array is
[97, 56, 29, 16, 7, 1]


Sorted array by InsertionSort is:
[1, 7, 16, 29, 56, 97]
Execution Time is 0.0014731999999963981 seconds
```

Input array is [97, 56, 29, 16, 7, 1], execution time: 0.001473199 seconds.

<u>Input 2:</u>



```
Enter you choice-
1. Mergesort
2. Heapsort
3. Regular quick sort
4. quick sort using 3 medians
5. Insertion Sort
6. Selection Sort
7. Bubble Sort
8. Exit Main Menu
5
Enter number of elements : 10
Enter the array
34
62
4
62
295
325
128
51
968
2
Given array is
[34, 62, 4, 62, 295, 325, 128, 51, 968, 2]


Sorted array by InsertionSort is:
[2, 4, 34, 51, 62, 62, 128, 295, 325, 968]
Execution Time is 0.0015068999996401544 seconds
```

Input array is [34, 62, 4, 62, 295, 325, 128, 51, 968, 2], execution time: 0.001506899 seconds.

3.6 Selection Sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts:

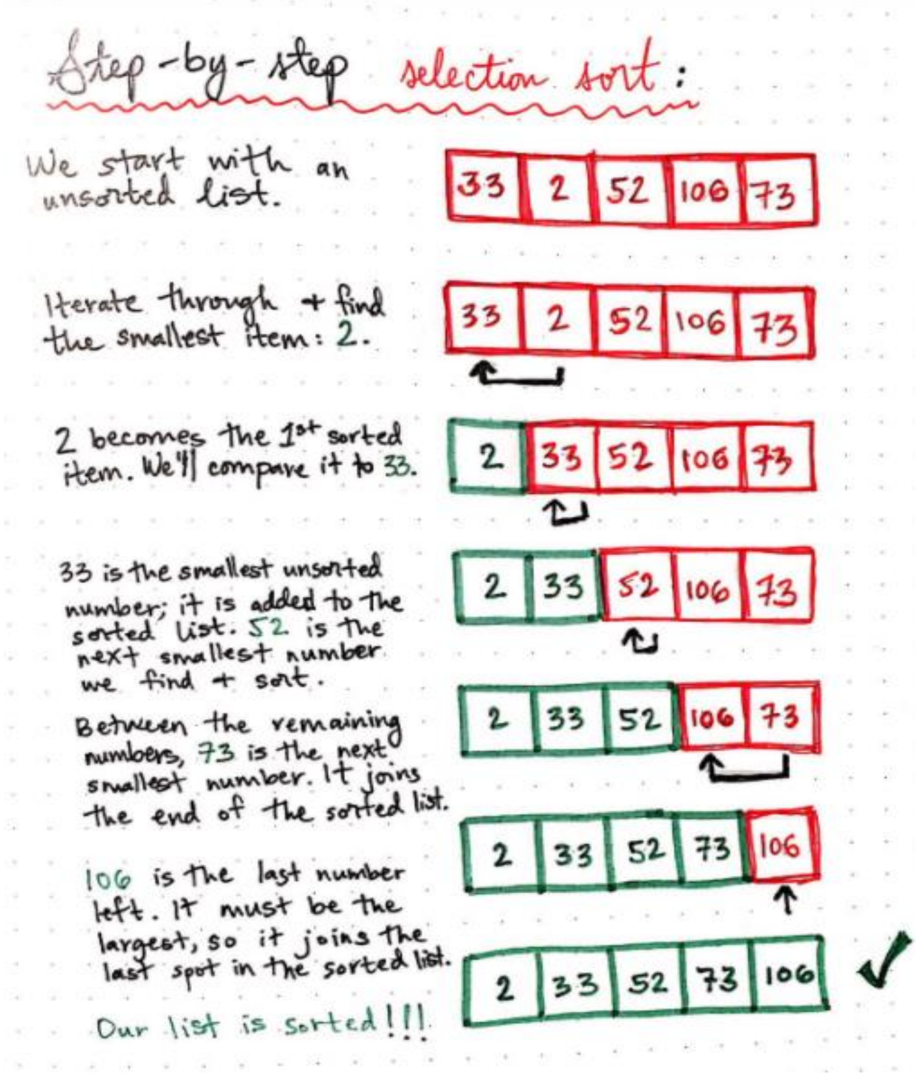
1. The subarray which is already sorted.
2. Remaining subarray which is unsorted.

Approach:

Initially, the sorted part is empty, and the unsorted part is the entire list.

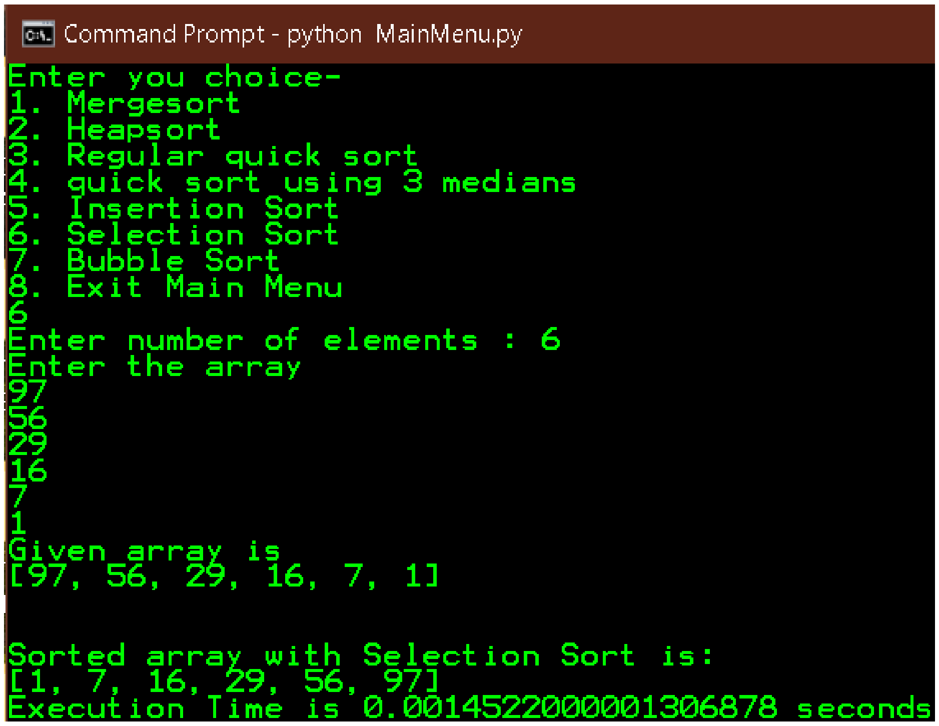
The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array.

This process continues moving unsorted array boundary by one element to the right.


Step-by-step selection sort:
We start with an unsorted list.
33 | 2 | 52 | 106 | 73
Iterate through + find the smallest item: 2.
33 | 2 | 52 | 106 | 73
2 becomes the 1st sorted item. We'll compare it to 33.
2 | 33 | 52 | 106 | 73
33 is the smallest unsorted number; it is added to the sorted list. 52 is the next smallest number we find + sort.
2 | 33 | 52 | 106 | 73
Between the remaining numbers, 73 is the next smallest number. It joins the end of the sorted list.
2 | 33 | 52 | 106 | 73
2 | 33 | 52 | 73 | 106
106 is the last number left. It must be the largest, so it joins the last spot in the sorted list.
2 | 33 | 52 | 73 | 106
Our list is sorted!!!

Time Complexity: O(n^2) for all cases **Space Complexity:** O(1)

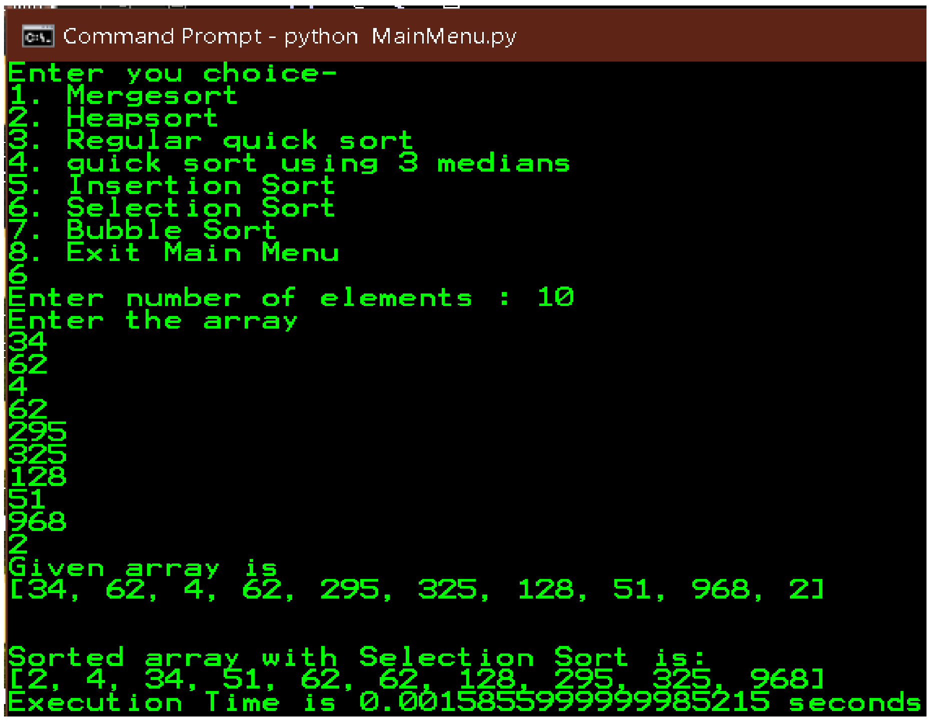
Input 1:



```
Command Prompt - python  MainMenu.py
Enter you choice-
1. Mergesort
2. Heapsort
3. Regular quick sort
4. quick sort using 3 medians
5. Insertion Sort
6. Selection Sort
7. Bubble Sort
8. Exit Main Menu
6
Enter number of elements : 6
Enter the array
97
56
29
16
7
1
Given array is
[97, 56, 29, 16, 7, 1]


Sorted array with Selection Sort is:
[1, 7, 16, 29, 56, 97]
Execution Time is 0.0014522000001306878 seconds
```

Input array is [97, 56, 29, 16, 7, 1], execution time: 0.00145220 seconds.

Input 2:



```
Command Prompt - python  MainMenu.py
Enter you choice-
1. Mergesort
2. Heapsort
3. Regular quick sort
4. quick sort using 3 medians
5. Insertion Sort
6. Selection Sort
7. Bubble Sort
8. Exit Main Menu
6
Enter number of elements : 10
Enter the array
34
62
4
62
295
325
128
51
968
2
Given array is
[34, 62, 4, 62, 295, 325, 128, 51, 968, 2]


Sorted array with Selection Sort is:
[2, 4, 34, 51, 62, 62, 128, 295, 325, 968]
Execution Time is 0.0015855999999985215 seconds
```

Input array is [34, 62, 4, 62, 295, 325, 128, 51, 968, 2], execution time: 0.001585599 seconds.

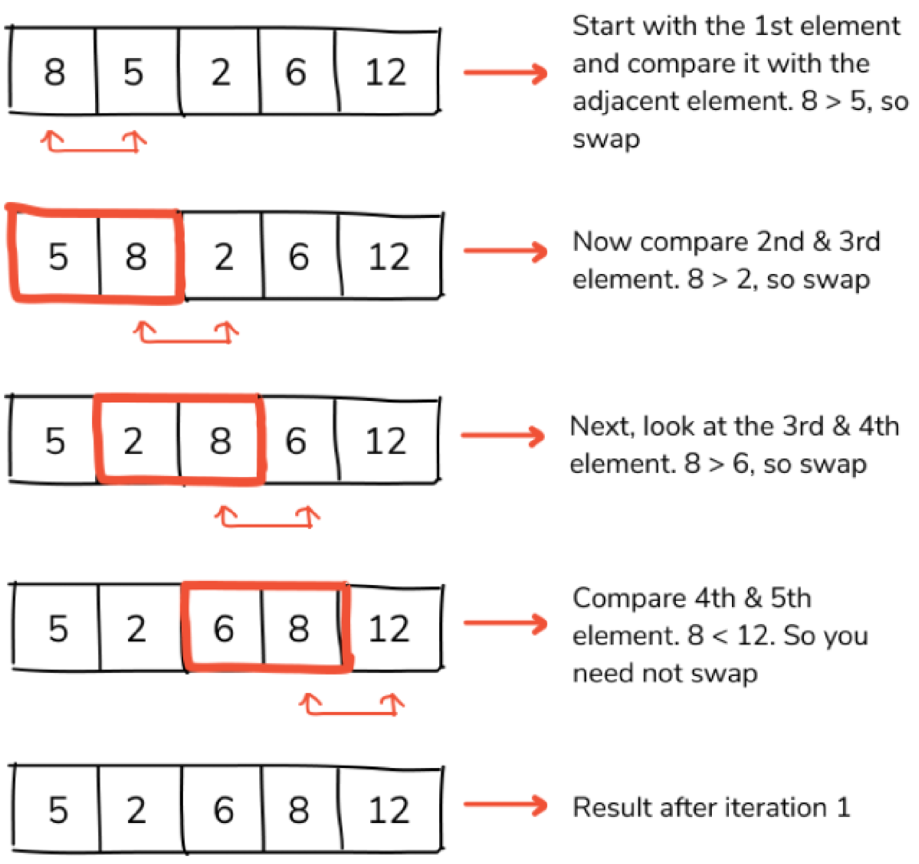
3.7 Bubble Sort

Bubble sort also knows as sinking sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in wrong order.

Approach:

Assume that A=[] is an unsorted array of elements. This array needs to be sorted in ascending order. The pseudo code is as follows:

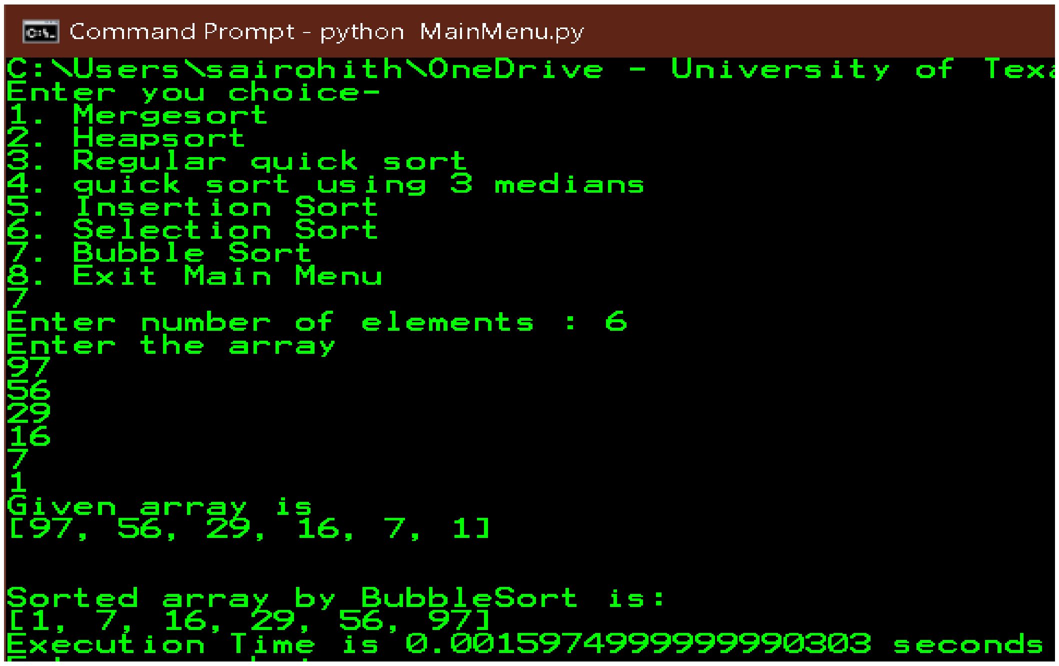
```
BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list
```


8 5 2 6 12
Start with the 1st element and compare it with the adjacent element. 8 > 5, so swap
5 8 2 6 12
Now compare 2nd & 3rd element. 8 > 2, so swap
5 2 8 6 12
Next, look at the 3rd & 4th element. 8 > 6, so swap
5 2 6 8 12
Compare 4th & 5th element. 8 < 12. So you need not swap
5 2 6 8 12
Result after iteration 1

Time Complexity: O(n) for best case, O(n^2) for average and worst case.

Space Complexity: O(1)

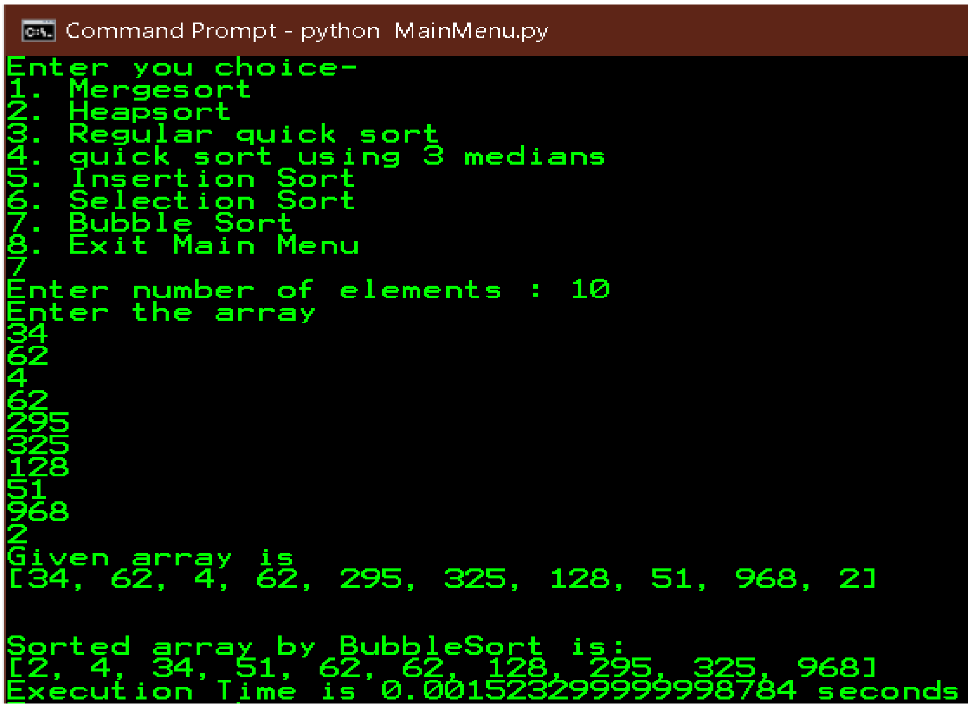
Input1:



```
Command Prompt - python  MainMenu.py
C:\Users\sairohith\OneDrive - University of Texa
Enter you choice-
1. Mergesort
2. Heapsort
3. Regular quick sort
4. quick sort using 3 medians
5. Insertion Sort
6. Selection Sort
7. Bubble Sort
8. Exit Main Menu
7
Enter number of elements : 6
Enter the array
97
56
29
16
7
1
Given array is
[97, 56, 29, 16, 7, 1]


Sorted array by BubbleSort is:
[1, 7, 16, 29, 56, 97]
Execution Time is 0.0015974999999990303 seconds
```

Input array is [97, 56, 29, 16, 7, 1], execution time: 0.001597499 seconds.

Input 2:



```
Command Prompt - python  MainMenu.py
Enter you choice-
1. Mergesort
2. Heapsort
3. Regular quick sort
4. quick sort using 3 medians
5. Insertion Sort
6. Selection Sort
7. Bubble Sort
8. Exit Main Menu
7
Enter number of elements : 10
Enter the array
34
62
4
62
295
325
128
51
968
2
Given array is
[34, 62, 4, 62, 295, 325, 128, 51, 968, 2]


Sorted array by BubbleSort is:
[2, 4, 34, 51, 62, 62, 128, 295, 325, 968]
Execution Time is 0.0015232999999998784 seconds
```

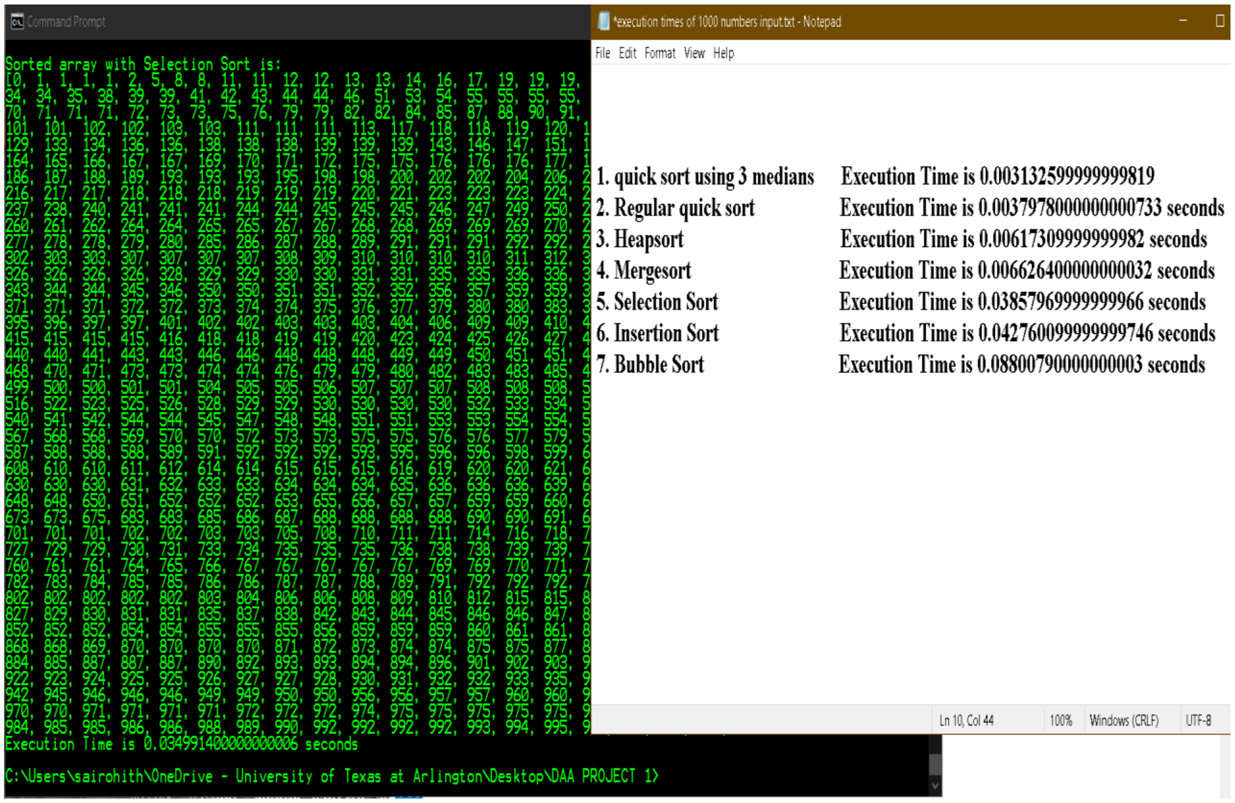
Input array is [34, 62, 4, 62, 295, 325, 128, 51, 968, 2], execution time: 0.001523299 seconds.

4. Comparison between algorithms

Algorithm	Execution time (seconds)	Worst case Execution time(seconds)	Ranking based on execution speed	Space Complexity
Merge Sort	0.00127820	0.00127820	**#2**	**O(n)**
Heap Sort	0.001222699	0.001222699	**#1**	**O(1)**
Quick Sort(reg)	0.00127390	0.001384200	**#4**	**O(log n)**
Quick Sort(3-median)	0.0012873	0.0012873	**#3**	**O(log n)**
Insertion Sort	0.001506899	0.001506899	**#5**	**O(1)**
Selection Sort	0.001585599	0.001585599	**#6**	**O(1)**
Bubble Sort	0.001523299	0.001597499	**#7**	**O(1)**

Comparison of sorting algorithms using 1000 random numbers as input:

Algorithm	Execution time (seconds)	Ranking based on execution speed	Space Complexity
Merge Sort	0.0066264000	**#4**	**O(n)**
Heap Sort	0.00617309999999	**#3**	**O(1)**
Quick Sort(reg)	0.00379780000000	**#2**	**O(log n)**
Quick Sort(3-median)	0.00313259999999	**#1**	**O(log n)**
Insertion Sort	0.0427600999	**#6**	**O(1)**
Selection Sort	0.0385796999	**#5**	**O(1)**
Bubble Sort	0.08800790000	**#7**	**O(1)**


Command Prompt
*execution times of 1000 numbers input.txt - Notepad
File Edit Format View Help
Ln 10, Col 44
100%
Windows (CRLF)
UTF-8

```
Sorted array with Selection Sort is:
[0, 1, 1, 1, 1, 2, 5, 8, 8, 11, 11, 12, 12, 13, 13, 14, 16, 17, 19, 19, 19,
34, 34, 35, 38, 39, 39, 41, 42, 43, 44, 44, 46, 51, 53, 54, 55, 55, 55, 55,
70, 71, 71, 71, 72, 73, 73, 75, 76, 79, 79, 82, 82, 84, 85, 87, 88, 90, 91,
101, 101, 102, 102, 103, 103, 111, 111, 111, 113, 117, 118, 118, 119, 120, 1
129, 133, 134, 136, 136, 138, 138, 138, 139, 139, 139, 143, 146, 147, 151, 1
164, 165, 166, 167, 167, 169, 170, 171, 172, 175, 175, 176, 176, 176, 177, 1
186, 187, 188, 189, 193, 193, 193, 195, 198, 198, 200, 202, 202, 204, 206, 2
216, 217, 217, 218, 218, 218, 219, 219, 219, 220, 221, 223, 223, 223, 224, 2
237, 238, 240, 241, 241, 241, 244, 244, 245, 245, 245, 246, 247, 249, 250, 2
260, 261, 262, 264, 264, 265, 265, 267, 267, 268, 268, 269, 269, 269, 270, 2
277, 278, 278, 279, 280, 285, 286, 287, 288, 289, 291, 291, 291, 292, 292, 2
302, 303, 303, 307, 307, 307, 307, 308, 309, 310, 310, 310, 310, 311, 312, 3
326, 326, 326, 326, 328, 329, 329, 330, 330, 331, 331, 335, 335, 336, 336, 3
343, 344, 344, 345, 346, 350, 350, 351, 351, 352, 352, 356, 357, 359, 359, 3
371, 371, 371, 372, 372, 373, 374, 374, 375, 376, 377, 379, 380, 380, 383, 3
395, 396, 397, 397, 401, 402, 402, 403, 403, 403, 404, 406, 409, 409, 410, 4
415, 415, 415, 415, 416, 418, 418, 419, 419, 420, 423, 424, 425, 426, 427, 4
440, 440, 441, 443, 443, 446, 446, 448, 448, 448, 449, 449, 450, 451, 451, 4
468, 470, 471, 473, 473, 474, 474, 476, 479, 479, 480, 482, 483, 483, 485, 4
499, 500, 500, 501, 501, 504, 505, 505, 506, 507, 507, 507, 508, 508, 508, 5
516, 522, 523, 525, 526, 528, 529, 529, 530, 530, 530, 530, 532, 533, 534, 5
540, 541, 542, 544, 544, 545, 547, 548, 548, 551, 551, 553, 553, 554, 554, 5
567, 568, 568, 569, 570, 570, 572, 573, 573, 575, 575, 576, 576, 577, 579, 5
587, 588, 588, 588, 589, 591, 592, 592, 592, 593, 595, 596, 596, 598, 599, 6
608, 610, 610, 611, 612, 614, 614, 615, 615, 615, 616, 619, 620, 620, 621, 6
630, 630, 630, 631, 632, 633, 633, 634, 634, 634, 635, 636, 636, 636, 639, 6
648, 648, 650, 651, 652, 652, 652, 653, 655, 656, 657, 657, 659, 659, 660, 6
673, 673, 675, 683, 683, 685, 686, 687, 688, 688, 688, 688, 690, 690, 691, 6
701, 701, 701, 702, 702, 703, 703, 705, 708, 710, 711, 711, 714, 716, 718, 7
727, 729, 729, 730, 731, 733, 734, 735, 735, 735, 736, 738, 738, 739, 739, 7
760, 761, 761, 764, 765, 766, 767, 767, 767, 767, 767, 769, 769, 770, 771, 7
782, 783, 784, 785, 785, 786, 786, 787, 787, 788, 789, 791, 792, 792, 792, 7
802, 802, 802, 802, 802, 803, 804, 806, 806, 808, 809, 810, 812, 815, 815, 8
827, 829, 830, 831, 831, 835, 837, 838, 842, 843, 844, 845, 846, 846, 847, 8
852, 852, 852, 854, 854, 855, 855, 855, 856, 859, 859, 859, 860, 861, 861, 8
868, 868, 869, 870, 870, 870, 870, 871, 872, 873, 874, 874, 875, 875, 877, 8
884, 885, 887, 887, 887, 890, 892, 893, 893, 894, 894, 896, 901, 902, 903, 9
922, 923, 924, 925, 925, 926, 927, 927, 928, 930, 931, 932, 932, 933, 935, 9
942, 945, 946, 946, 946, 949, 949, 950, 950, 956, 956, 957, 957, 960, 960, 9
970, 970, 971, 971, 971, 971, 972, 972, 972, 974, 975, 975, 975, 975, 975, 9
984, 985, 985, 986, 986, 988, 989, 990, 992, 992, 992, 992, 993, 994, 995, 9
Execution Time is 0.03499140000000006 seconds

C:\Users\sairohith\OneDrive - University of Texas at Arlington\Desktop\DAA PROJECT 1>
```

```
1. quick sort using 3 medians     Execution Time is 0.0031325999999999819
2. Regular quick sort             Execution Time is 0.0037978000000000733 seconds
3. Heapsort                       Execution Time is 0.0061730999999999982 seconds
4. Mergesort                      Execution Time is 0.0066264000000000032 seconds
5. Selection Sort                 Execution Time is 0.0385796999999999966 seconds
6. Insertion Sort                 Execution Time is 0.0427600999999999746 seconds
7. Bubble Sort                    Execution Time is 0.0880079000000000003 seconds
```

- From above 2 tables we can observe that **Merge sort, Heap Sort, and quicksort 3-median is slow** when **the input array is very small,** and they are **very fast when input is large.**

- It can be easily observed that the **quicksort 3 median** is the **fastest algorithm** when the **input is very large**, and **quicksort** is the **fastest algorithm** when **input is small** when compared to other algorithms.

- Knowing which algorithm is best possible depends heavily on details of the application and implementation. In most practical situations, **quicksort** is a popular algorithm for sorting large input arrays because its expected running time is O(n log n). It also outperforms heap sort in practice.

- **Heap Sort, Selection Sort and Quick Sort** are **not so stable** because their **execution time varies** with **change of order (worst case gives high execution times)** of elements with the same value.

- **Merge Sort** occupies “extra space” and “memory” when it divides the list into sub lists , thereby space complexity for **small input(best case) is O(n log n)** and Space Complexity is **O(n) for larger inputs.**