Name: Ahmed Lotfey Siam
ID: 4129

Problem Statement:-

In this assignment, you're required to implement some basic procedures and show how they could be used in a sorting algorithm:

- The MAX-HEAPIFY procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The BUILD-MAX-HEAP procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The HEAPSORT procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The MAX-HEAP-INSERT, and HEAP-EXTRACT-MAX procedures, which run in $O(\lg n)$ time, allow the heap data structure to implement a priority queue.

You're required to implement the above procedures, pseudo code for the above procedures are explained in details in the textbook attached with this problem statement: "Cormen, Thomas H., et al. Introduction to algorithms. Vol. 2. Cambridge: MIT press, 2001". Feel free to select the programming language of your choice.

- You are required to implement the "heapsort" algorithm as an application for binary heaps. You're required to compare the running time performance of your algorithms against:
  - An $O(n^2)$ sorting algorithm such as Selection Sort, Bubble Sort, or Insertion sort.
  - An $O(n \lg n)$ sorting algorithm such as Merge Sort or Quick sort algorithm in the average case[1].

  In addition to heapsort, select **one** of the sorting algorithms from each class mentioned above.

- To test your implementation and analyze the running time performance, generate a dataset of random numbers and plot the relationship between the execution time of the sorting algorithm versus the input size.

3 Bonus

The following parts could you considered as a bonus:

- Implementing more sorting techniques (example: implementing both the quick sort and merge sort)
- Graphical Illustration for the operation of different sorting algorithms (Example: http://www.sorting-algorithms.com/heap-sort)

- **Pseudocode:-**
  - Heap.
    - Heapify

```
Max-Heapify(A, i, size)
    l = Left(i)
    r = RIGHT(i)
    if(l <= size and A[l] > A[i])
        largest = l
    else
        largest - i
    if(r <= size A[r] > A[largest])
        largest = r
    if(largest != i)
        swap(A[i], A[largest])
        MAX-Heapify(A, largest)
```

    - add

```
add(Object)
    A.append(Object)
    Index = A.heap-size()
    while(parent(index) != 0 && A[parent(index)] <
A[index])
        swap(A[index], A[parent(index)])
        Index = parent(index)
```

- poll

```
Poll ()
    if(isEmty())
        Return
    Object top = A[1]
    swap(A[1], A[A.heap-size])
    A.remove(A.heap-size);
    Max-Heapify(A, 1)
    Return top
```

- getParent

```
getParent(index)
    Return floor(index / 2)
```

- getLeft

```
getLeft(index)
    Return 2 * index
```

- getRight

```
getRight(index)
    Return 2 * index + 1
```

- BuildHeap

```
Build-Heap(A)
    For (index = A.size downTo 0)
        Max-Heapify(A, index, A.size)
```

- Heap-Sort

```
HeapSort(A)
    Build-Heap(A)
    For (index = A.size downTo 1)
        swap(A[index], A[1])
        Max-Heapify(A, 1, index - 1)
```

- QuickSort

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q − 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

- Merge sort

```
func mergesort( var a as array )
    if ( n == 1 ) return a

    var l1 as array = a[0] ... a[n/2]
    var l2 as array = a[n/2+1] ...
a[n]

    l1 = mergesort( l1 )
    l2 = mergesort( l2 )

    return merge( l1, l2 )
end func

func merge( var a as array, var b
as array )
    var c as array

    while ( a and b have elements )
        if ( a[0] > b[0] )
            add b[0] to the end of c
            remove b[0] from b
        else
            add a[0] to the end of c
            remove a[0] from a
    while ( a has elements )
        add a[0] to the end of c
        remove a[0] from a
    while ( b has elements )
        add b[0] to the end of c
        remove b[0] from b
    return c
end func
```

- SelectionSort

```
selectionSort(Array)
        for (int i = 1 upTo Array.size)
           int pivot = i;
           for (int j = i + 1 upTo Array.size)
                  if (Array[pivot] > Array[j]))
                     pivot = j;
           swap(array, i, pivot);
```

- BubbleSort

```
procedure bubbleSort( A : list of sortable
items )
   n = length(A)
   repeat
     swapped = false
     for i = 1 to n-1 inclusive do
       if A[i-1] > A[i] then
         swap( A[i-1], A[i] )
         swapped = true
       end if
     end for
   until not swapped
end procedure
```

Sample run

[982035, 303946, 826244, 833386, 681733, 888315, 904027, 553382, 525546, 221369, 908826, 991748, 973113, 928058, 268923, 644275, 189339, 4538, 102659, 809290]

Type	Size	Time
HeapSort	20	1
QuickSort	20	0
MergeSort	20	1
BubbleSort	20	1
SelectionSort	20	0

[4538, 102659, 189339, 221369, 268923, 303946, 525546, 553382, 644275, 681733, 809290, 826244, 833386, 888315, 904027, 908826, 928058, 973113, 982035, 991748]

- <u>Complexity Analysis</u>

This is the run time in (ms) for each algorithm, all running on the same data set.
The data set is generated randomly from integers in range (1:int.MaxValue).

Data Size	Heap Sort	Quick Sort	Merge Sort	Bubble Sort	Selection Sort
5000	12	5	31	126	64
10000	5	2	79	270	109
15000	5	2	168	554	191
20000	6	3	289	1005	341
25000	7	4	465	1558	594
30000	8	5	673	2471	877
35000	9	5	905	3497	1213
40000	11	6	1193	4714	1654
45000	48	27	1579	7632	4707
50000	17	8	1893	9490	5821
55000	17	8	2297	8892	7029
60000	18	9	2757	10310	8327