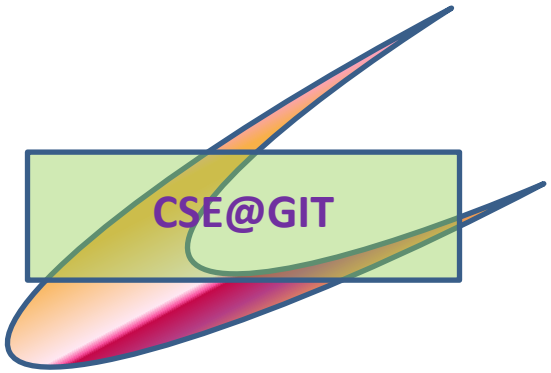
Heap Sort Algorithm

Problem Definition: Implement HeapSort Algorithm to sort a given set of elements and determine the time required to sort the elements. Plot the graph of Computing V/s Problem size.



Objectives of the Experiment:

1. To apply the Transform and conquer strategy
2. Present the working of HEAPSORT
3. Learn to write Algorithm in standard form
4. Analyze the Algorithm & Estimate computing time

General Idea of Transform & Conquer

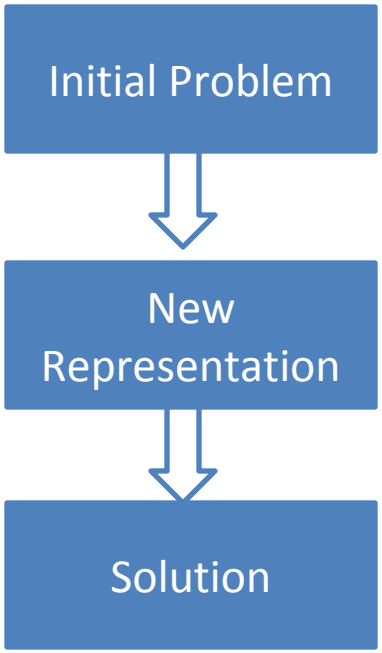
1. Transform the original problem instance into a different problem instance
2. Solve the new instance
3. Transform the solution of the new instance into the solution for the original instance

HeapSort – Transform and Conquer

Transform and Conquer is a **two stage** procedure.

The first stage is called the transformation stage as it involves changing the problem into something easier to solve.

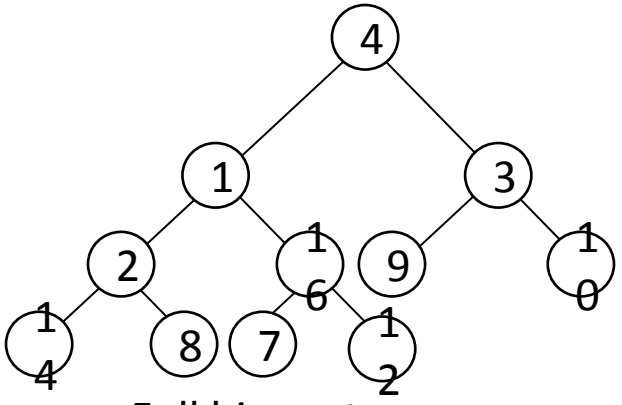
The second stage is then the solution part**.**



- Two Stage procedure/Solution
  - Transform into Another Problem
  - Solve New Problem

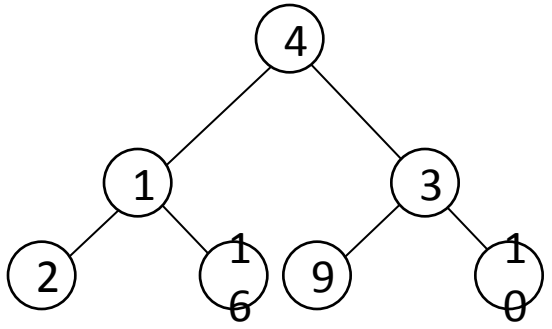
Basic definition

- *Def:* **Full binary tree** = a binary tree in which each node is either a leaf or has degree exactly 2.



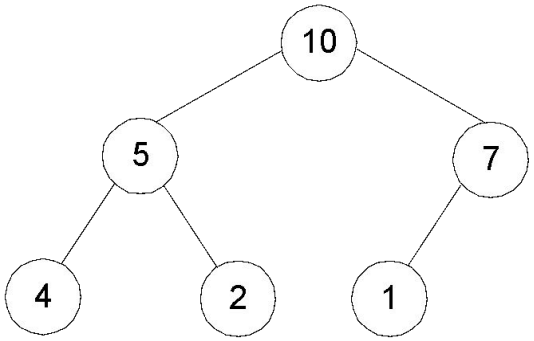
Full binary tree

- *Def:* **Complete binary tree** = a binary tree in which all leaves are on the same level and all internal nodes have degree 2.

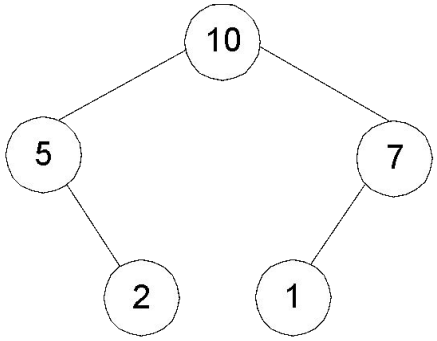


Complete binary tree

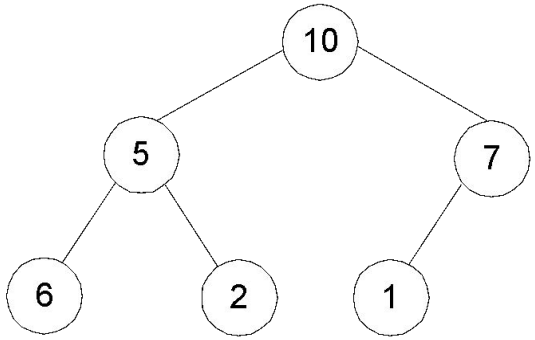
Illustration of the heap's definition



a heap



not a heap



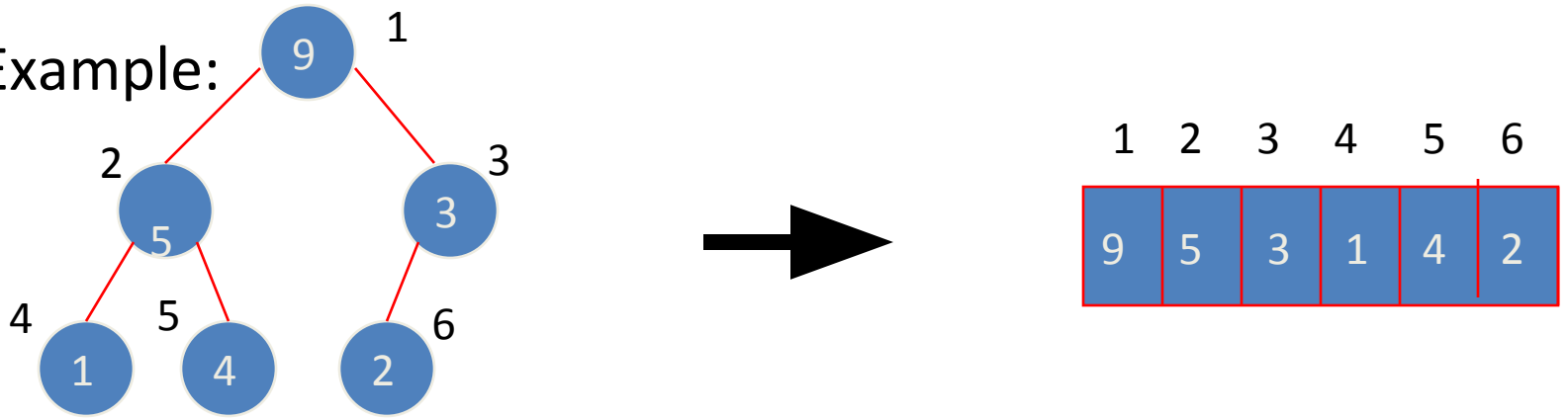
not a heap

Note: Heap's elements are ordered top down (along any path down from its root), but they are not ordered left to right

Heap's Array Representation

Store heap's elements in an array (whose elements indexed,for convenience, 1 to *n*) in top-down left-to-right order

Example:

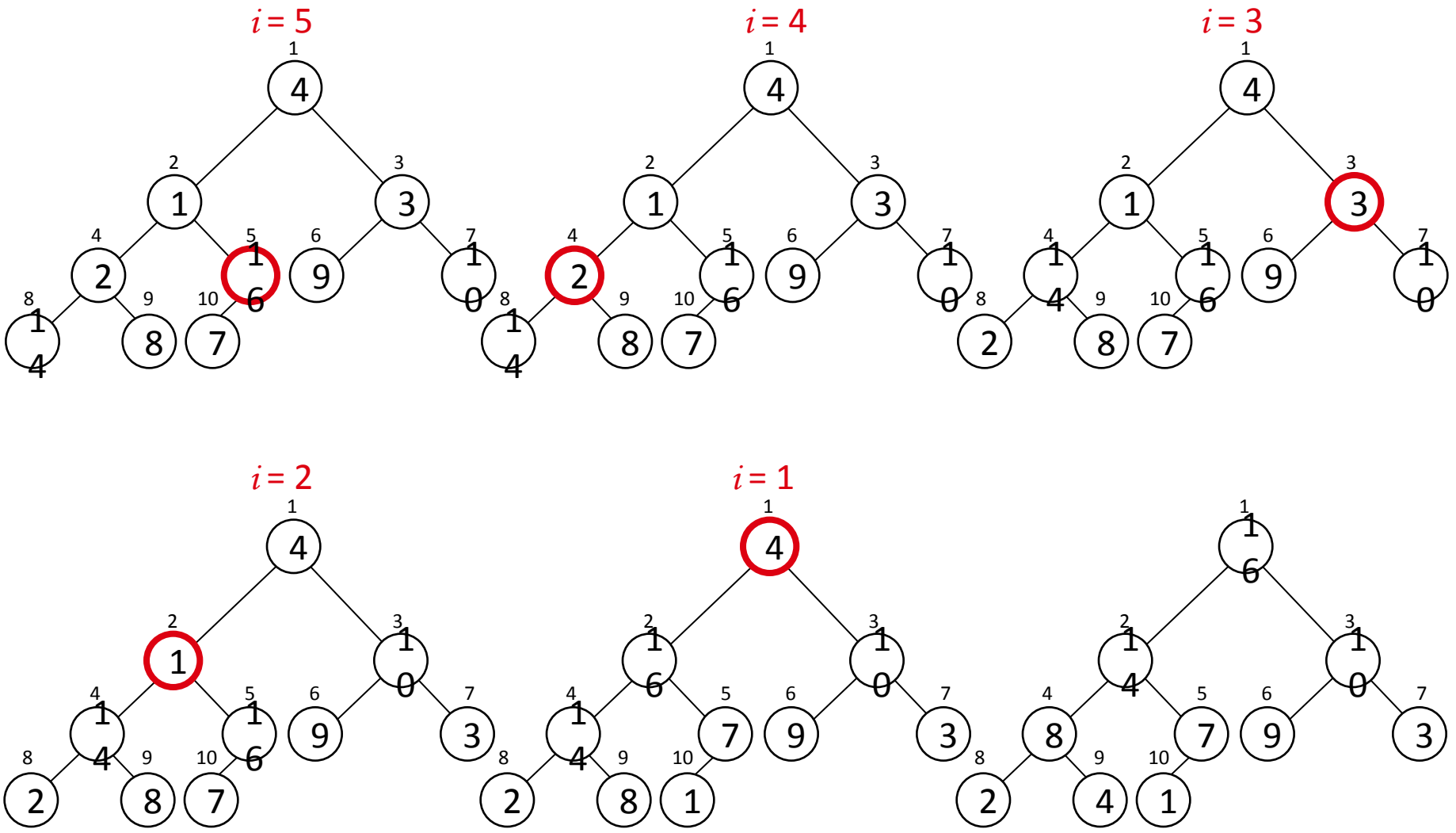


- Left child of node ***i*** is at ***2i***
- Right child of node ***i*** is at ***2i+1***
- Parent of node ***i*** is at ***⌊i/2⌋***
- Parental (i.e interior) nodes are in the ***first : n/2*** locations
- leaf nodes are at *locations* ***n/2+1*** to n

  i.e The elements in the subarray $A[\lfloor(n/2\rfloor+1) .. n]$ are leaves

Example: A

4	1	3	2	16	9	10	14	8	7



Algorithm

STEP 1: Logically, think the given array as Complete Binary Tree,

STEP 2: For sorting the array in ascending order, check whether the tree is satisfying Max-heap property at each node,

(For descending order, Check whether the tree is satisfying Min-heap property)

STEP 3: If the tree is satisfying Max-heap property, then largest item is stored at the root of the heap.

(At this point we have found the largest element in array, Now if we place this element at the end(nth position) of the array then 1 item in array is at proper place.)

We will remove the largest element from the heap and put at its proper place(nth position) in array.

Algorithm

After removing the largest element, which element will take its place?

We will put last element of the heap at the vacant place. After placing the last element at the root, The new tree formed may or may not satisfy max-heap property. So, If it is not satisfying max-heap property then first task is to make changes to the tree, So that it satisfies max-heap property.

(**Heapify process:** The process of making changes to tree so that it satisfies max-heap property is called **heapify**)

When tree satisfies max-heap property, again largest item is stored at the root of the heap. We will remove the largest element from the heap and put at its proper place(n-1 position) in array.

Repeat step 3 until size of array is 1 (At this point all elements are sorted.)

Heapify Process with Example

Heapify process checks whether item at parent nodes has larger value than its left and right child.

If parent node is not largest compared to its left and right child, then it finds the largest item among parent, its left and right child and replaces largest with parent node.

It repeat the process for each node and at one point tree will start satisfying max-heap property.
At this point, stop heapify process and largest element will be at root node.

We found the largest element, Remove it and put it at its proper place in array,
Put the last element of the tree at the place we removed the node(that is at root of the tree)
Placing last node at the root may disturbed the max-heap property of root node.
So again repeat the Heapify process for root node. Continue heapify process until all nodes in tree satisfy max-heap property.

Initially, From which node we will start heapify process? Do we need to check each and every node that they satisfy heap property?

We do not have to look into leaf nodes as they don't have children and already satisfying max-heap property.

So, we will start looking from the node which has at least one child present.

Bottom Up Heap construction

Algorithm HeapSort(A,n)

```
   // Construct a heap from the elements of  given array by
   // the bottom up approach
   // Input : An array A[1,n] of orderable items
   // Output : A heap array A[1.n]
  Begin
  for i= n/2 downto 1 do
          Heapify(A, n, i)
For i=n down to 2 do
    exchange(A[1], A[i])
    Heapify(A, i-1, 1)
End
```

Algorithm Heapify(A, n, i)

// Construct a heap from the elements of given an array
// by the bottom up approach once the exchange of
// root's key with key K of the heap is done.
// Input : An array H[1,n] of orderable items
// Output : A heap array H[1.n]

```
Begin
largest=i
l=2*i     r=2*i+1
If( l <=n && A[l]>A[largest])
      largest=l
If( r <=n && A[r]>A[largest])
      largest=r
If(largest!=i)
     exchange(A[i], A[largest])
     Heapify(A,n,largest)
End
```

Heap Sort Time Complexity

- **1.** Heap sort has the best possible worst case running time complexity of **O(n Log n).**
  2. It **doesn't need any extra storage** and that makes it good for situations where array size is large.

Heap Sort (cont'd)

- Time complexity?
  - First, MakeHeap(A)
    - Easy analysis: O($n$lg$n$) since Heapify is O(lg$n$)
    - More exact analysis: O(n)
  - Then:
    - n-1 swaps = O(n)
    - n-1 calls to Heapify = O($n$lg$n$)
  - ➔ O(n) + O(n) + O($n$lg$n$) = O($n$lg$n$)

Time and Space Complexity

- Quick Sorts worst-case running time is : $\Theta(n^2)$
  (This case occurs when list is almost sorted)

- But Heaport worst and average-case running time are: $\Theta(n*\log n)$

- Space complexity of Heapsort and Quicksort are same.

18 Analysis of Heapsort

Stage 1: Build heap for a given list of n keys (of height h)

worst-case

$$C(n) = \sum_{i=0}^{h-1} 2(h-i)\, 2^i = 2\,(n - \log_2(n+1)) \in \Theta(n)$$

nodes at level i

Stage 2: Repeat operation of root removal n-1 times (fix heap)

worst-case

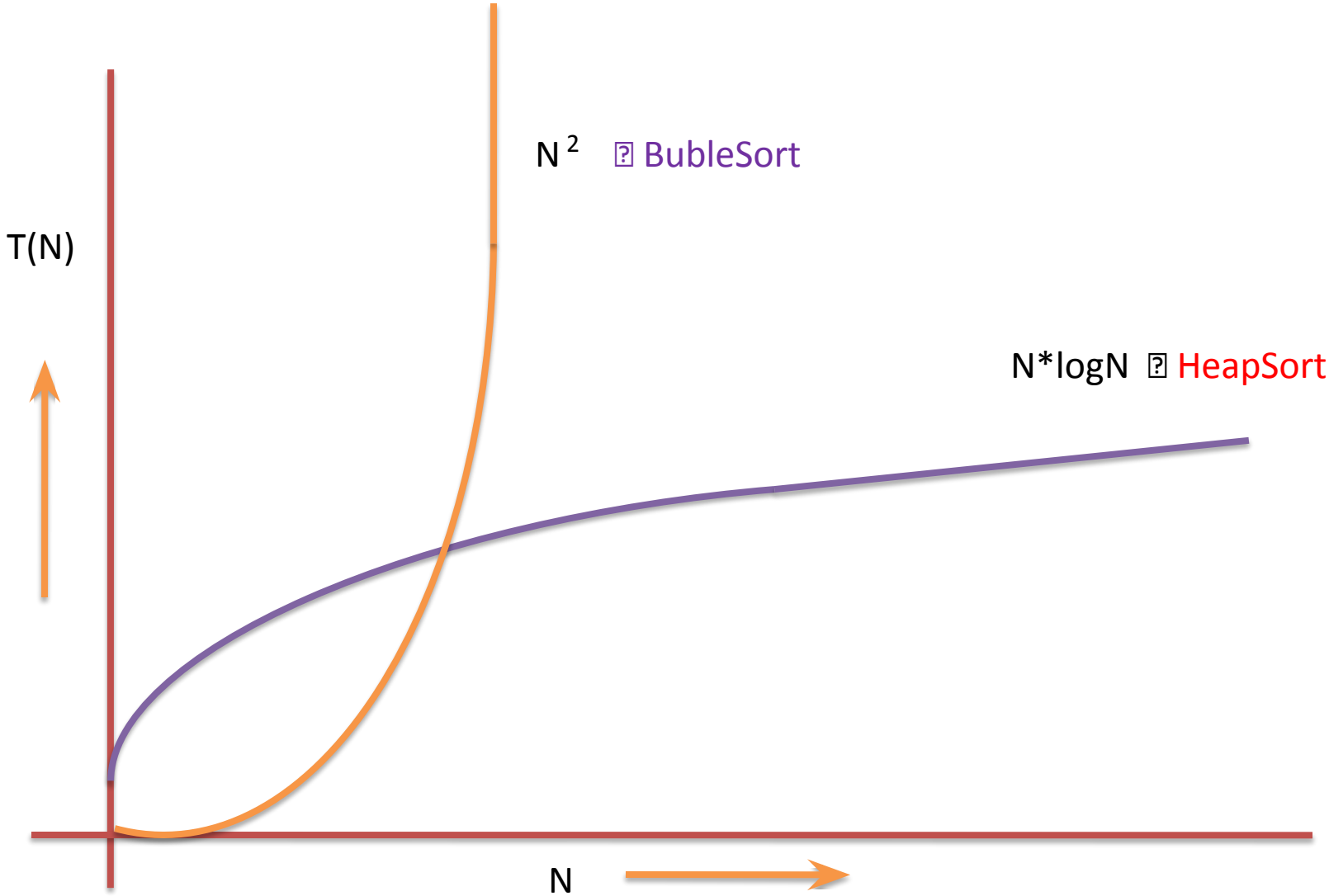
$$C(n) = \sum_{i=1}^{n-1} 2\log_2 i \in \Theta(n \log n)$$

Both worst-case and average-case efficiency: $\Theta(n \log n)$

In-place: yes

Stability: no (e.g., apply heapsort to 1 1)

Time and Space Complexity



Learning Outcome of the Experiment and Conclusion

At the end of the session, students should be able to :

1. Explain the working of Transform and Conquer Strategy
2. Demonstrate the working of Heapsort algorithm on a given set of size n
3. Write the program in Java to implement HeapSort Algorithm and estimate the computing time using appropriate time functions.
4. Plot a graph of Computing time V/s Size of the input and draw conclusions