Question 1)

(a) Given an integer list (an array) of size n, write an algorithm in **C++/Java** that adds up all values in the list and displays the result of the summation on screen.

(b) Count the number of basic operations in your algorithm. Explain and **show all your work**.

(c) Write the function f(n) expressing the time complexity of your algorithm correspondingly. Come up with the Big-O notation, then. Explain.

Answer)

(a)

```
class Code{
    public static void main(String args[]){
        // n = 10
        int[] array = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        int n = array.length;
        int sum = 0;
        for(int i = 0;i < n;i++){
            sum += array[i];
        }
        System.out.println("Sum is " + sum);
    }
}
```

(b)

Code	Cost	Number of times it runs
int[] array = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}	C1	1
int n = array.length	C2	1
int sum = 0	C3	1
int i = 0	C4	1
i < n	C5	N + 1
i++	C6	N
sum += array[i];	C7	N
System.out.println("Sum is " + sum)	C8	1

(c)

The Function is

$f(N) = C1 + C2 + C3 + C4 + C5 * (N + 1) + C6 * N + C7 * N + C8$

$= (C5 + C6 + C7) * N + (C1 + C2 + C3 + C4 + C5 + C8)$

Let $C = C5 + C6 + C7$

$C' = C1 + C2 + C3 + C4 + C5 + C8$

$f(N) = C * N + C'$

To find the Big-O Notation of f(N),

We can ignore C' because C * N changes with N, but C' is a constant.

Similarly, we can ignore C in C * N since C doesn't greatly influence the growth.

So, Big-O Notation of f(N) or Order of Magnitude of f(N) is **O(N)**

Explanation:

$f(N) = C1 + C2 + C3 + C4 + C5 * (N + 1) + C6 * N + C7 * N + C8$

Let C > 10 * (C1 + C2 + C3 + C4 + C5 + C6 + C7 + C8)
Then, f(N) will always be <= C * N for all N >= 1
According to Upper Bound definition in Asymptotic Notations, f(N) = O(g(N)) if for some constants C & N0 f(N) <= C * g(N) for all N >= N0
Here g(N) = N, N0 = 1 and C = 10
Therefore f(N) = O(N) for all N >= 1

Question 2)

Consider the array below:

24, 12, 38, 3, 45, 2, 56, 8, 100, 6.

a) Construct a binary tree out of the given array. Explain how you do so.

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

c) Analyze the algorithm you used to construct the heap in part b. Provide enough detail.

Assume the array size is n.

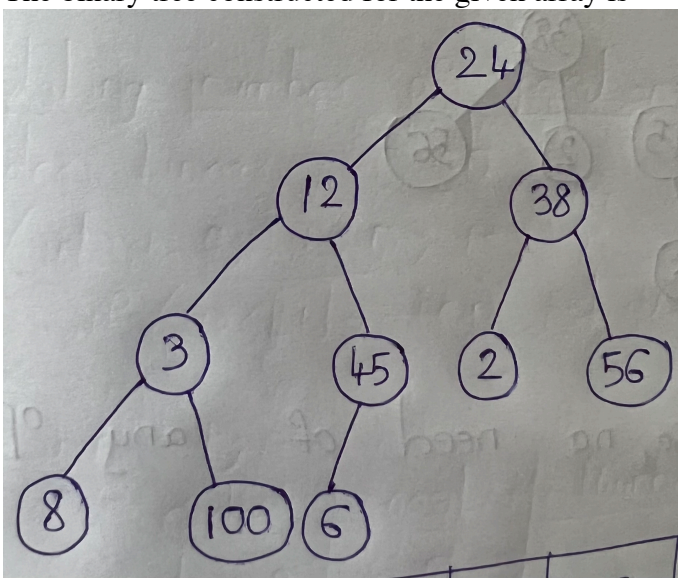
Answer)

(a)

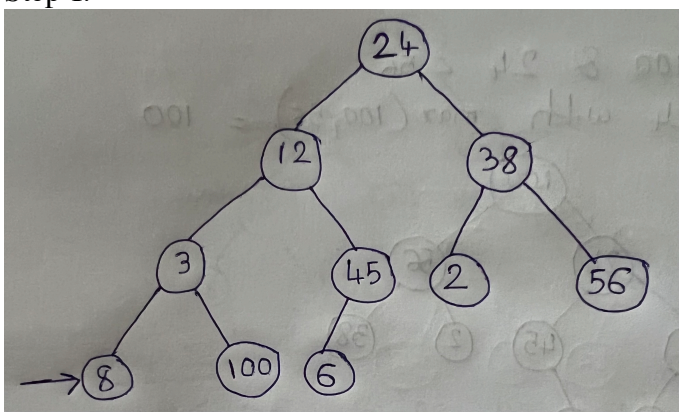
The algorithm to construct a binary tree from a given array is

```
class Node{
    int value;
    Node left;
    Node right;
    Node(int value, Node left, Node right){
        this.value = value;
        this.left = left;
        this.right = right;
    }
}
Node constructTree(int[] values, int index){
    if(index >= values.length) return null;
    Node root = new Node(values[index], null, null);
    root.left = constructTree(values, 2 * index + 1);
    root.right = constructTree(values, 2 * index + 2);
    return root;
}
```

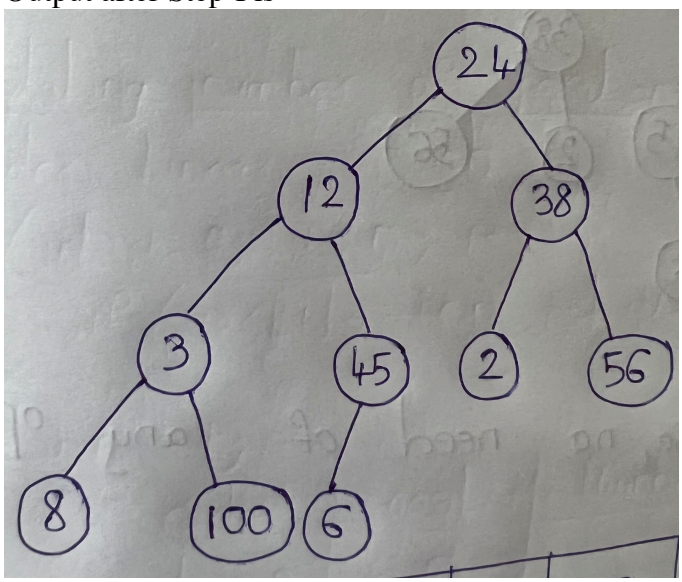
The binary tree constructed for the given array is



(b)
Steps to Convert Binary Tree to Max Heap are
Step-I:

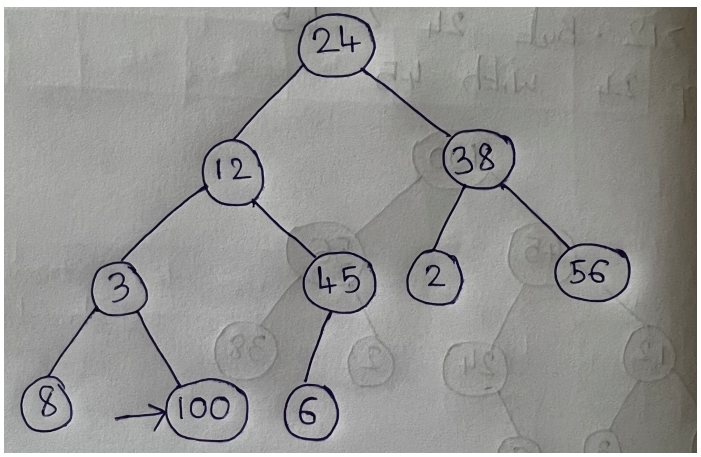


8 is a Leaf Node. So, no need of any operations
Output after Step-I is

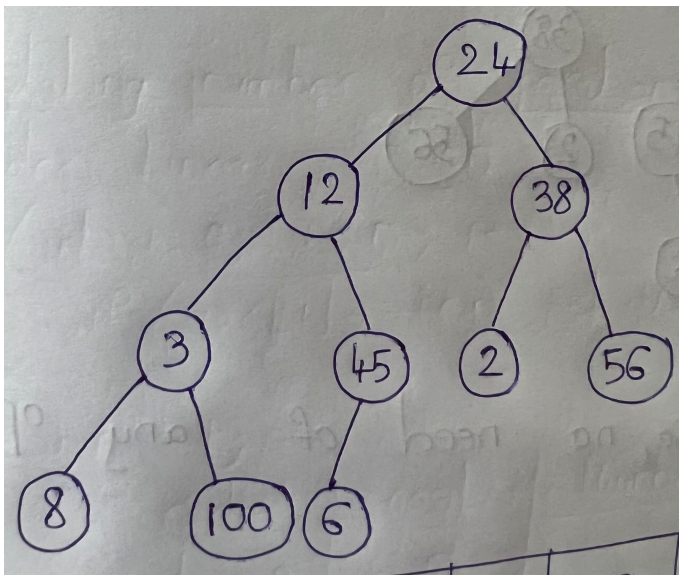


24	12	38	3	45	2	56	8	100	6

Step-II:

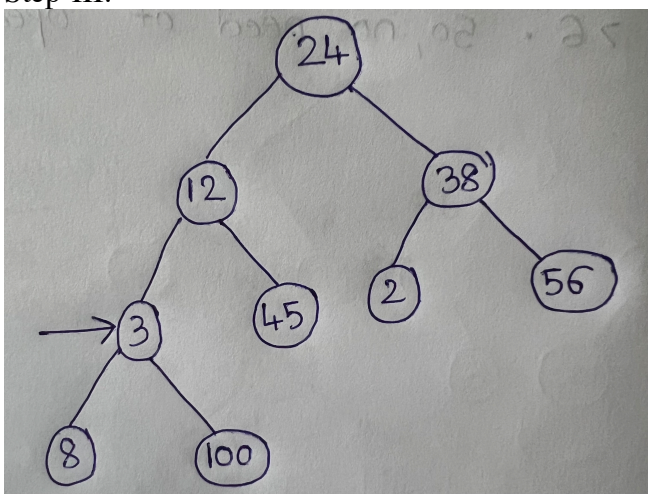


100 is a Leaf Node. So, no need of any operations
Output after Step-II is

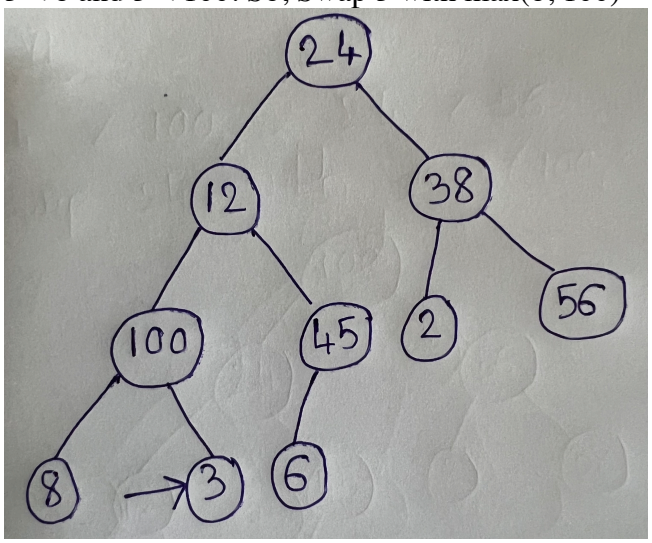


24	12	38	3	45	2	56	8	100	6

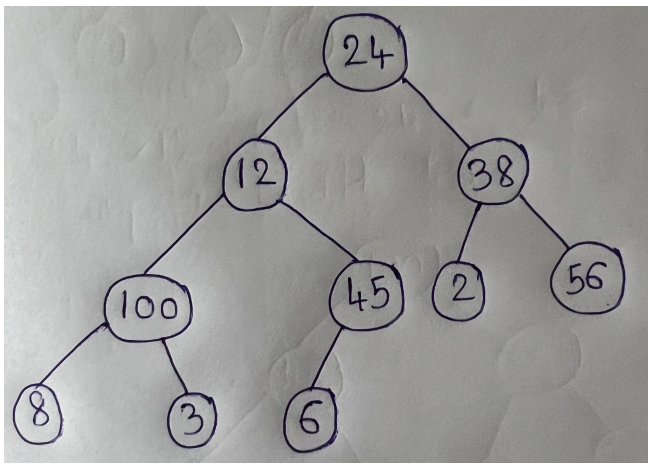
Step-III:



3 < 8 and 3 < 100. So, Swap 3 with max(8, 100) = 100

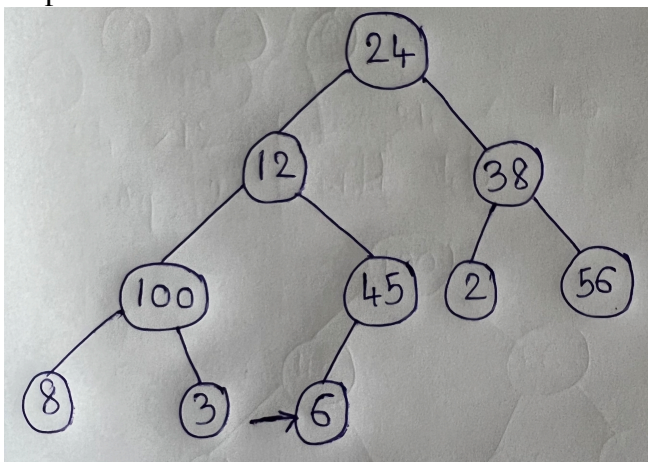


3 is a Leaf Node. So, no need of any operations.
Output after Step-III is

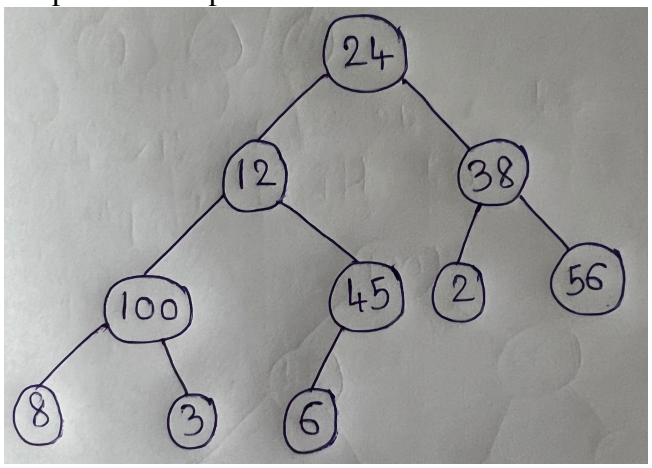


24	12	38	100	45	2	56	8	3	6

Step-IV:

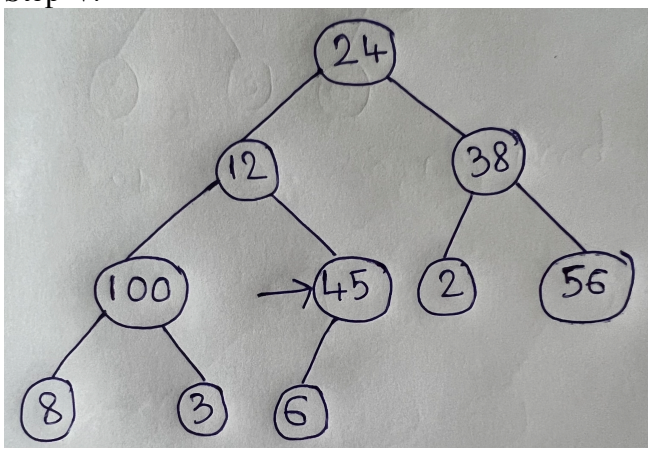


6 is a leaf node. So, no need of any operations.
Output after Step-IV is



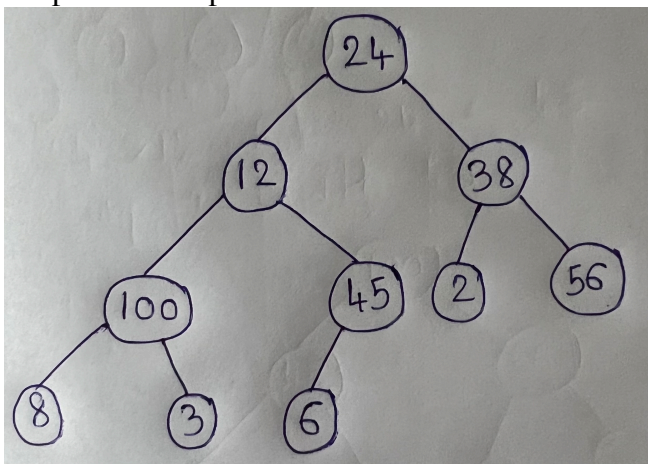
24	12	38	100	45	2	56	8	3	6

Step-V:



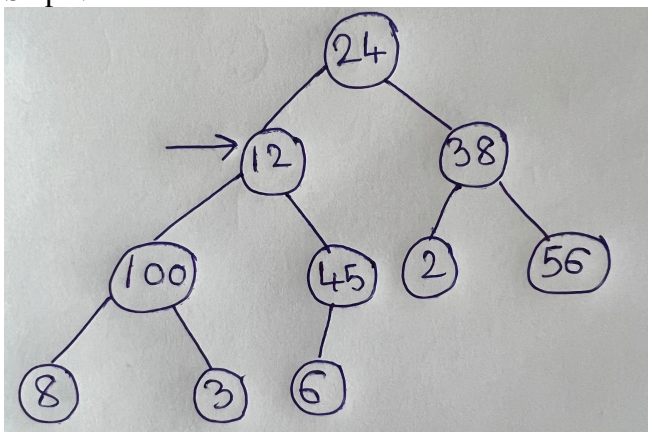
45 > 6. So no need of any operations.

Output after Step-V is

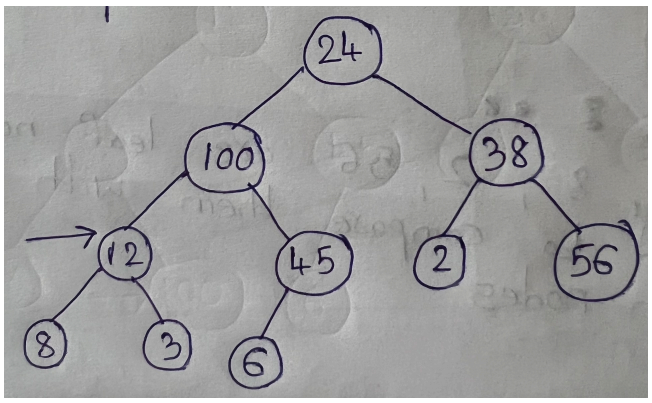


24	12	38	100	45	2	56	8	3	6

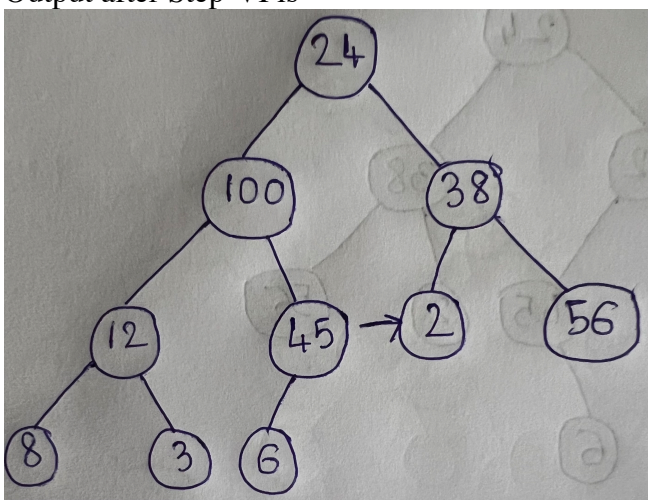
Step-VI:



12 < 100 and 12 < 45. So, Swap 12 with max(100, 45) = 100

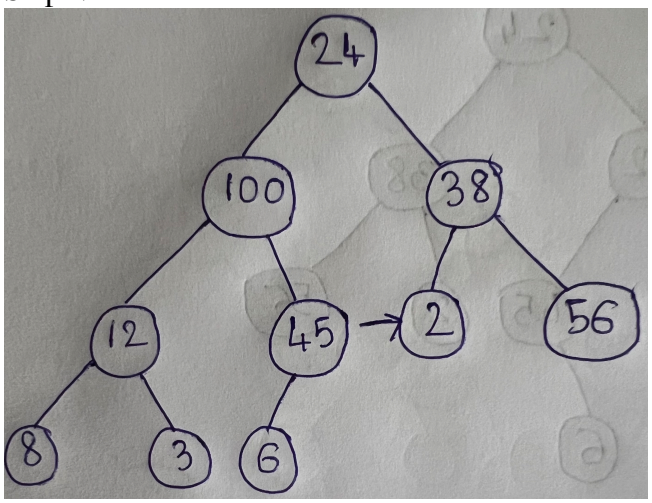


12 > 8 and 12 > 3. So, no need of any operations.
Output after Step-VI is

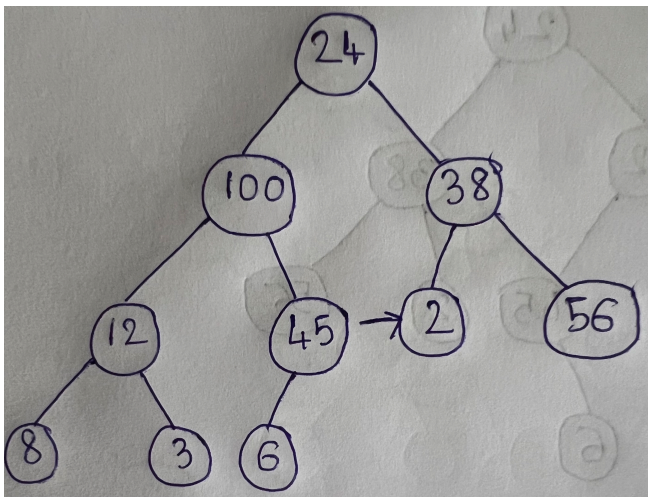


24	100	38	12	45	2	56	8	3	6

Step-VII:

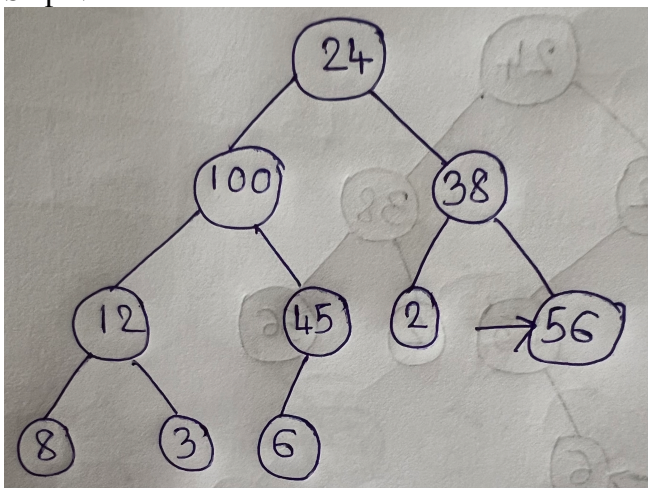


2 is a leaf node. So, no need of any operations.
Output after Step-VII is

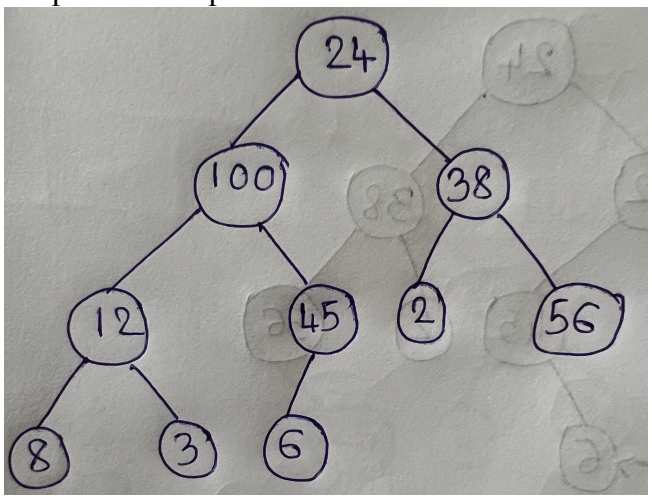


24	100	38	12	45	2	56	8	3	6

Step-VIII:

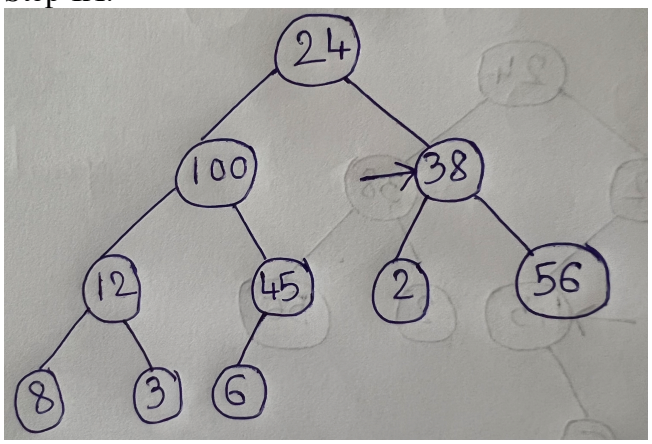


56 is a Leaf Node. So, no need of any operations.
Output after Step-VIII is

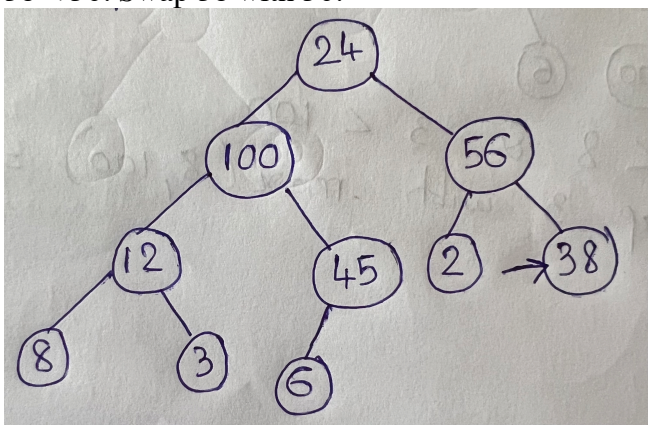


24	100	38	12	45	2	56	8	3	6

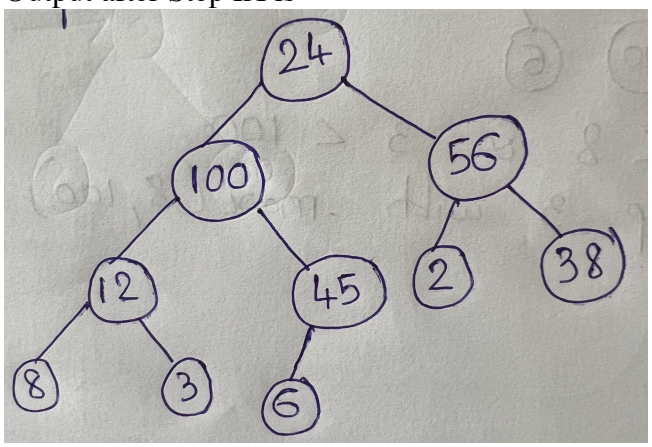
Step-IX:



38 < 56. Swap 38 with 56.

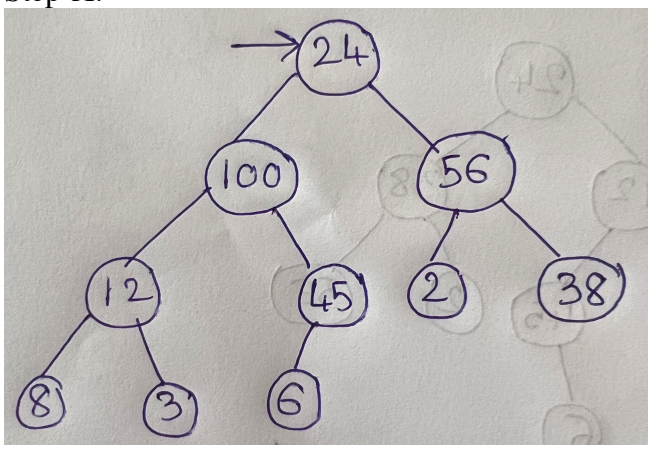


38 is a Leaf Node. So, no need of any operations.
Output after Step IX is

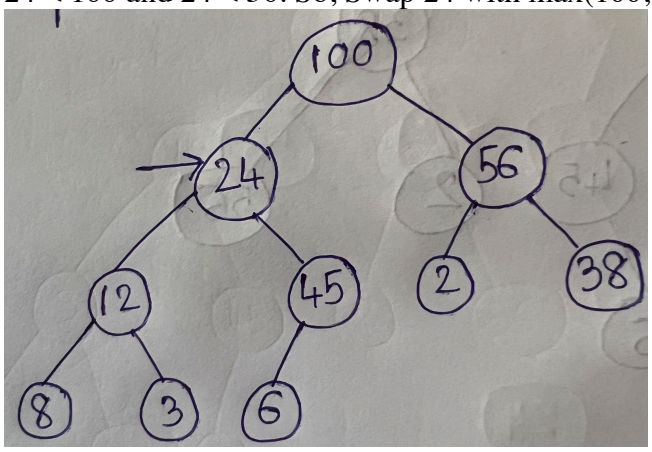


24	100	56	12	45	2	38	8	3	6

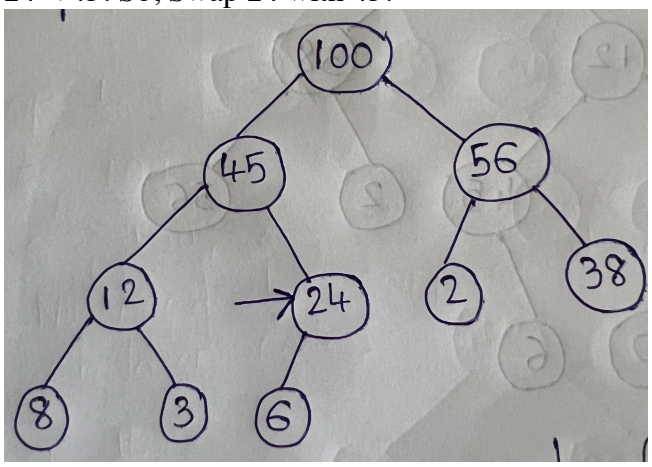
Step-X:



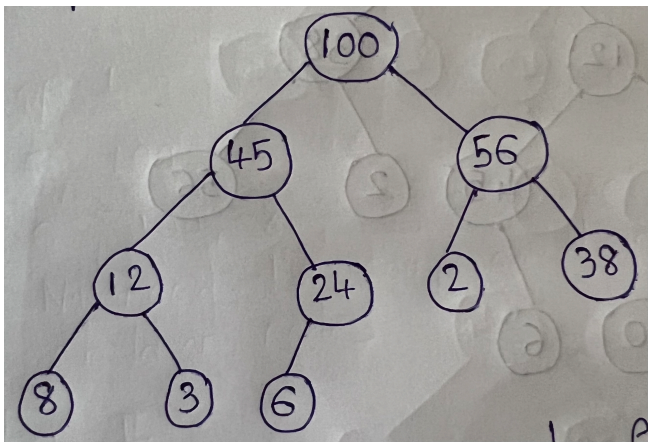
24 < 100 and 24 < 56. So, Swap 24 with max(100, 56) = 100



24 < 45. So, Swap 24 with 45.



24 > 6. So, no need of any operations.
Output after Step-X is



100	45	56	12	24	2	38	8	3	6

(c)

Algorithm I used to convert Binary Tree to Max Heap is

```
void heapify(Node root){
        if(root == null) return;
        heapify(root.left);
        heapify(root.right);
        swapNodes(root);
}
void swapNodes(Node root){
        if(root.left == null && root.right == null) return;
        if(root.left == null){
                if(root.value < root.right.value){
                        int value = root.value;
                        root.value = root.right.value;
                        root.right.value = value;
                        swapNodes(root.right);
                }
        }
        else if(root.right == null){
                if(root.value < root.left.value){
                        int value = root.value;
                        root.value = root.left.value;
                        root.left.value = value;
                        swapNodes(root.left);
                }
        }
        else{
                if(root.value < root.left.value && root.value > root.right.value){
                        int value = root.value;
                        root.value = root.left.value;
                        root.left.value = value;
                        swapNodes(root.left);
                }
                else if(root.value < root.right.value && root.value > root.left.value){
                        int value = root.value;
                        root.value = root.right.value;
```

```
                root.right.value = value;
                swapNodes(root.right);
            }
            else if(root.value < root.right.value && root.value < root.left.value){
                if(root.left.value < root.right.value){
                    int value = root.value;
                    root.value = root.right.value;
                    root.right.value = value;
                    swapNodes(root.right);
                }
                else{
                    int value = root.value;
                    root.value = root.left.value;
                    root.left.value = value;
                    swapNodes(root.left);
                }
            }
        }
    }
```

Analysis of Algorithm:

Let n be the size of given binary tree.
heapify() function is called exactly n number of times, since it needs to go to every node and convert that particular subtree to max heap.
Inside heapify(), we call swapNodes() functions to swap nodes from current root node and their child nodes to maintain heap constraints. swapNodes() at any level of node takes a max of height of that subtree to convert that subtree to a Max Heap. The height of subtree will be log(k) where k is number of nodes in that subtree.
So, time complexity of swapNodes() at worst case scenario will be log(n).
Since we call swapNodes() every time we run heapify(), we have to multiply both functions time complexities to get the time complexity of this algorithm.
Therefore, the time complexity of this algorithm is n * log(n).

Question 3)

(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)).

$f1(n)= n$

$f2(n)= n^{10}$

$f3(n)= 2^n$

$f4(n)=100^n$

$f5(n)=n \log n$

$f6(n)= n^2 \log n$

$f7(n)= n^n$

$f8(n)=n!$

(b) In your arrangement in Q3 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)).

Answer)

(a)

The Ascending order of the given functions according to Growth Rate is,

$f1(n)= n$

$f5(n)=n \log n$

$f6(n)= n^2 \log n$

$f2(n)= n^{10}$

$f3(n)= 2^n$

$f8(n)=n!$

$f4(n)=100^n$

$f7(n)= n^n$

(b)

(i) f1(n) = n, f5(n) = n log n

f1(n) = O(f5(n))

Explanation:

f(n) = O(g(n)) if f(n) <= C * g(n) for all n >= n0

Let C = 1

n	f1(n) = n	C * f5(n) = nlogn
1	1	0
2	2	2
3	3	4.75
4	4	8
5	5	11.6

As we can see for n >= 2, f1(n) is always <= C * f5(n) where C = 1

Therefore, f1(n) = O(f5(n)) = O(nlogn)

(ii) f5(n) = nlogn, f6(n) = $n^2 \log n$

f5(n) = O(f6(n))

Explanation:

f(n) = O(g(n)) if f(n) <= C * g(n) for all n >= n0

Let C = 1

n	f5(n) = nlogn	C* f6(n) = $n^2 \log n$
1	0	0
2	2	4
3	4.75	14.26
4	8	32
5	11.6	58.04

As we can see for n >= 1, f5(n) is always <= C * f6(n) where C = 1

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

(iii) f6(n)= n^2 log n, f2(n)= n^{10}

f6(n) = O(f2(n))

Explanation:

f(n) = O(g(n)) if f(n) <= C * g(n) for all n >= n0

Let C = 1

n	f6(n) = n^2 log n	C * f2(n)= n^{10}
1	0	1
2	4	1024
3	14.26	59049

As we can see for n >= 1, f6(n) is always <= C * f2(n) where C = 1

Therefore, f6(n) = O(f2(n)) = O(n^{10})

(iv) f2(n)= n^{10}, f3(n)= 2^n

f2(n) = O(f3(n))

Explanation:

f(n) = O(g(n)) if f(n) <= C * g(n) for all n >= n0

Let C = 1

For small numbers like 10, we can clearly see that f2(n) > f3(n). But, if we increase the n value, at some point the inequality changes.

Apply log() on both functions

log($_n$10) and log($_2$n)

== 10log(n) and nlog(2)

== 10log(n) and n

Now, consider n == 64

Then, 10log(64) will be 60 because 64 is 2^6.

So, value we get from f2(n) is 60 which is less than f3(n) which is 64.

We can say that for n >= 64, f2(n) is always <= f3(n) where C = 1

Therefore, f2(n) = O(f3(n)) = O($_2$n)

(v) f3(n)= 2^n ,f8(n)=n!

f3(n) = O(f8(n))

Explanation:

f(n) = O(g(n)) if f(n) <= C * g(n) for all n >= n0

Let C = 1

n	f3(n)= 2^n	C * f8(n)=n!
1	2	1
2	4	2

3	8	6
4	16	24
5	32	120

As we can see for n >= 4, f3(n) is always <= C * f8(n) where C = 1
Therefore, f3(n) = O(f8(n)) = O(n!)

(vi) f8(n)=n!, f4(n)=100^n
f8(n) = O(f4(n))
Explanation:
f(n) = O(g(n)) if f(n) <= C * g(n) for all n >= n0
Let C = 1

n	f8(n)=n!	C * f4(n)=100^n
1	1	100
2	4	10000
3	6	1000000
4	24	100000000
5	120	10000000000

As we can see for n >= 1, f8(n) is always <= C * f4(n) where C = 1
Therefore, f8(n) = O(f4(n)) = O(100^n)

(vii) f4(n)=100^n ,f7(n)= n^n
f4(n) = O(f7(n))
Explanation:
f(n) = O(g(n)) if f(n) <= C * g(n) for all n >= n0
Let C = 1

n	f4(n)=100^n	C * f7(n) = n^n
100	100 power 100	100 power 100
101	100 power 101	101 power 101
102	100 power 102	102 power 101

As we can see for n >= 100, f4(n) is always <= C * f7(n) where C = 1
Therefore, f4(n) = O(f7(n)) = O(n^n)