Data Structures 2

Project 1

Group :

1- Aly Ahmed Aly Aboelnasr - 6511

2- Fares Mohamed ElSayed Ramdan – 6537

3- Maged Magdy Mohamed Zearban – 6395

Project Description :

The Project is designed to calculate the time taken to sort random arrays of different sizes (1,000 – 500,0000) , using 6 different sorting techniques to identify the best sorting option . 3 of the sorting techniques have a complexity of O(n^2) and 3 have a complexity O(n log(n)) .

Pseudo Code :

Selection Sort:

```
SelectionSort(A,n)
{ for i ← 0 to n-2
{ iMin ← i
for j ← i+1 to n-1
{ if(A[j] < A[iMin]) iMin ← j }
if (i != iMin) Swap(A[i] ,A[iMin]) }
}
```

Bubble Sort:

```
BubbleSort(A,n)
{
for k ← 1 to n-1
{
for i ← 0 to n-2
{
if(A[i] > A[i+1])
{Swap(A[i] , A[i+1]) }
}
}
}
```

Insertion Sort :

```
InsertionSort(A,n)
{
for i ← 1 to n-1
{ key = A[i]
hole ← i
while(hole > 0 && A[hole-1] > key)
{
A[hole] = A[hole-1]
hole ←hole-1
}
A[hole] ← key
}
}
```

Merge Sort :

```
merge (L,R,A)
{ nL← Length (L); nR← Length (R);
i← 0;j← 0;k← 0;
While (i< nL && j< nR)
{ if (L[i]<=R[j])
{A[k] ←L[i]; i ←i+1; k ←k+1;}
else {A[k] ←R[j]; j ←j+1; k ←k+1;}
}
While (i< nL) {A[k] ←L[i]; i ←i+1; k ←k+1;}
While (j< nR) {A[k] ←R[j]; j ←j+1; k ←k+1;}
}
mergesort(theArray, first, last)
if (first < last) {
mid = (first + last)/2;
// index of midpoint
mergesort(theArray, first, mid);
mergesort(theArray, mid+1, last);
for i ← 0 to mid-1
Left[i] ←theArray[i]
for i ← mid to n-1
Right[i-mid] ←theArray[i]
// merge the two halves
merge (Left, Right, theArray)
}
```

Quick Sort :

```
partition(theArray, first, last)
{ lastS1 ← first
firstUnknown ← first + 1
while (firstUnknown <= last)
{ if (theArray[firstUnknown] < theArray[first])
{ lastS1 ← lastS1 + 1
swap(theArray[firstUnknown], theArray[lastS1])
}
firstUnknown ← firstUnknown + 1
}
swap(theArray[first], theArray[lastS1])
return lastS1
}
quicksort(theArray,first,last)
{ if (first < last)
{ q ← partition(theArray,first,last)
quicksort(theArray,first,q-1)
quicksort(theArray,q+1,last)
}
}
```

Heap Sort :

```
Heapsort(A) {
  BuildHeap(A)
  for i <- length(A) downto 2 {
    exchange A[1] <-> A[i]
    heapsize <- heapsize -1
    Heapify(A, 1)
}
```

```
BuildHeap(A) {
  heapsize <- length(A)
  for i <- floor( length/2 ) downto 1
    Heapify(A, i)
}
```

```
Heapify(A, i) {
  le <- left(i)
  ri <- right(i)
  if (le<=heapsize) and (A[le]>A[i])
    largest <- le
  else
    largest <- i
  if (ri<=heapsize) and (A[ri]>A[largest])
    largest <- ri
  if (largest != i) {
    exchange A[i] <-> A[largest]
    Heapify(A, largest)
  }
}
```

Sample Runs :

1,000(Time in milliseconds):

```
Sorting 1000 Elements :
Time Elapsed Quick Sort= 0
Time Elapsed Merge Sort= 0
Time Elapsed Heap Sort= 1
Time Elapsed Insertion Sort= 1
Time Elapsed Selection Sort= 1
Time Elapsed Bubble Sort= 2
```

Sort 1000	Try 1	Try 2	Try3	Try 4	Try 5	Average	Time in Seconds
Quick	1	0	0	0	0	0.20	0.00
Merge Sort	1	1	1	1	1	1.00	0.00
Heap Sort	0	0	0	0	0	0.00	0.00
Insertion	3	1	1	1	1	1.40	0.00
Selection	3	1	1	1	1	1.40	0.00
Bubble	3	3	2	3	1	2.40	0.00

10,000 **<u>(Time in milliseconds)</u> :**

```
Sorting 10000 Elements :
Time Elapsed Quick Sort= 2
Time Elapsed Merge Sort= 2
Time Elapsed Heap Sort= 2
Time Elapsed Insertion Sort= 33
Time Elapsed Selection Sort= 67
Time Elapsed Bubble Sort= 155
```

Sort 10000						
Quick	2	1	1	1	2	1.40
Merge Sort	1	3	3	2	2	2.20
Heap Sort	1	3	2	1	1	1.60
Insertion	33	36	34	34	34	34.20
Selection	67	80	66	65	65	68.60
Bubble	172	170	158	157	157	162.80

50,000 (Time in milliseconds) :

```
Sorting 50000 Elements :
Time Elapsed Quick Sort= 6
Time Elapsed Merge Sort= 10
Time Elapsed Heap Sort= 8
Time Elapsed Insertion Sort= 881
Time Elapsed Selection Sort= 1655
Time Elapsed Bubble Sort= 4611
```

Sort 50000						
Quick	4	4	4	4	5	4.20
Merge Sort	13	9	9	9	10	10.00
Heap Sort	8	8	9	9	8	8.40
Insertion	829	812	842	851	865	839.80
Selection	1644	1621	1608	1646	1641	1632.00
Bubble	4578	4550	4490	4533	4542	4538.60

100,000 (Time in milliseconds) :

```
Sorting 100000 Elements :
Time Elapsed Quick Sort= 7
Time Elapsed Merge Sort= 16
Time Elapsed Heap Sort= 17
Time Elapsed Insertion Sort= 3316
Time Elapsed Selection Sort= 6560
Time Elapsed Bubble Sort= 18498
```

Sort 100000						
Quick	8	10	7			8.33
Merge Sort	6	24	16			15.33
Heap Sort	17	17	17			17.00
Insertion	3322	3285	3270			3292.33
Selection	6488	6474	6416			6459.33
Bubble	18135	18244	18263			18214.00

200,000 (Time in milliseconds) :

Sort 200000	
Quick	14
Merge Sort	35
Heap Sort	35
Insertion	12885
Selection	25798
Bubble	73126

300,000 (Time in milliseconds):

Sort 300000	
Quick	20
Merge Sort	52
Heap Sort	56
Insertion	29046
Selection	57730
Bubble	165221

500,000 (Time in milliseconds):

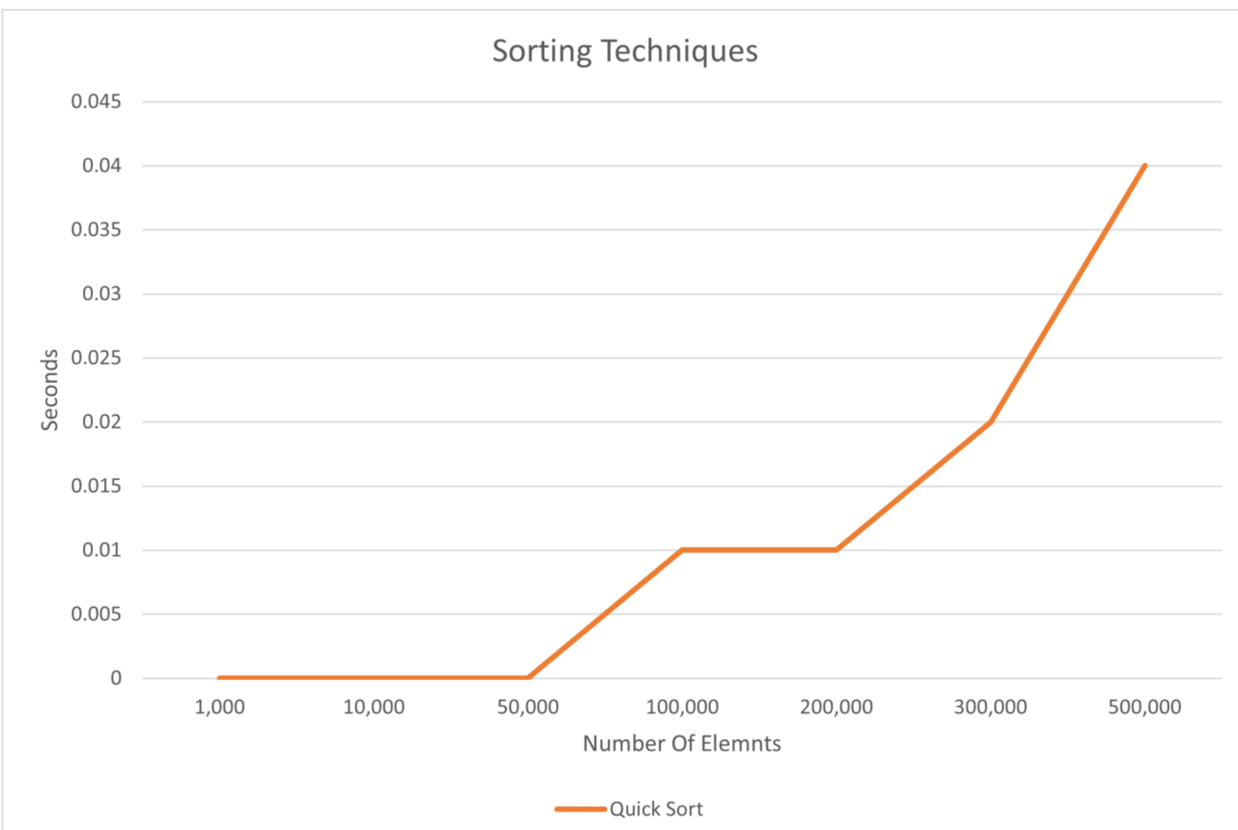
Sort 500000	
Quick	39
Merge Sort	95
Heap Sort	119
Insertion	81131
Selection	161643
Bubble	464746

Final table (time in seconds) :

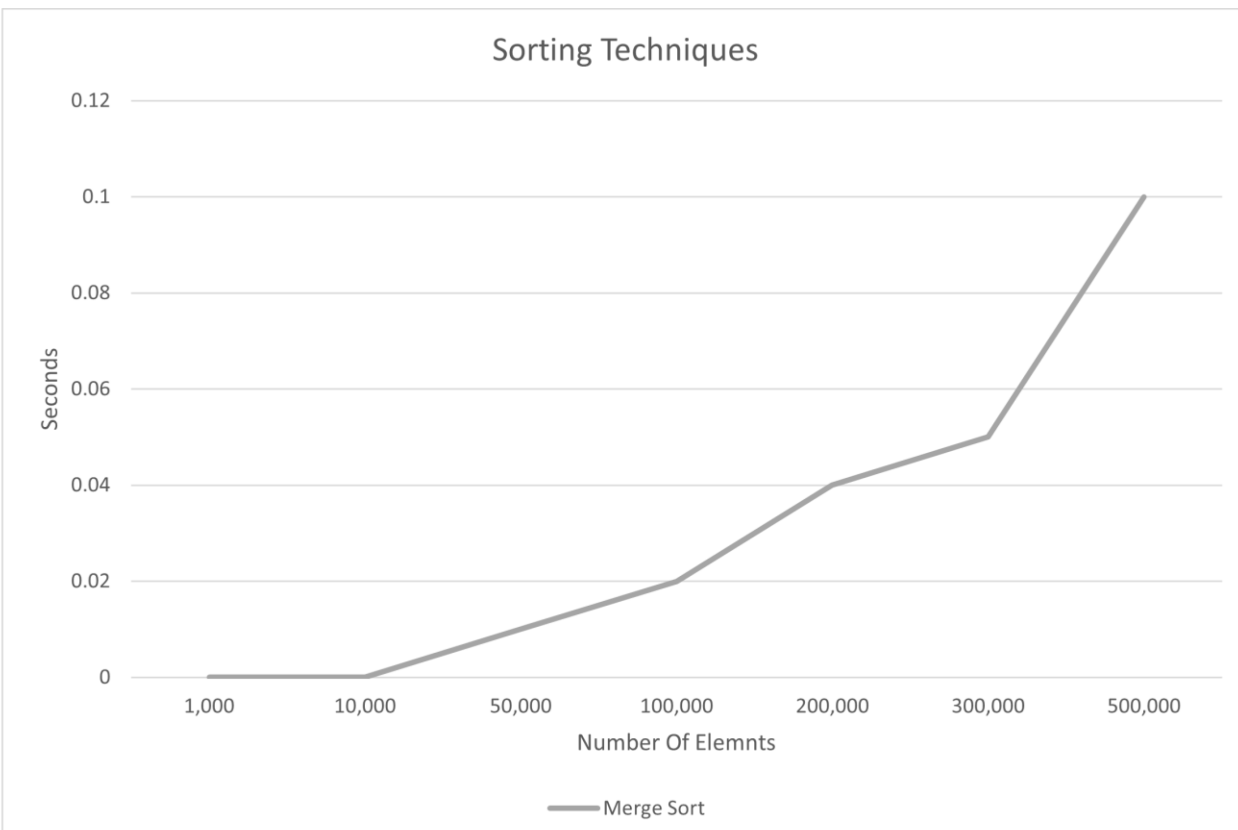
Sorting Techs	Quick Sort	Merge Sort	Heap Sort	Insertion	Selection	Bubble
1,000	0.00	0.00	0.00	0.00	0.00	0.00
10,000	0.00	0.00	0.00	0.03	0.07	0.16
50,000	0.00	0.01	0.01	0.84	1.63	4.54
100,000	0.01	0.02	0.02	3.29	6.46	18.21
200,000	0.01	0.04	0.04	12.89	25.8	73.13
300,000	0.02	0.05	0.06	29.05	57.73	165.22
500,000	0.04	0.1	0.12	81.13	161.64	464.75

Graphs :

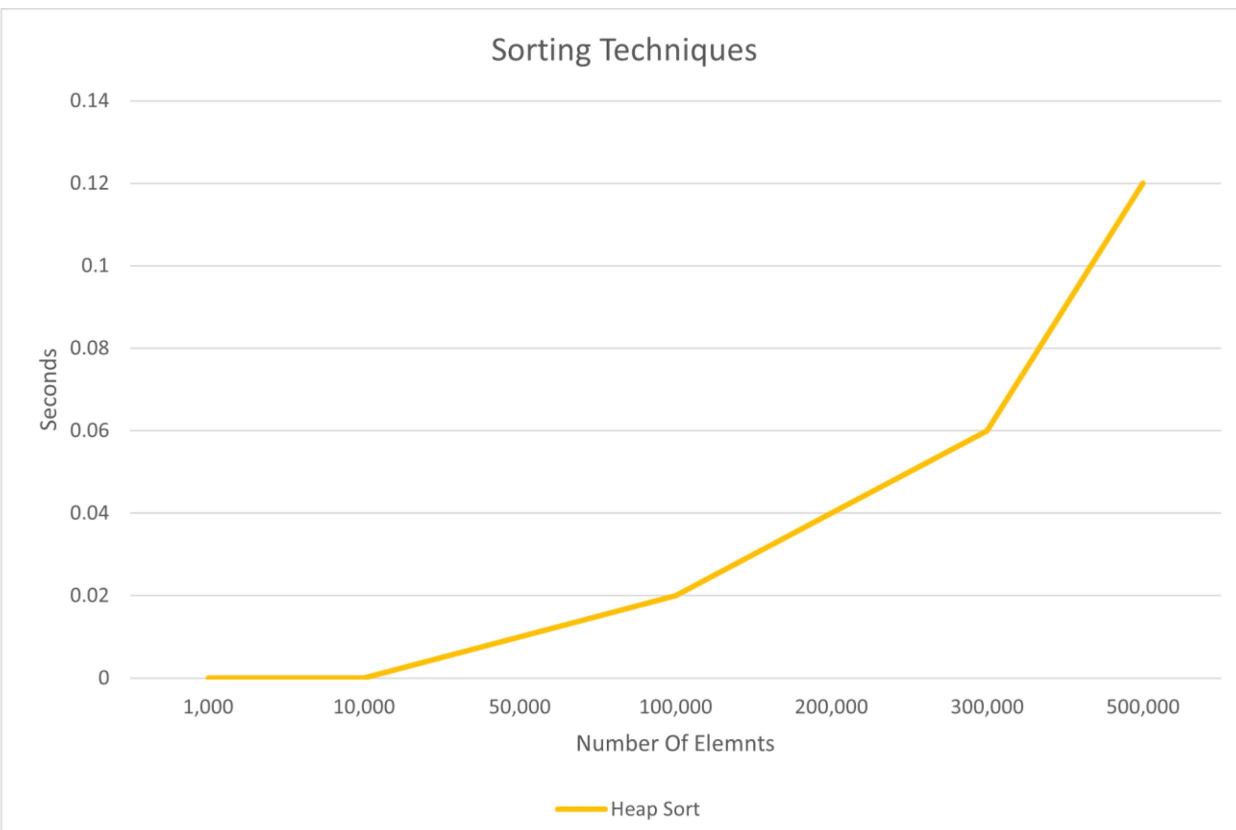
Quick Sort O(n log(n)) :



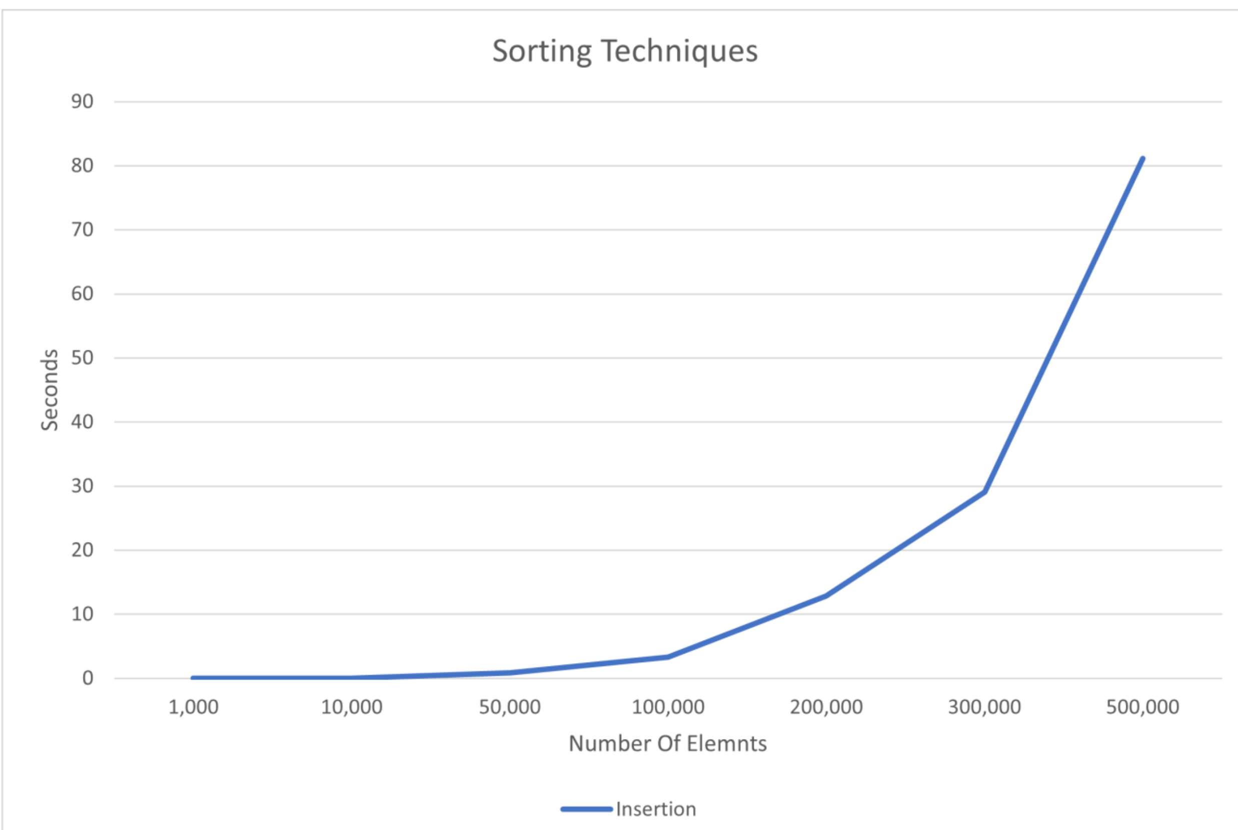
Merge Sort O(n log(n)) :



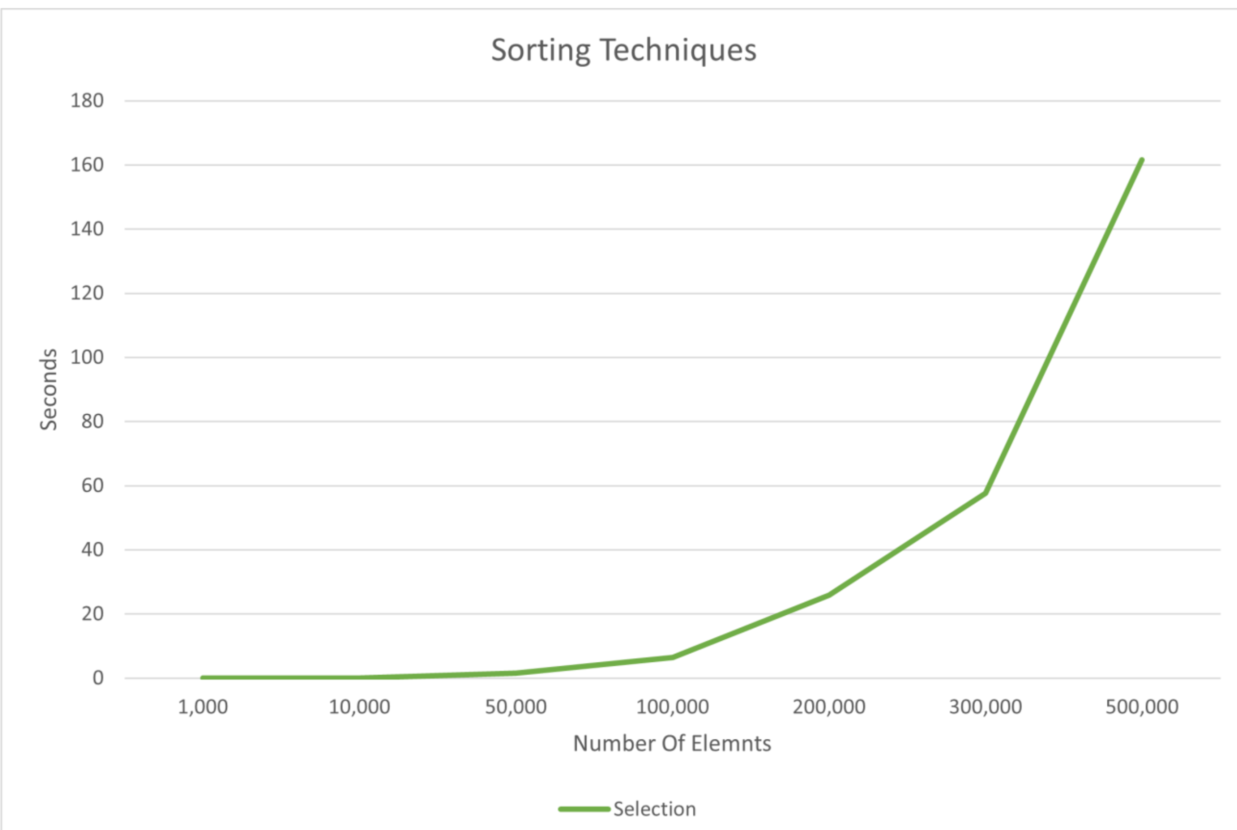
Heap Sort O(n log(n)) :



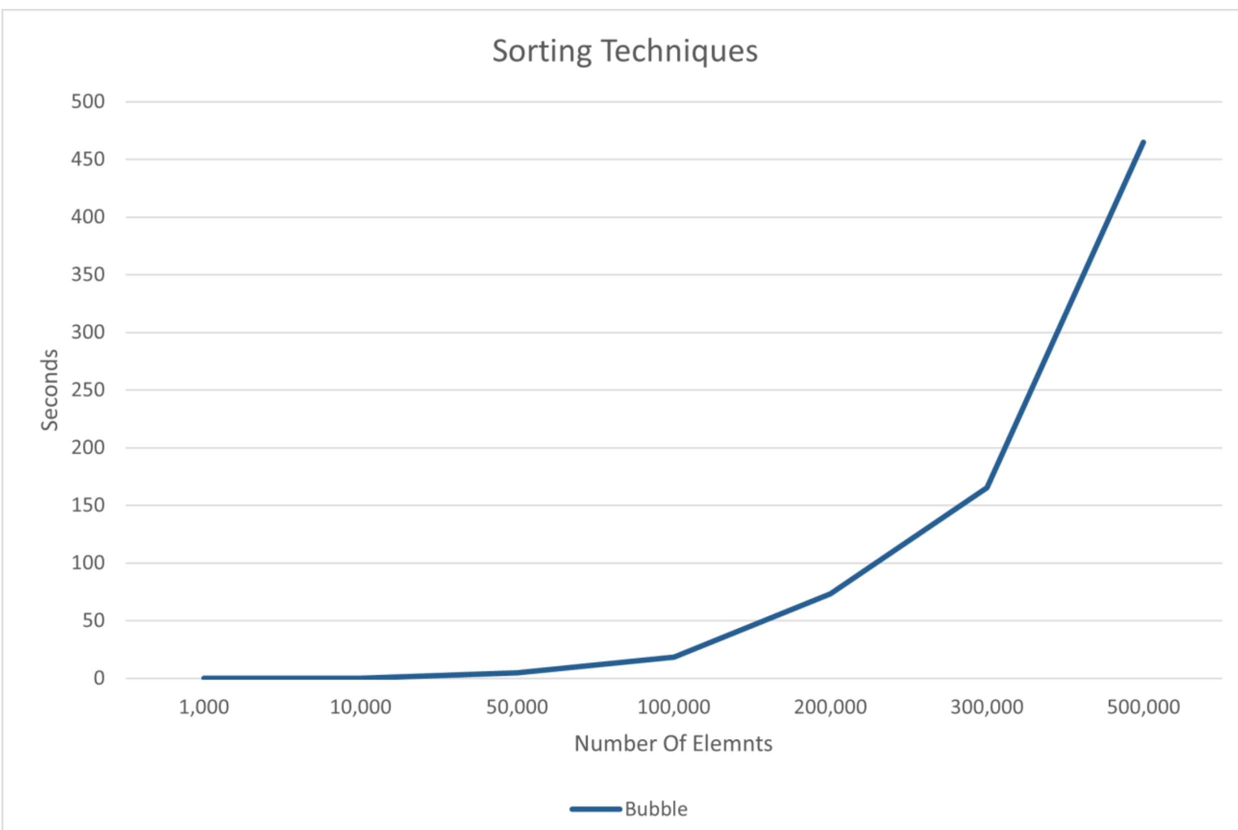
Insertion Sort O(n^2) :



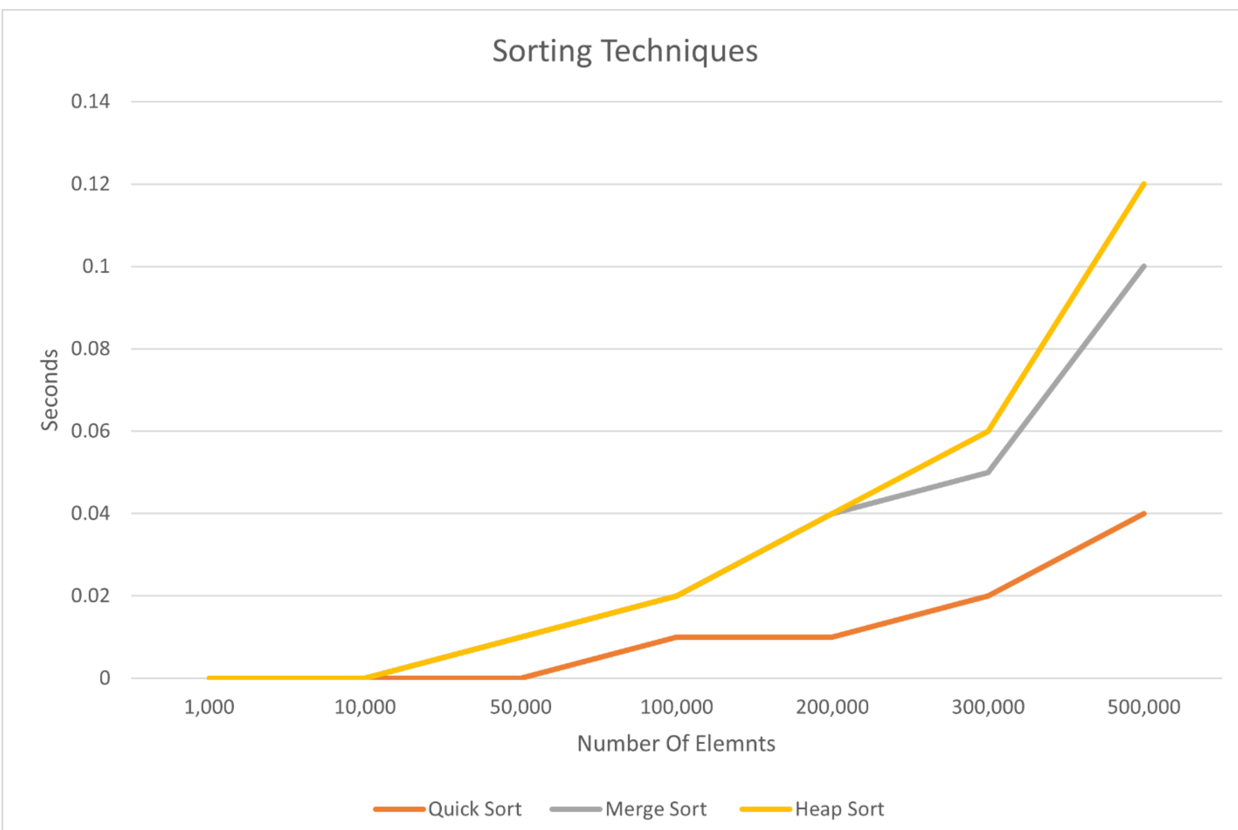
Selection Sort O(n^2) :



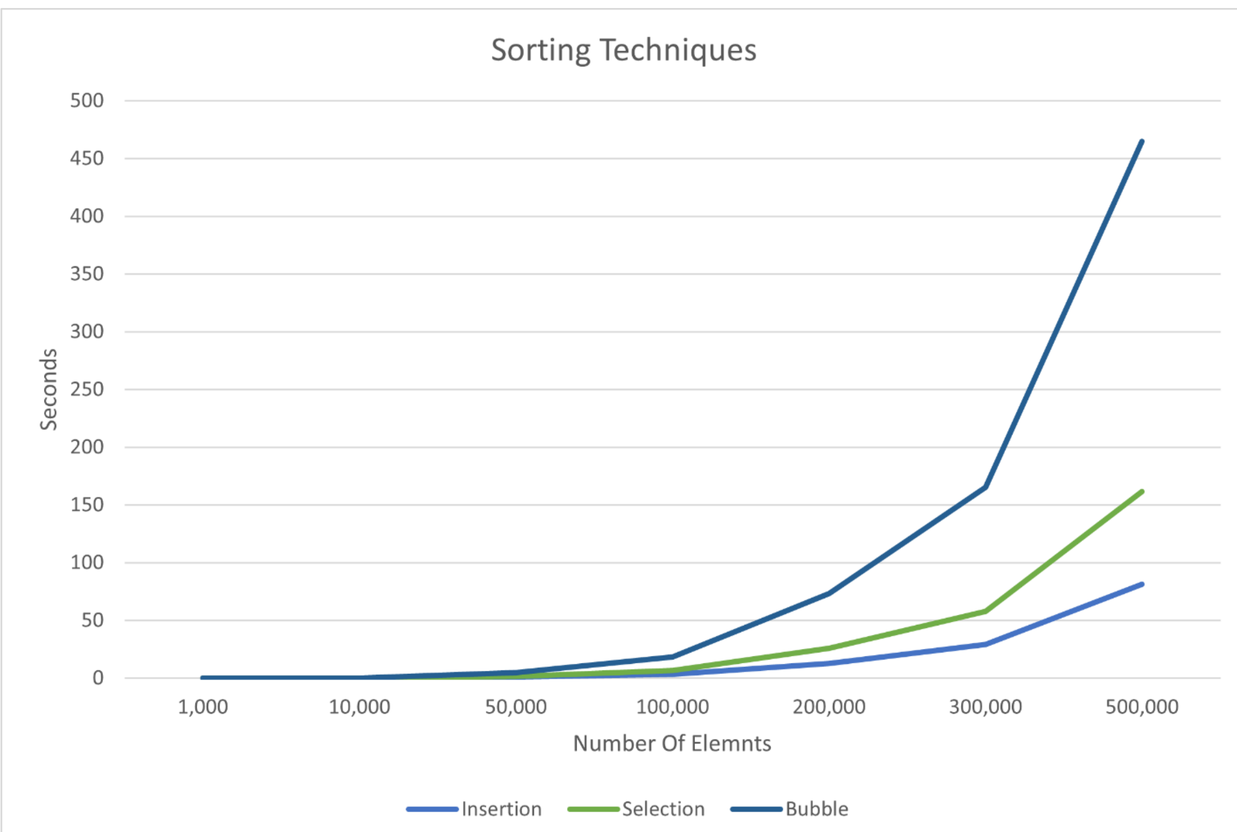
Bubble Sort O(n^2) :



3 O(n log(n)) :


Sorting Techniques
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
Seconds
1,000
10,000
50,000
100,000
200,000
300,000
500,000
Number Of Elemnts
Quick Sort
Merge Sort
Heap Sort

3 O(n^2) :


Sorting Techniques
500
450
400
350
300
250
200
150
100
50
0
Seconds
1,000
10,000
50,000
100,000
200,000
300,000
500,000
Number Of Elemnts
Insertion
Selection
Bubble

All 6 Sorting Techniques :