MARMARA UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

CSE2046 - ANALYSIS OF ALGORITHMS PROJECT

Muhammed Murat Dilmaç - 150116017

Elif Gülay - 150119732

Beyzanur Çabuk - 150119632

Purpose of the Project

The aim of the project is to design an experiment to compare different algorithms for the selection problem, i.e. finding k-th smallest element in an unsorted list of n numbers. In this experiment, we are going to compare seven methods and we will analyze our results both theoretically and empirically.

1. INSERTION SORT

Insertion Sort algorithm is based on the principle of handling unsorted array elements one by one and placing each one in its proper place in the sorted part of the array. In the algorithm, starting from the second element, the larger elements are shifted to the right in the array by comparing the element with the previous elements. Thus, a sorted array is obtained.

Time Complexity:

- Best Case: In the best case, the array is already sorted and time complexity is O(n).
- Worst Case: In the worst case, the array is sorted in reverse and time complexity is $O(n^2)$.
- Average Case: $O(n^2)$

ALGORITHM *InsertionSort*$(A[0..n-1])$

```
//Sorts a given array by insertion sort
//Input: An array A[0..n − 1] of n orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← v
```

2. MERGE SORT

It is one of the sorting algorithms developed to keep the data (in our case, an unsorted array) sorted in memory. It simply divides the array to be sorted in two continuously until its two elements are reduced to the remaining parts. Then he merges these parts by arranging them among themselves. The resulting array is the sorted array itself. In this respect, it is a divide and conquer approach.

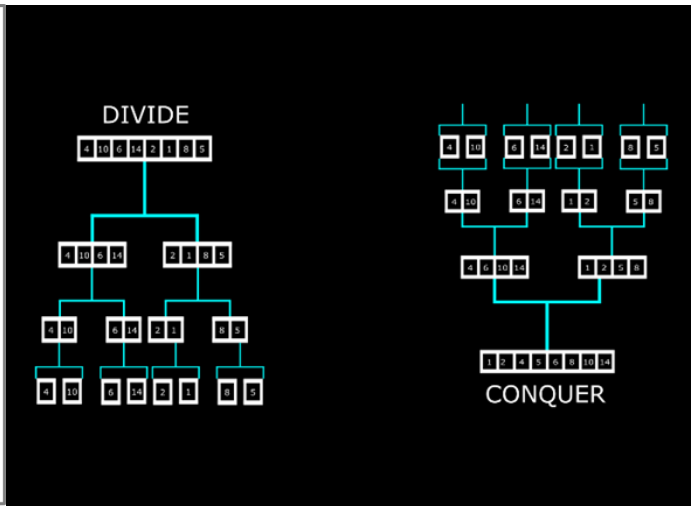
Time Complexity:

Since it is a recursive function, it always calls itself and divides the array into two. The complexity will be O(N*logn) since the length of the array is N for the Merge operation of each split array:

- Best case: O(N*logn)
- Average case: O(N*logn)
- Worst case: O(N*logn)

```
MERGE(A, p, q, r)
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
 4  for i ← 1 to n1
 5        do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7        do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13        do if L[i] ≤ R[j]
14              then A[k] ← L[i]
15                   i ← i + 1
16              else A[k] ← R[j]
17                   j ← j + 1
```

Pseudo Code of Merge Sort



Operations of Merge Sort

3. QUICK SORT

Quicksort is a Divide and Conquer algorithm, it picks an element as pivot and partitions the given array around the picked pivot. There are different versions of quicksort that pick pivots in different ways. In our case, we can take all elements as pivots. But we choose the pivot element as the first element in an array while partitioning.

Time Complexity:

Quick sort is a divide and conquer algorithm as mentioned and it selects the first element as the pivot and partitions the array.

- Best case, inputs occur when the partitions are as evenly balanced as possible. Our random case is more suitable for that. Time complexity of the best case is O(nlogn).
- Worst case, we used inputs with reversed sorted and sorted arrays. Because the worst case occurs when the largest or smallest element is selected as the pivot. The time complexity of the worst case is O(n2).
- For the average case, we generated random inputs with different sizes. The time complexity of the average case is O(nlogn).

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

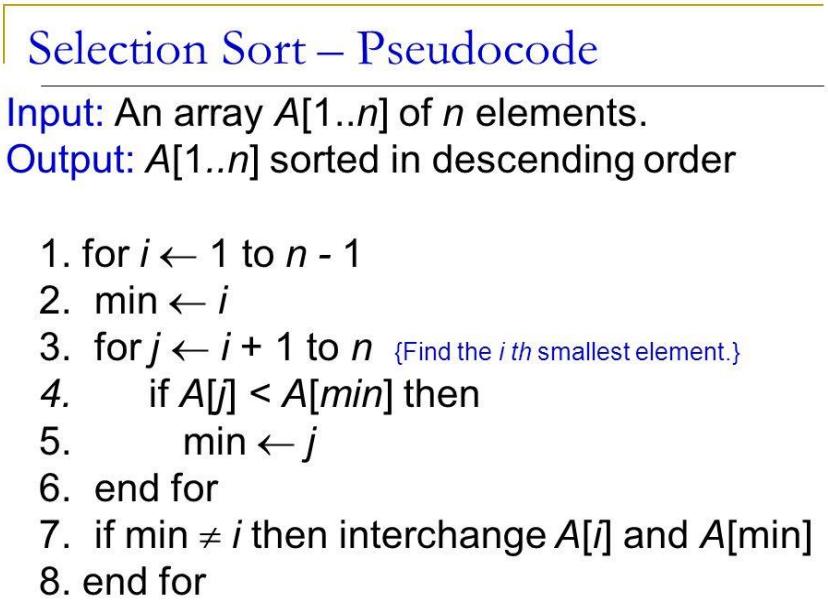
4. PARTIAL SELECTION SORT

For the partial selection sort we will talk about the selection algorithm. This is an algorithm for finding the k.th smallest (or largest) number in a list or an array. That number is called the kth order statistic. For finding the minimum (or maximum) element by iterating through the list, we keep track of current minimum (or maximum) elements that occur so far and it is related to the selection sort.

Time Complexity:

Sorting the list or an array then selecting the required element can make selection easy. This method is inefficient for selecting a single element but is efficient when many selections need to be made from an array for which it only needs sorting of an array. For selection in a linked list is O(n) even if the linked list is sorted due to lack of random access.

Instead of sorting the entire list or an array, we can use partial sorting to select the kth smallest(or largest) element in a list or an array. Then the kth smallest(or largest) is the largest(or smallest) element of the partially sorted list. This takes O(1) to access in an array and O(k) to access in a list.



5. HEAP SORT (In our case, max-heap sort)

A binary tree structure is referred to as a heap. Maximum and minimum are the two options. The data is arranged in descending order in the maximum heap type, with the element in the root (main) node being the largest, whereas the data is placed in ascending order in the minimum heap type, with the element in the root being the smallest. As a result, the data can be sorted ascending or descending. When utilizing the maximum heap, the root node is destroyed and placed at the end of the directory, and it is sorted from least to largest using the minimum heap.

Time Complexity:

- Best case: O(N*logn)
- Average case: O(N*logn)
- Worst case: O(N*logn)

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)


BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)


MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4      largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7      largest = r
8  if largest ≠ i
9      exchange A[i] with A[largest]
10     MAX-HEAPIFY(A, largest)
```

6. QUICK SELECT

Quick Select is an algorithm based on finding the k-th smallest element in an unordered list. It finds the k-th smallest element and then partitions the array around that element. Quick Select algorithm is very similar to the Quick Sort algorithm. The difference is that Quick Sort algorithm sorts the entire array while Quick Select algorithm is searching for only one element.

Time Complexity:

- Best Case: O(n)
- Worst Case: $O(n^2)$
- Average Case: O(n)

```
function quickSelect(list, left, right, k)

  if left = right
    return list[left]

  Select a pivotIndex between left and right

  pivotIndex := partition(list, left, right)
  if k-1 = pivotIndex
    return list[pivotIndex]
  else if k-1 < pivotIndex
    quickSelect(arr, left, pivotIndex – 1, k)   //left
  else
    quickSelect(arr, pivotIndex + 1, right, k) //right array
```

Deciding on Metrics

We have two different methods for doing our empirical analysis:

1. Using a physical time unit.
2. Counting the number of basic operations of executions.

Due to the problems that can arise from deciding basic operations of executions, we chose to measure the physical unit of time to arrive at more reasonable conclusions. That's why we put the time counters of Java's System.nanoTime() method exactly before and after our functions. We measured the time in microseconds (μs) for each of our experiments by finding the difference between these two counters. We preferred the microsecond (μs) unit to achieve more precise results. We tested the algorithms 10 times with different inputs to find comparable results of each algorithm.

Illustrating and Analyzing Results

Insertion Sort:

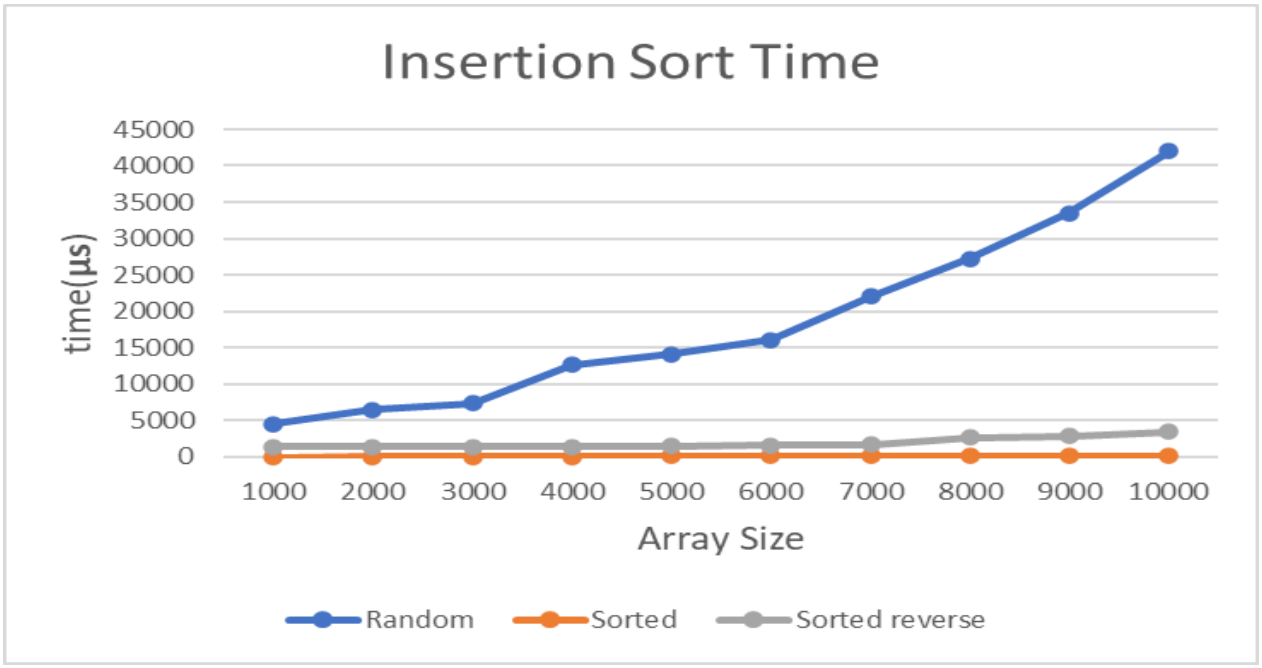
In order to get the best performance from insertion sort algorithm, the input array must be sorted. As we can see in the table, when we use a sorted array, measured time will be smaller compared to the random and decreased sorted arrays. For this reason, we can clearly say that it is most appropriate to use sorted arrays for the best case.

When the elements of the input array are in random order, the insertion sort algorithm performs the sorting in the longest time. Therefore, we used an array in reverse order for the worst case.

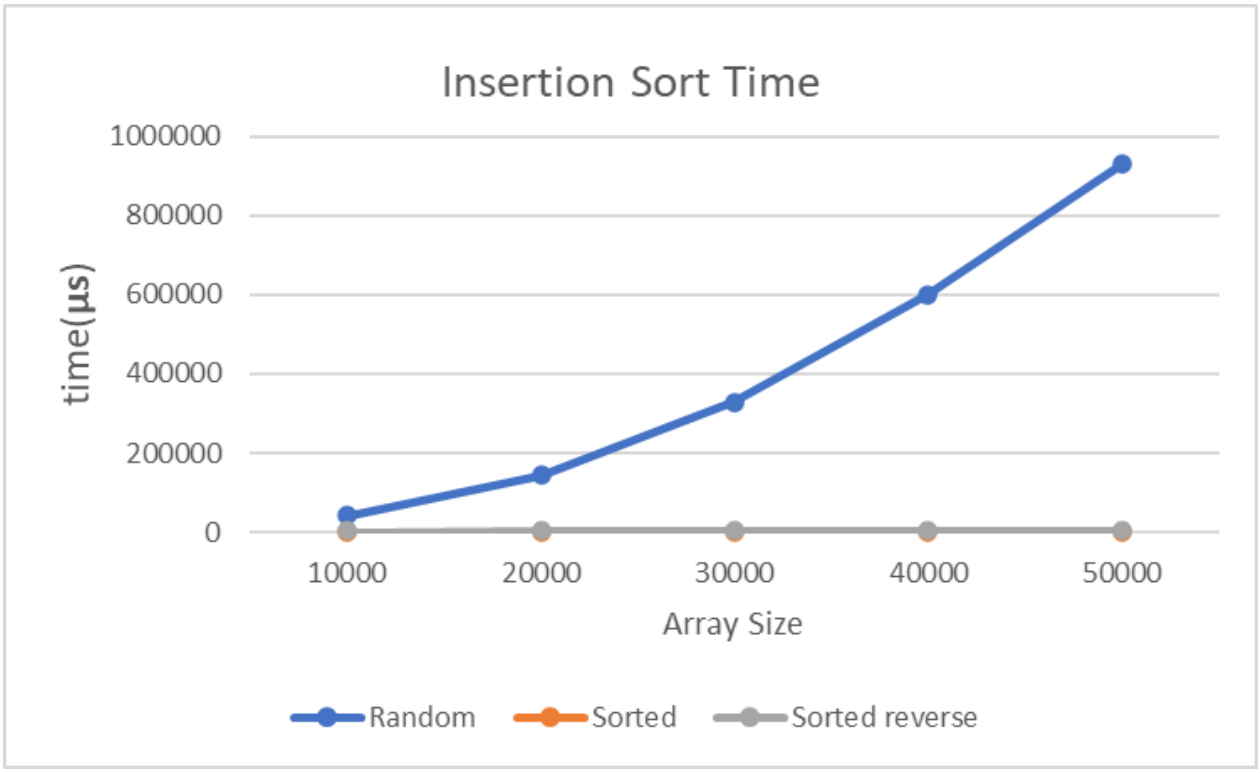
For the average case, we used randomly ordered input arrays and calculated their average. As we can see in the table, when we increase the input size, the time also increases as expected.

Insertion Sort			
Array size	Random	Sorted	Sorted reverse
100	990	20	728
200	1049	28	752
300	1169	29	769
400	1454	31	777
500	1765	32	829
600	2398	32	928
700	2819	32	1160
800	2922	33	1195
900	3662	33	1331
1000	4505	37	1334

Insertion Sort			
Array size	Random	Sorted	Sorted reverse
1000	4505	37	1334
2000	6412	39	1334
3000	7386	41	1337
4000	12627	44	1392
5000	14059	50	1488
6000	16061	58	1551
7000	22065	67	1633
8000	27243	85	2653
9000	33500	99	2837
10000	42044	117	3441



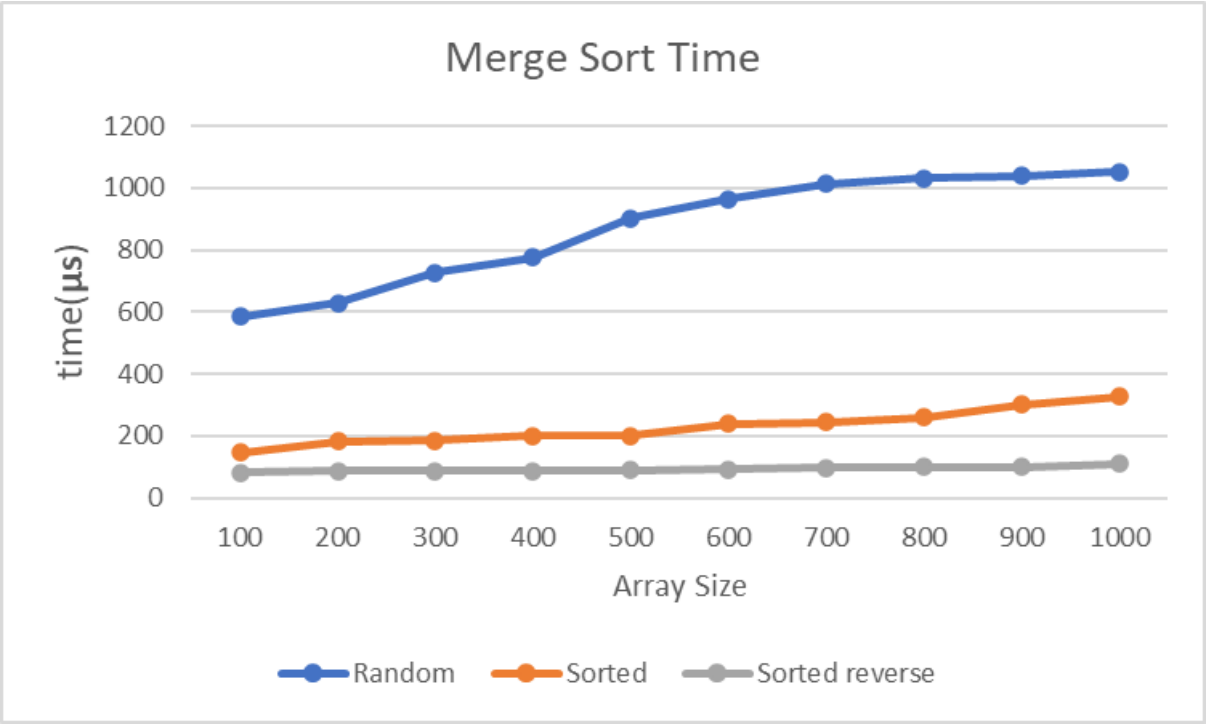
Insertion Sort			
Array size	Random	Sorted	Sorted reverse
10000	42044	117	3441
20000	143218	150	3596
30000	328871	237	3941
40000	601223	365	4453
50000	929107	406	4486



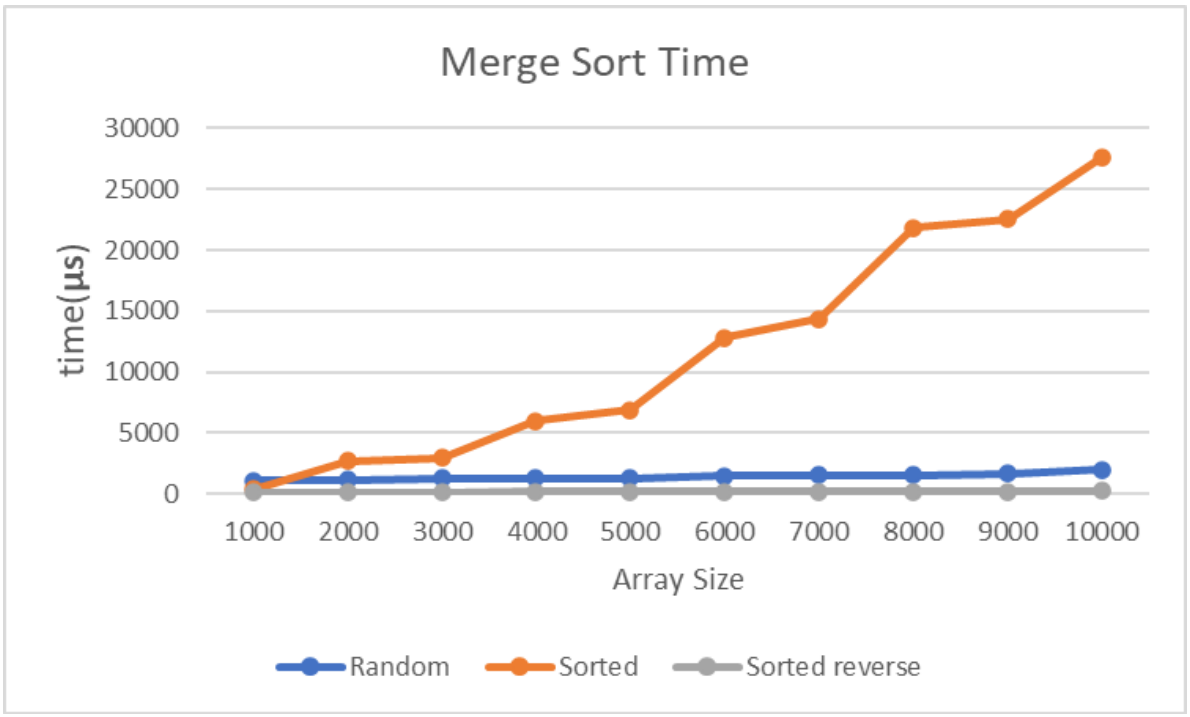
MERGE SORT:

To find the best case of merge sort, using an unordered array (descending) is optimal. We observed that in this state our arrays gave the lowest time values compared to the other cases. Theoretically, we knew that the complexity of the merge sort in all cases was O(n.logn).
To have maximum comparison, we used an random array whose values are unique from each other. As we expected, we saw that as the peak time and input size increased, the time (n.logn) increased in proportion.

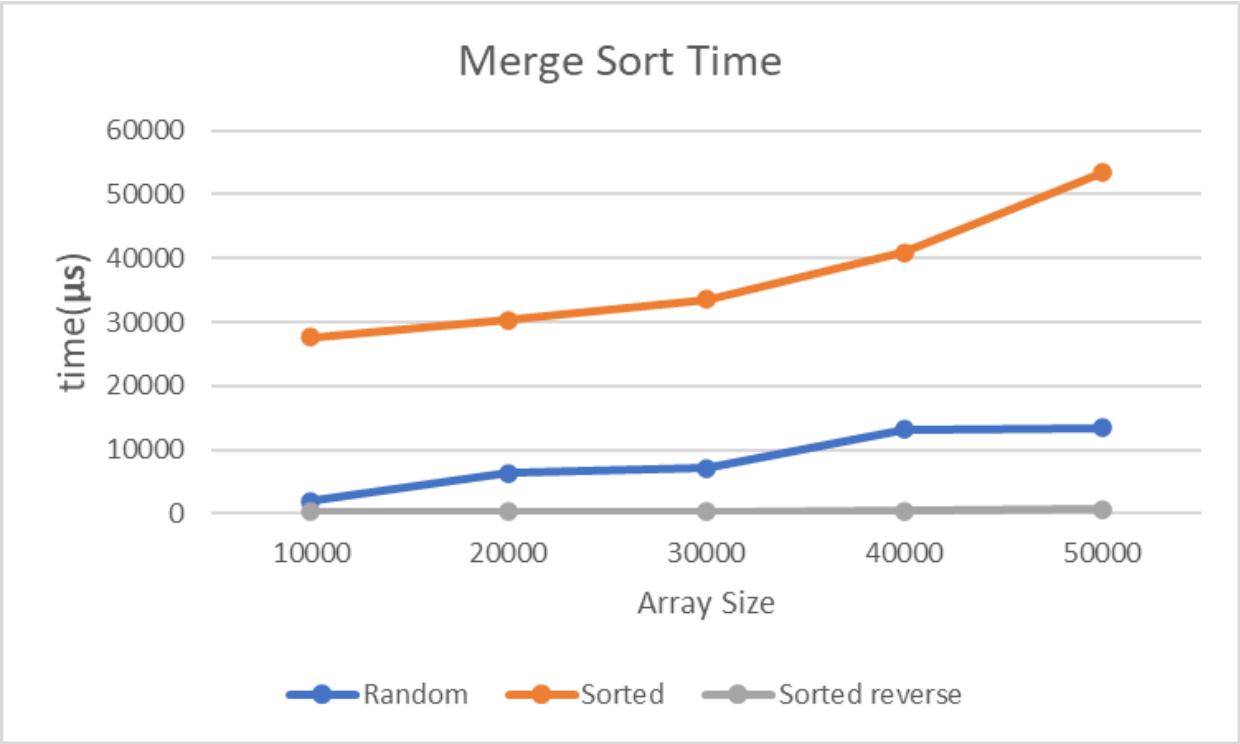
Merge Sort			
Array Size	Random	Sorted	Sorted reverse
100	583,9	147,3	80,8
200	628	183,7	87,5
300	726,6	185,3	87,9
400	776,6	200,3	88,2
500	901,8	200,4	90,8
600	964,1	238,8	92,5
700	1013,5	243,6	98,8
800	1031,1	259,2	99,3
900	1039,5	300,8	99,9
1000	1053,3	327,4	110,5



Merge Sort			
Array Size	Random	Sorted	Sorted reverse
1000	1053,3	327,4	110,5
2000	1103,3	2693,9	112,7
3000	1235,8	2929,7	114,9
4000	1268,7	5978,1	133,2
5000	1277,8	6891,8	140,9
6000	1444,1	12811,6	165
7000	1499	14367,2	182,2
8000	1523,4	21840,7	184,9
9000	1642,5	22533,5	192,5
10000	1942	27656,5	204,7



Merge Sort			
Array Size	Random	Sorted	Sorted reverse
10000	1942	27656,5	204,7
20000	6282,7	30298,7	238,6
30000	7090,2	33549,9	277,5
40000	13202,5	40946	343,4
50000	13416	53454,7	622,5

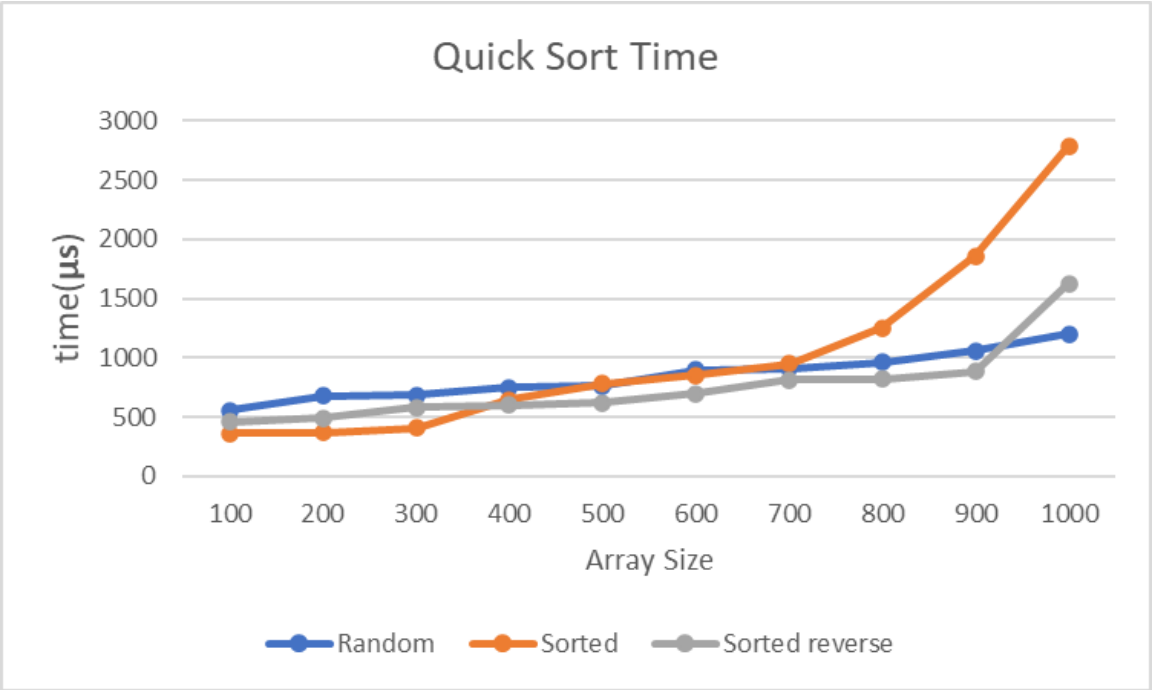


QUICK SORT:

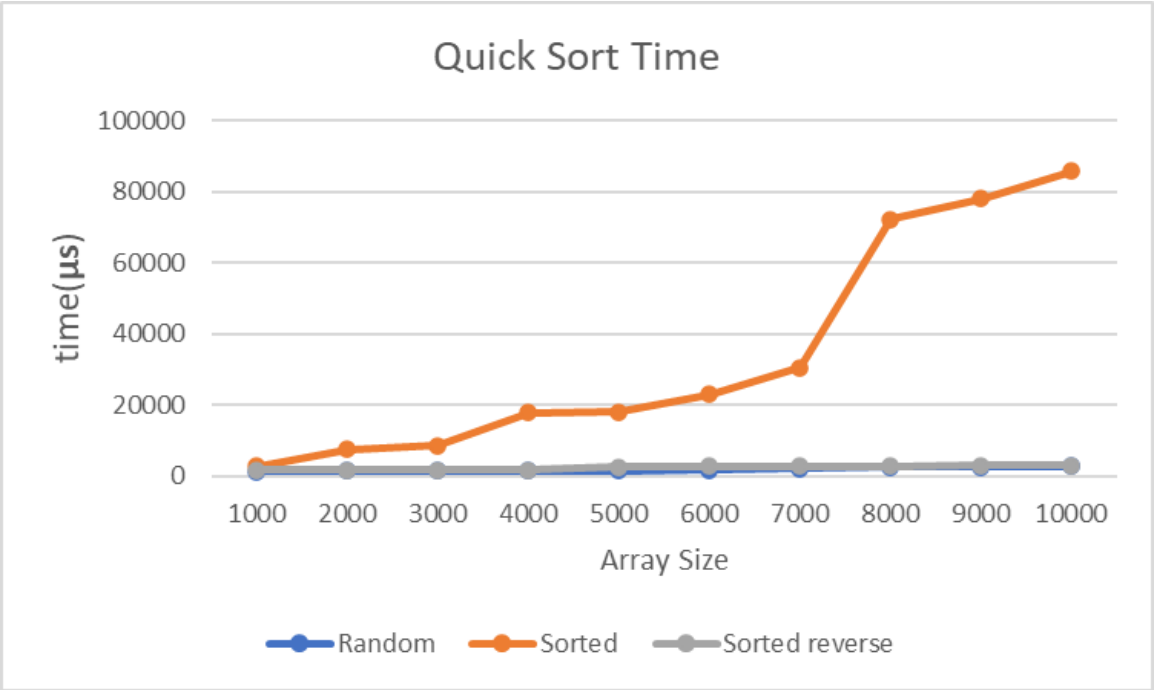
The quick sort algorithm performs best when the pivot element is in the middle. According to the results we obtained, we can say that it is more appropriate to use a decreased sorted array for the best case, since the decreased sorted array is completed in a shorter time compared to the others.

The quick sort algorithm shows the lowest performance when the pivot element is the largest or the smallest element. According to our observations, when we use a sorted array, a longer time is spent compared to others. For this reason, we can say that it is more appropriate to use a sorted array for the worst case.

We took a stack overflow error after 20000 array size, it shows us Quick sort is not suitable for large amount of data.



Quick Sort			
Array Size	Random	Sorted	Sorted reverse
100	557,6	359,7	461,4
200	680,5	369,7	494,5
300	681,9	404,4	583
400	751,4	644,4	600,1
500	761,8	784,5	622,9
600	898,1	853,8	700,4
700	910,6	950,9	814,1
800	963,8	1251	818,9
900	1061,3	1859,7	887,3
1000	1200,9	2784,5	1628,4

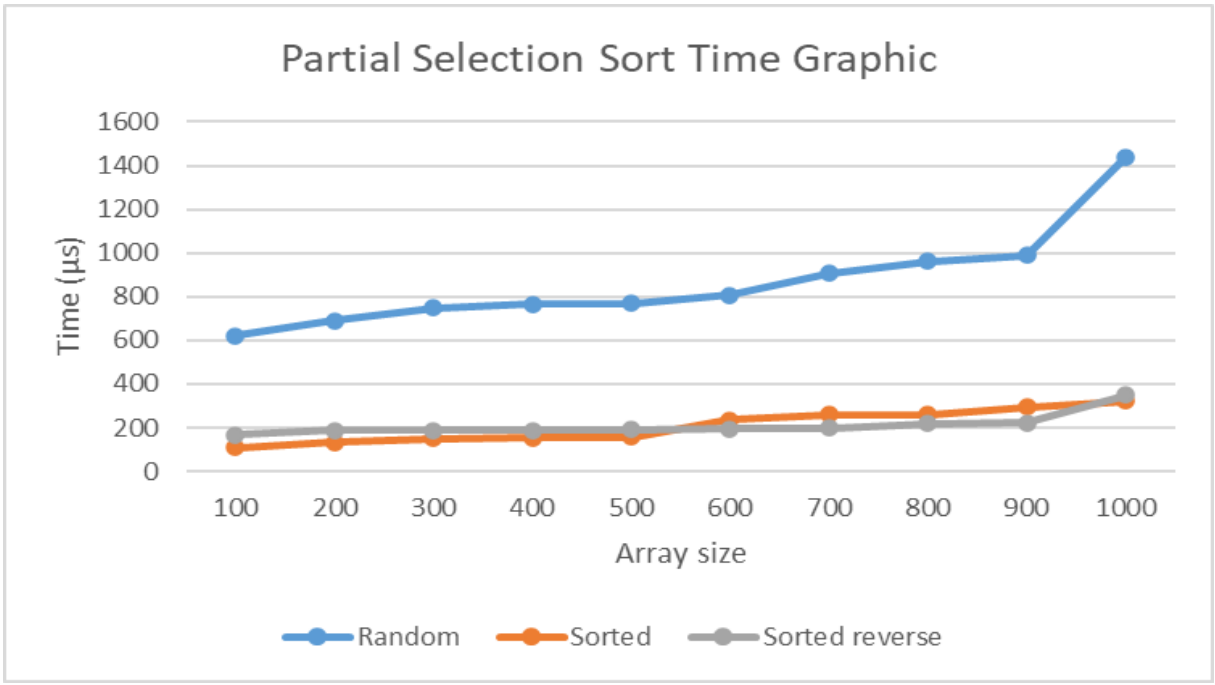


Quick Sort			
Array Size	Random	Sorted	Sorted reverse
1000	1200,9	2784,5	1628,4
2000	1419,9	7549,9	1646,5
3000	1483,4	8705,3	1675,8
4000	1520,7	17830,1	1742,2
5000	1566,3	18012	2619,1
6000	1797	22986,2	2741,6
7000	2178,1	30632,6	2758,2
8000	2528,4	72222,6	2855
9000	2534,5	78006,1	2919,4
10000	2878,5	85847,9	2990,3

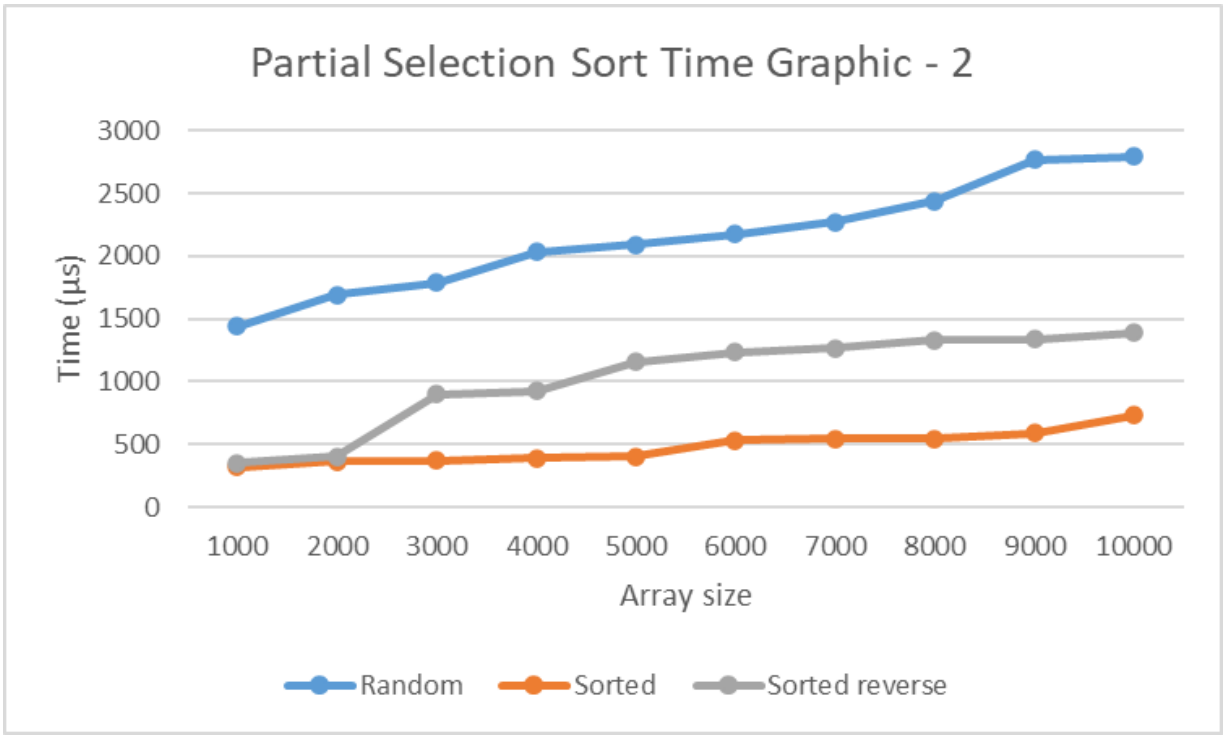
PARTIAL SELECTION SORT:

Based on the results we obtained, when we use a decreased array, the time spent is minimized. For this reason, it would be more accurate to use a decreased sorted array for the best case. Again, as a result of our observations, we see that the time spent is the longest when we use a randomly sorted array. Therefore, we can interpret that it would be more appropriate to use a random array for the worst case.

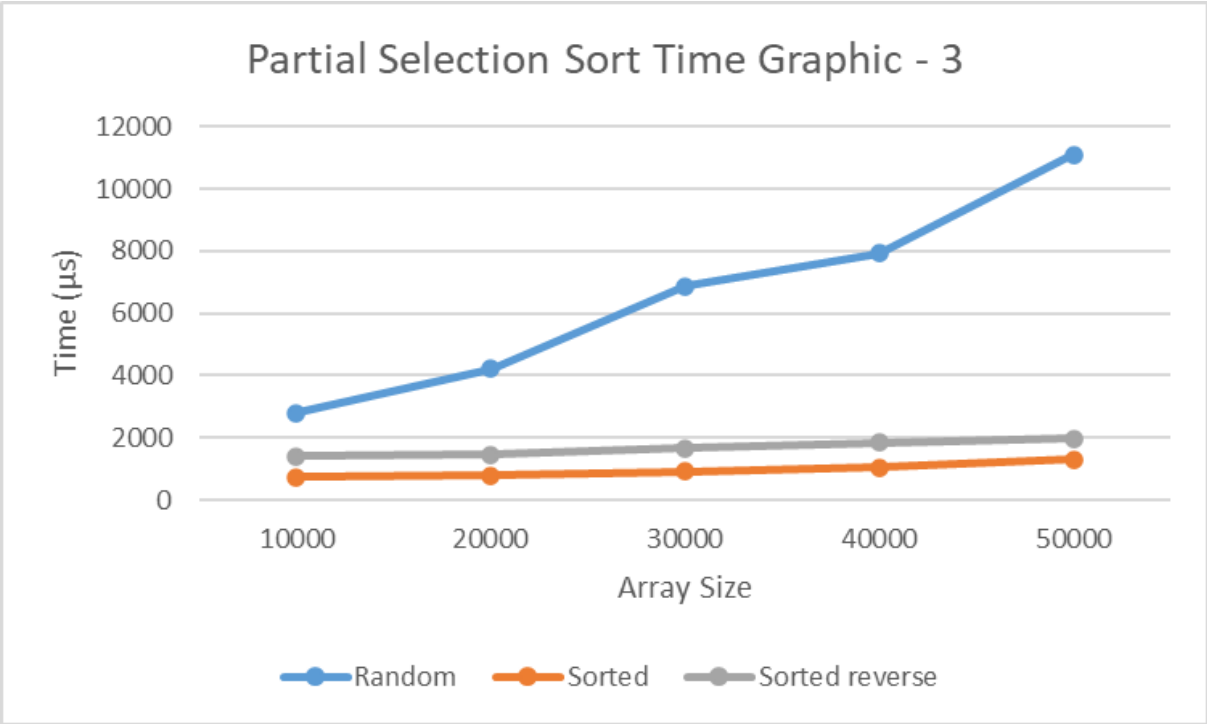
Partial Selection Sort			
Array Size	Random	Sorted	Sorted reverse
100	622	109,1	168,3
200	690	133,4	186,3
300	747,5	149,9	189,1
400	765,1	154,4	189,3
500	767,5	158,4	192,6
600	805,7	237,2	195,8
700	905,7	259	198,2
800	961,1	260,9	220,1
900	989,5	292,6	220,6
1000	1440,7	321,6	350



Partial Selection Sort			
Array Size	Random	Sorted	Sorted reverse
1000	1440,7	321,6	350
2000	1695,1	364,9	400
3000	1787,8	367,5	897,3
4000	2037,3	391,6	927
5000	2091,1	405,5	1156,7
6000	2175,2	531,6	1231,9
7000	2272,9	541,5	1268
8000	2440,3	544,9	1334,2
9000	2771,2	588,1	1336,7
10000	2797,9	733,9	1392



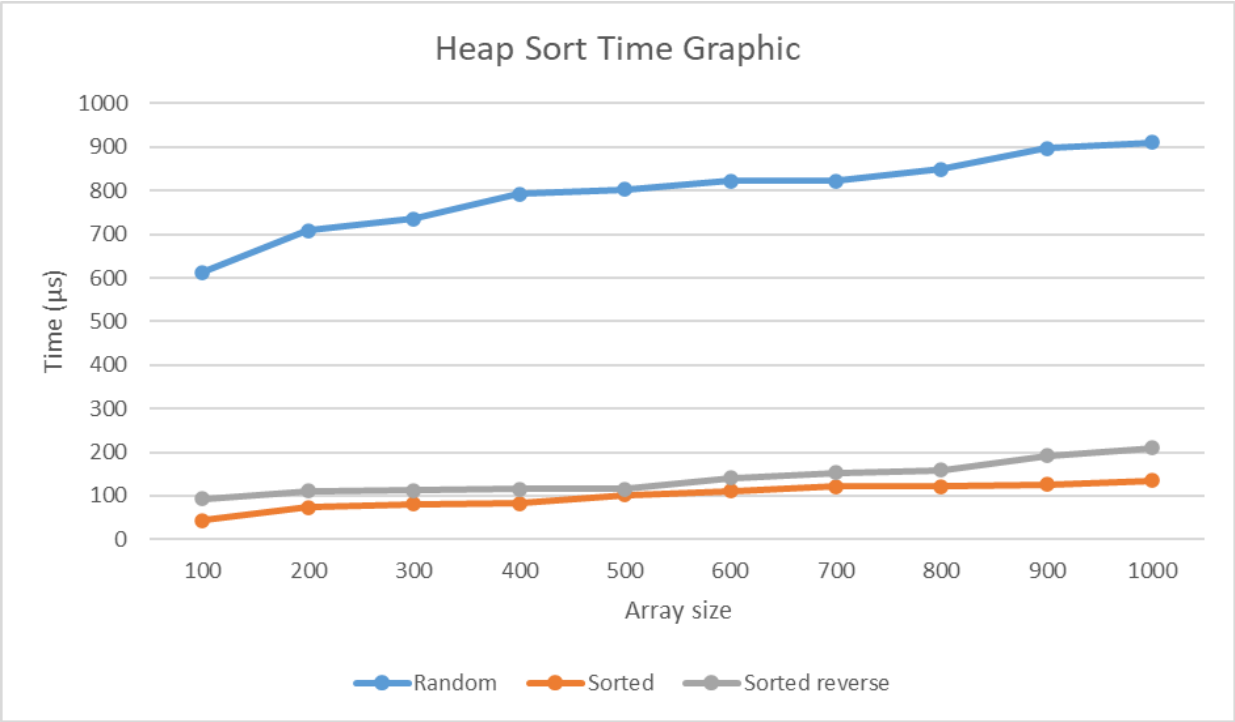
Partial Selection Sort			
Array Size	Random	Sorted	Sorted reverse
10000	2797,9	733,9	1392
20000	4218,4	792,6	1445,5
30000	6871,3	906,1	1666,9
40000	7922,8	1049,7	1834,4
50000	11106,6	1312,8	1982,8



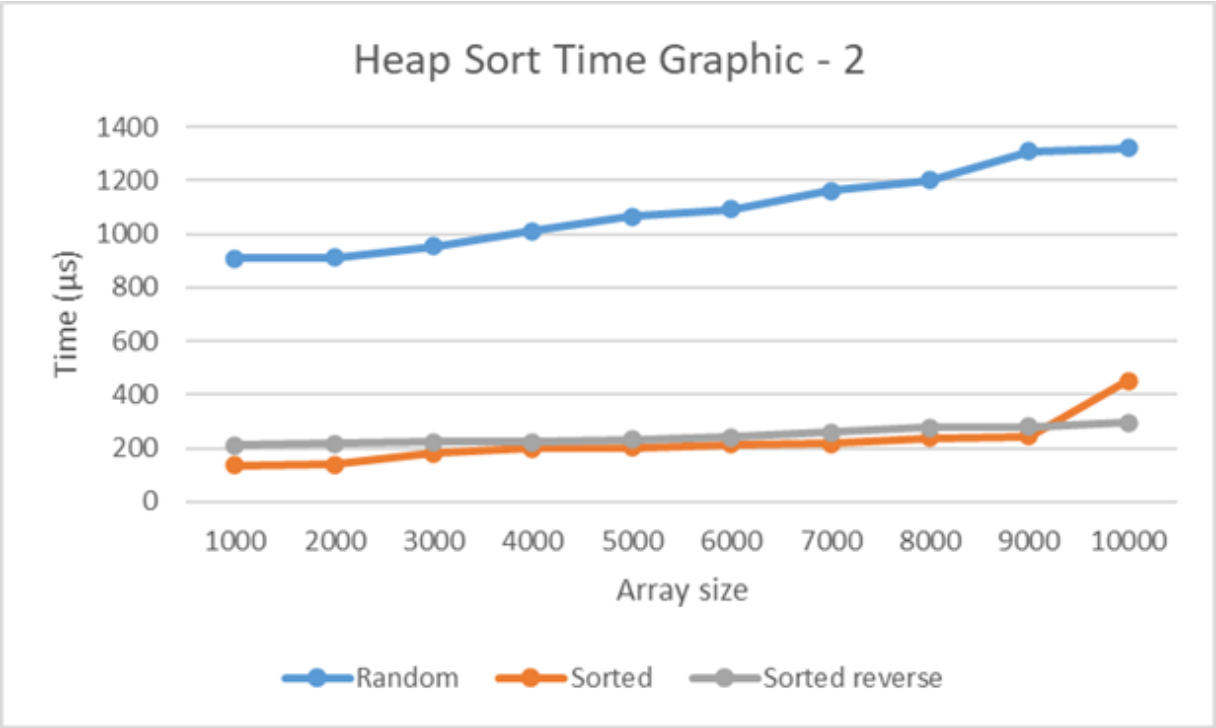
HEAP SORT:

The heap-sort algorithm performs best when all elements in the input array are the same. According to our observations, when we use a decreased sorted array, the time spent is minimized. For this reason, according to our results, we can deduce that a decreased sorted array is more suitable for the best case.

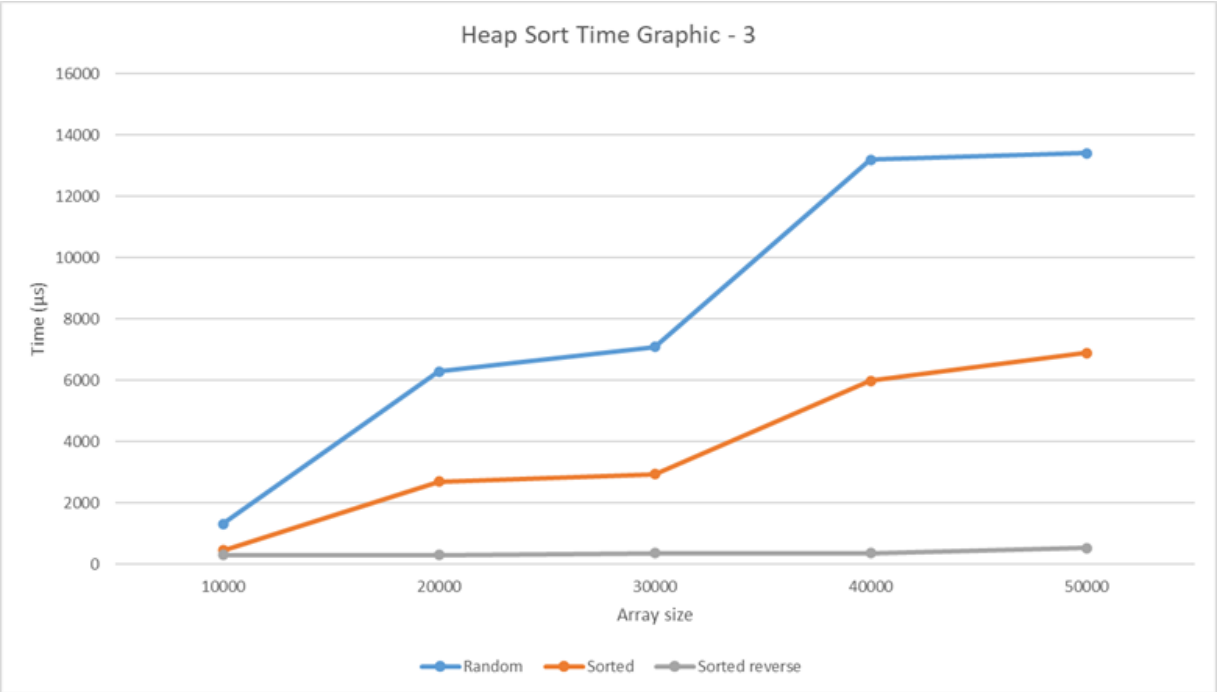
As can be seen in the table, we can say that it is more appropriate to use a random array for the worst case, since the elapsed time is the longest when we use a randomly sorted input array.



Heap Sort			
Array Size	Random	Sorted	Sorted reverse
100	612,4	44	93,7
200	708,7	73,6	110,5
300	735,7	81	112,7
400	793	82,3	114,9
500	802,8	101,7	115,8
600	822,4	111,3	141,7
700	822,8	121,7	152,3
800	849	121,8	159,5
900	897,9	126,6	192,5
1000	910,4	135,7	210,5



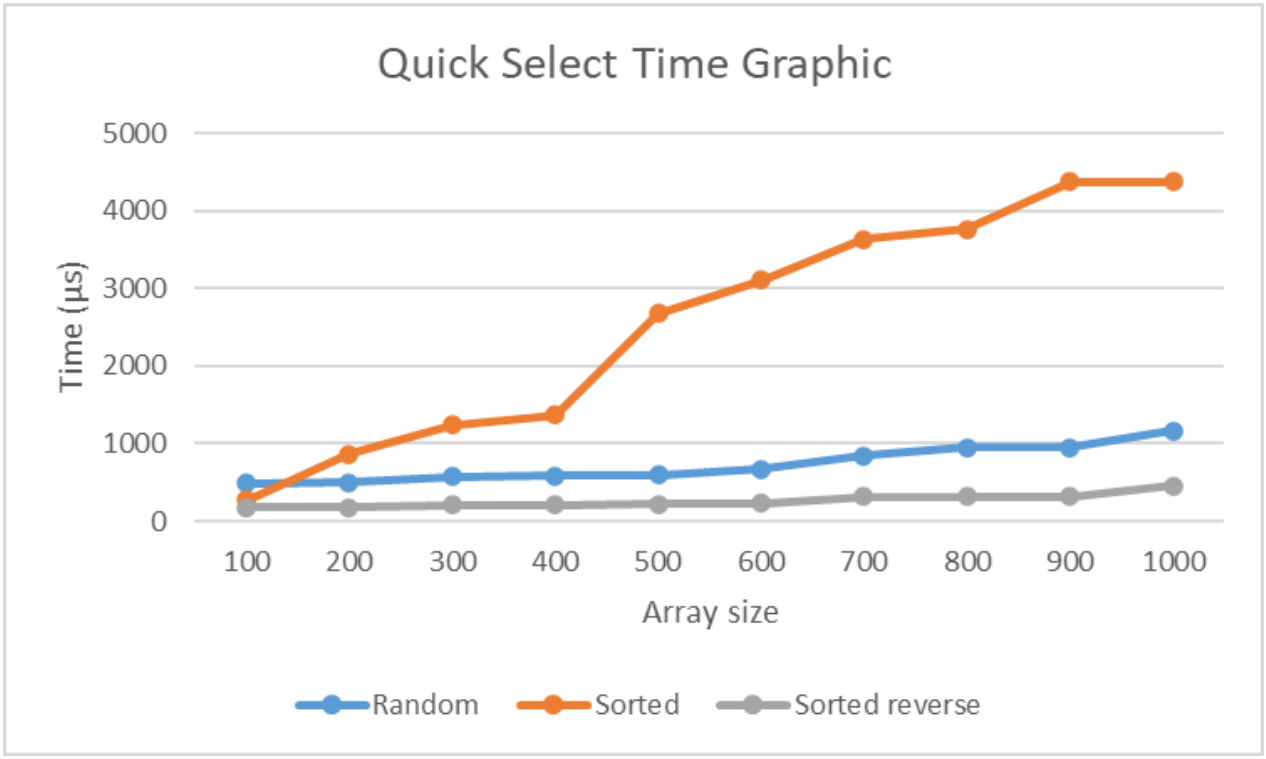
Heap Sort			
Array Size	Random	Sorted	Sorted reverse
1000	910,4	135,7	210,5
2000	910,8	138,9	215,9
3000	954,4	181,5	223
4000	1012,9	199,4	224
5000	1067,1	202,4	231,2
6000	1092,9	215,4	241,5
7000	1163,2	217,4	258,8
8000	1202,5	238,6	278
9000	1310,6	242,7	281,4
10000	1320,2	451,5	295



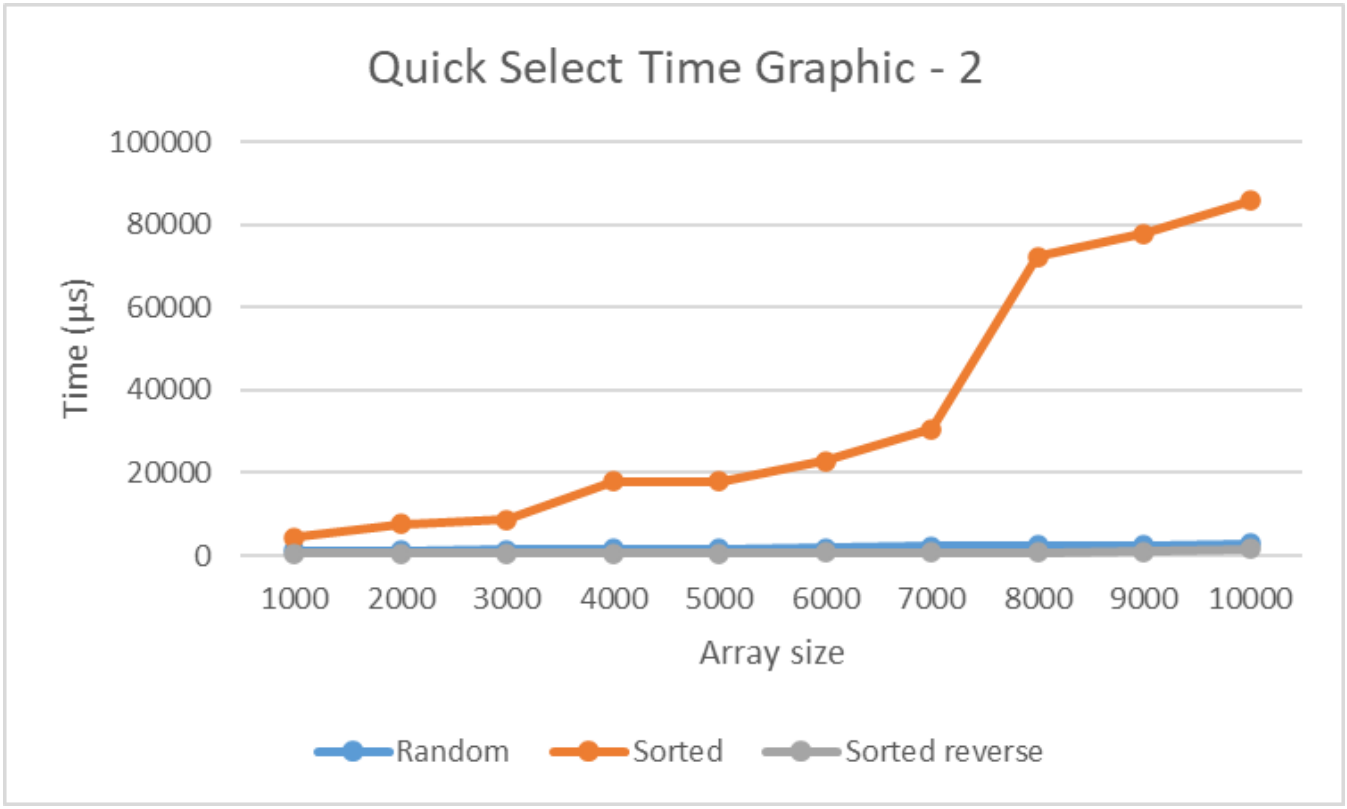
Heap Sort			
Array Size	Random	Sorted	Sorted reverse
10000	1320,2	451,5	295
20000	6282,7	2693,9	296,4
30000	7090,2	2929,7	351,5
40000	13202,5	5978,1	357,6
50000	13416	6891,8	525,3

QUICK SELECT (the pivot element as the first element in an array):

According to the results we obtained, the quick select algorithm performs the longest in a sorted array whose first element is taken as a pivot. Based on this, we can deduce that the worst case scenario occurs when a sorted array is used.
Again, according to our observations, when a decreased sorted array is used, less time is spent compared to the others. On this basis, we can say that the quick select algorithm shows the best performance in a decreased array whose first element is selected as the pivot.



Quick Select			
Array Size	Random	Sorted	Sorted reverse
100	493,6	277,6	176,7
200	504,1	862,5	179,7
300	576,9	1245,4	211,1
400	590,3	1365,3	211,1
500	596,7	2682,1	223,7
600	669,9	3107,4	232,6
700	842,9	3639,3	315
800	948	3766,5	315,7
900	955	4374,3	320,7
1000	1170,7	4381,4	461,4



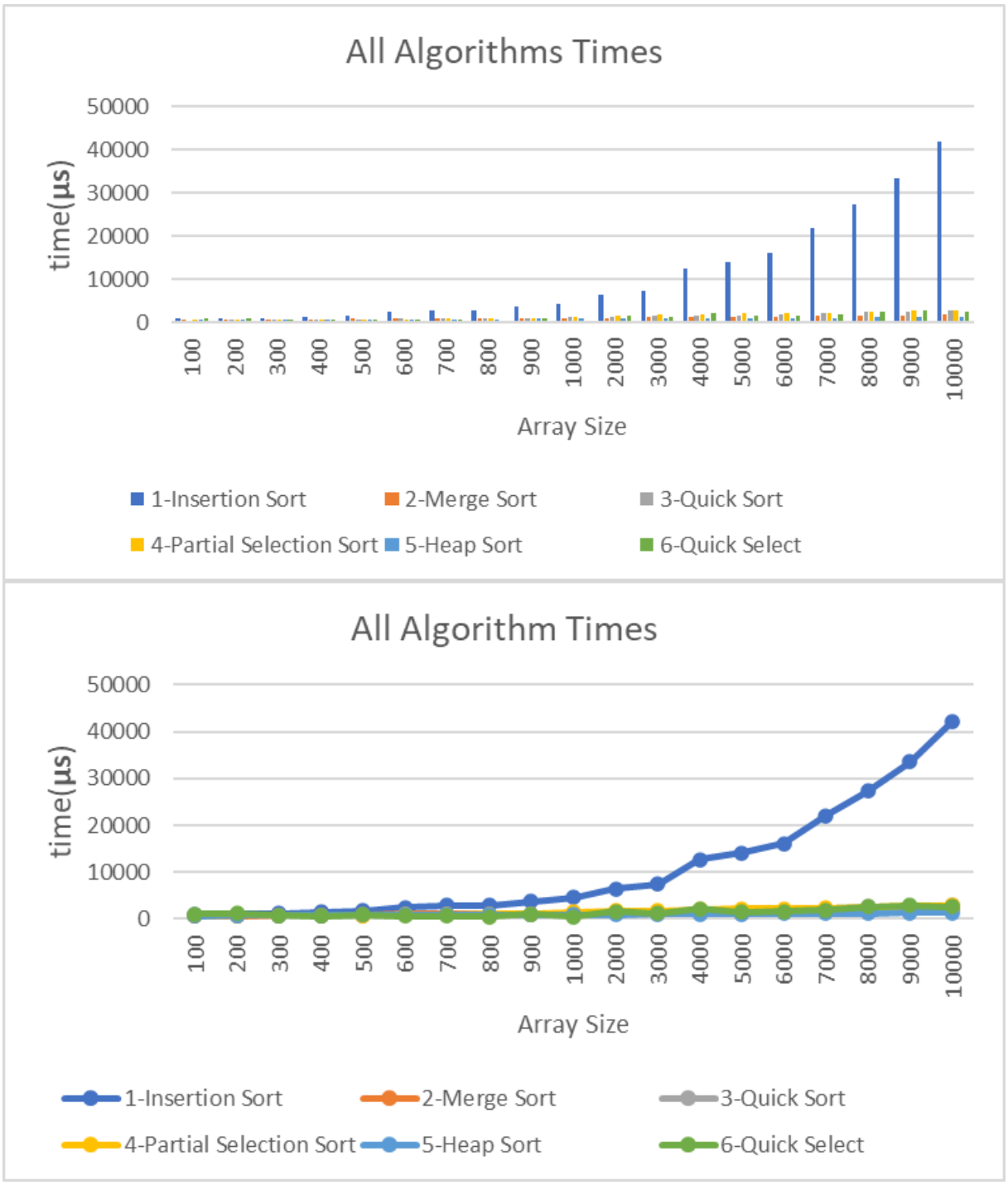
Quick Select			
Array Size	Random	Sorted	Sorted reverse
1000	1170,7	4381,4	461,4
2000	1200,9	7549,9	494,5
3000	1483,4	8705,3	583
4000	1520,7	17830,1	600,1
5000	1566,3	18012	622,9
6000	1797	22986,2	700,4
7000	2178,1	30632,6	814,1
8000	2528,4	72222,6	818,9
9000	2534,5	78006,1	887,3
10000	2878,5	85847,9	1647,4

CONCLUSION:

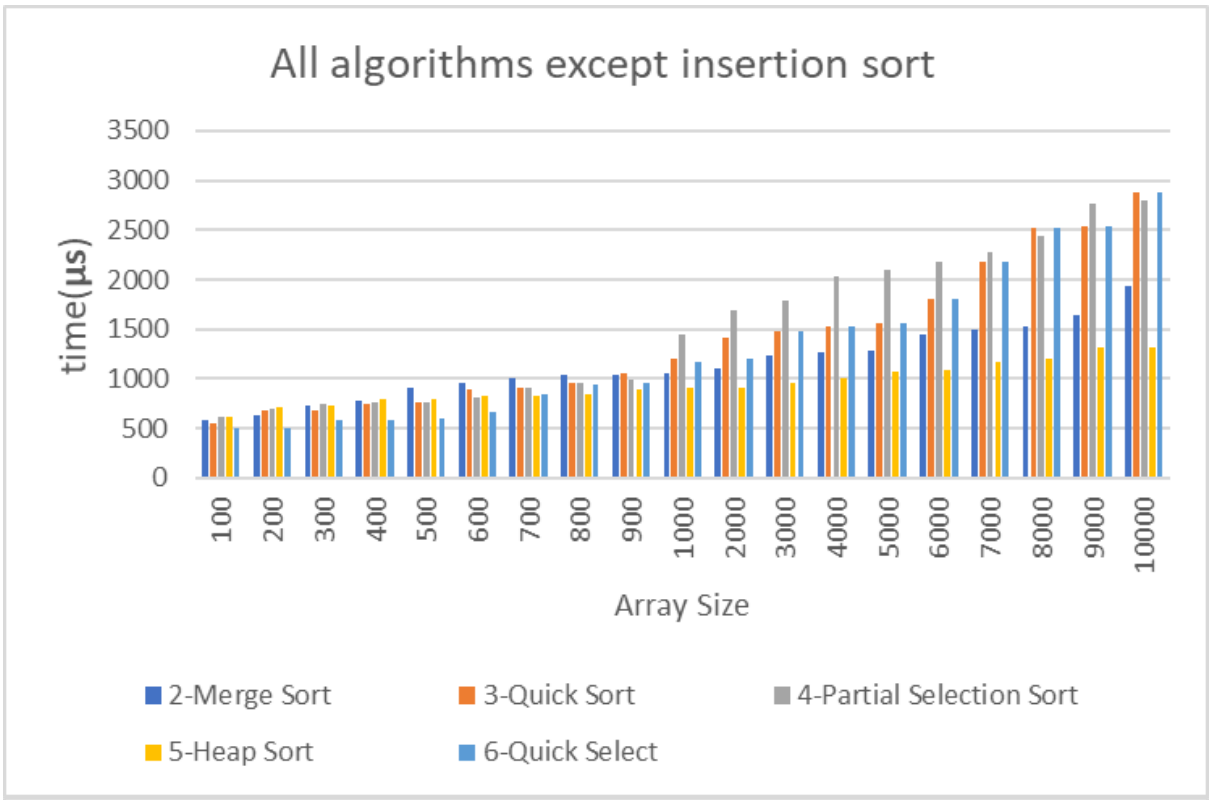
To all appearances, the large input heap sort is better than others. This algorithm is also consistent and uses low memory, unlike the Merge Sort or Quick Sort. But Heap Sort is not very efficient at complex data so the other choice is quick sort for the complex datas. But we saw that from our tests, Quick sort is a recursive algorithm and large datas can't be handled. It couldn't handle 20000 array sizes. If we want to sort and find the K-th smallest element in the array, partial selection sort is the best choice for that. But it can't handle large data too and Quick sort is better than this algorithm. Here is the comparison of all algorithms and we can clearly see that insertion sort is the worst sort algorithm for sorting.

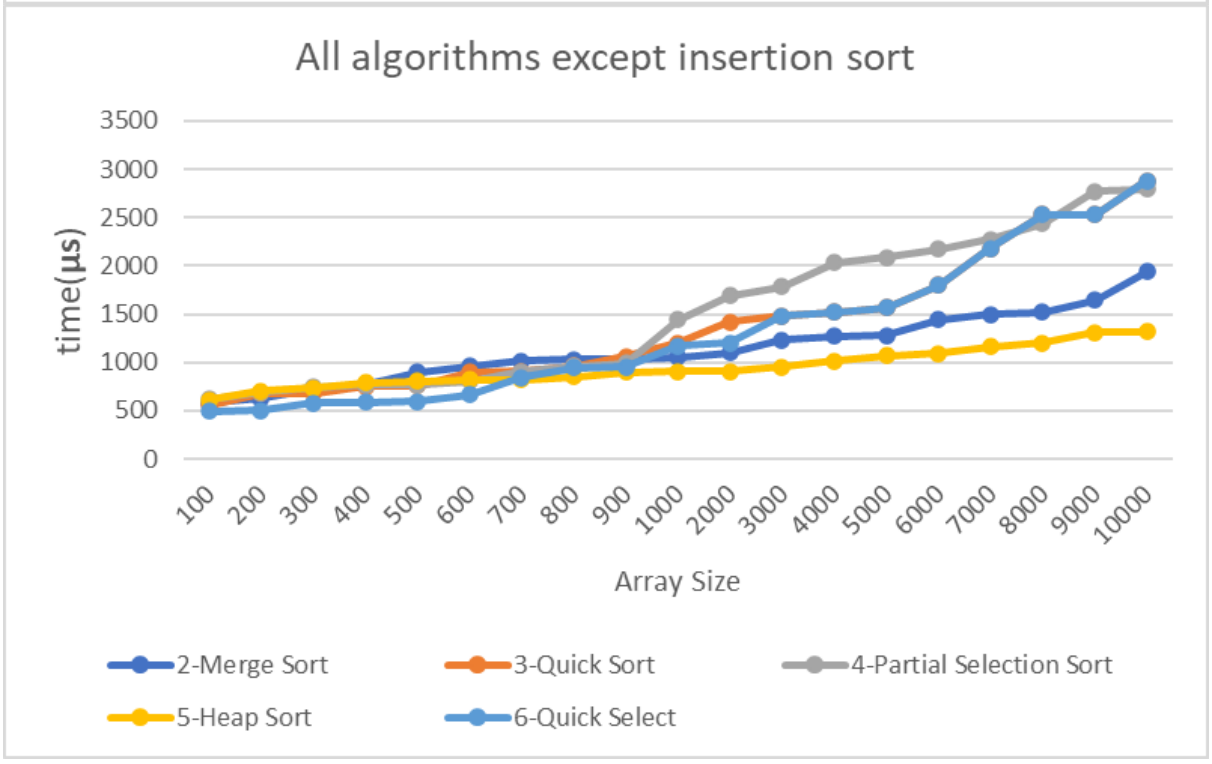
Array Size	Insertion Sort	2-Merge Sort	3-Quick Sort	4-Partial Selection Sort	5-Heap Sort	6-Quick Select
100	990	583,9	557,6	622	612,4	955
200	1049	628	680,5	690	708,7	1170,7
300	1169	726,6	681,9	747,5	735,7	669,9
400	1454	776,6	751,4	765,1	793	590,3
500	1765	901,8	761,8	767,5	802,8	842,9
600	2398	964,1	898,1	805,7	822,4	576,9
700	2819	1013,5	910,6	905,7	822,8	596,7
800	2922	1031,1	963,8	961,1	849	504,1
900	3662	1039,5	1061,3	989,5	897,9	948
1000	4505	1053,3	1200,9	1440,7	910,4	493,6
2000	6412	1103,3	1419,9	1695,1	910,8	1520,7
3000	7386	1235,8	1483,4	1787,8	954,4	1200,9
4000	12627	1268,7	1520,7	2037,3	1012,9	2178,1
5000	14059	1277,8	1566,3	2091,1	1067,1	1483,4
6000	16061	1444,1	1797	2175,2	1092,9	1566,3
7000	22065	1499	2178,1	2272,9	1163,2	1797
8000	27243	1523,4	2528,4	2440,3	1202,5	2528,4
9000	33500	1642,5	2534,5	2771,2	1310,6	2878,5
10000	42044	1942	2878,5	2797,9	1320,2	2534,5

Time of all algorithm tables (µs)



We seperated the insertion sort from the below graphics to see other algorithm's performance. So as we mentioned before for the large dataset, partial selection is not useful except k's value is smaller. Here is k is small and we can see that it's faster than others despite large dataset.


All algorithms except insertion sort
time(µs)
3500
3000
2500
2000
1500
1000
500
0
100
200
300
400
500
600
700
800
900
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Array Size
2-Merge Sort
3-Quick Sort
4-Partial Selection Sort
5-Heap Sort
6-Quick Select


All algorithms except insertion sort
time(µs)
3500
3000
2500
2000
1500
1000
500
0
100
200
300
400
500
600
700
800
900
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Array Size
2-Merge Sort
3-Quick Sort
4-Partial Selection Sort
5-Heap Sort
6-Quick Select

DIVISION OF LABOR:

As 3 group members, we divided all algorithms into 2 algorithms for each person. After preparing the appropriate codes and research, we met and discussed the appropriate inputs together and identified the inputs. We did the tests together and tabulated them. We focused on the results and compiled our results and tables into reports.

REFERENCES:

1. https://www.geeksforgeeks.org/insertion-and-deletion-in-heaps/
2. https://www.geeksforgeeks.org/merge-sort/
3. https://www.geeksforgeeks.org/insertion-sort/
4. https://www.geeksforgeeks.org/quickselect-algorithm/
5. https://www.geeksforgeeks.org/quick-sort/
6. https://tr.wikipedia.org/
7. https://stackoverflow.com/