**Data structures II**

**Assignment 1**

**Students:**

|  |  |
| --- | --- |
| ID | Name |
| 6668 | Youssef Victor |
| 6656 | Muhanned Basher |
| 6670 | Mohamed Zayton |

**Pseudocode of algorithms**

### Bubble Sort

Let arr be the array we want to sort, and arr[i] the ith (0-indexed) element of the array.

Let N be the length of the array

For i From 0 To N-1

MadeSwaps ← False

For j from 0 to N-2-i

If arr[j] > arr[j+1]

Swap arr[j], arr[j+1]

MadeSwaps ← True

End If

End For

If Not MadeSwaps: Exit For: End If

End For

### ****Insertion Sort****

INSERTION\_SORT(unsorted):

length = unsorted.length

for i = 1 to length:

hole ← i

key ← unsorted[i]

while hole > 0 && unsorted[hole - 1] > key:

unsorted[hole] ← unsorted[hole-1]

hole ← hole - 1

unsorted[hole] ← key

### ****Selection Sort****

SELECTION\_SORT(unsorted):

length = unsorted.length

for i = 0 to length

min\_index = i

for j = i to length

if unsorted[min\_index] > unsorted[j]:

min\_index = j

SWAP( unsorted[i], unsorted[min\_index])

**Pseudocode of algorithms**

### Heap Sort

|  |
| --- |
| Function HeapSort(unsorted)  For i From unsorted.Length / 2 To 0  Heapify(unsorted, length, i)  End For  For i From unsorted.Length - 1 To 0  Swap unsorted[0], unsorted[i]  Heapify(unsorted, i, 0)  End For  End Function  Function Heapify(array, length, i)  IndexOfMaximumElement ← i  IndexOfLeftChild ← 2 \* i + 1  IndexOfRightChild ← 2 \* i + 2  If IndexOfLeftChild < length AndAlso arr[IndexOfLeftChild ] > array[IndexOfMaximumElement] Then  IndexOfMaximumElement ← IndexOfLeftChild  EndIf  If IndexOfRightChild < length AndAlso array[IndexOfRightChild] > arr[IndexOfMaximumElement] Then  IndexOfMaximumElement ← IndexOfRightChild  End If  If i <> IndexOfMaximumElement Then  Swap array[i], arr[IndexOfMaximumElement]  Heapify(array, length, IndexOfMaximumElement)  End If  End Function |

窗体顶端

窗体底端

### Quick Sort

QUICK\_SORT\_LAST\_PIVOT(unsorted):

QUICK\_SORT(unsorted, 0, len(unsorted), GET\_PIVOT\_LAST)

QUICK\_SORT\_RANDOM\_PIVOT(unsorted):

QUICK\_SORT(unsorted, 0, len(unsorted), GET\_PIVOT\_RANDOM)

QUICK\_SORT(unsorted, first, last, GET\_PIVOT)

if first < last

return

p = PARTITION(unsorted, first, last, GET\_PIVOT)

QUICK\_SORT(unsorted, first, p, GET\_PIVOT)

QUICK\_SORT(unsorted, p+1, last, GET\_PIVOT)

PARTITION(unsorted, first, last, GET\_PIVOT):

i = first - 1

j = first

piv\_ind = GET\_PIVOT(unsorted, first, last)

piv = piv\_ind[0]

ind = piv\_ind[1]

unsorted[ind], unsorted[last] = unsorted[last], unsorted[ind]

while i < (last) and j < (last)

if unsorted[j] < piv:

i ++

SWAP(unsorted[i], unsorted[j)

j ++

SWAP(unsorted[i+1], unsorted[ind)

return i + 1

GET\_PIVOT\_LAST(unsorted, first, last):

return unsorted[last], last

GET\_PIVOT\_RANDOM(unsorted, first, last):

rand:= random(first, last)

return unsorted[rand], rand

### ****Merge Sort****

MERGE\_SORT(unsorted):

if unsorted.length <= 1:

return unorted

unsorted\_half1, unsorted\_half2 = divide(unsorted)

merge\_sort(unsorted\_half1)

merge\_sort(unsorted\_half2)

merge(unsorted\_half1, unsorted\_half2, unsorted)

DIVIDE(unsorted):

half\_i = unsorted.length / 2

return unsorted[:half\_i], unsorted[half\_i:]

MERGE(sorted\_half1, sorted\_half2, unsorted):

i, j, k = 0, 0, 0

half1\_length, half2\_length = sorted\_half1.length, sorted\_half2.length

while i != half1\_length and j != half2\_length:

if sorted\_half1[i] < sorted\_half2[j]:

unsorted[k] = sorted\_half1[i]

i ← i + 1

else:

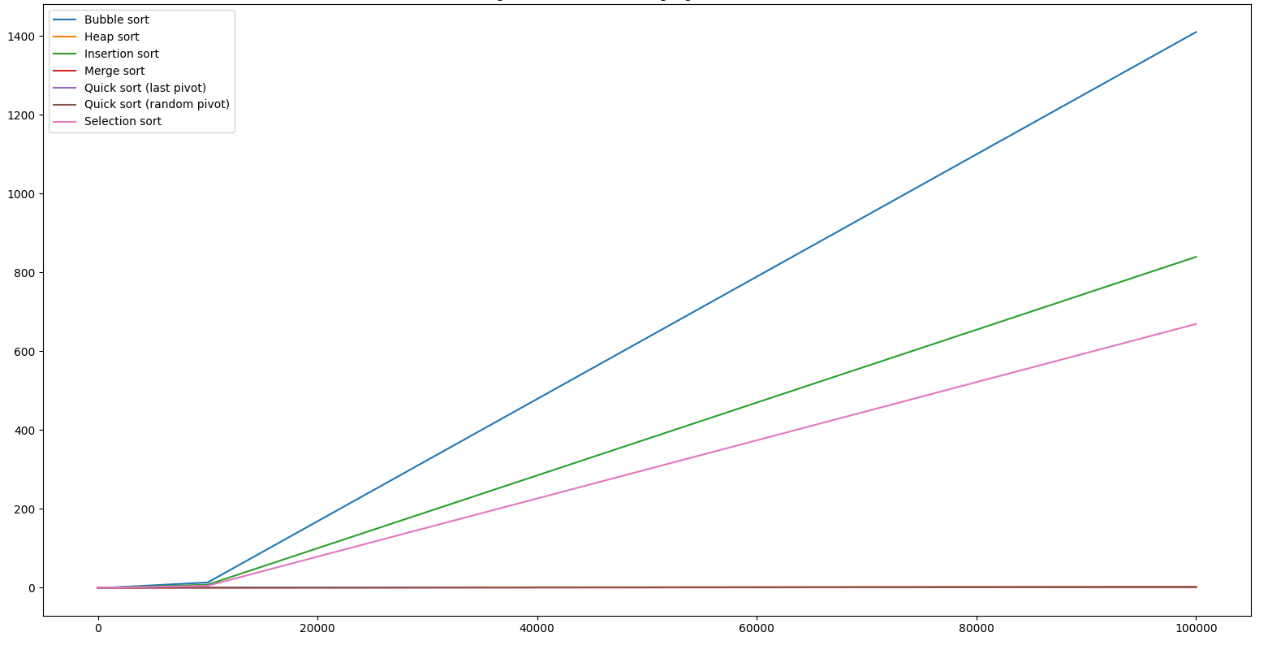
unsorted[k] = sorted\_half2[j]

j ← j + 1

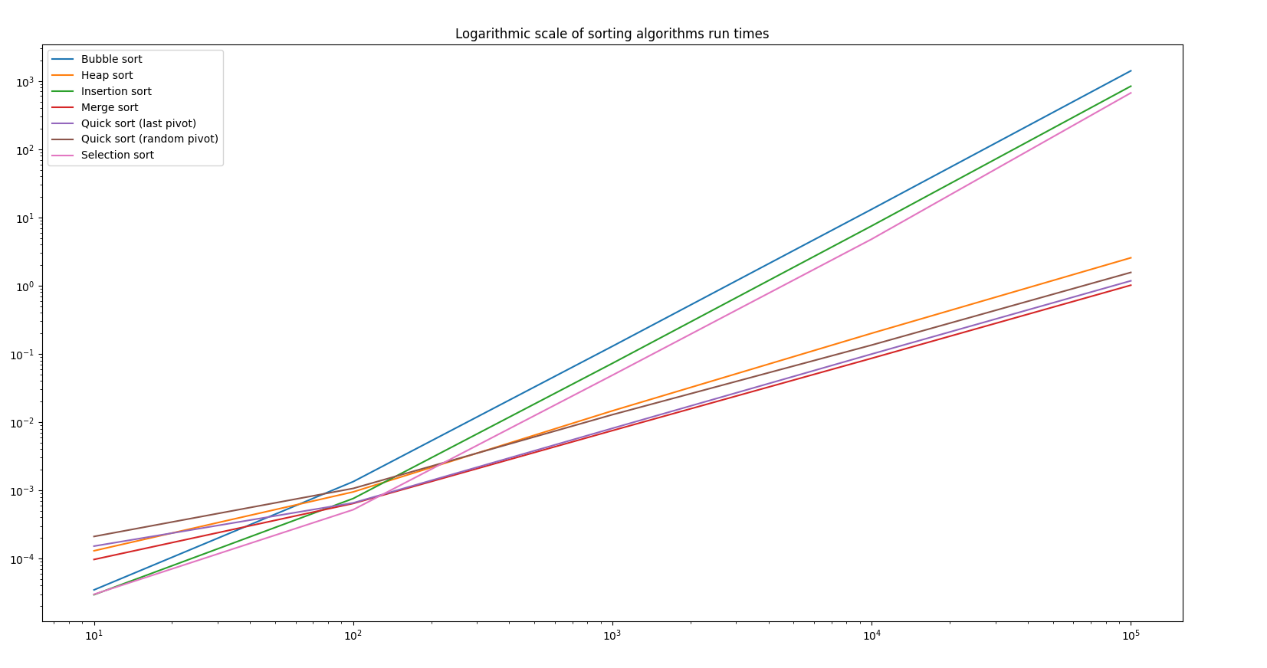
k ← k + 1

unsorted[k:] = sorted\_half1[i:] if i != half1\_length else sorted\_half2[j:]

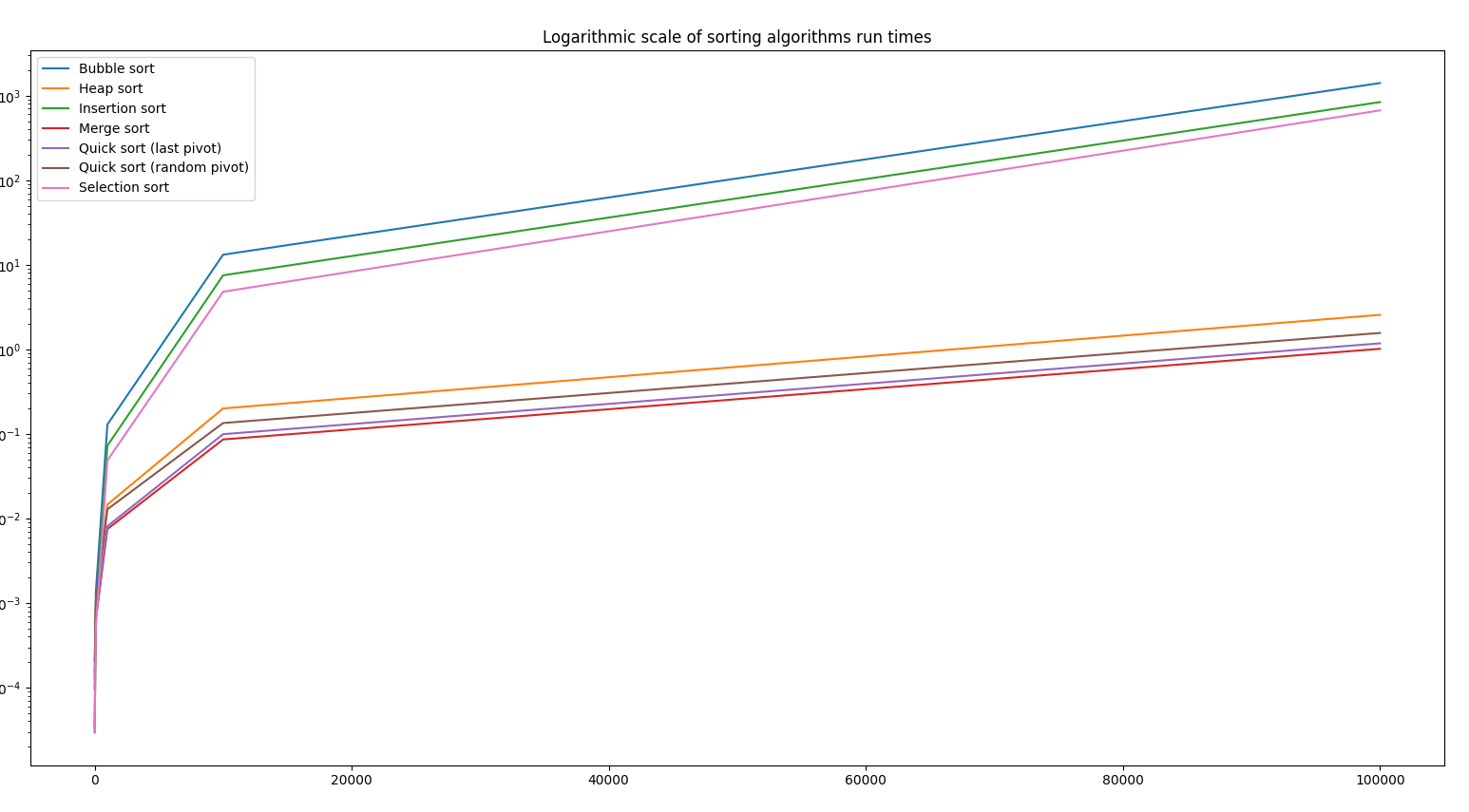
**Sample graph:**



**Sample graph (both x-axis and y-axis are logarithmic scale):**



**Sample graph (y-axis logarithmic scale):**



**NOTE: The graphs may not look accurate, this is a result of taking small number of samples (n = 10, 100, 1000, 10000, 100000) which is stated in the assignment requirements PDF.**