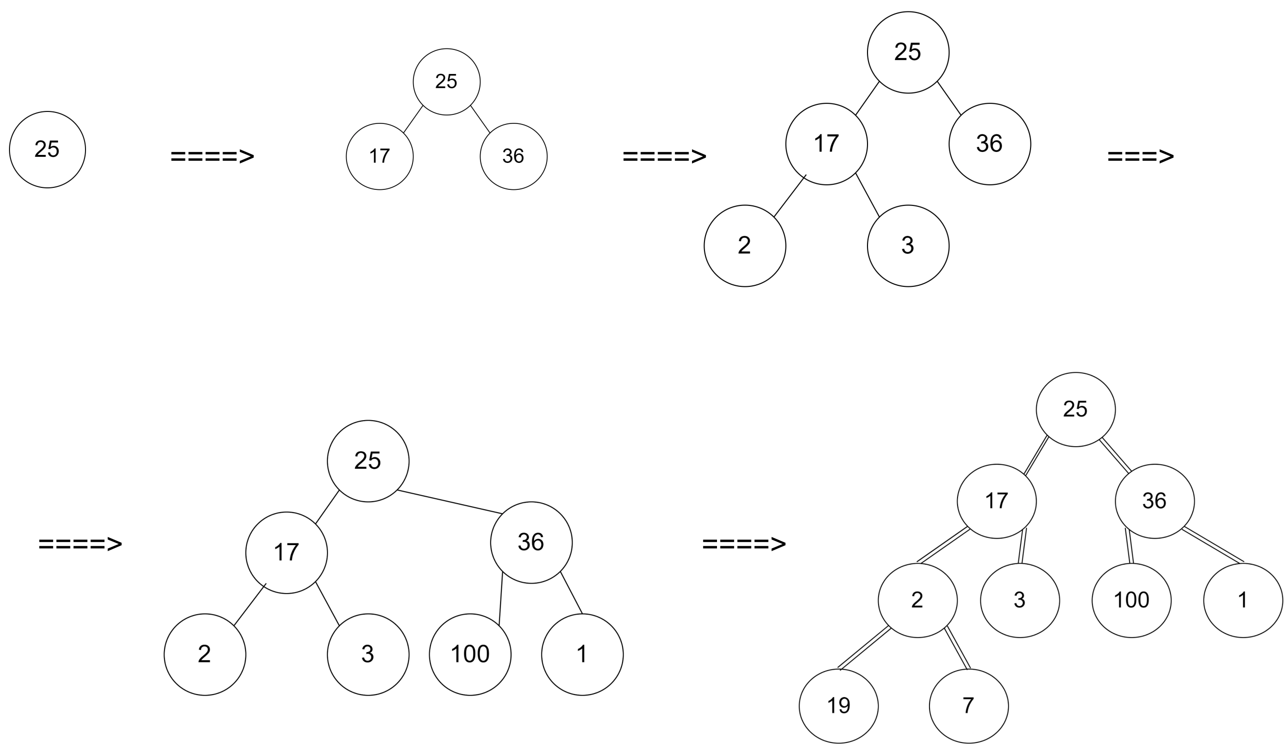
# CS - 601 Advanced Algorithms

## Programming Assignment -1

**Question 1**:
Consider the array below:
25, 17, 36, 2, 3, 100, 1, 19, 7.
a) Construct a binary tree out of the given array



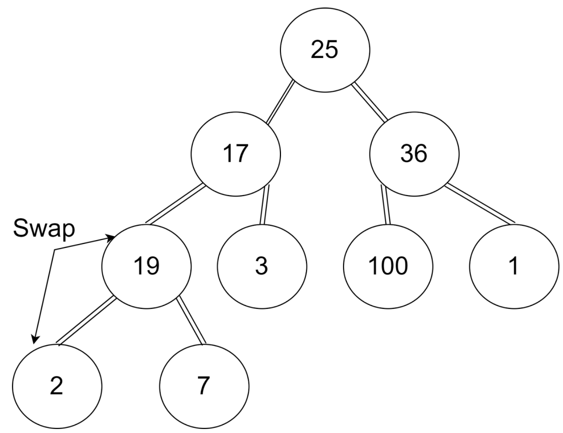

⇒ Binary Tree off the given array

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 25 | 17 | 36 | 2 | 3 | 100 | 1 | 19 | 7 |

b) Build a heap out of that. Show all your work step by step. In each step (i) draw the corresponding tree as well as (ii) the resulting array.

Heapify 1: Check max of both child nodes and swap with the parent node if the parent node has a value less than the max child node.

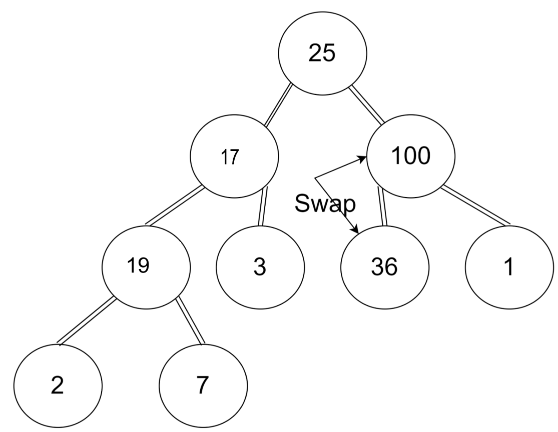
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 25 | 17 | 36 | **19** | 3 | 100 | 1 | **2** | 7 |




⇒ Swapped 2 and 19, since 19 is greater than 2

Heapify 2 :

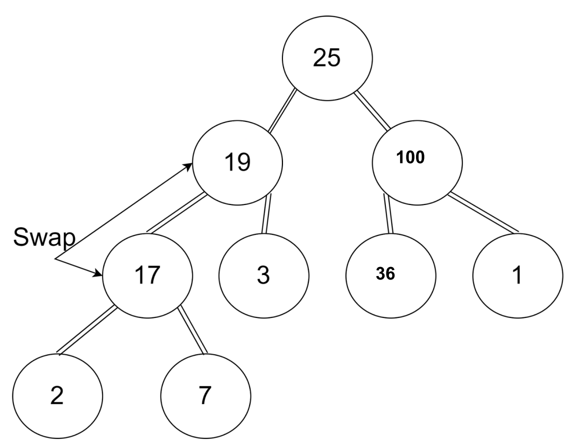
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 25 | 17 | **100** | 19 | 3 | **36** | 1 | 2 | 7 |




⇒ Swapped 17 and 19, since 19 is greater than 17

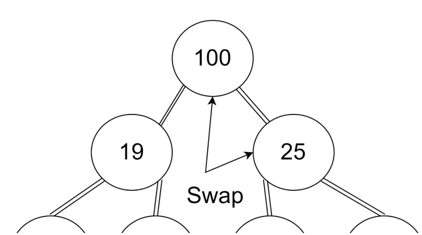
Heapify 3 :

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 25 | **19** | 100 | **17** | 3 | 36 | 1 | 2 | 7 |




⇒ Swapped 100 and 36, since 100 is greater than 36

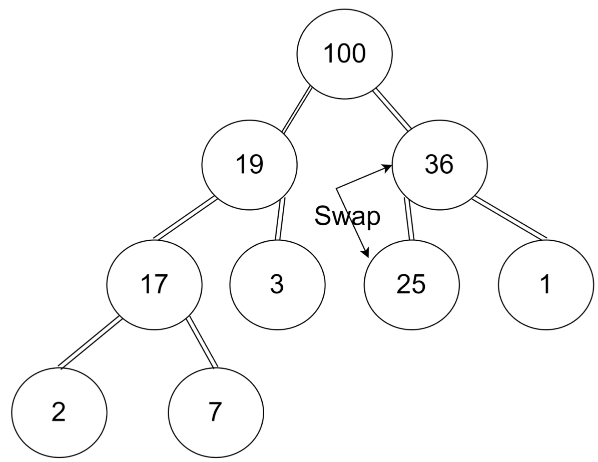
Heapify 4:



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **100** | 19 | **25** | 17 | 3 | 36 | 1 | 2 | 7 |

⇒ Swapped 100 and 25, since 100 is greater than 25

Heapify 5 :

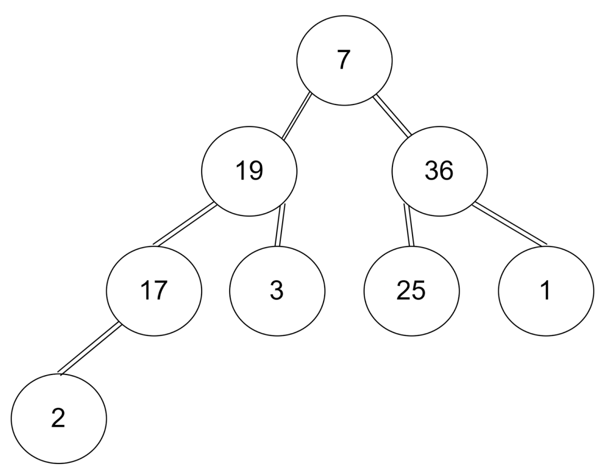



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 100 | 19 | **36** | 17 | 3 | **25** | 1 | 2 | 7 |

⇒ Swapped 36 and 25, since 36 is greater than 25

c) Now use the heapsort algorithm to sort the array in ascending order. Again, show all your work step by step. In each step (i) draw the corresponding tree as well as (ii) the resulting array.

**Iteration 1:**



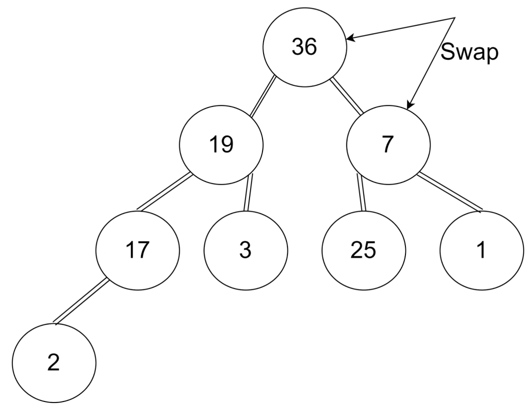

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **7** | 19 | 36 | 17 | 3 | 25 | 1 | 2 | **100** |

⇒ Root element (100) is deleted to get the max value of heap.
⇒ The last leaf node is shifted to the root node.

⇒ Heapify process is executed to maintain the heap.

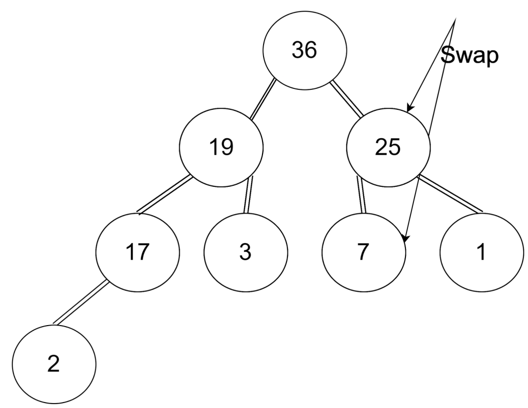
After heapify (Iteration 1):

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **36** | 19 | **7** | 17 | 3 | 25 | 1 | 2 | **100** |




⇒ swapped 7 with 36 , since 36 is greater than 7

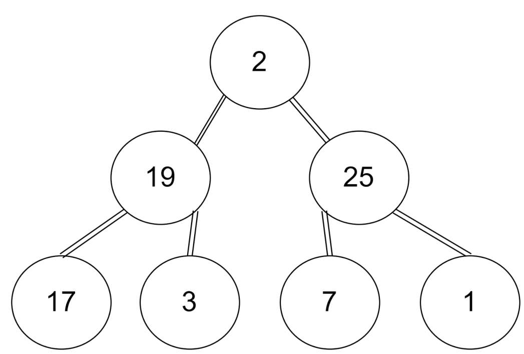
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 36 | 19 | **25** | 17 | 3 | **7** | 1 | 2 | **100** |




⇒ swapped 7 with 25, since 25 is greater than 7

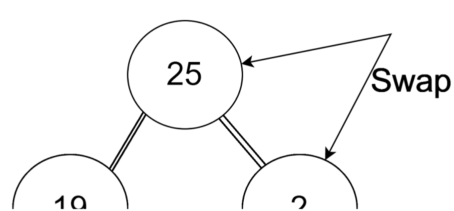
**Iteration 2:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2 | 19 | 25 | 17 | 3 | 7 | 1 | **36** | **100** |




⇒ Root element (36) is deleted to get the max value of heap.
⇒ The last leaf node is shifted to the root node.
⇒ Heapify process is executed to maintain the heap.

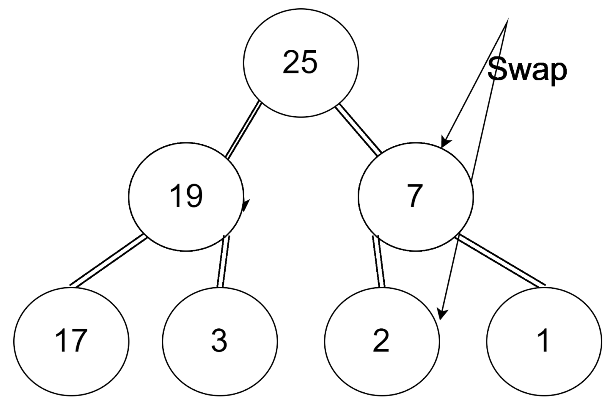
After Heapify (Iteration 2):



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **25** | 19 | **2** | 17 | 3 | 7 | 1 | **36** | **100** |

⇒ swapped 2 with 25, since 25 is greater than 2

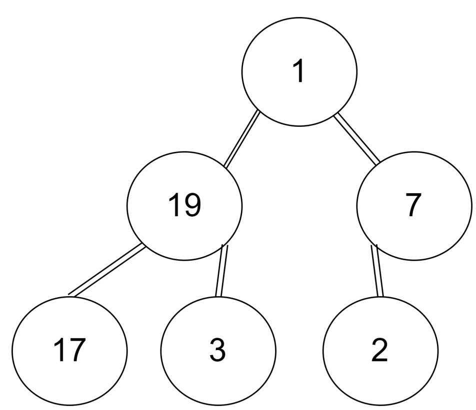
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 25 | 19 | **7** | 17 | 3 | **2** | 1 | **36** | **100** |



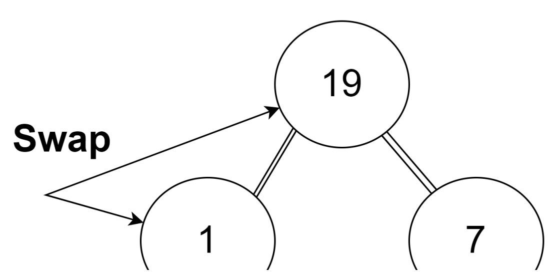

⇒ swapped 2 with 7, since 7 is greater than 2

**Iteration 3:**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 19 | 7 | 17 | 3 | 2 | **25** | **36** | **100** |



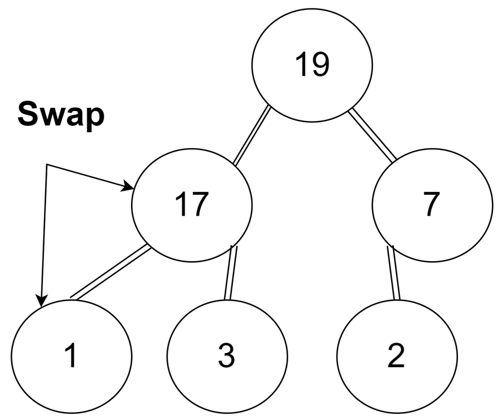

⇒ Root element (25) is deleted to get the max value of heap.
⇒ The last leaf node is shifted to the root node.
⇒ Heapify process is executed to maintain the heap.
After heapify (Iteration 3):



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **19** | **1** | 7 | 17 | 3 | 2 | **25** | **36** | **100** |

⇒ swapped 19 with 1 , since 1 is greater than 19

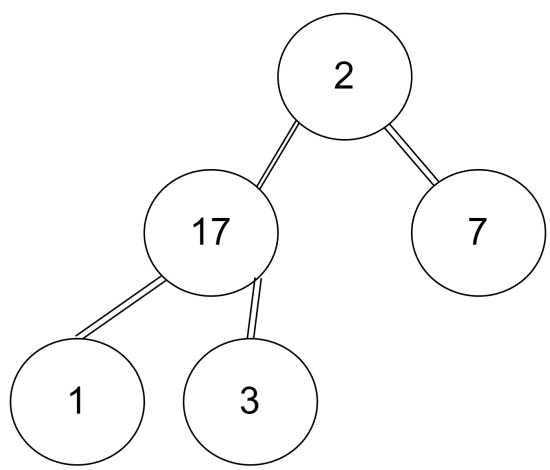
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 19 | **17** | 7 | **1** | 3 | 2 | **25** | **36** | **100** |



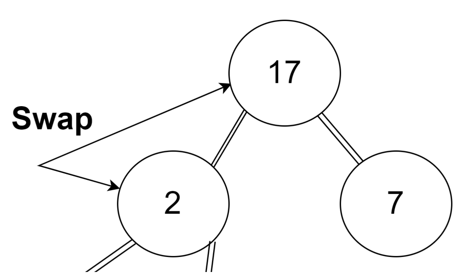

⇒ swapped 17 with 1 , since 17 is greater than 1

Iteration 4:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2 | 17 | 7 | 1 | 3 | **19** | **25** | **36** | **100** |



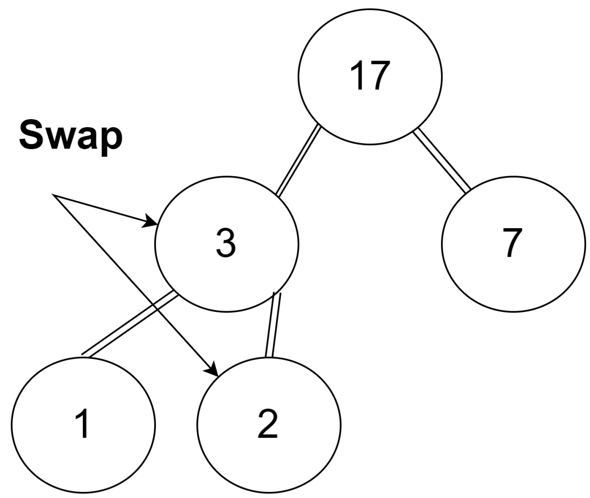

⇒ Root element (19) is deleted to get the max value of heap.
⇒ The last leaf node is shifted to the root node.
⇒ Heapify process is executed to maintain the heap.
After heapify (iteration 4):



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **17** | **2** | 7 | 1 | 3 | **19** | **25** | **36** | **100** |

⇒ swapped 17 with 2 , since 17 is greater than 2

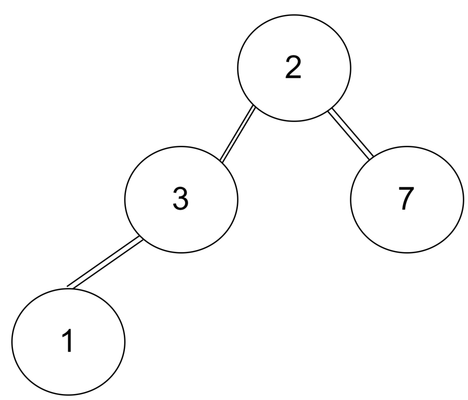
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 17 | **3** | 7 | 1 | **2** | **19** | **25** | **36** | **100** |



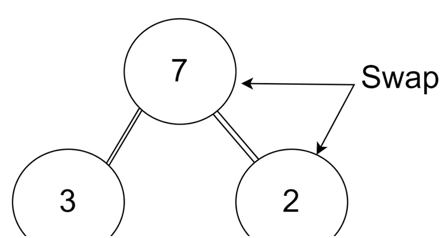

⇒ swapped 3 with 2 , since 3 is greater than 2

Iteration 5:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **2** | **3** | 7 | 1 | **17** | **19** | **25** | **36** | **100** |




⇒ Root element (17) is deleted to get the max value of heap.
⇒ The last leaf node is shifted to the root node.
⇒ Heapify process is executed to maintain the heap.
After heapify (iteration 5):

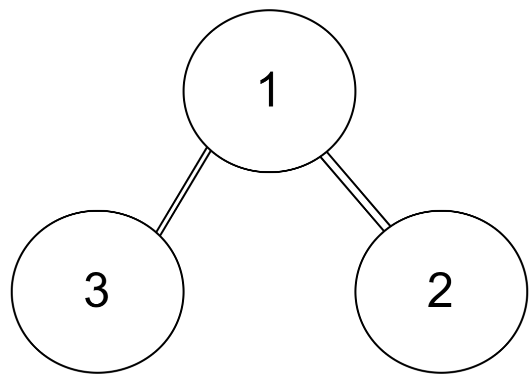


| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **7** | 3 | **2** | 1 | **17** | **19** | **25** | **36** | **100** |

⇒ swapped 7 with 2 , since 7 is greater than 2

Iteration 6:

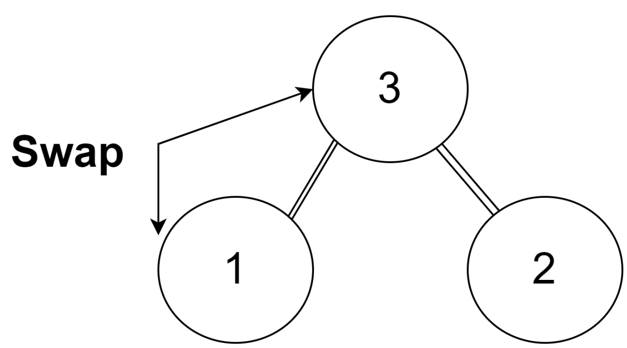
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **1** | 3 | 2 | **7** | **17** | **19** | **25** | **36** | **100** |




⇒ Root element (7) is deleted to get the max value of heap.
⇒ The last leaf node is shifted to the root node.
⇒ Heapify process is executed to maintain the heap.

After heapify (iteration 6):

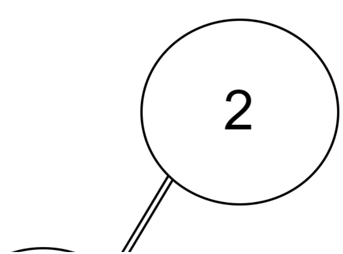
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **3** | 1 | 2 | **7** | **17** | **19** | **25** | **36** | **100** |




⇒ swapped 3 with 1 , since 3 is greater than 1

Iteration 7:

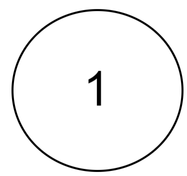
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **2** | 1 | **3** | **7** | **17** | **19** | **25** | **36** | **100** |



⇒ Root element (3) is deleted to get the max value of heap.
⇒ The last leaf node is shifted to the root node.
⇒ Heapify process is executed to maintain the heap.

Iteration 8:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 7 | 17 | 19 | 25 | 36 | 100 |




⇒ Root element (2) is deleted to get the max value of heap.
⇒ The last leaf node is shifted to the root node.
⇒ Heapify process is executed to maintain the heap.

Iteration 9:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 7 | 17 | 19 | 25 | 36 | 100 |

⇒ Array sorted using heap sort in ascending order.

d) Implement the Heapsort algorithm in C++/Java. Paste your code entirely in the solution file . Also, attach the code to your submission to check and execute by the grader. No points for the code if the code files are not provided.

```
public class HeapSort {
	public void heapSort(int array[]) {
		int n = array.length;
		for (int i = n / 2 - 1; i >= 0; i--)
			heapify(array, n, i);
		System.out.println("Array after building max heap");
		displayArray(array);
		System.out.println("Array after each main loop");
		for (int i = n - 1; i > 0; i--) {
			int temp = array[0];
			array[0] = array[i];
			array[i] = temp;
			displayArray(array);
			heapify(array, i, 0);
		}
	}

	void heapify(int array[], int n, int i) {
		int maximum = i;
		int l = 2 * i + 1;
		int r = 2 * i + 2;
		if (l < n && array[l] > array[maximum])
			maximum = l;
		if (r < n && array[r] > array[maximum])
			maximum = r;
		if (maximum != i) {
			int temp = array[i];
			array[i] = array[maximum];
			array[maximum] = temp;
			heapify(array, n, maximum);
		}
	}

	static void displayArray(int array[]) {
		int n = array.length;
		for (int i = 0; i < n; i++)
			System.out.print(array[i] + " ");
		System.out.println();

	}

	public static void main(String args[]) {
		int array[] = { 25, 17, 36, 2, 3, 100, 1, 19, 7 };
```

```
            HeapSort ob = new HeapSort();
            ob.heapSort(array);
            System.out.println("============================================");
            System.out.println("Sorted Array");
            System.out.println("============================================");
            displayArray(array);
            System.out.println("============================================");
    }
}
```

e) Analyze the time complexity of your program in the worst-case. Explain in detail first line by line. Then come up with the Big-O notation for the whole program.

**Will start with heapsort function:**

| **Operation** | **Costs** | **Time** |
|---|---|---|
| int n = array.length; | c1 | 1 |
| for (int i = n / 2 - 1(c2); i >= 0(**c3**); i--(**c4**)) | c2 | 1 |
| | c3 | n/2 |
| | c4 | n/2-1 |
| heapify(array, n, i); | c5 | This depends on the height of the tree, so height of tree can be calculated using logn. Hence in worst case it would be logn |
| System.out.println("Array after building max heap"); | c6 | 1 |
| displayArray(array); | c7 | 1 |
| System.out.println("Array after each main loop"); | c8 | 1 |
| for (int i = n - 1(**c9**); i > 0(**c10**); i–(**c11**)) | c9 | 1 |
| | c10 | n |

| | | |
|---|---|---|
| | c11 | n-1 |
| int temp = array[0]; | c12 | n-1 |
| array[0] = array[i]; | c13 | n-1 |
| array[i] = temp; | c14 | n-1 |
| displayArray(array); | c15 | n-1 |
| heapify(array, i, 0); | c16 | This will be called recursively inside based on the height, In the worst case this will be called **n-1 logn** times based on height. |

**For heapify function :**

| | | |
|---|---|---|
| int maximum = i; | c17 | 1 |
| int l = 2 * i + 1; | c18 | 1 |
| int r = 2 * i + 2; | c19 | 1 |
| if (l < n && array[l] > array[maximum]) | c20 | 1 |
| maximum = l; | c21 | 1 |
| if (r < n && array[r] > array[maximum]) | c22 | 1 |
| maximum = r; | c23 | 1 |
| if (maximum != i) | c24 | 1 |
| int temp = array[i]; | c25 | 1 |
| array[i] = array[maximum]; | c26 | 1 |
| array[maximum] = temp; | c27 | 1 |
| heapify(array, n, maximum); | c28 | This will be recursive call |

| | | |
|---|---|---|
| | | inside so in worst case it would be logn |

**For displayArray function :**

| | | |
|---|---|---|
| int n = array.length; | c29 | 1 |
| for (int i = 0(c30); i < n(c31); i++(c32)) | c30 | 1 |
| | c31 | n+1 |
| | c32 | n |
| System.out.print(array[i] + " "); | c33 | n |
| System.out.print(array[i] + " "); | c34 | 1 |

**For main function**

| | | |
|---|---|---|
| int array[] = { 25, 17, 36, 2, 3, 100, 1, 19, 7 }; | c35 | 1 |
| HeapSort ob = new HeapSort(); | c36 | 1 |
| ob.heapSort(array); | c37 | 1 |
| System.out.println("========================================="); | c38 | 1 |
| System.out.println("Sorted Array"); | c39 | 1 |
| displayArray(array); | c40 | 1 |
| System.out.println("========================================="); | c41 | 1 |

f(n) can be obtained by adding all cost * times

$$
\begin{aligned}
f(n) = {} & C1 + C2 + (n/2)C3 + C4(n/2 - 1) + C5(\log n) + C6(1) + C7(1) + C8(1) + C9(1) + \\
& C10(1) + C11(n-1) + C12(n-1) + C13(n-1) + C14(n-1) + C15(n-1) + C16((n-1)\log n) \\
& C17(1) + C18(1) + C19(1) + C20(1) + C21(1) + C(22) + C23(1) + C24(1) + \\
& C25(1) + C26(1) + C27(1) + \cancel{C28(n\log n)}\ \cancel{C28(n\log n) + (n-1)\log n +} \\
& C28(\log n) + C29(1) + C30(1) + C31(n+1) + C32(n) + C33(n) + C34(1) \\
& + C35(1) + C36(1) + C37(1) + C38(1) + C39(1) + C40(1) + C41(1)
\end{aligned}
$$

Considering only higher magnitude values for simpler calculation

$$
\begin{aligned}
\Rightarrow {} & C4(n/2 - 1) + C5(\log n) + C11(n-1) + C12(n-1) + C13(n-1) + C14(n-1) + C15(n-1) \\
& C16((n-1)\log n) + C28(\log n) + C31(n+1) + C32(n) + C33(n)
\end{aligned}
$$

consider only higher magnitude values

$$
\Rightarrow n\log n(C16) + \log n(C5 - C16 + C28) + n(C4/2 + C11 + C12 + C13 + C14 + C15 + C31 + C32 + C33)
$$

$$
\Rightarrow n\log n\, c' + \log n\, c'' + n\, c'''
$$

$c', c''$ & $c'''$ are constants.

consider only higher as n increase the whole runtime increases rapidly

$$
\boxed{\Rightarrow f(n) = n\log n}
$$

f) Give your code the array above as an input. Provide the screenshot of the array you get after each execution of the main loop and at the end once the array is sorted.

```
<terminated> HeapSort [Java Application] C:\Users\STSC\.p2\pool\plugins\or
Array after building max heap
100 19 36 17 3 25 1 2 7
Array after each main loop
7 19 36 17 3 25 1 2 100
2 19 25 17 3 7 1 36 100
1 19 7 17 3 2 25 36 100
2 17 7 1 3 19 25 36 100
2 3 7 1 17 19 25 36 100
1 3 2 7 17 19 25 36 100
2 1 3 7 17 19 25 36 100
1 2 3 7 17 19 25 36 100
==========================================
Sorted Array
==========================================
1 2 3 7 17 19 25 36 100
==========================================
```

**Question 2:**

(a) Take the following list of functions and arrange them in ascending order of growth rate. That is, if function g(n) immediately follows function f(n) in your list, then it should be the case that f(n) is O(g(n)).
f1(n)= n ,f2(n)= n10 ,f3(n)= 2n ,f4(n)=100n, f5(n)=n log n, f6(n)= n2 log n, f7(n)= n^n, f8(n)=n!

From the given list of functions:

From the above equations f7(n) = n^n grows rapidly when compared to other functions and then the function n factorial f8(n) = n! and other exponentials f4(n) = 100^n, f3(n) = 2^n, f2(n) = n^10 and then the exponentials f6(n) = n^2 log n, f5(n) = n log n have logarithms so they grow fast compared to f1(n) = n

Arranging the functions in the order of growth rate in ascending order:

**f1(n) <= f5(n) <= f6(n) <= f2(n) <= f3(n) <= f4(n) <= f8(n) <= f7(n)**

2 (b): In your arrangement in Q2 part (a), if you have fi(n) <=fj(n) that means fi(n)=O(fj(n)).Pick each of such consecutive pair of functions in your arrangement (let's call them fi(n) and fj(n), where fi(n)<=fj(n) in the arrangement) and prove formally why you believe fi(n)=O(fj(n)).

As fi(n)=O(fj(n)), where for fi(n) there exists positive constants c & n0 such that
⇒ **fi(n) <= c * fj(n)** for all n >= n0

Considering following pairs,

-> f1(n) <= f5(n)
f1(n) = n, f5(n)=n log n
f1(n) = n = O(n)
f5(n) = n log n = O(n log n)

n <= n log n
1 <= log n , for n0=2

Therefore, **f1(n) <= O(n log n) where c=1 & n0=2**
**f1(n) = O(f5(n))**

-> f5(n) <= f6(n)
f5(n)=n log n, f6(n)= n^2 log n

f5(n) = n log n = O(n log n)
f6(n) = n^2 log n = O(n^2 log n)

n log n <= n^2 log n //canceling log n on both sides
n <= n^2
1 <= n, for n0 = 1

Therefore, **f5(n) <= O(n^2 log n) where c=1 & n0=1**
**f5(n) = O(f6(n))**

-> f6(n) <= f2(n)
f6(n)= n^2 log n, f2(n)= n^10

f6(n)= n^2 log n = O(n^2 log n)
f2(n)= n^10 = O(n^10)

n^2 log n <= n^10 //canceling n^2 on both sides
log n <= n^8, for n0=1

Therefore, **f6(n) <= O(n^10) where c=1 & n0=1**
**f6(n) = O(f2(n))**

-> f2(n) <= f3(n)
f2(n)= n^10, f3(n)= 2^n

f2(n)= n^10 = O(n^10)
f3(n)= 2^n = O(2^n)

n^10 <= 2^n //Applying log on both sides
log(n^10) <= log(2^n)
10 log n <= n` log 2 //As log 2 base 2 = 1

10 log n <= n, for n0 = 60

Therefore, **f2(n) <= O(2^n) where c=1 & n0=60**
**f2(n) = O(f3(n))**

-> f3(n) <= f4(n)
f3(n)= 2^n, f4(n)=100^n

f3(n)= 2^n = O(2^n)
f4(n)= 100^n = O(100^n)

2^n <= 100^n //Applying log base 2 on both sides
log(2^n) <= log(100^n)
n log 2 <= n log 100
log 2 <= log 100, for n0 = 1

Therefore, **f3(n) <= O(100^n) where c=1 & n0=1**
**f3(n) = O(f4(n))**

-> f4(n) <= f8(n)
f4(n)=100^n, f8(n)=n!

f4(n)= 100^n = O(100^n)
f8(n)= n! = O(n!)

100^n <= n! //Applying log base 2 on both sides

log(100^n) <= log(n!)
n log 100 <= n log n - n + O(log n) //log(n!) = n log n - n + O(log n)
n log 100 <= n log n //n * log n grows quickly compared to "n + O(log n)"
log 100 <= n log n

OR

log(n!) = log(n) + log(n-1) + ... log(2) + log(1)
which is greater than n * log 100

Therefore, **f4(n) <= O(n!)**
**f4(n) = O(f8(n))**

-> f8(n) <= f7(n)
f8(n)=n!, f7(n)= n^n

f8(n)= n! = O(n!)
f7(n)= n^n = O(n^n)

n! <= n^n
log(n!) <= log(n^n) //taking log on both side
n log n - n + O(log n) <= log(n^n) //log(n!) = n log n - n + O(log n)
n log n <= n log n //n * log n grows quickly compared to "n + O(log n)"

Therefore, **f8(n) <= O(n^n)**
**f8(n) = O(f7(n))**