EMPRICAL ANALYSIS ON SEVERAL SORTING ALGORITHMS

	Dept	Name Surname	Contribution
1	CSE	MÜSLİM YILMAZ	Insertion Sort Algorithm and report, partial selection sort algorithm and report, partial heapsort and report, quick-select Algorithm and report, Quick Select Algorithm and Report.
2	CSE	EVREN KOYMAT	Merge Sort Algorithm and report, Quick Sort Algorithm and report, Quick Select Algorithm report.

PURPOSE

Aim of this project is comparing several algorithms for the given selection problem. Comparison will be made by empirical experiment based on their time complexities.

Algorithms:

- (1) Sort all elements with insertion-sort and return k'th element.
- (2) Sort all elements with merge-sort and return k'th element.
- (3) Sort all elements with Quick-sort and return k'th element
- (4) Partial Selection-sort
- (5) Partial Heapsort
- (6) Quick-select (not sorted list)
- (7) Quick-select (median-off-three pivot selection)

Note that 1-2 and 3rd algorithms are presorted selection algorithms.

EXPERIMENT

- **Algorithm (1): Sorting all elements with insertion-sort and return k'th element**.
    1. ***Idea*:** First sort all elements with insertion than select the k'th smallest element.
    2. ***Empirical experiment explanation*:** We will do the comparison by counting how many times the basic operation is executed. Since after sorting the elements, finding k'th smallest elements takes just constant time, we will just analyze regular insertion sort.
    3. ***Sample Inputs:*** We will use inputs with different sizes and different characteristic as:
       Characteristic: Random distribution, ascending order distribution, descending order distribution
       Size variation: Input with 1000-5000-10000-50000-100000 elements
    4. ***Best Case:*** For the best case of insertion sort, only requires one comparison per iteration and algorithm never go into while loop. This is the case where list is already sorted. Addition to that, number of comparisons is calculated as: $\boldsymbol{N-1}$, where N is the number of elements.

Table 1.1

Input Size	# Comparisons (Theoretically)	#Comparisons (Calculated)
1000	999	999
5000	4999	4999
10000	9999	9999
50000	49999	49999
10000	99999	99999

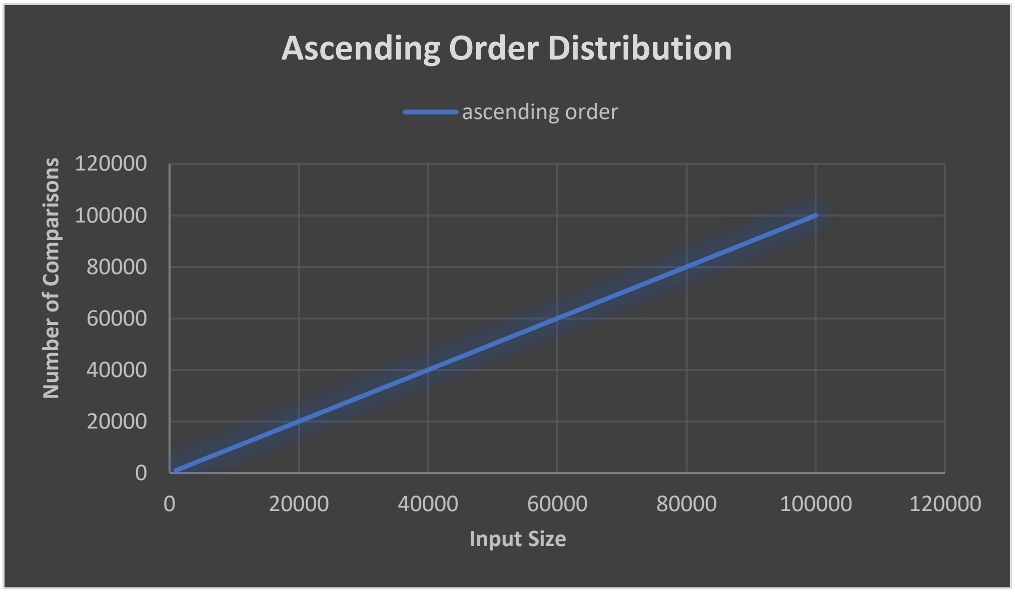


Figure 1.a

- Figure 1.a agree that; in best case scenario, the relationship between number of comparisons and input size is linear.

5. ***Worst Case:*** Worst case of insertion sort is the case where list is reverse sorted. Algorithm always go into while loop in every iteration. Addition to that, theoretically number of comparisons is: $\frac{1}{2}(\mathbf{N}^2 - \mathbf{N})$, where N is the number of elements in the list.

Input Size	# Comparisons (Theoretically)	#Comparisons (Calculated)
1000	499500	500495
5000	12497500	12502360
10000	49995000	50004521
50000	1249975000	1250012547
10000	4999950000	5000000041

Table 1.2

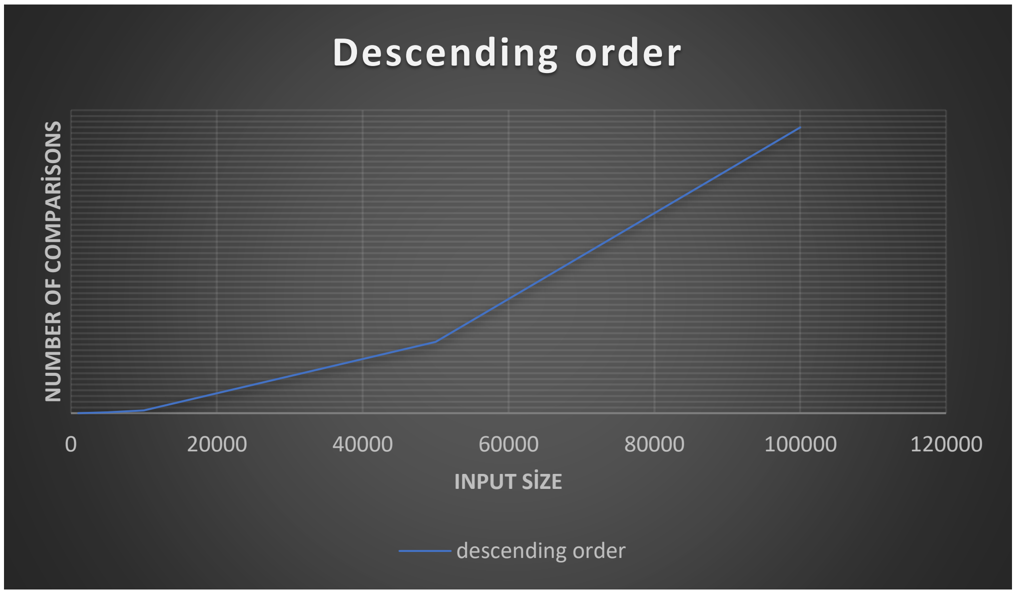


Figure 1.b

- Figure 1. b agrees that; in worst case scenario, the relationship between number of comparisons and input size is quadratic. This would happen when list is reverse sorted. We use descending order distribution to observe this.

6. Average Case: We will analyze the average case for insertion sort by using random input distribution. Addition to that, number of comparisons is approximately: $\frac{1}{4}(\mathbf{N^2} - \mathbf{N})$, where N is the number of elements in the list.

Table 1.3

Input Size	# Comparisons (Theoretically)	#Comparisons (Calculated)
1000	249750	254594
5000	6248750	6172648
10000	24997500	24909694
50000	624987500	626717362
100000	2499975000	2495833058

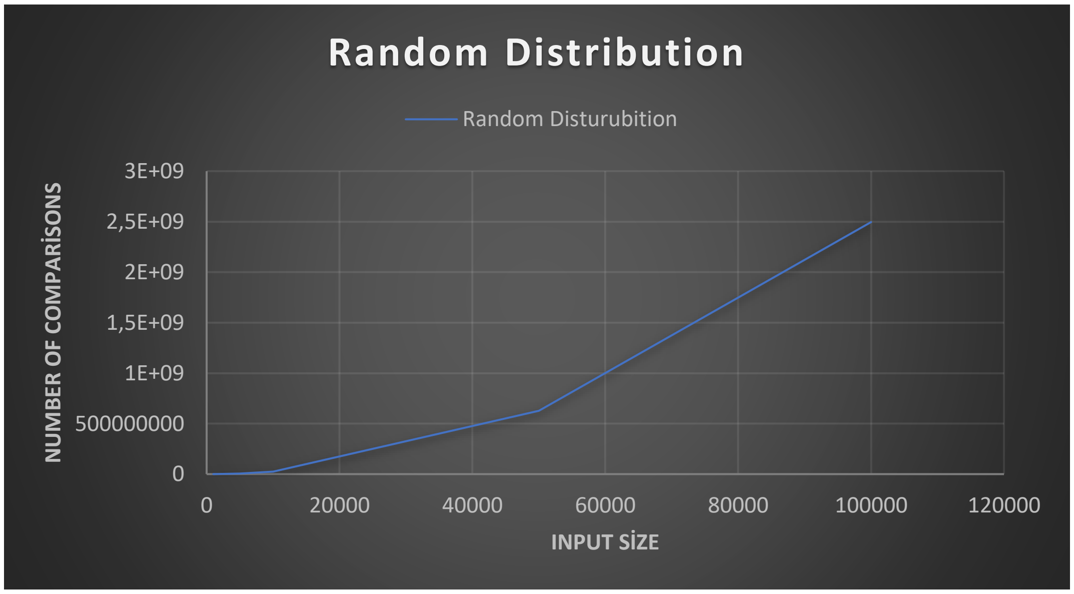


Figure 1.c

- Figure 1. c agrees that; in average case scenario, the relationship between number of comparisons and input size is again quadratic; but the comparison made by algorithm for random input type, most likely less than the reverse order input type. (Differences can be seen in table 1.2 and 1.3)

7. Result and conclusion: In this experiment we observe that insertion sort best case is linear whereas average and worst case is quadratic. It means that when we have sorted list or almost sorted list and acceptable input size; it makes sense to use algorithm (1) due to linear relationships. However, when we use reverse order input type in algorithm, the function becomes quadratic (Figure1.b) and it is not good idea to use insertion sort when you have reverse order sorted list.

There is figure in below showing that number of comparisons made by algorithm with different characteristic of inputs.

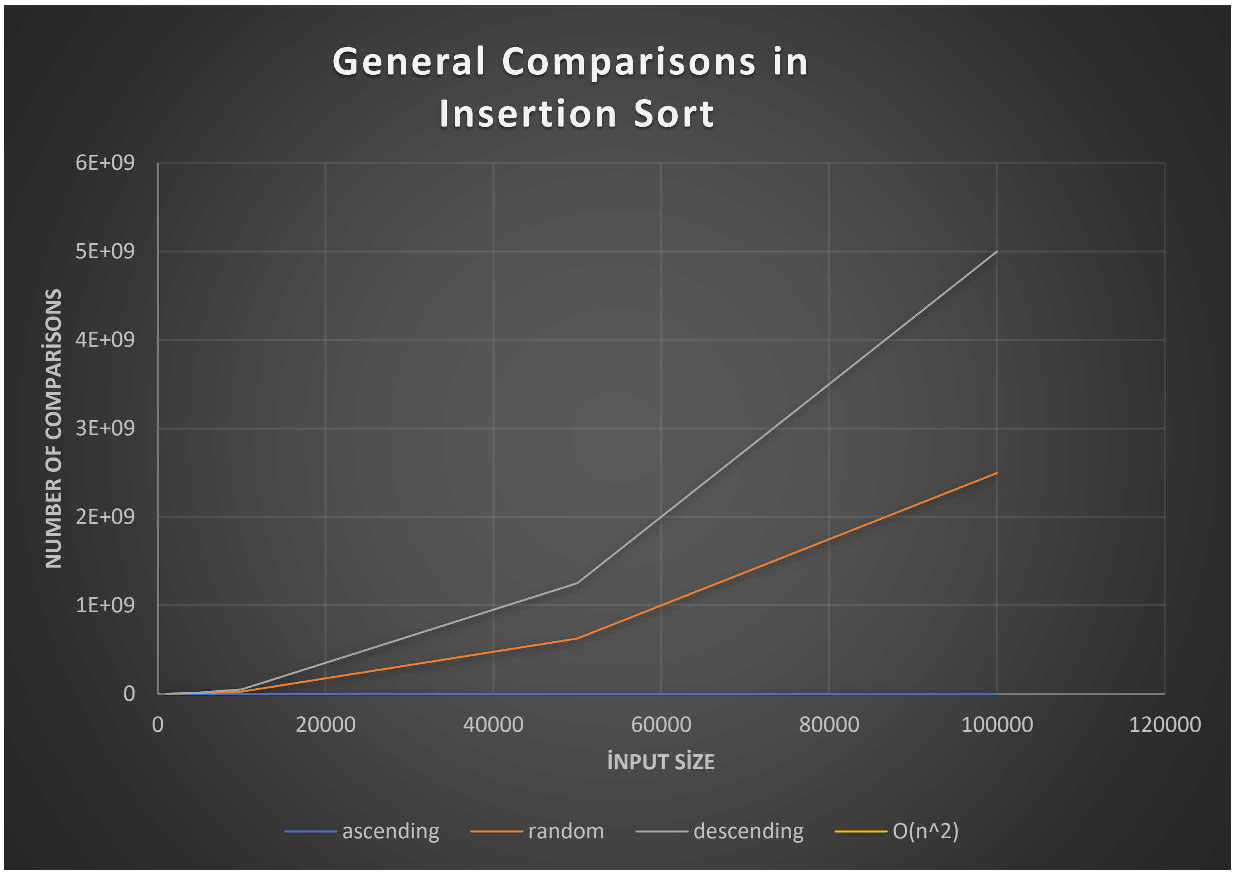


Figure 1.d

- <u>**Algorithm (2):**</u> **Sorting all elements with merge-sort and return k'th element.**
  1. ***Idea:*** First divide the array until reach a single element, then merge the single elements according to ascending order. Then select the k'th smallest element.
  2. ***Empirical experiment explanation:*** Like sorting algorithm, we <u>count the number of comparisons</u> when merging the numbers. Finding k'th smallest elements takes constant time so we only check the comparison count.
  3. ***Sample Inputs*:** For best case we select the ordered lists (ascending, descending).
     For average case we select the list that sorted half and randomly distributed.
     For worst case, we select the list that divided half of it randomly even and other half is randomly odd. So, every number need to check with all elements when comparing.
  4. ***Best Case:*** If the given array is sorted, that ensures, for ascending order, the left subarray's last item is always lower than right subarray's first item. So, there is only one comparison in each merge step. That reduces the number of comparisons in each merge step to N/2, and the best-case number of comparison formula is:
     $0.5\ x\ N * \log_2 N$ where N is the number of elements and log base is 2.

Input Size	# Of Comparisons (Theoretically)	#of Comparisons Descending Order (Calculated)	# Of Comparisons Ascending Order (Calculated)
1000	4982	5044	4932
5000	30719	32004	29888
10000	66438	69008	64903
50000	390241	401952	390003
100000	830482	853904	842909

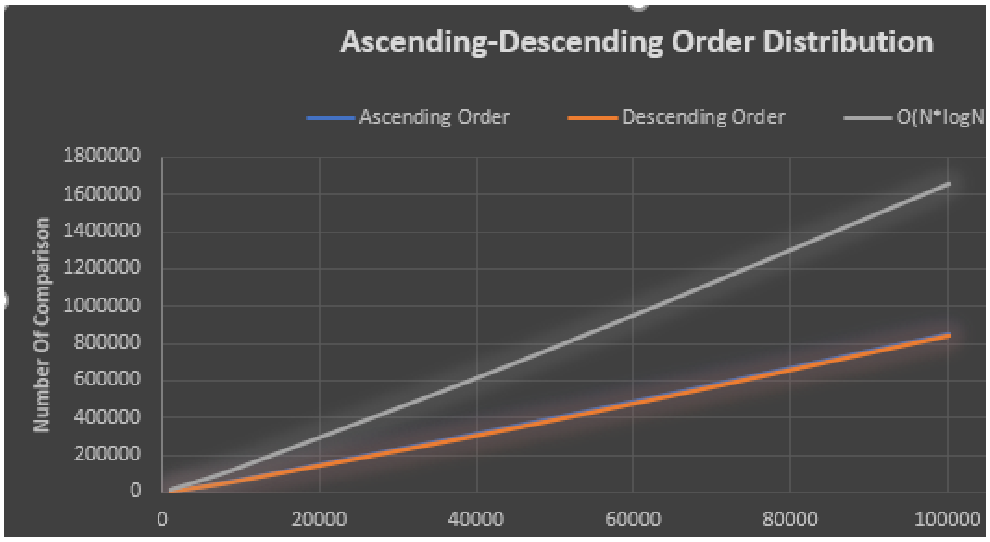


Figure 2.a

In Figure 2.a, we see that descending order makes less comparison than ascending order.

5. ***Worst Case:*** The given list's elements are randomly and half of it even, the other half is odd. We need to check all the elements before merging and sorting. The number of comparisons is **N * logN**.

Input Size	# of Comparisons (Theoretically)	# of Comparisons (Calculated)
1000	9964	8692
5000	61438	55283
10000	132876	120488
50000	780482	717909
100000	1660964	1536633

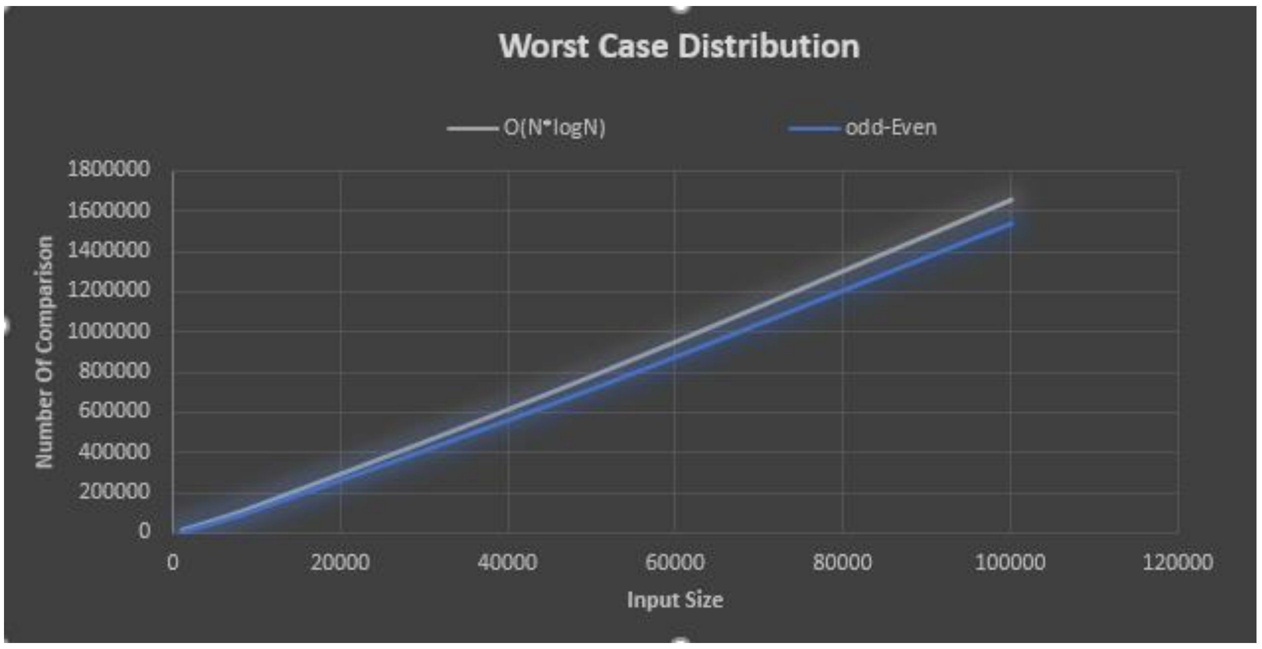


Figure 2.b

*In Figure 2.b, worst case's comparison close to O(N*logN) time complexity.*

6. ***Average Case:*** When the half of the list is ordered and the other half is randomly separated, this is the chosen case for average case. Number of comparison is **0.74*N*logN** (where N is the number of numbers, log base is 2).

Input Size	# of Comparisons (Theoretically)	# of Comparisons (Calculated)
1000	7373	6745
5000	45464	44860
10000	98328	97234
50000	577556	572543
100000	1229113	1220115

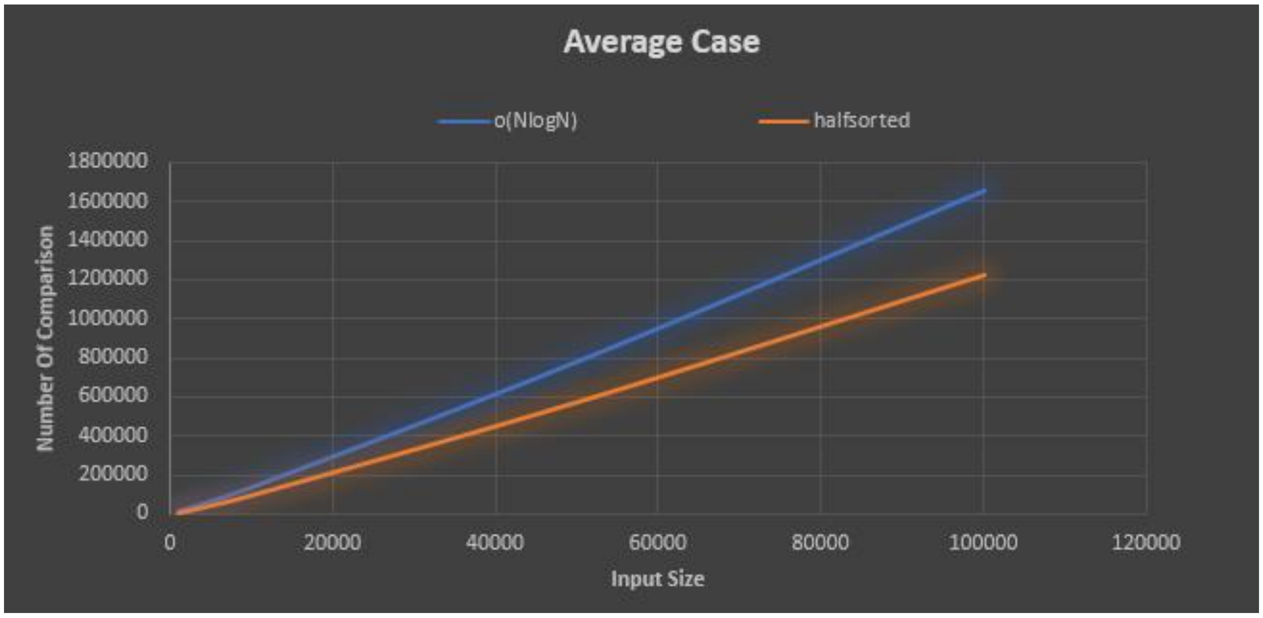


Figure 2.c

7. ***Result and Conclusion***: Merge sort have the same time complexity **(O(NLogN**) for all three cases. The number of comparisons for best case **is 0.5*N*logN**, for average case is **0.74*N*logN** (where N is # of numbers, log base is 2). In our given input lists ensures that for best and average cases. For worst case # of comparison is **O(NlogN).** Our input list is close but not equal to **N*logN.**

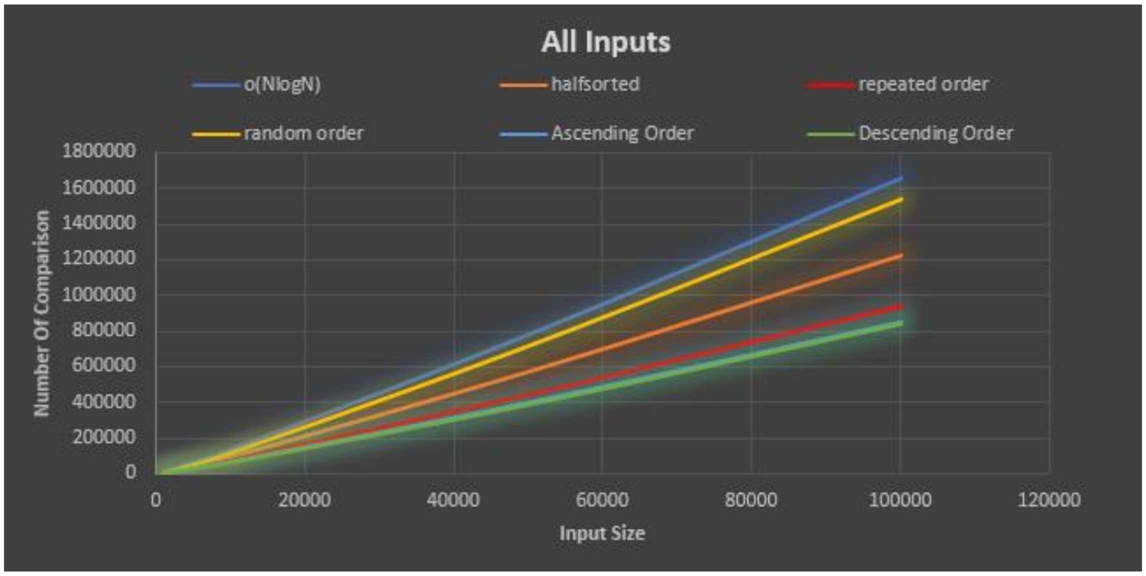


Figure 2.d

In Figure 2.d, we see the best, worst and average case together. The merge sort is efficient for worst case because it has O(NlogN) time complexity. But we can't say same thing for space complexity.

Algorithm (3): Sort all elements by Quick-sort and return the k'th element in the list. While partitioning chooses the pivot element as the first element in an array.

1. **Idea:** : Select the first element of array as a pivot. According to pivot , sort the list with Quick Sort Algorithm.
2. **Empirical experiment explanation**: Quick Sort algorithm states that, In every end of iteration, the pivot element settles in the right place. The left part of Pivot indexes is less than pivot value, and the right part is greater than pivot value. When we select the first element as a pivot, we are <u>dealing with worst time complexity</u> with Quick sort and It is **<u>O(N ^ 2) (</u>**where N is the # of numbers). Number of comparison is nearly **<u>N ^ 2 / 2</u>**. We are counting number of comparison according to pivot values.
3. **Sample Inputs:** We have 4 different inputs. Ascending order inputs, Descending order inputs, randomly separated inputs, and repeated number inputs.
4. **Case Analyses:** In the below there is table for number of comparisons done by algorithm with difference sizes and characteristic.

5. **Result and Conclusion:** As a result, even if we deal with the worst case of Quick Sort algorithm, the input is determining the # of comparisons. As we seen in the table, repeated and randomize inputs comparisons are much lower than ascending & descending orders. Because in ascending & descending order inputs, every number should be check. But repeated & randomize inputs, sometime the numbers settle already in the right place. That's why there is a huge difference.

Input Sizes	Time Complexity (o(N^2))	# Of comparisons theoretically (n^2/2)	# of comparisons of repeated input	# of comparisons of randomize input	# of comparisons of ascending order input	# of comparisons of descending order input
1000	1000000	500000	38638	14341	501494	500587
5000	25000000	12500000	893198	87877	12494773	12308232
10000	100000000	50000000	3536398	193442	49920414	Overflow
50000	2500000000	1250000000	87402699	1151013	Overflow	Overflow
100000	10000000000	5000000000	349869336	2459994	Overflow	overflow

Table 3.1

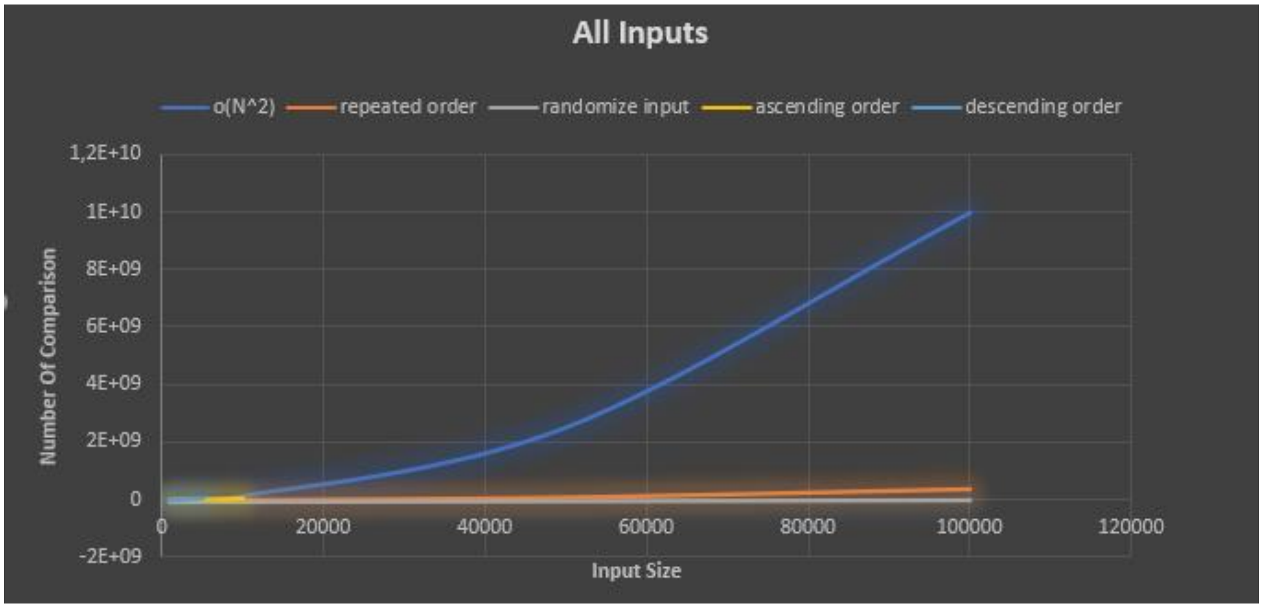


Figure 3.1

Now we will compare above three algorithms to observe which input type is suitable and acceptable for which algorithm.

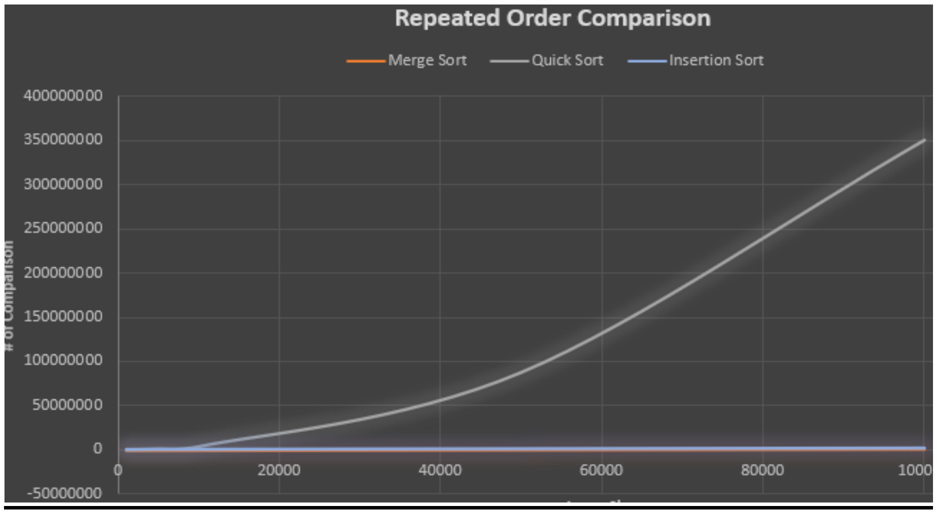
Comparison of Insertion, Merge and Quick-Sort Algorithms:

Algorithms: Insertion sort, Merge Sort and Quick Sort

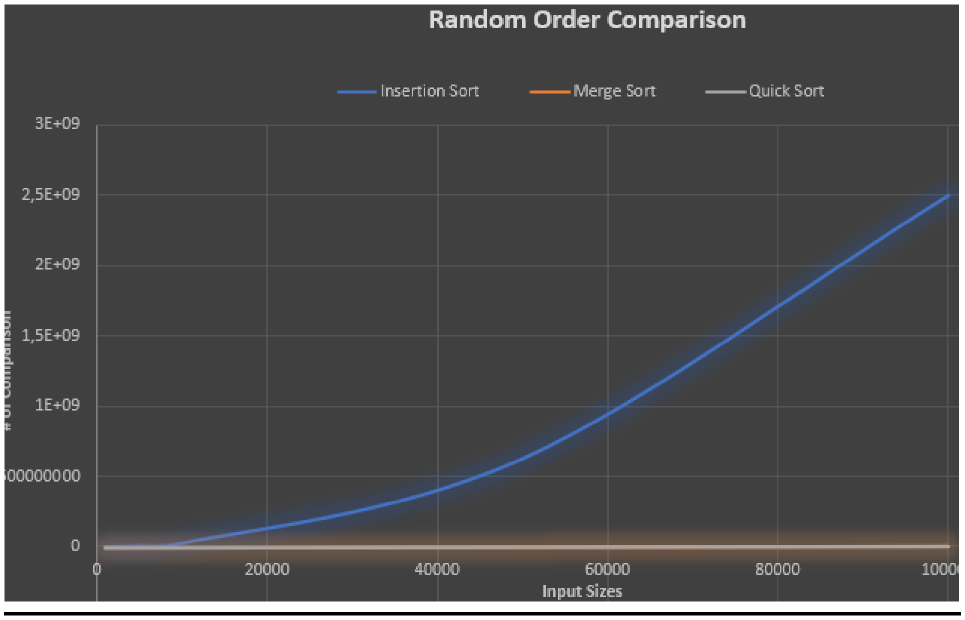
Input sizes: 1000-5000-10000-50000-100000

Input characteristic: Random order, Descending order, Ascending order, Repeated order, half-sorted order, half-even half-odd order

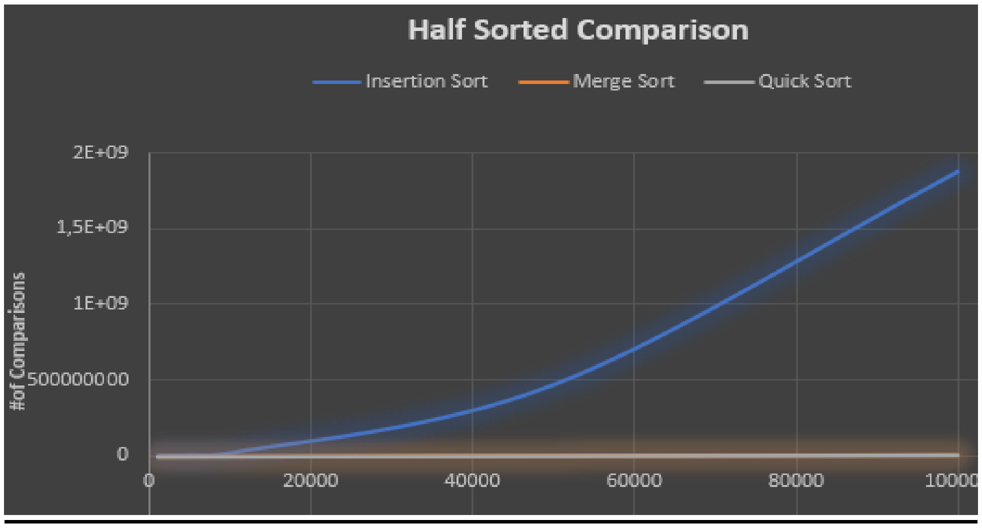
- **Note:** We compare first 3 sorting due to other sorting algorithms includes recurrences about selection k times.



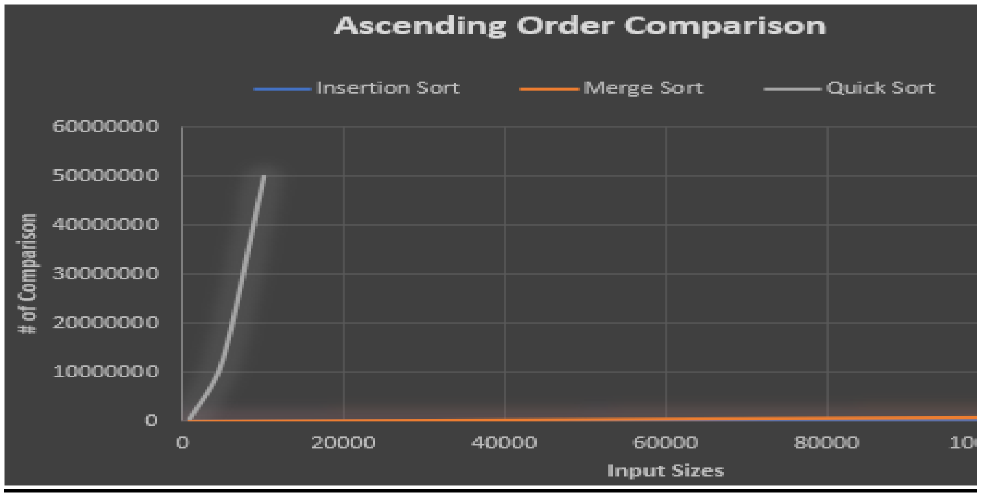
- In repeated order input type, we see that Quick-Sort Algorithm is worst algorithm for sorting because when the # of comparisons are increase, our time complexity is also increase. If time complexity is increase, we sort slower. Merge sort compares less than insertion sort. Thus, if we select merge sort algorithm, the # of comparisons smaller, and the time complexity is less than others.



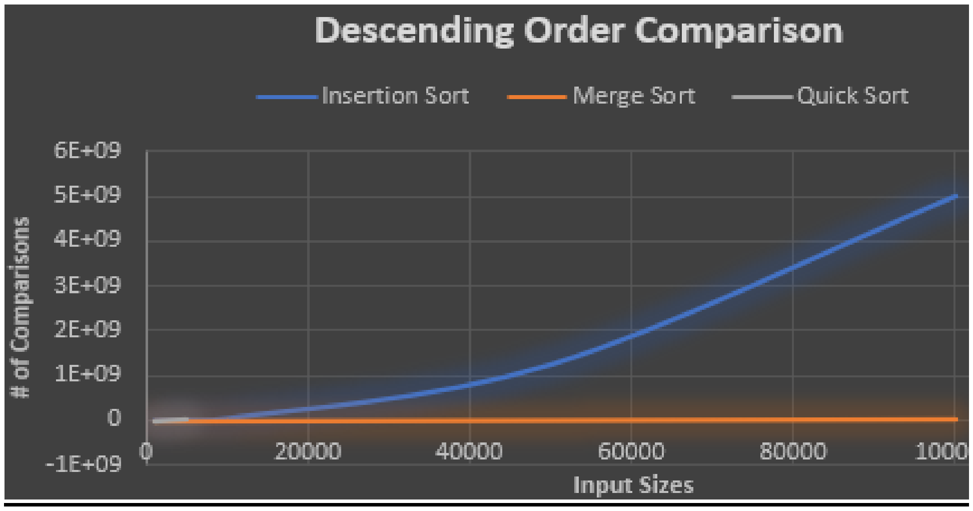
- In random order input type, we see that Insertion Sort Algorithm is worst algorithm because of # comparisons. Merge sort compares less than Quick sort. Thus, if we select merge sort algorithm, the # of comparisons smaller and the time complexity is less than others.



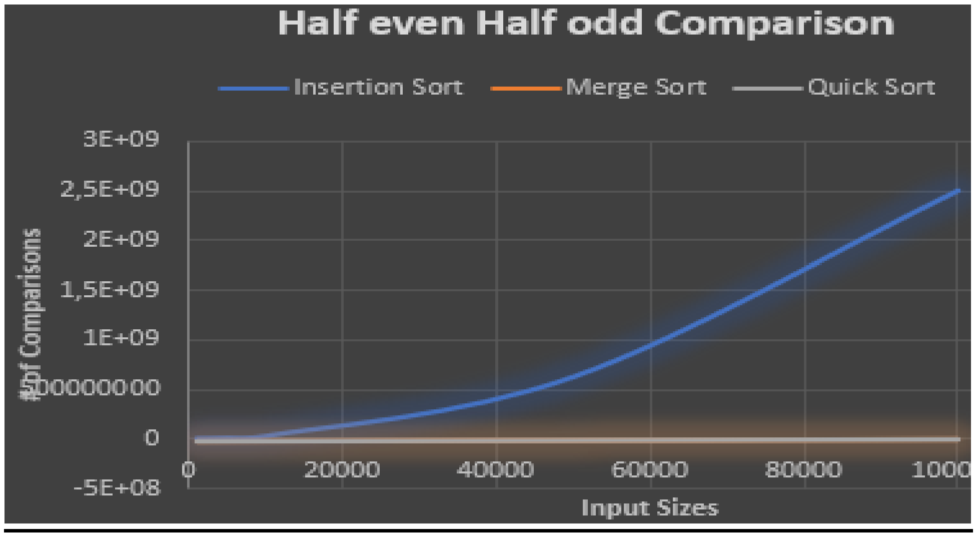
- In Half-sorted input type, we see that Insertion Sort Algorithm is worst algorithm. Because it has more comparisons than others. Merge sort compares less than Quick Sort. Thus, if we select merge sort algorithm, the # of comparisons smaller and the time complexity is less than others.



- In ascending order input type, Quick Sort Algorithm is worst and over 10k input sizes, it overflows. On the other hand, Insertion Sort compares less than Merge Sort. Thus, if we select Insertion sort, the # of comparisons smaller and the time complexity is less than others.



- In descending order input type, Quick Sort again overflows over 5k input size. The worst algorithm for this input type is going to be Quick Sort or Insertion sort. Both algorithms are comparing a lot. We should choose the Merge Sort algorithm for best time complexity.



- In half even & half odd input type, Insertion Sort is worst Algorithm because the # of comparisons bigger than other sort algorithms. The Merge Sort Algorithm is best algorithm for this input type. Because in merge sort, # of comparisons less than Quick and Insertion Sort.

Algorithm (4): Partial Selection-Sort; finding the minimum element *k* times to find the *k*'th smallest element.

1. ***Idea***: Just applying regular selection sort. Only difference is algorithm is not sorting all inputs; instead, it sorts first k elements where k is the parameter of function.
2. ***Empirical experiment explanation***: We will do the comparison by counting how many times the basic operation is executed. The importing think is we are trying to find minimum element k times to find k'th smallest element.
3. ***Sample Inputs:*** We will use inputs with different sizes and different characteristic as:
   Characteristic: Random distribution, ascending order distribution, descending order distribution
   Size variation: Input with 1000-5000-10000-50000-100000 elements
   Different k values: k= input size * (1/4) -> (25th percentile of list)
   k= input size * (1/2) -> (50th percentile of list)
   k= input size * (3/4) -> (75th smallest element in the list)
   k= input size (largest number in list)

4. ***Case analyzes***:

First, we will observe whether the characteristics of the inputs affect the count number. For this purpose, we will analyze 3 different input characteristics (ascending order, descending order, and random distribution) with same input size.

Table for input characteristic of random, ascending order and descending order distribution *(Table4.1)*

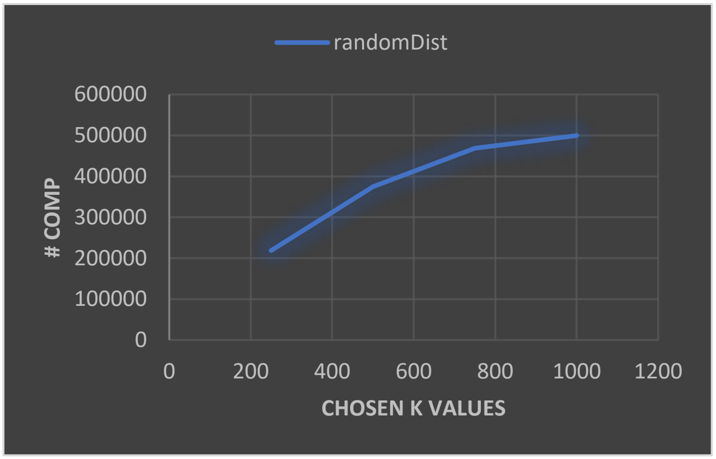
Value of k	Number of Comparison	O(k*N)
250	218625	250000
500	374750	500000
750	468375	750000
1000	499500 *	1000000

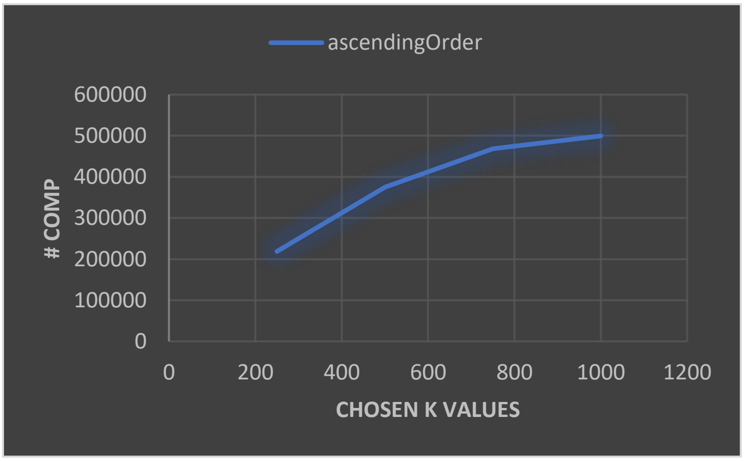
For the case k = N where N is the input size the number of comparisons made by algorithm is theoretically equals to:

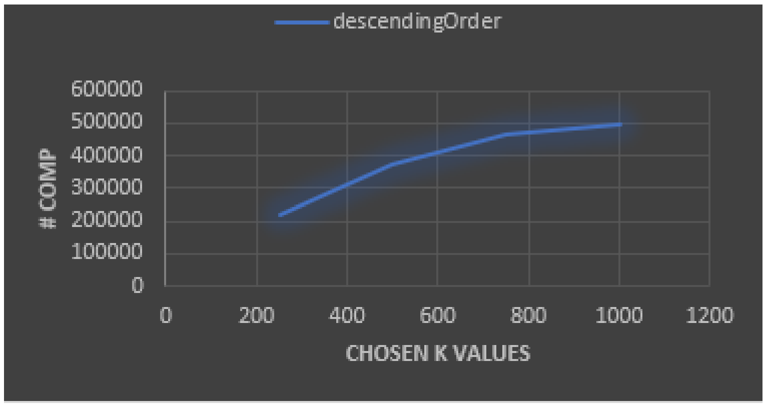
$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \frac{(n-1)(n)}{2}; \quad C(1000)$$

$$= 999 * \frac{1000}{2} = 499500$$

which is the same with table (*) value.







Figures 2.c

- Figures 2.c and table 2.a conclude that the characteristic of the inputs does not affect how many times the basic operation will execute. It makes sense due to algorithm doesn't know where the smallest element in the list. So, regardless of the characteristic of inputs, algorithm will search the minimum key at the end of the list.

Now, we will analyze how the algorithm behaves against increase in the number of inputs. To do that we will keep k constant against the increase in inputs. (For simplicity we always search for the middle element which is 50th percentile of list.)

Input Size	Number of Comparison
1000	374750
5000	9373750
10000	37497500
50000	937487500
100000	3749975000

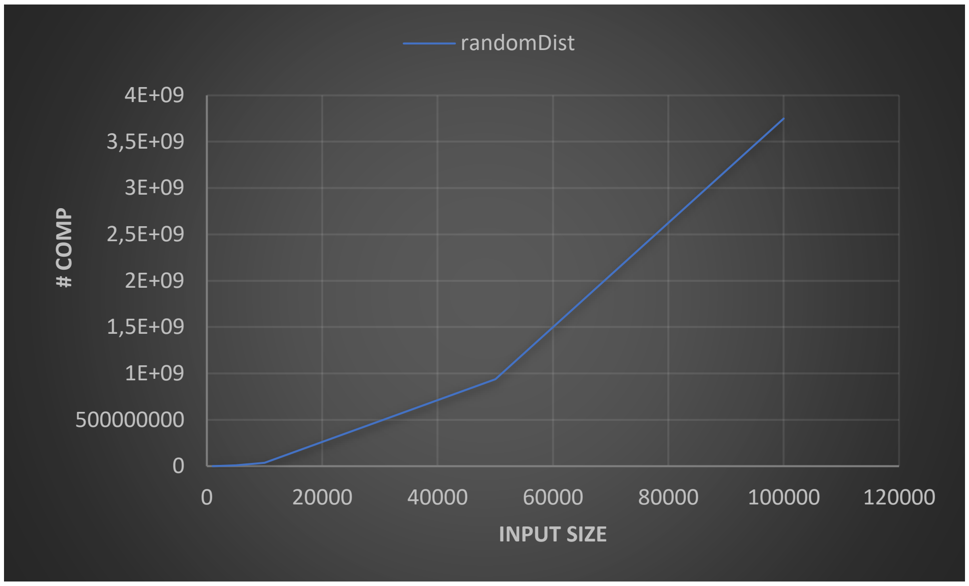


Figure 1.c

- Figure 1.c shows that the relationship between input size and number of comparisons made by algorithm is linear and never exceeds k*N where k is the smallest k'th element and N is the total input size.

5. Result and conclusion: First we observe that, changing of characteristic of inputs does not affect algorithm. After that, for same value of k, the relationship between input size and number of comparisons done by algorithm is linear and it upper bound is O(n*k). Lastly, we observe that the increase in k value also increases the number of comparisons done by algorithm.

We conclude that, this specific algorithm runs better when k is small (see in Figure 1.d) and is easy to implement.

Algorithm (5): Partial Heapsort; storing all elements in a max-heap and apply n-k times max removal. At the end, return the max element in the root

1. ***Idea*:** We use max heap property to find kth smallest element in the list. To do that, we will use well-known operation in max heap which is remove max operation. Each iteration we remove the max key which located in root and rearrange the heap. We do this process N-K times to find kth smallest element in the list.

2. ***Empirical experiment explanation***: We will do the comparison by counting how many times the basic operation is executed. Key point of this algorithm is, we just sort until k'th smallest element. Rest is kept unsorted.
3. ***Sample Inputs:*** We will use inputs with different sizes and different characteristic as:
   Characteristic: Random distribution, ascending order distribution, descending order distribution and input with exactly same values.
   Size variation: Input with 100-200-300-400-500 elements
   Different k values: k= input size * (1/4) -> (25th percentile of list)
   k= input size * (1/2) -> (50th percentile of list)
   k= input size * (3/4) -> (75th percentile of list)
   k= input size (largest number in list)
4. ***Case analyzes***: First we analyze the number of comparisons in chosen inputs with same k value to find how to algorithm response the increasing in the input size. We chose arbitrary k as always middle element to list which is the 50th percentile of the list. We will choose random input distribution to observe this:

Input Size	Number of Comparison	O(N + N*longN)
100	677	764
200	1544	1728
300	1878	2768
400	2196	3857
500	2554	4982

Table 5.1

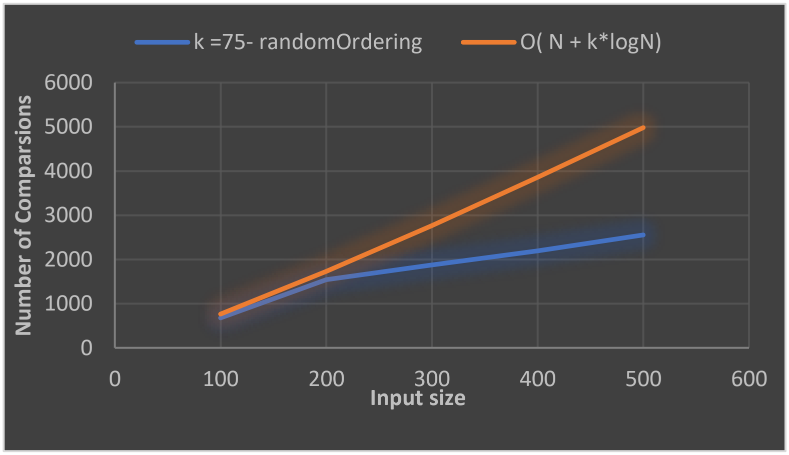


Figure 6.1

- Figure 6.1 indicates that the relationship between input size and number of comparisons in partial heap sort algorithm close the linear but bigger than linear. Since we use small input values in our experiment, function behavior may not clearly observe in figure 6.1. To compensate this, we use table 5.1 to prove that function in line N*logN.

- As theoretically we know that, when we search for the biggest element in the list which is exactly regular heap sort (not partial); time complexity of this algorithm is **N * log N**. It means that when we doubled the input size, time complexity must be increase **N * log N times:**

  Input size 100- > Theoretically: $100\log(100) = 200$
  Input size 200- > Theoretically: $200\log(200) = 460$

  460 / 200 = 2.3, which means that it should be equal number of comparisons done by algorithm with input size 100 over number of comparisons done by algorithm with input size 200 in table 5.1.

  Input size 100- > Calculated: 764
  Input size 200- > Calculated: 1728

  1728 / 764 = 2.26 ~ 2.3, it is proven that algorithm fit in N*logN line.

Now we will analyze that what is the algorithm response different characteristic type of inputs. To clear observe, we always search for 25th percentile of input size.

Ascending order

k	size	#comp
25	100	424
50	200	942
75	300	1504
100	400	2082
125	500	2668

Descending order

k	size	#comp
25	100	214
50	200	1366
75	300	2144
100	400	2930
125	500	3674

Random

k	size	#comp
25	100	214
50	200	1200
75	300	1878
100	400	2590
125	500	3394

Repeated value

k	size	#comp
25	100	150
50	200	300
75	300	450
100	400	600
125	500	750

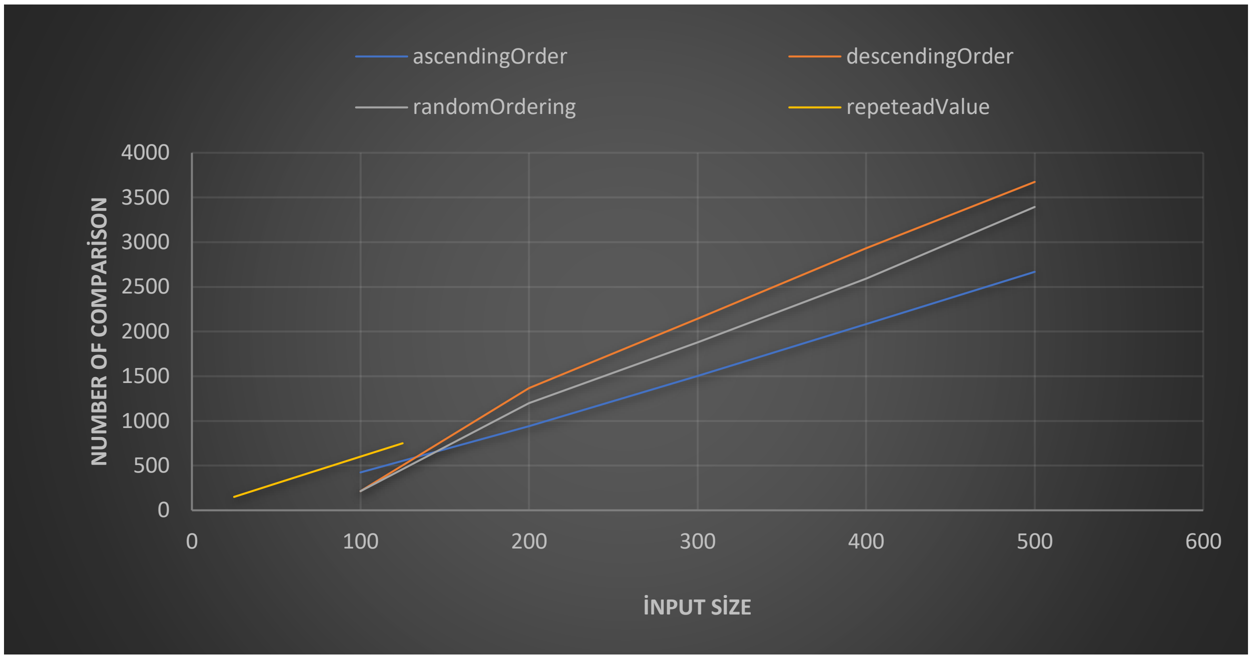


Figure 6.2

- Figure 6.2 conclude that best case for partial sorting algorithm is the input type which is include repeated values. Algorithm works very fast when input is included repeated values or mostly included repeated values. Also, we observe that, ascending order distribution works better than descending order distribution. As we expect, average case is between ascending and descending order distribution which we represented as random order distribution.

5. Result and conclusion: Above algorithm, we use binary max-heap to find kth smallest element is the list. We observe that the upper bound of partial heap sort is:

- each insert operation takes O (logk) time and finding in heap size N; total (N*logk). Also, we must consider the build heap time which is O(N). So total: 0(N+klogn).

The algorithms works well when we have input such that includes many repeated values. Another advantage of the algorithm is that the algorithm does not sort the entire list unless the condition k = input size holds. It just sorts the list until we find kth smallest element in the list.

Algorithm (6): Quick-Select algorithm, which based on array partitioning (Pivot element chosen as the first element in an array.

1. ***Idea*:** The algorithm implementation is very similar to regular quicksort. There is slight difference is that: instead of sort all element in the list recursively, we just recur only one part which includes k'th smallest element.
2. ***Empirical experiment explanation*:** We will do the comparison by using physical unit of time. We will use nanosecond (10^-9 times second) to measure elapsed time.
3. ***Sample Inputs:*** We will use inputs with different sizes and different characteristic as:
   Characteristic: Random distribution, ascending order distribution, descending order distribution
   Size variation: Input with 100-200-300-400-500 elements.
   Different k values: k= input size * (1/4) -> (25th percentile of list)
   k= input size * (1/2) -> (50th percentile of list)
   k= input size * (3/4) -> (75th smallest element in the list)
   k= input size (largest number in list)
4. ***About the reliability of measurement with physical unit of time:*** Since we run this algorithm on the computer, depending on the hardware features of the computers, different computers may produce different results. To addition to that even different result can be observed in the same computer when we run algorithm several times.

   To minimize this error, we will be run algorithm several times on the same computer and data will be the average of each iteration.

   Here is the process:

```
iteration0 -> 65200 ns          iteration95 -> 105300 ns
iteration1 -> 270600 ns         iteration96 -> 102500 ns
iteration2 -> 574100 ns         iteration97 -> 102000 ns
iteration3 -> 905100 ns         iteration98 -> 81300 ns
iteration4 -> 188500 ns         iteration99 -> 101000 ns
```

->Total second: 14877300 ns
->Average: 148773 ns (last from of data)

5. ***Case analyzes***: First we analyze the elapsed time in chosen input type with same k value to find how to algorithm response the increasing in the input size We will choose random input distribution to observe this:

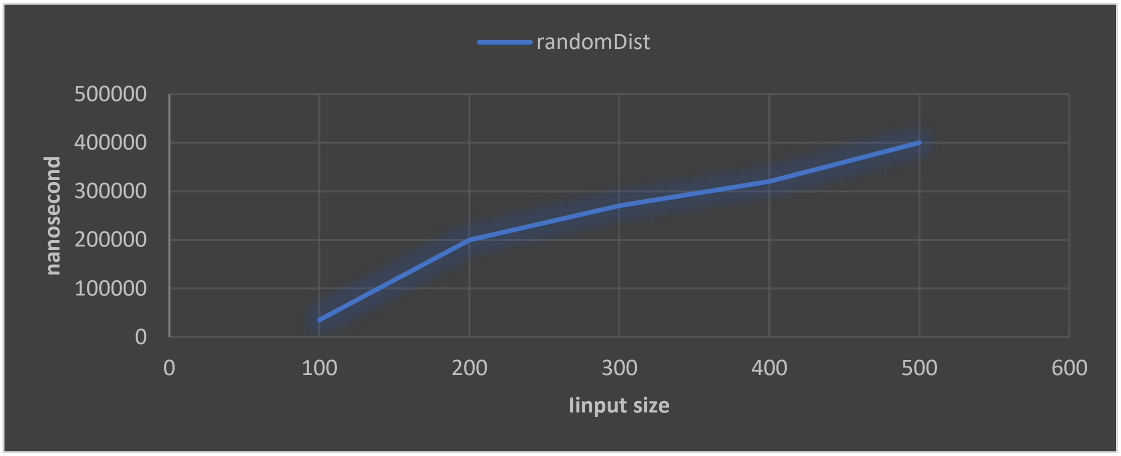


Figure 6.1

- Figure 6.1 conclude that for small input sizes the algorithm is may not sufficient but as input size grows, algorithm efficiency increases. Also, the function is in the same line with N*logN. It is close the linear but bigger than linear line.

Now we will analyze what is the algorithm response for different characteristic type of inputs. To clear observe, we always search for 50th percentile of input size.

Random distribution	input size	time
k= 50	100	62480
k=100	200	164322
k=150	300	215986
k=200	400	256999
k=250	500	312206

Ascending order distribution	input size	time
k= 50	100	59298
k=100	200	168608
k=150	300	221607
k=200	400	286485
k=250	500	341781

descending order Distribution

	input size	time
k= 50	100	39570
k=100	200	159169
k=150	300	213250
k=200	400	279942
k=250	500	346925

repeated number Distribution

	input size	time
k= 50	100	30257
k=100	200	146297
k=150	300	187588
k=200	400	224676
k=250	500	268334

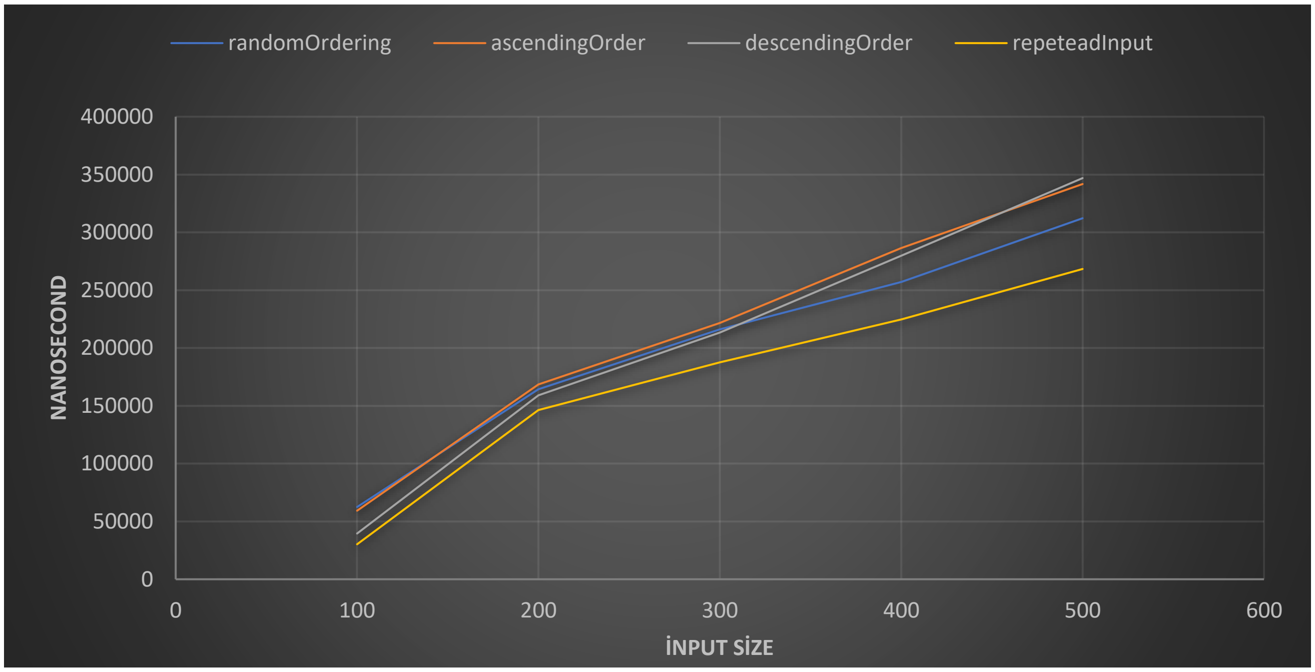


Figure 6.2

- Figure 6.2 conclude that the algorithm's response to inputs of different characters is approximately the same. From this point of view, we can say that the input characteristic does not have a significant effect on the algorithm. What speeds up or slows down the algorithm depends on the chosen pivot. Since always the first element is chosen as the pivot in this algorithm, the reaction of the algorithm is generally the same for different characteristic inputs.

6. ***Result and conclusion:*** In this algorithm logic is: we divide array two parts. If index is equal to the k, the algorithm is done we found the kth smallest element in the list, if index is less than k we do recursively solve for right part, if index is greater than k we do recursively solve for left part.

<u>Algorithm (7):</u> Apply quick select algorithm, but this time use median-of-three pivot selection.

1. ***Idea***: The median of three means, we look at the first, middle and last elements of the array and choose the median of those three elements as the pivot.
2. ***Empirical experiment explanation:*** We will do the comparisons by counting how many times the basic operation is executed. Since after sorting the elements, finding k'th smallest elements takes just constant time, we will just analyze regular insertion sort.
3. ***Sample Inputs:*** We will use inputs with different sizes and different characteristics as:
   <u>Characteristic:</u> Random distribution, ascending order & descending order distribution
   <u>Size variation:</u> 1000-5000-10000-50000-100000 elements

4. ***Case analyzes***:

RO Dist.

k	input size	#Comp
1000	1000	18527
5000	5000	82357
10000	10000	189613
50000	50000	1107976
100000	100000	2891590

AO Dist.

k	input size	#Comp
1000	1000	500409
5000	5000	12502499
10000	10000	50004999
50000	50000	overflow
100000	100000	overflow

DO Dist.

k	input size	#Comp
1000	1000	250507
5000	5000	6073416
10000	10000	23879610
50000	50000	overflow
100000	100000	overflow

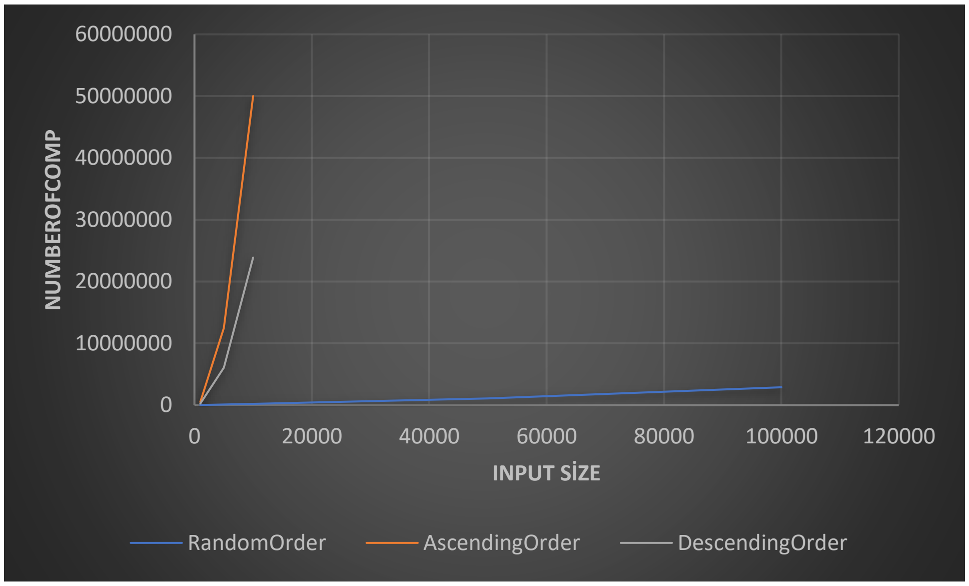


Figure 7.1

- Figure 7.1 conclude that for random ordering inputs, median of three quick-select algorithm is the best algorithm among six algorithms above. However, we can't say same think for ascending and descending order.
-

5. Result and conclusion: This algorithm based on regular quick select algorithm. The difference is pivot element index. We chose as pivot median of first middle and last index. Worst case is O(n^2) and average case time complexity is O(n) as we can see in figure 7.1

REFERENCES

IMPORTANT NOTE: Since these algorithms are well-known algorithms in Computer Science, the source code of these algorithms has been taken by reference from various internet sites and has been modified to be necessary for the experiment.

Merge Sort Algorithm:

https://www.studytonight.com/data-structures/merge-sort

https://www.geeksforgeeks.org/merge-sort/

https://iq.opengenus.org/time-complexity-of-merge-sort/

https://iq.opengenus.org/merge-sort/

https://cs.fit.edu/~pkc/classes/writing/hw13/luis.pdf

https://www.baeldung.com/cs/merge-sort-time-complexity#:~:text=It%20occurs%20when%20the%20left,merge%20operations%20have%20alternate%20elements.

Quick Select Algorithm:

https://stackoverflow.com/questions/6740183/quicksort-with-first-element-as-pivot-example

https://www.bigocheatsheet.com/

https://www.javatpoint.com/quick-sort#:~:text=The%20best%2Dcase%20time%20complexity,O(n*logn).

https://www.softwaretestinghelp.com/quicksort-in-java/#:~:text=Quicksort%20uses%20a%20divide%2Dand,not%20be%20equal%20in%20size.

Insertion Sort:

https://www.geeksforgeeks.org/insertion-sort/#:~:text=Insertion%20sort%20is%20a%20simple,Algorithm

http://watson.latech.edu/book/algorithms/algorithmsSorting2.html

Partial Selection Sort:

https://www.geeksforgeeks.org/selection-algorithms/#:~:text=Partial%20selection%20sort,time%20to%20sort%20the%20array.

https://en.wikipedia.org/wiki/Partial_sorting

https://www.programiz.com/dsa/selection-sort

https://runestone.academy/ns/books/published/pythonds/SortSearch/TheSelectionSort.html

https://www.softwaretestinghelp.com/selection-sort-java/

Partial Heap Sort:

https://en.wikipedia.org/wiki/Partial_sorting#:~:text=Offline%20problems-,Heap%2Dbased%20solution,and%20remove%20the%20largest%20element.

https://stackoverflow.com/questions/24407555/partial-sorting-to-find-the-kth-largest-smallest-elements

https://www.geeksforgeeks.org/heap-sort-for-decreasing-order-using-min-heap/

Quick Select:

https://www.geeksforgeeks.org/quickselect-algorithm/

https://en.wikipedia.org/wiki/Quickselect

https://www.techiedelight.com/quickselect-algorithm/

Median of 3:

https://gist.github.com/epomp447/4c0d0676d9f013788647cbe6e60ae0df