Sorting techniques

Names : *Ahmed Tarek Ahmed 6646*

Hazem Hussien Elsayed 6152

Ramez Ragaa Adly Gerges 6135

1-Description of the program

we compare the running time performance of each algorithm using (**java 11**) by generating a dataset of random numbers and plot relation between the time and size .

The program takes 2 (or more) command line arguments. The first argument takes the number of times the test is repeated for each array size. The remaining arguments are the array sizes to be tested. Ex. "Main 4 50 60 70" sorts 4 random arrays of length 50, 4 random arrays of length 60 and 4 random arrays of length 70 using each sorting algorithm and prints out the average time each algorithm took for each array size.

the program consists of six sorting techniques

Bubble Sort: is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order and **complexity are O(n^2)**.

selection sort: It finds the first smallest element, swaps it with the first element of the unordered list, finds the second smallest element, swaps it with the second element of the unordered list, and Similarly, continues to sort the rest of the elements and **complexity are O(n^2)** .

Insertion sort is a simple sorting algorithm that is appropriate for small inputs, The list is divided into two parts: sorted and unsorted In each pass, the first element of the unsorted part is picked up, transferred to the sorted sublist, and inserted at the appropriate place and **complexity are O(n^2)** .

Merge sort is a Divide and Conquer algorithm algorithm that divides the list into halves,Sort each halve recursively, and Then merges the sorted halves into one sorted array **and complexity are O(nlogn)** .

Divide**:** Partition the array into two subarrays **.**

Conquer**:** Sort the two subarrays by recursive calls.

Quick sort type of Divide and Conquer algorithm just like merge sort. It Picks a random element as a pivot. And does a procedure called Partitioning which places the pivot in its correct position in the array (all elements before it are smaller and all the elements after it are bigger), Then perform the same procedure recursively on the rest of the array until it is sorted.

complexity are O(n^2) in the worst case and O(nlogn) on average .

Heapsort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the minimum element and place the minimum element at the beginning. We repeat the same process for the remaining elements **and complexity are O(nlogn) .**

2-pseudo code

Bubble sort

Array arr   , size of array n

```
Bubble sort(array,n)

{  for i=0 to n-1

        { Flag =0

         For j=0 to n-i-1

              { if(arr[j]>arr[j+1])

                   { Swap (arr[j],arr[j+1])

                     Flag=1

                    }

              }

      If(flag =0 )

          Break

}
```

<u>Selection sort</u>

Array arr   , size of array n

```
Selection sort(arr,n)
{
For i=0 to n-2
   {  min index = i
    For j = i+1 to n-1
        {  if(arr[j]<arr[min index])
                min index = j
        }
      Swap(arr[min index],arr[i])
     }
 }
```

Insertion sort

Array arr   , size of array n

```
  Insertion sort(arr,n)
  {     for i=1 to n-1
        {  key = arr[i]
         j= i-1
        while(j>=0 && arr[j]>key)
           { arr[i+1] =arr[j]
              j = j-1
           }
          arr[j+1]=key
         }
      }
```

Merge sort

Assume arr : array , l : left array , r : right array

Merge(arr, l , r)

```
{   nl=length(l)
    nr=length(r)
     i=0
     j=0
     k=0
    while( i<nl && j <nr)
     {  if( l[i] <= r[j] )
           {  arr[k]= l[i]
              i=i+1
              k=k+1
            }
       Else
       {  arr[k]=r[ j ]
          j=j+1
         k=k+1   }
}
While(i < nl)
{ arr[k] = l[i]
 i=i+1
k=k+1  }
While(i < nr)
{ arr[k] = r[j]
 j=j+1
k=k+1  }
```

first : first index , end : last index

```
merge_ sort(array , first , end )
{  if (first < last)
   Mid = first + end /2
Merge_sort(array ,first,mid)
Merge_sort(array ,mid+1,end)
For i=0 to mid -1
      l[i] =array[i]
for i= mid to n-1
      r[i-mid] = array[i]
merge(array, l , r)
}
```

Quick sort

first : first index , end : last index

```
 partition(arr,first ,end)
  {pivot = arr[end]
    i= end -1
    for j = first   to   end -1
{  if(arr[i]<pivot)
     {    i=i+1
    swap(arr[i],arr[j])
       }
  }
Swap(arr[i+1],a[end])
Return i+1
   }


Quick_sort(arr, first , end)
{ if(first< end)
    {  F= partition(arr , first , end )
        Quick_sort(arr,first, F-1)
         Quick_sort(arr,F+1,end)
    }
}
```

Heap sort

Assume arr : array , i : index

Heapify(arr , i)

```
{  l = left(i)
   r= right(i)
if(l <=arr.heap size &&arr[l]>arr[i])
     largest =l
 else
     largest=i
if(r<=arr.heap size &&arr[r]>arr[largest])
     largest =r
 if(largest!= i)
   swap(arr[i],arr[largest])
heapify(arr,largest)
}
```

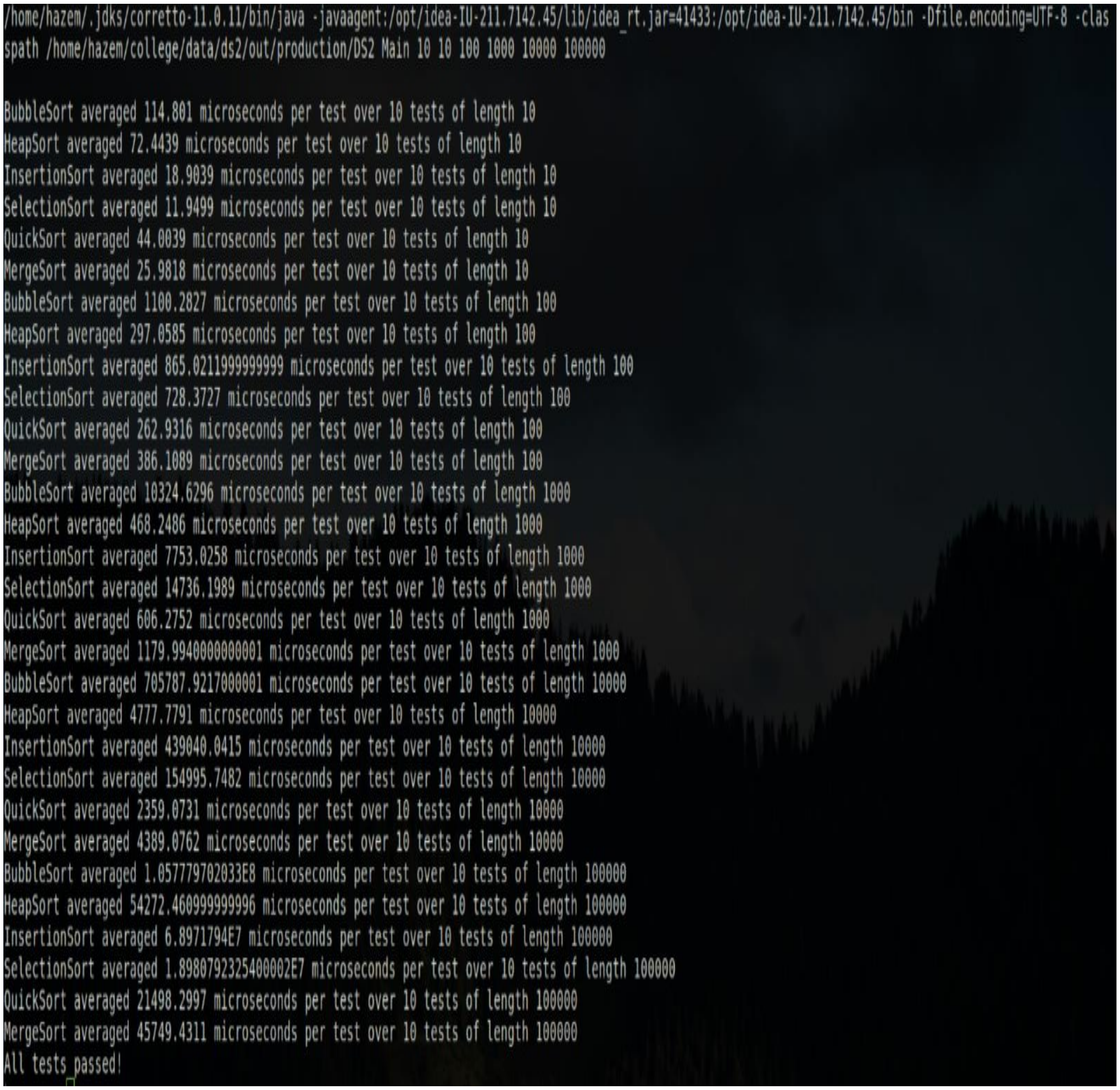
Build_heap(arr)

```
{  arr.heap size =arr.length
   For i= arr.length/2 to 1
     Heapify(arr,i)    }
```

Heap_sort(arr)

```
{ build heap(arr)
For i=arr.length to 2
   Swap(arr[l],a[i])
Arr.heap size= arr.heap size -1
 Heapify(arr , l)  }
```

3-Sample run:

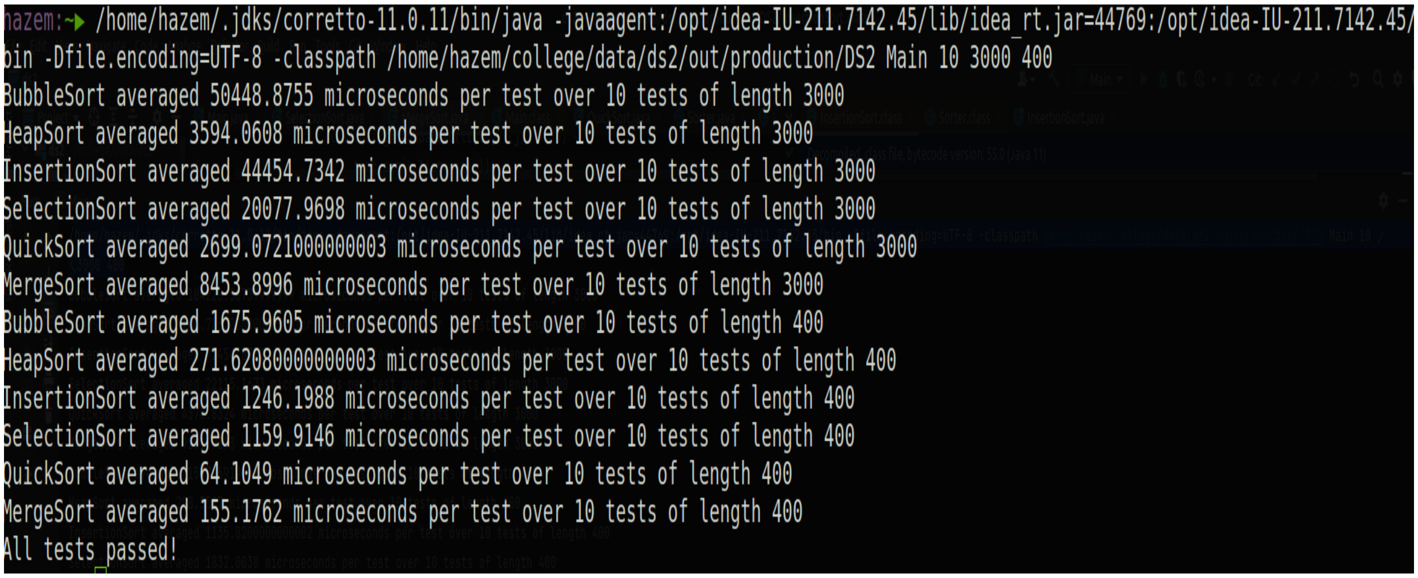


```
/home/hazem/.jdks/corretto-11.0.11/bin/java -javaagent:/opt/idea-IU-211.7142.45/lib/idea_rt.jar=41433:/opt/idea-IU-211.7142.45/bin -Dfile.encoding=UTF-8 -clas
spath /home/hazem/college/data/ds2/out/production/DS2 Main 10 10 100 1000 10000 100000

BubbleSort averaged 114.801 microseconds per test over 10 tests of length 10
HeapSort averaged 72.4439 microseconds per test over 10 tests of length 10
InsertionSort averaged 18.9039 microseconds per test over 10 tests of length 10
SelectionSort averaged 11.9499 microseconds per test over 10 tests of length 10
QuickSort averaged 44.0039 microseconds per test over 10 tests of length 10
MergeSort averaged 25.9818 microseconds per test over 10 tests of length 10
BubbleSort averaged 1100.2827 microseconds per test over 10 tests of length 100
HeapSort averaged 297.0585 microseconds per test over 10 tests of length 100
InsertionSort averaged 865.0211999999999 microseconds per test over 10 tests of length 100
SelectionSort averaged 728.3727 microseconds per test over 10 tests of length 100
QuickSort averaged 262.9316 microseconds per test over 10 tests of length 100
MergeSort averaged 386.1089 microseconds per test over 10 tests of length 100
BubbleSort averaged 10324.6296 microseconds per test over 10 tests of length 1000
HeapSort averaged 468.2486 microseconds per test over 10 tests of length 1000
InsertionSort averaged 7753.0258 microseconds per test over 10 tests of length 1000
SelectionSort averaged 14736.1989 microseconds per test over 10 tests of length 1000
QuickSort averaged 606.2752 microseconds per test over 10 tests of length 1000
MergeSort averaged 1179.9940000000001 microseconds per test over 10 tests of length 1000
BubbleSort averaged 705787.9217000001 microseconds per test over 10 tests of length 10000
HeapSort averaged 4777.7791 microseconds per test over 10 tests of length 10000
InsertionSort averaged 439040.0415 microseconds per test over 10 tests of length 10000
SelectionSort averaged 154995.7482 microseconds per test over 10 tests of length 10000
QuickSort averaged 2359.0731 microseconds per test over 10 tests of length 10000
MergeSort averaged 4389.0762 microseconds per test over 10 tests of length 10000
BubbleSort averaged 1.057779702033E8 microseconds per test over 10 tests of length 100000
HeapSort averaged 54272.460999999996 microseconds per test over 10 tests of length 100000
InsertionSort averaged 6.8971794E7 microseconds per test over 10 tests of length 100000
SelectionSort averaged 1.8980792325400002E7 microseconds per test over 10 tests of length 100000
QuickSort averaged 21498.2997 microseconds per test over 10 tests of length 100000
MergeSort averaged 45749.4311 microseconds per test over 10 tests of length 100000
All tests passed!
```

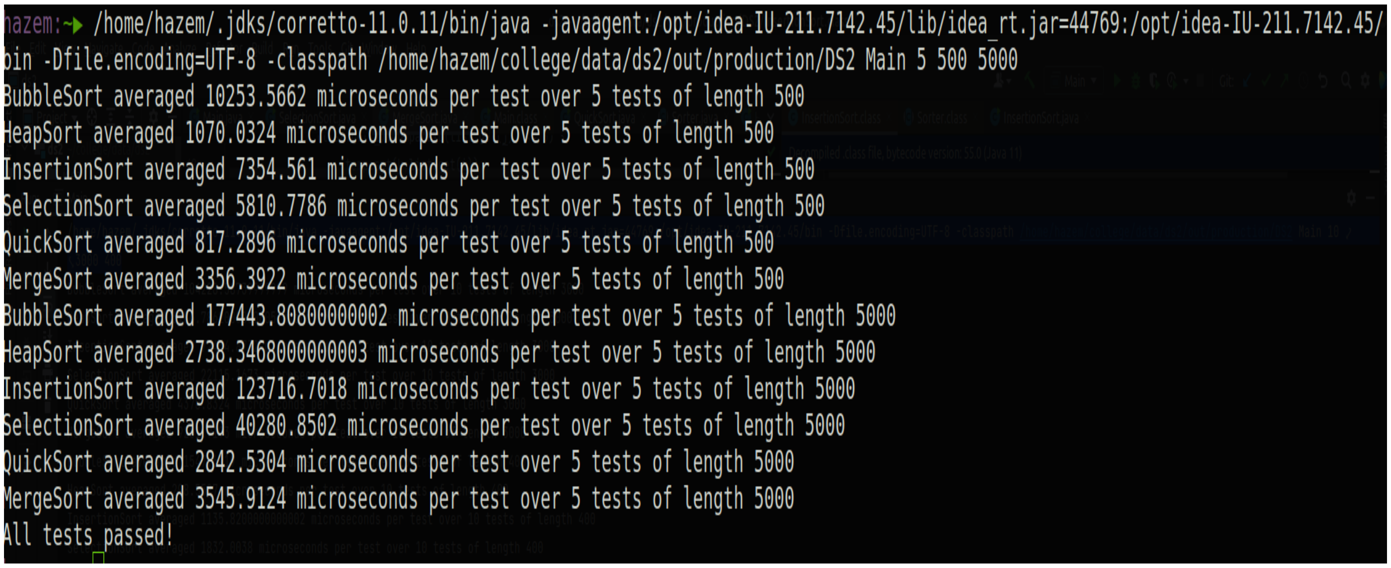
Fig 3.1

More sample runs with smaller arguments given:



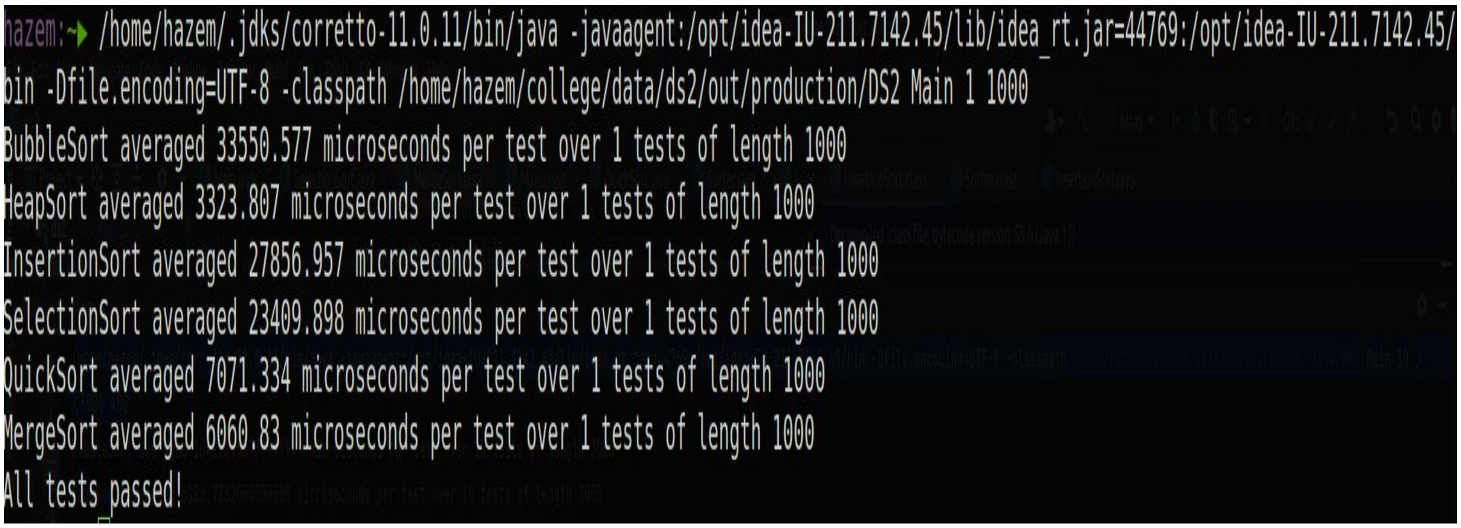
```
hazem:~> /home/hazem/.jdks/corretto-11.0.11/bin/java -javaagent:/opt/idea-IU-211.7142.45/lib/idea_rt.jar=44769:/opt/idea-IU-211.7142.45/
bin -Dfile.encoding=UTF-8 -classpath /home/hazem/college/data/ds2/out/production/DS2 Main 10 3000 400
BubbleSort averaged 50448.8755 microseconds per test over 10 tests of length 3000
HeapSort averaged 3594.0608 microseconds per test over 10 tests of length 3000
InsertionSort averaged 44454.7342 microseconds per test over 10 tests of length 3000
SelectionSort averaged 20077.9698 microseconds per test over 10 tests of length 3000
QuickSort averaged 2699.0721000000003 microseconds per test over 10 tests of length 3000
MergeSort averaged 8453.8996 microseconds per test over 10 tests of length 3000
BubbleSort averaged 1675.9605 microseconds per test over 10 tests of length 400
HeapSort averaged 271.62080000000003 microseconds per test over 10 tests of length 400
InsertionSort averaged 1246.1988 microseconds per test over 10 tests of length 400
SelectionSort averaged 1159.9146 microseconds per test over 10 tests of length 400
QuickSort averaged 64.1049 microseconds per test over 10 tests of length 400
MergeSort averaged 155.1762 microseconds per test over 10 tests of length 400
All tests passed!
```

Fig 3.2



```
hazem:~> /home/hazem/.jdks/corretto-11.0.11/bin/java -javaagent:/opt/idea-IU-211.7142.45/lib/idea_rt.jar=44769:/opt/idea-IU-211.7142.45/
bin -Dfile.encoding=UTF-8 -classpath /home/hazem/college/data/ds2/out/production/DS2 Main 5 500 5000
BubbleSort averaged 10253.5662 microseconds per test over 5 tests of length 500
HeapSort averaged 1070.0324 microseconds per test over 5 tests of length 500
InsertionSort averaged 7354.561 microseconds per test over 5 tests of length 500
SelectionSort averaged 5810.7786 microseconds per test over 5 tests of length 500
QuickSort averaged 817.2896 microseconds per test over 5 tests of length 500
MergeSort averaged 3356.3922 microseconds per test over 5 tests of length 500
BubbleSort averaged 177443.80800000002 microseconds per test over 5 tests of length 5000
HeapSort averaged 2738.3468000000003 microseconds per test over 5 tests of length 5000
InsertionSort averaged 123716.7018 microseconds per test over 5 tests of length 5000
SelectionSort averaged 40280.8502 microseconds per test over 5 tests of length 5000
QuickSort averaged 2842.5304 microseconds per test over 5 tests of length 5000
MergeSort averaged 3545.9124 microseconds per test over 5 tests of length 5000
All tests passed!
```

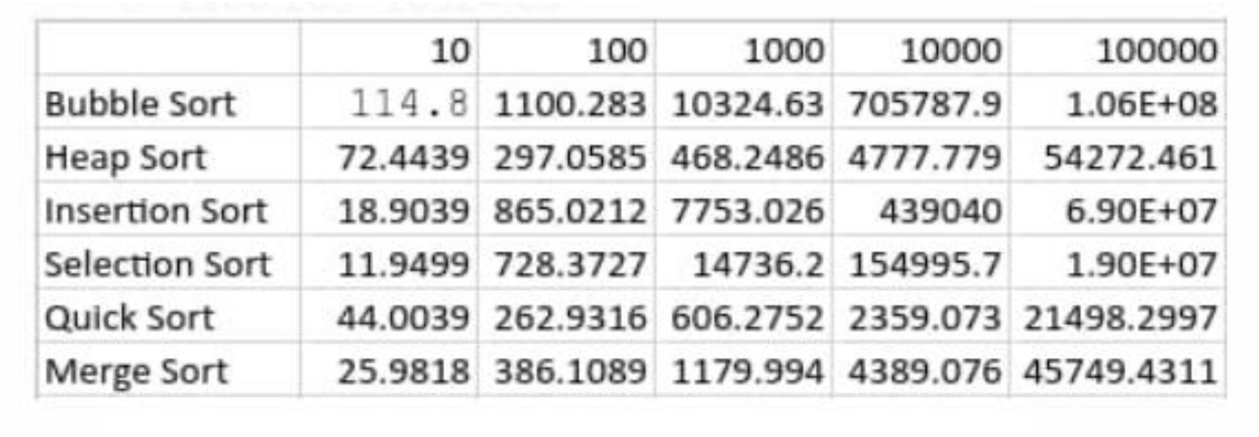
Fig 3.3



```
hazem:~> /home/hazem/.jdks/corretto-11.0.11/bin/java -javaagent:/opt/idea-IU-211.7142.45/lib/idea_rt.jar=44769:/opt/idea-IU-211.7142.45/
bin -Dfile.encoding=UTF-8 -classpath /home/hazem/college/data/ds2/out/production/DS2 Main 1 1000
BubbleSort averaged 33550.577 microseconds per test over 1 tests of length 1000
HeapSort averaged 3323.807 microseconds per test over 1 tests of length 1000
InsertionSort averaged 27856.957 microseconds per test over 1 tests of length 1000
SelectionSort averaged 23409.898 microseconds per test over 1 tests of length 1000
QuickSort averaged 7071.334 microseconds per test over 1 tests of length 1000
MergeSort averaged 6060.83 microseconds per test over 1 tests of length 1000
All tests passed!
```

Fig 3.4

4-graphs



	10	100	1000	10000	100000
Bubble Sort	114.8	1100.283	10324.63	705787.9	1.06E+08
Heap Sort	72.4439	297.0585	468.2486	4777.779	54272.461
Insertion Sort	18.9039	865.0212	7753.026	439040	6.90E+07
Selection Sort	11.9499	728.3727	14736.2	154995.7	1.90E+07
Quick Sort	44.0039	262.9316	606.2752	2359.073	21498.2997
Merge Sort	25.9818	386.1089	1179.994	4389.076	45749.4311

Fig 4.1 table

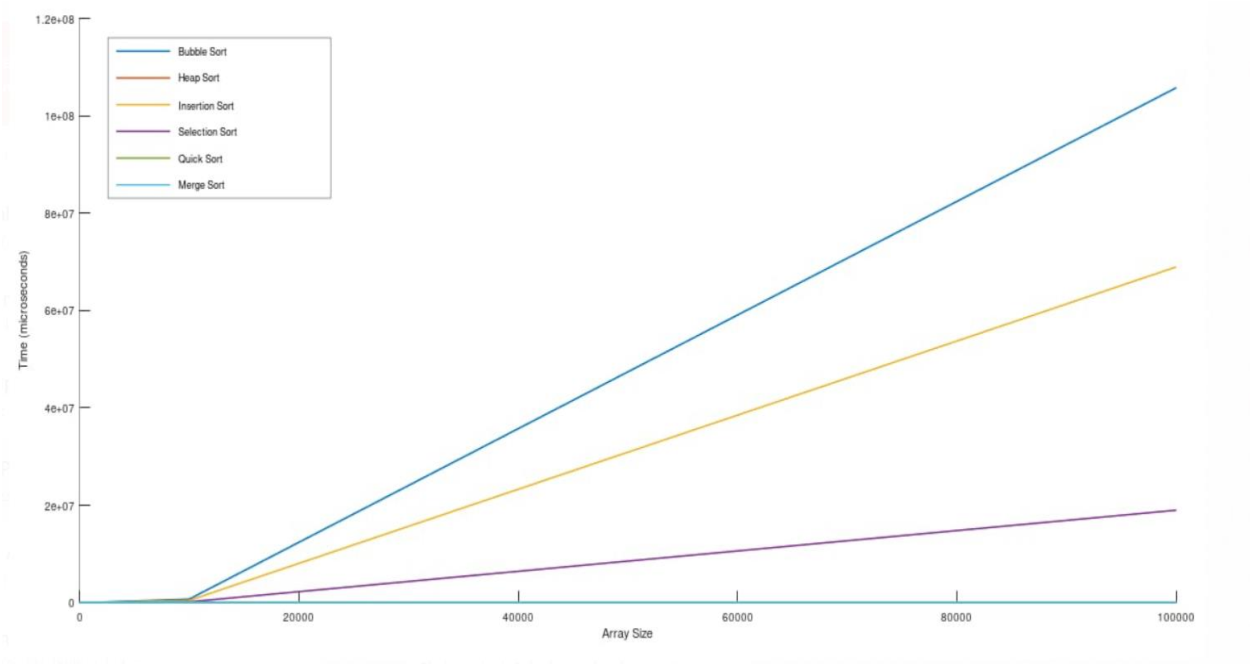


Fig 4.2 original graph

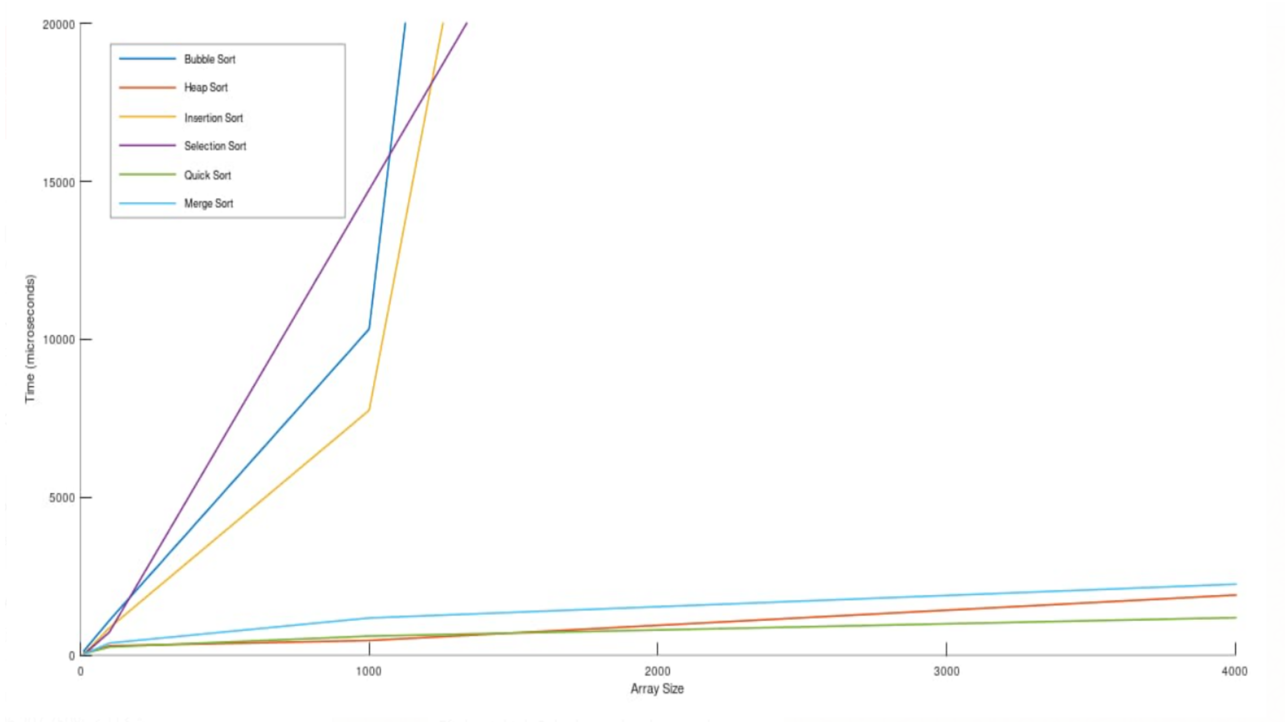


Fig 4.3 zoomed graph

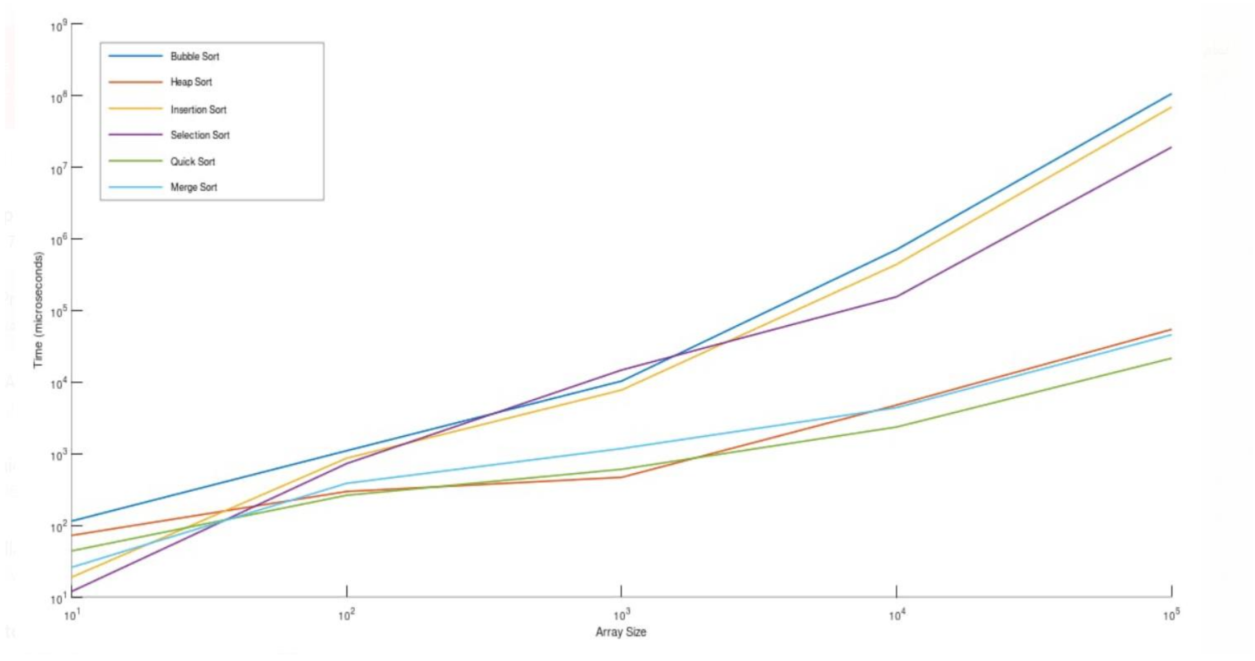


Fig 4.4 log-log graph