Analysis Of Algorithms
Homework 1

Berk Yıldız 150117052
Esraa ismail taha turky 150119529

A. Purpose :

Purpose of our project is to design the different sorting algorithms and meusers the running time for each one of them and see if our results meet the theoretical assumption . Also at the end we will compare between the different algorithms and decide which one has better time complexity .

Those algorithms are :

1. Insertion-sort,
2. Binary Insertion-sort,
3. Merge-sort,
4. Quick-sort (pivot is always selected as the first element),
5. Quick-sort with median-of-three pivot selection
6. Heap-sort,
7. Counting-sort.

B. Input and Metrics:

We used three different combinations of inputs which are sorted inputs, reversed order sorted inputs, and random inputs.we also provided six different sizes which are 250, 500, 1000,2000, 8000 .the range of the numbers inside the arrays is from 0 to size-1 . In order to analyze the complexity of these algorithms, we used a physical unit of time in nanoseconds and counted the actual number of basic operations.

C.Illustrating and Analyzing Results:

We make definitions of each sorting type and provide their time complexities for different cases. We prepare a group of tables and graphs from the information we get from our program. These tables and graphs are about how long an algorithm takes to sort the list. Also we prepare another group of tables and graphs that about operation counts. We make our comments about these tables and graphs by comparing them with theoretical values.

1. Insertion Sort

Main principle in insertion sort is progressing by putting the input element on the place where it belongs and lowering the comparison on each loop. Input element will be compared with the next element and if it is bigger they are swapped and now will be compared with the next element. It will go like this until there is no more smaller element than our input element.
Time Complexities:

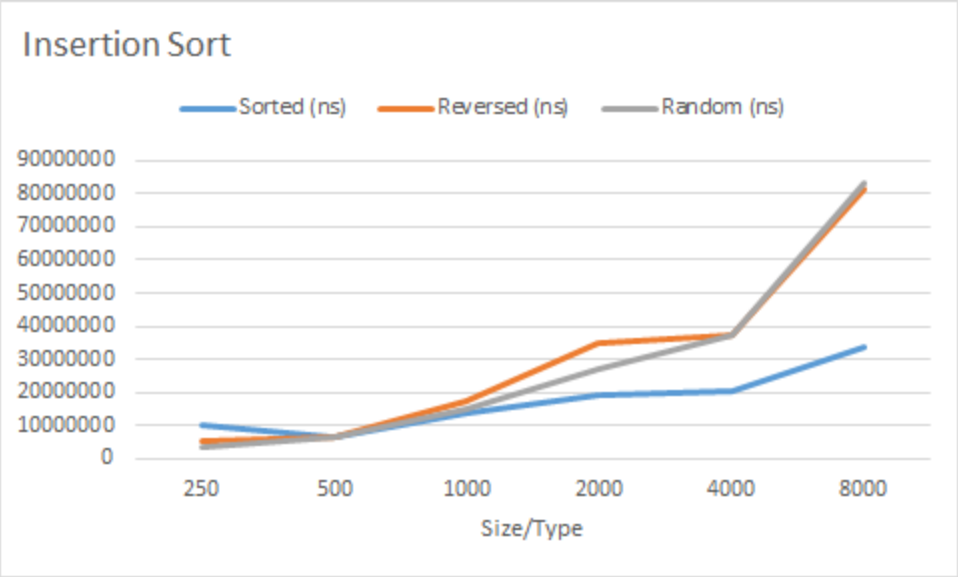
Best Case: If our list is already a sorted one, then our algorithm will only compare elements with each other. There will be no swap operation. With an array length n, total comparison will be n-1 and O(n) time complexity.

Worst Case: If our list is sorted reversely, this is the worst case. Because now we have to make n-1 comparisons and n-1 swap operations. Total operation is n(n-1)/2 and time complexity is $O(n^2)$.

Average Case: Each element is about halfway in order.For every position in the array on an average half the elements are greater than the number in that position, and hence they will contribute to a comparison. So unlike the worst case we will do half of its computation. Result will be n(n-1)/4 and time complexity is going to be the same $O(n^2)$.

Time:

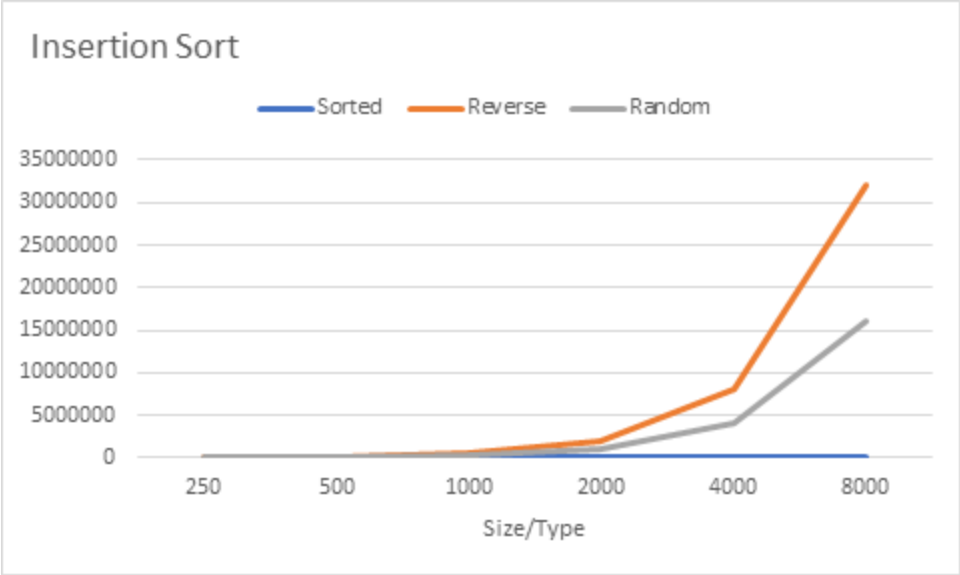
Size/Type	Sorted (ns)	Reversed (ns)	Random (ns)
250	9906199	5278200	3439500
500	6370000	6411800	6398100
1000	13837000	17547200	15196200
2000	19265901	34731800	27017799
4000	20301601	37351399	37304499
8000	33916900	81668000	83234000



As expected from theory, sorted type is the best case.Mostly reversed type is the worst case as expected. For high sizes our experiment got similar results for random and reversed lists.

Count of Basic Operation:

Size/Type	Sorted	Reversed	Random
250	**250**	**31375**	**15068**
500	**500**	**125250**	**63695**
1000	**1000**	**500500**	**254533**
2000	**2000**	**2001000**	**1000040**
4000	**4000**	**8002000**	**4002059**
8000	**8000**	**32004000**	**16129862**



Reverse is clearly the worst case. Sorted is the best case which meets the theoretical assumption .

2. Binary Insertion Sort

Unlike insertion sort, when deciding where to place the selected array data among the preceding data, instead of checking backward sequentially, it is aimed to perform a binary search by looking at the data set as a whole. After the target location is determined, the process is completed by scrolling again.

Time Complexities:

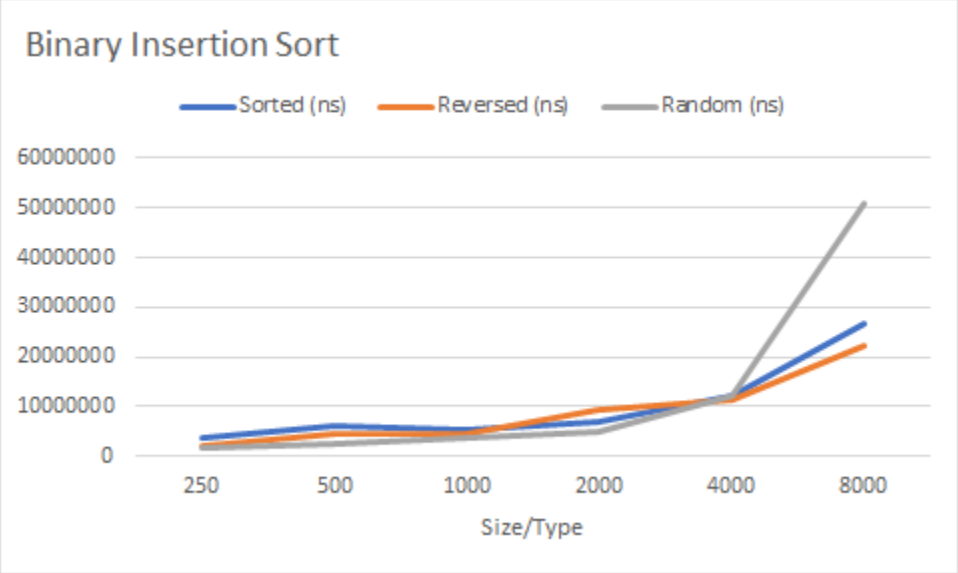
Best Case: Reversed array is the best case. Because it needs n(logn - nloge) comparisons. Its time complexity is O(nlogn).

Worst Case: Because of the swap of series needed for each iteration it is O(n2) time complexity.

Average Case: It is the same as the worst case, time complexity is O(n2).

Time:

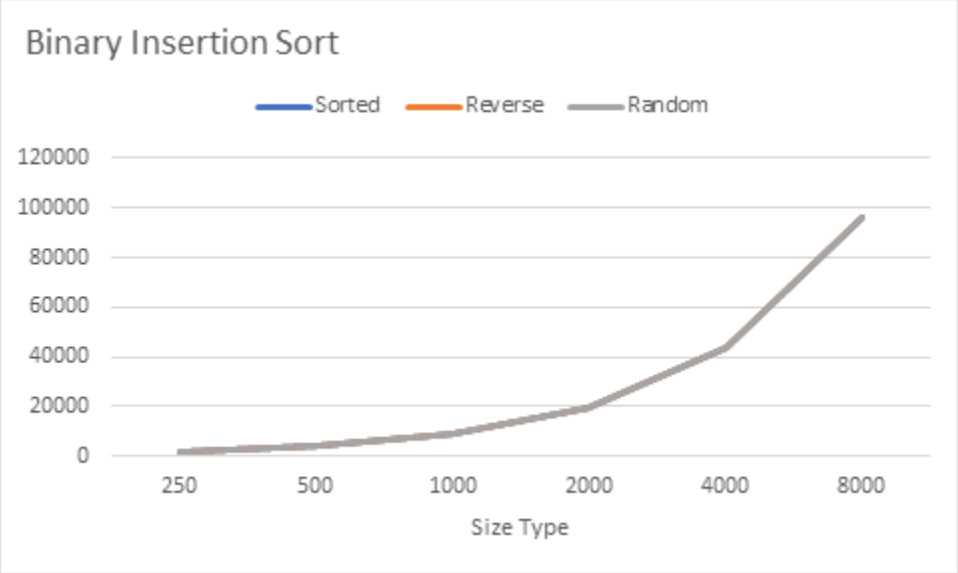
Size/Type	Sorted (ns)	Reversed (ns)	Random (ns)
250	3569600	2268600	1682801
500	5982300	4431201	2501899
1000	5355301	4666000	3866800
2000	6991301	9338801	4932901
4000	12196100	11320501	12333500
8000	26531401	22272600	50987000



Just like theory, the reversed one is the best case. The other cases should be similar and our results are similar too but in large sizes it differs a few higher than expected.

Count of Basic Operation:

Size/Type	Sorted (ns)	Reversed (ns)	Random (ns)
250	**1745**	**1745**	**1745**
500	**3989**	**3989**	**3989**
1000	**8977**	**8977**	**8977**
2000	**19953**	**19953**	**19953**
4000	**43905**	**43905**	**43905**
8000	**95809**	**95809**	**95809**



It should be different for the best case. Best case is the reversed one. This graph shows us we have made a mistake while counting operations. We count the comparisons, because comparison is the basic operation here. And it gives us unexpected results like that.

3. Merge Sort

It is a recursive algorithm that halves the array consecutively down to its smallest sub-sequences and then combines them together in order. The halving process continues until the largest sub-sequence has at most two elements. Then, with the merge operation, the subarrays are combined in a higher sequence sequentially in the order of two-by-two division. At the end of the process, a sorted list is reached.

Time Complexities:

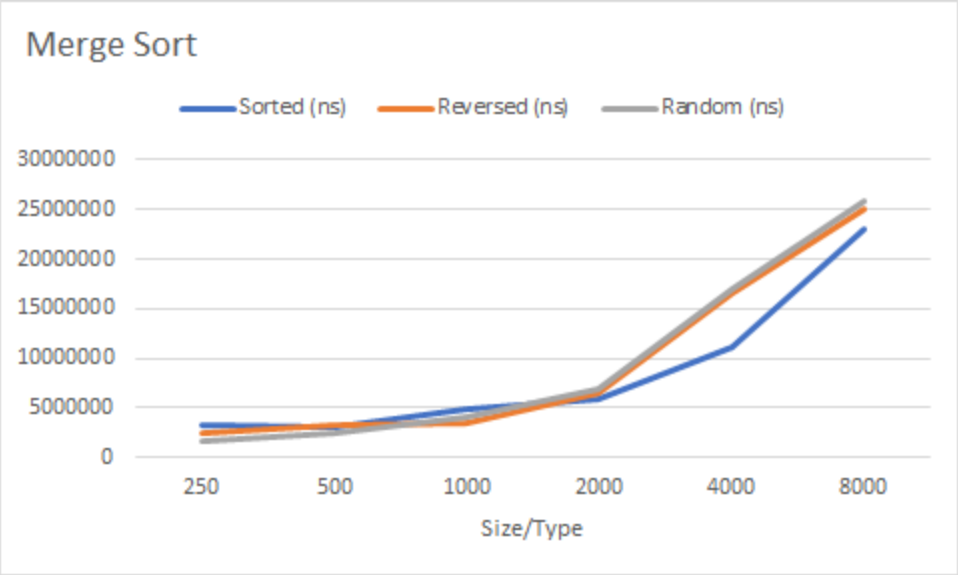
Best Case: A sorted list is going to be the easiest one to sort by merge sort. With an array length n, by binary search logic dividing is logn and and merging is n, so it is O(nlogn) time complexity.

Worst Case: It has to do a maximum number of comparisons. So alternate elements are stored in the left and right subarray. Every element of arrays are going to be compared. It is O(nlogn) time complexity.

Average Case: Merge sort always divides the array in two halves and takes linear time to merge them so this one's time complexity is also O(nlogn).

Time:

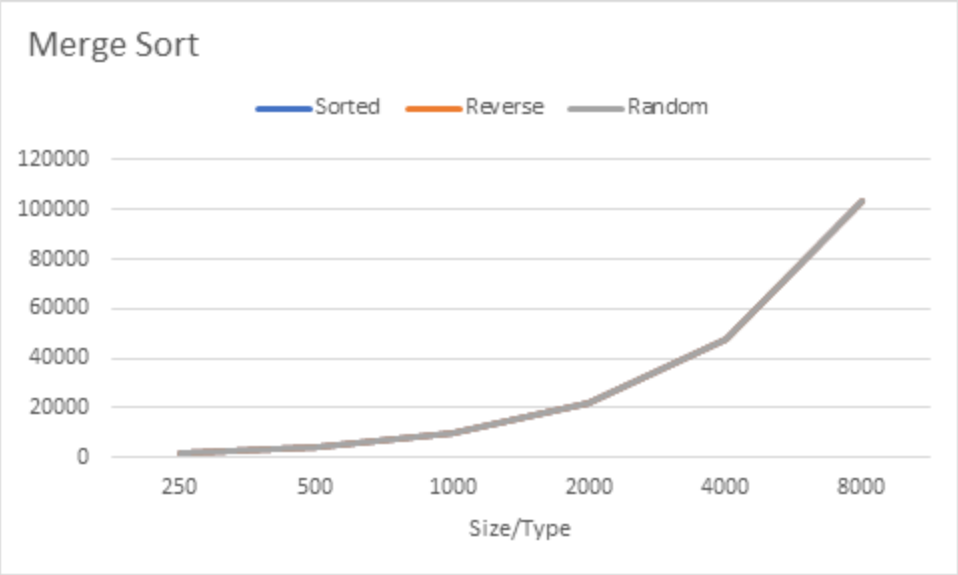
Size/Type	**Sorted (ns)**	**Reversed (ns)**	**Random (ns)**
250	3356400	2416600	1603401
500	3146001	3331401	2560900
1000	4910400	3535200	4151900
2000	5954800	6477100	6823601
4000	11238000	16536600	16978800
8000	22985700	25136600	25827801



Out time graph values for different types are really close to each other like theory. It should be the same, it is very similar. Increasing the rate is like theory O(nlogn).

Count of Basic Operation:

Size/Type	Sorted	Reversed	Random
250	**1994**	**1994**	**1994**
500	**4488**	**4488**	**4488**
1000	**9976**	**9976**	**9976**
2000	**21952**	**21952**	**21952**
4000	**47904**	**47904**	**47904**
8000	**103808**	**103808**	**103808**



Just like the theory, values are the same for different input types and it meets the theoretical assumption O(nlogn).

4. Quick-sort (pivot is always selected as the first element):

The main idea for quick sort is that it's a divide and conquer algorithm . The first step in quick sorting is choosing the number which we will divide and sort the array around it we call this number pivot . Usually there are 3 ways to choose the pivot but in our case we will use the first number as the pivot every time and then compare it with the rest of the array , put all the numbers smaller than the pivot before it and the numbers greater than the pivot after it.

Time Complexities:

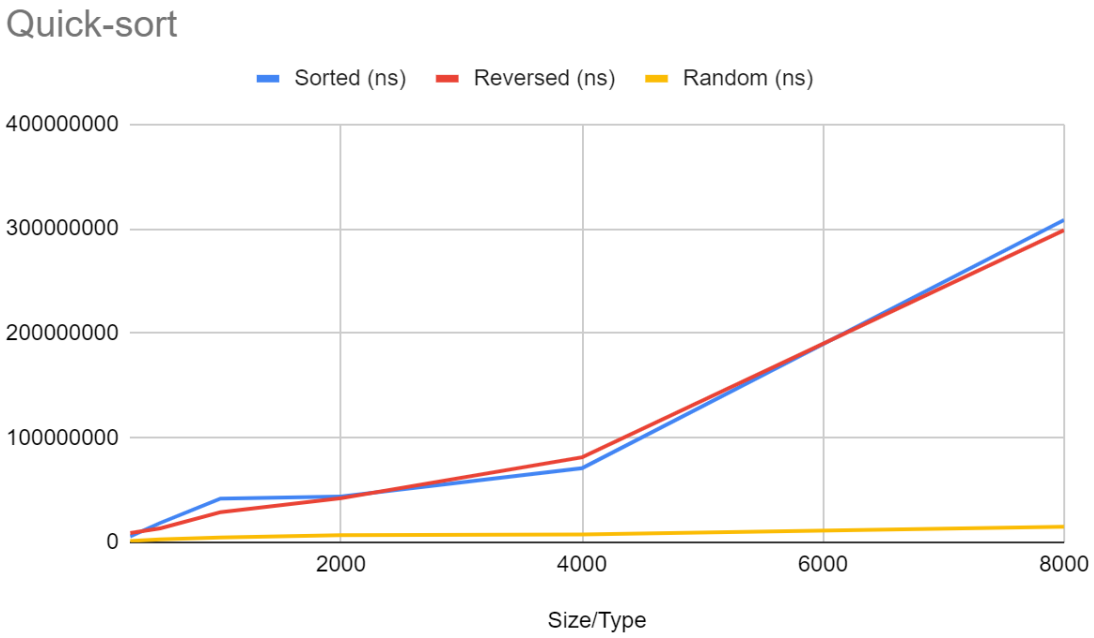
Worst case: the worst case occurs in the quick sort when we choose the pivot as the first element in the array in two cases. If our array is already sorted or sorted in reversed order in those cases the pivot will split the array to two unbalanced arrays the time complexity will be $O(n^2)$

Best case: the best case occurs when the pivot divides the array to two balanced partitions each time in this case the time complexity will be $O(n \log n)$

Average case:the average case have the same time complexity for the best case O(nlogn)

Time:

Size/Type	Sorted (ns)	Reversed (ns)	Random (ns)
250	5578100	8931000	1188800
500	18488800	13413700	2881100
1000	41831400	28878700	4722700
2000	43810900	42166900	6915300
4000	70913400	81357400	7601500
8000	308386900	298689500	14810000

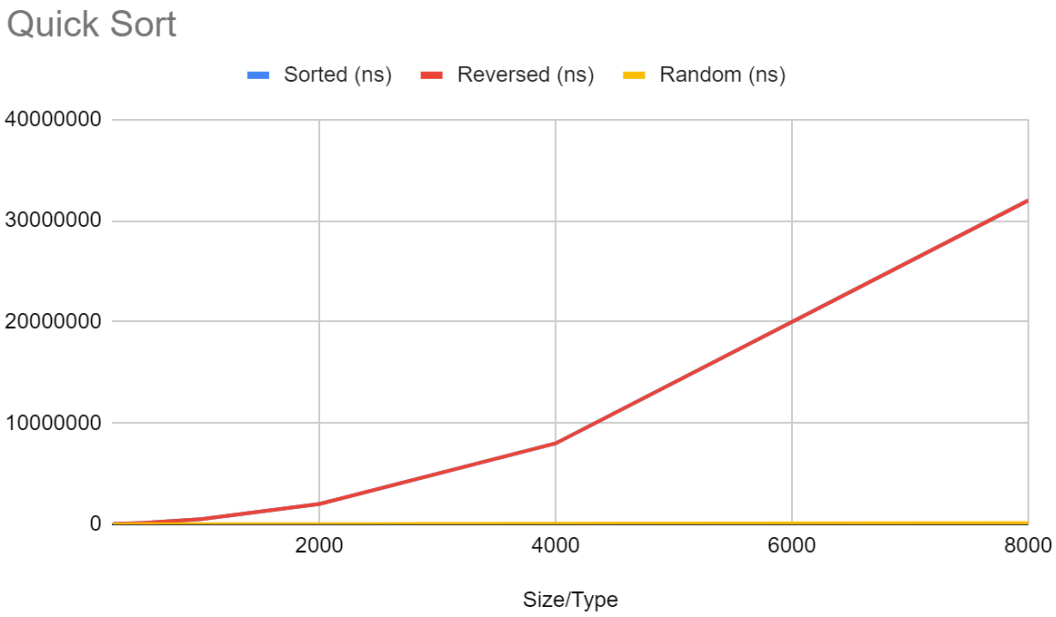


To test the quick-sort algorithm.we measured the time taken by different arrays with different sizes and in different orders as we can see the table above and as we expected the sorted and reversed arrays are given the worst case time complexity and there lines looks like $O(n^2)$

For the random arrays it gives us the best case time complexity also as we expected and as we can see in the graph it's line meets the O(nlogn) shape.

Count of basic operation :

Size/Type	Sorted (ns)	Reversed (ns)	Random (ns)
250	31125	31125	1999
500	125250	125250	4710
1000	500500	500500	10658
2000	1999000	1999000	24618
4000	7998000	7998000	52751
8000	31996000	31996000	123719



We counted the basic operation for the quick-sort algorithm and as we expected the number of operations for the random array is very low compared with the sorted and reversed arrays which have almost the same number of operations and their lines look like $O(n^2)$ as it should be.

5. Quick-sort with median-of-three pivot selection:

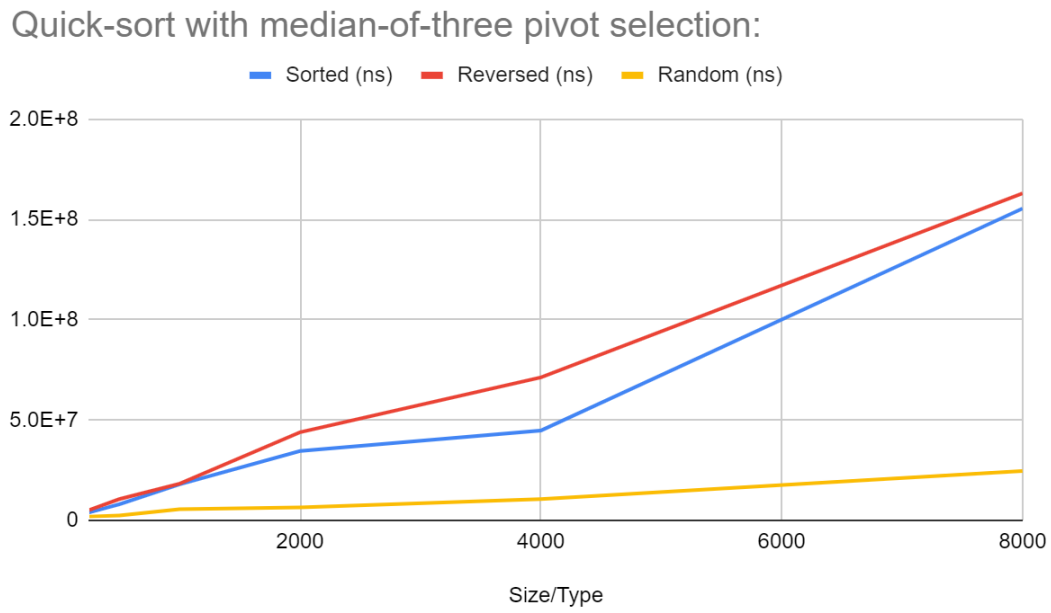
The quick-sort with median of three pivot selection is an improvement in the original quick sort algorithm . The main technique in this algorithm is comparing between the first ,last and middle elements in the array and choosing the median as the pivot .

Time Complexity:

As we said in the algorithm description this improvement should help us avoid the worst case scenario and all the cases should have time complexity of O(n)

Time:

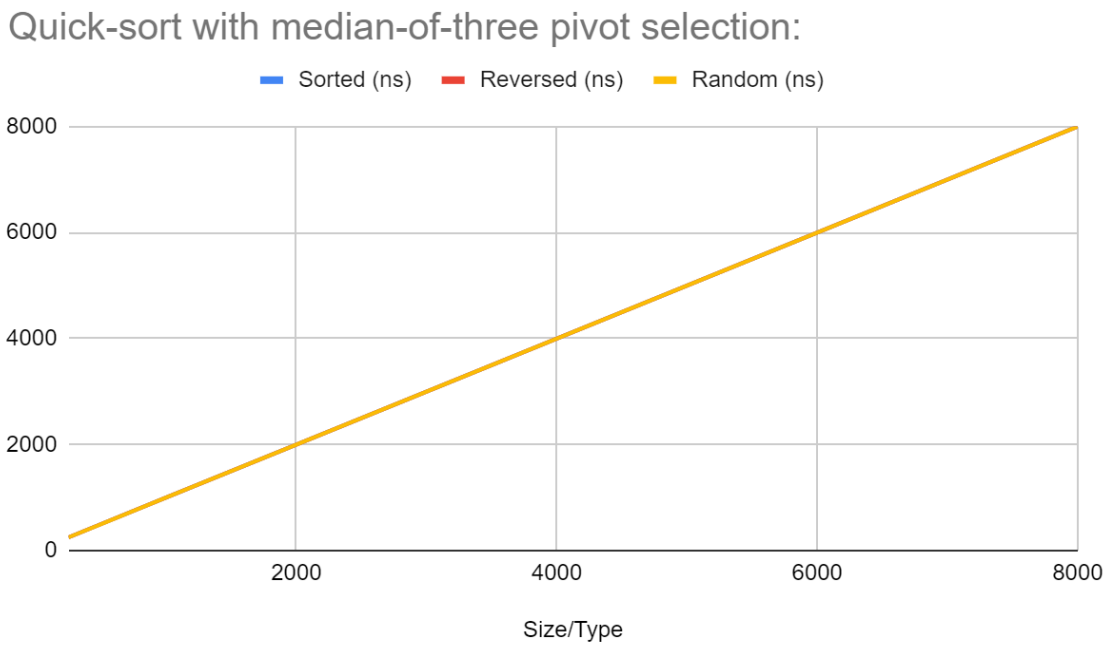
Size/Type	Sorted (ns)	Reversed (ns)	Random (ns)
250	3903400	5128600	1915200
500	8031800	10602800	2458300
1000	17939900	18237700	5528800
2000	34589800	43943700	6413000
4000	44719300	71235100	10648800
8000	155455600	163031400	24588000



Since we used our computers to calculate the real time, we got these unexpected results which don't fit the theoretical time complexity .But as we can see, the sorted and reversed lines are close to each other and look like O(n) lines.

Count of basic operation :

Size/Type	Sorted (ns)	Reversed (ns)	Random (ns)
250	249	249	249
500	500	500	499
1000	1000	1000	999
2000	1999	1999	1999
4000	3999	3999	3999
8000	7999	7999	7999



We counted the number of basic operations for the algorithm and as we can see the different arrangements of the arrays didn't change the number of the operations and perfectly fits the O(n) as we can see in the graph.

6. Heap-sort:

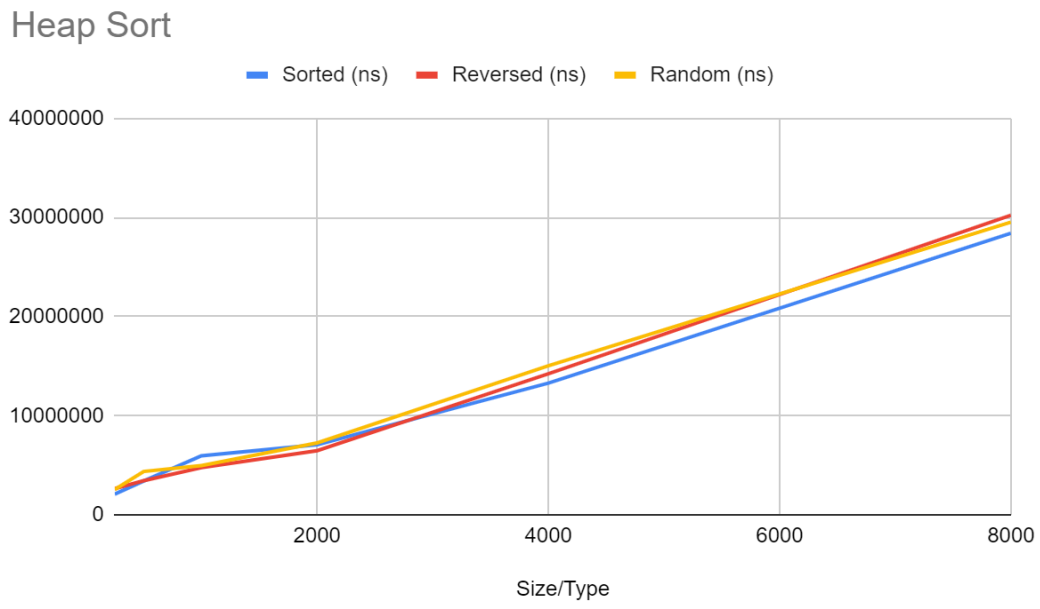
Heap-sort is a comparison based algorithm. The main idea in the heap-sort is to design a tree and sort it so the root will be the largest number and the child is the less number .

Time complexity :

In the heap-sort best,worst and average cases have the same time complexity which is O(nlog n)

Time:

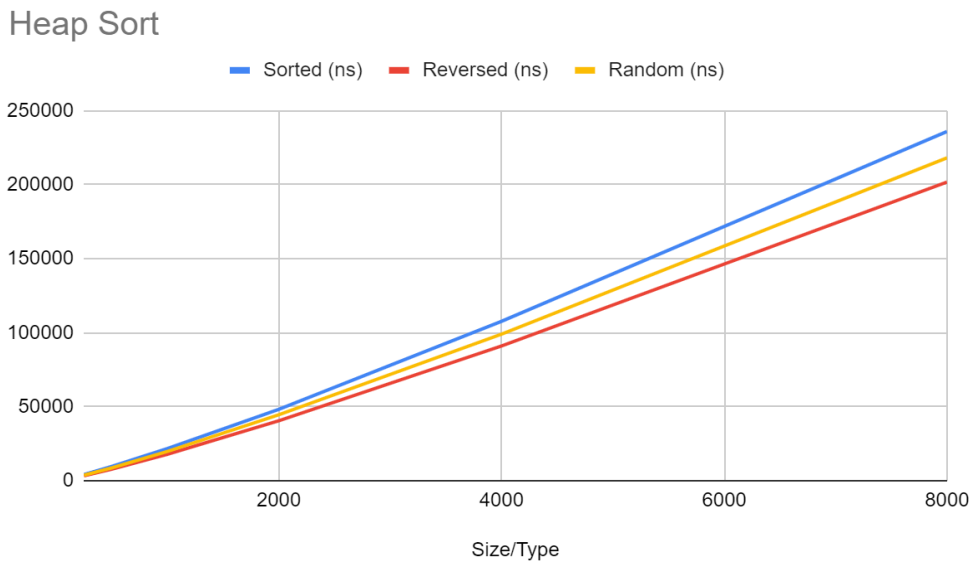
Size/Type	Sorted (ns)	Reversed (ns)	Random (ns)
250	2080800	2636500	2550400
500	3411700	3443600	4373900
1000	5947500	4762300	4971200
2000	7071100	6469700	7255600
4000	13291000	14228700	15043400
8000	28419000	30218200	29529800



After calculating the time for all the arrays we had provided as input we can see that our results fit the theoretical assumption and in all the cases the time is almost the same and close to the O(nlog n)

Count of basic operation :

Size/Type	Sorted (ns)	Reversed (ns)	Random (ns)
250	4179	3207	3727
500	9605	7701	8621
1000	21646	17844	19802
2000	48151	40493	44487
4000	107657	90994	99040
8000	235939	201761	218147



We counted the basic operation for the heap-sort algorithm and as we can see all the cases are very close to each other and close to theoretical assumption O(nlog n)

7. Counting Sort

The counting algorithm uses the range information of the values in the list to create a new list. Each row of the new list created shows the number of elements in the main list that have the value of that row number. The element value numbers in the new list are then used to put all the values in the main list in the correct position.

Time Complexities:

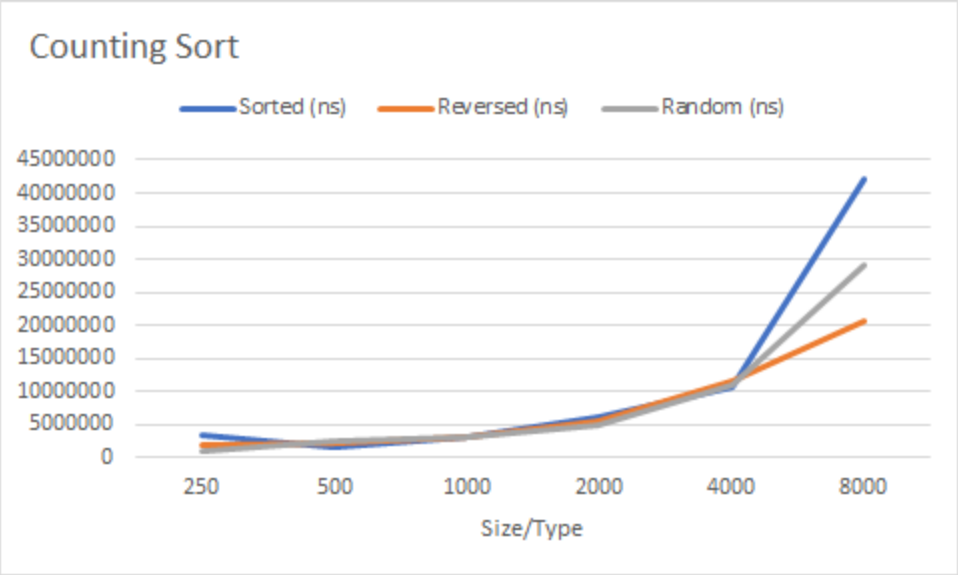
Best Case: There is no best case.

Worst Case: There are loops that iterate to n and k.n is the number of elements to sort and k is the size of the number range. So time complexity is O(n + k).

Average Case: It is the same as the worst case, time complexity is O(n + k).

Time:

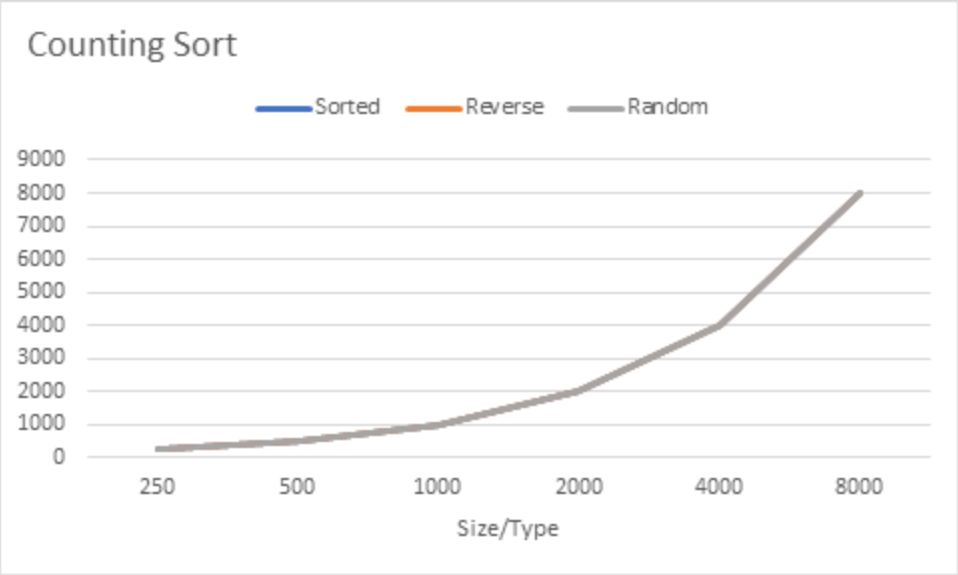
Size/Type	Sorted (ns)	Reversed (ns)	Random (ns)
250	3288400	1822900	1079900
500	1507400	2126700	2526900
1000	3108499	3103300	3179700
2000	6066200	5560801	4957901
4000	10586099	11683701	11099500
8000	42118601	20632600	29177199



Our time graph is really close to the theory until the high size of inputs. After 4000 sized arrays it goes a little higher than expected.Increase rate is like theory.

Count of Basic Operation:

Size/Type	Sorted (ns)	Reversed (ns)	Random (ns)
250	249	249	249
500	499	499	499
1000	999	999	999
2000	1999	1999	1999
4000	3999	3999	3999
8000	7999	7999	7999



Just like theory, values are the same for different input types. Increase rate is the same as what we expected.

Conclusion

As a result we find out that quicksort with median-of-three pivot selection is the fastest algorithm for our purpose. In all circumstances it does not matter if our array is sorted, reversed or random; it has time complexity O(n). Also from our time measurements table and graphs, it proves that this is the fastest. We should talk about counting sort too. It is also a very fast algorithm. It has O(n+k) time complexity for all circumstances. But we have to remember it is a very memory consuming algorithm.

Goal of this project is to see for different types of lists, different types of sorting algorithms are useful. One algorithm may perform best at a reversed sorted array (binary insertion sort), the other one (insertion sort)may perform its worst. Also in general we see from both theory and our measurements which algorithm performs best in general. Our experimental values are mostly similar with the theoretical ones.