

**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

$$\begin{aligned}f(N) &= C1 + C2 + C3 + C4 + C5 * (N + 1) + C6 * N + C7 * N + C8 \\&= (C5 + C6 + C7) * N + (C1 + C2 + C3 + C4 + C5 + C8)\end{aligned}$$

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  $C > 10 * (C_1 + C_2 + C_3 + C_4 + C_5 + C_6 + C_7 + C_8)$

Then,  $f(N)$  will always be  $\leq C * N$  for all  $N \geq 1$

According to Upper Bound definition in Asymptotic Notations,  $f(N) = O(g(N))$  if for some constants  $C$  &  $N_0$   $f(N) \leq C * g(N)$  for all  $N \geq N_0$

Here  $g(N) = N$ ,  $N_0 = 1$  and  $C = 10$

Therefore  $f(N) = O(N)$  for all  $N \geq 1$

## 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);
```

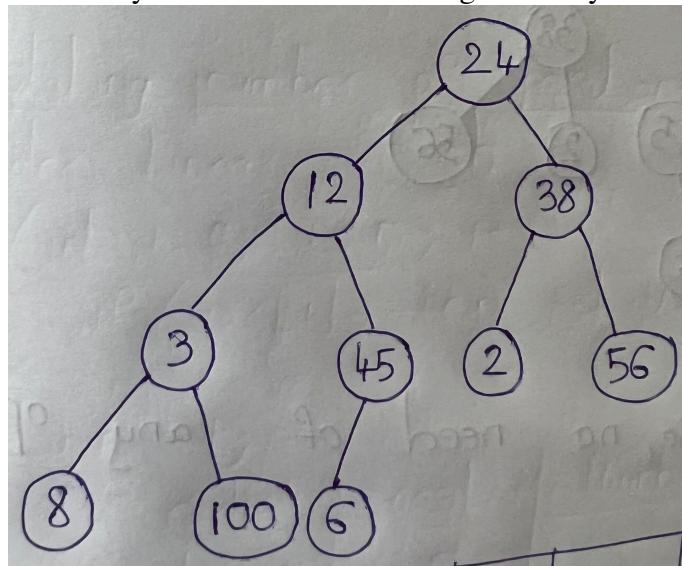
```
    root.left = constructTree(values, 2 * index + 1);
```

```
    root.right = constructTree(values, 2 * index + 2);
```

```
    return root;
```

```
}
```

The binary tree constructed for the given array is



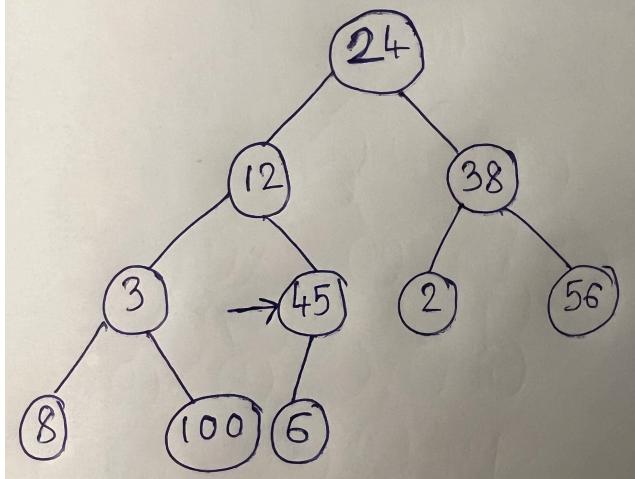
(b)

Steps to Convert Binary Tree to Max Heap are

Step-I:

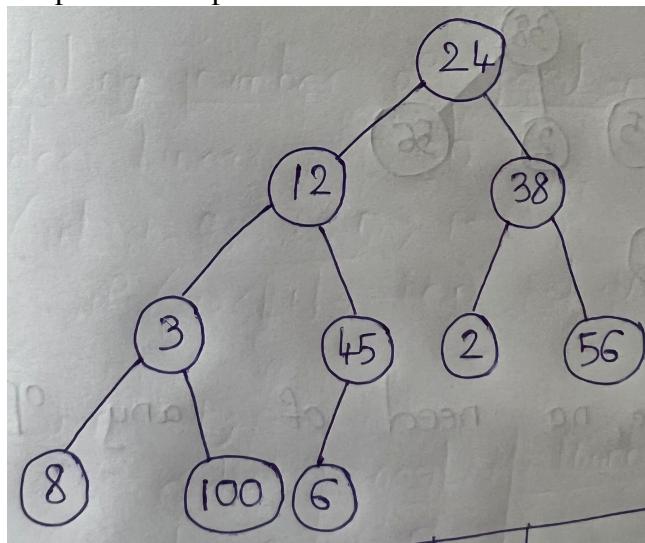
We start at index  $n / 2 - 1$ . Given array has size 10 i.e;  $n = 10$ . So, we start at index  $10 / 2 - 1 = 4$ .

Element at index 4 is 45



$45 > 6$ . So, no need of any operations.

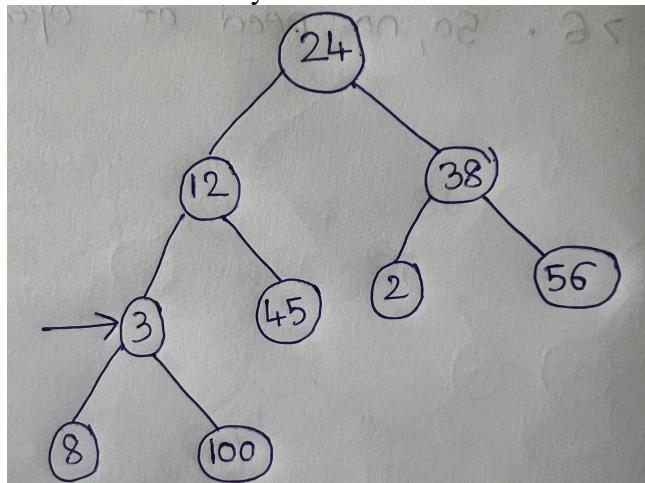
Output after Step-I is



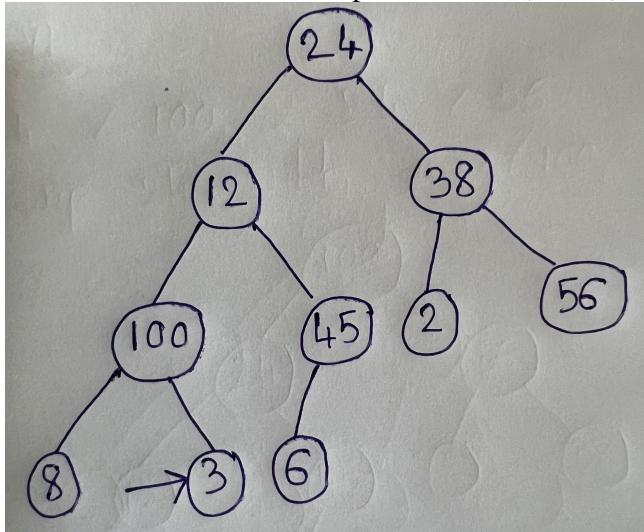
24	12	38	3	45	2	56	8	100	6
----	----	----	---	----	---	----	---	-----	---

Step-II:

Decrement index by 1. Index = 3. Element at index 3 is 3

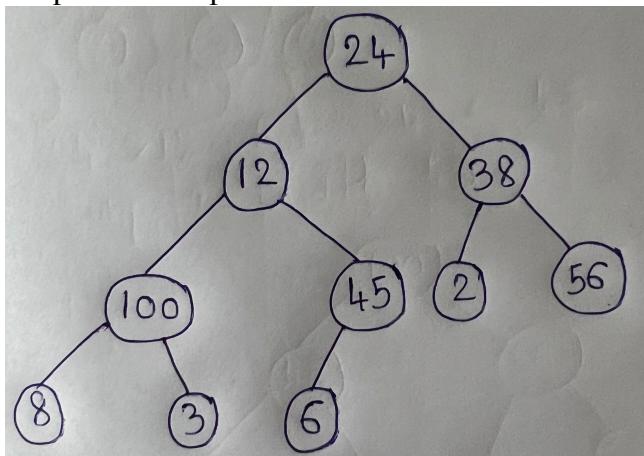


$3 < 8$  and  $3 < 100$ . So, Swap 3 with  $\max(8, 100) = 100$



3 is a Leaf Node. So, no need of any operations.

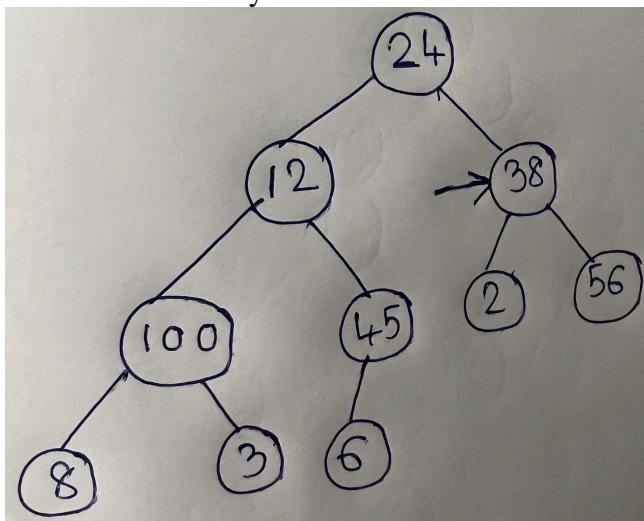
Output after Step-II is



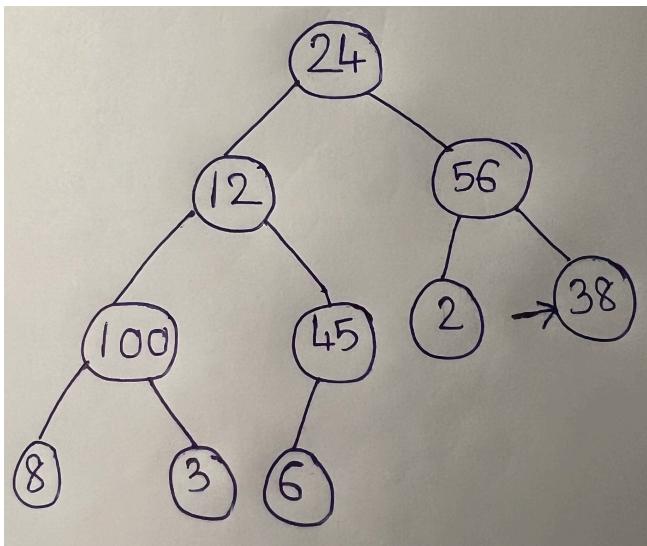
24	12	38	100	45	2	56	8	3	6
----	----	----	-----	----	---	----	---	---	---

Step-III:

Decrement index by 1. Index = 2. Element at index 2 is 38

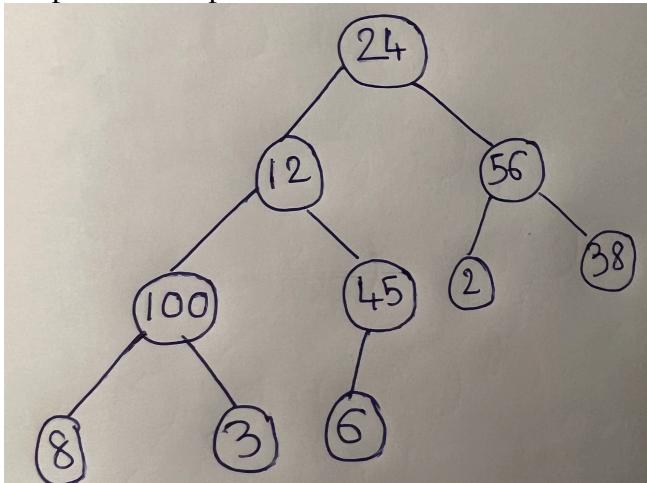


$38 < 56$ . Swap 38 with 56.



38 is a leaf node. So, no need of any operations.

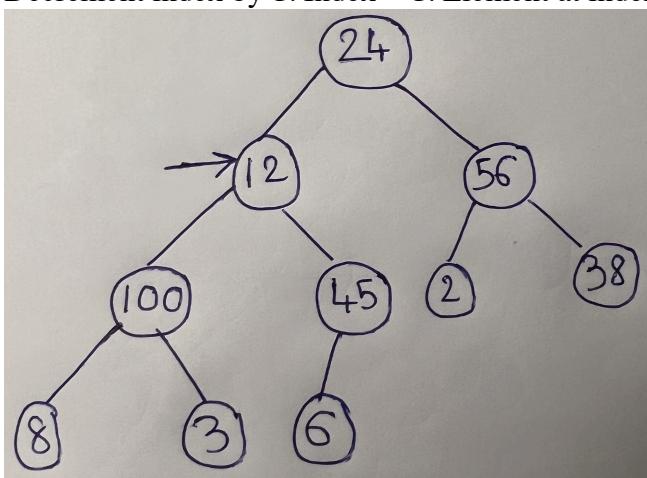
Output after Step-III is



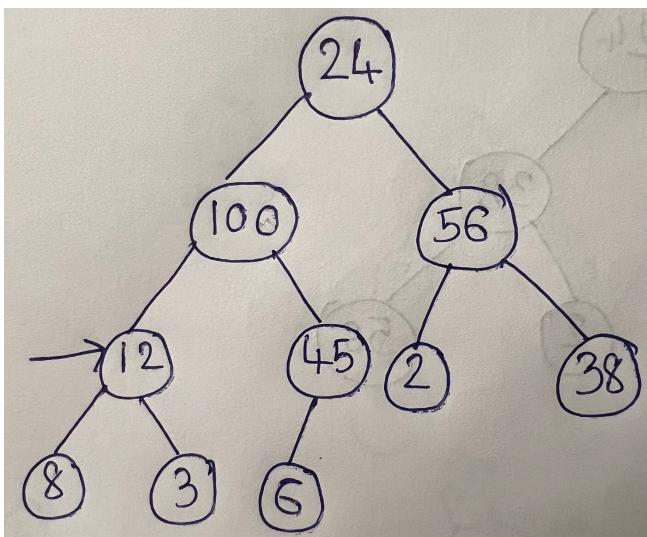
24	12	56	100	45	2	38	8	3	6
----	----	----	-----	----	---	----	---	---	---

Step-IV:

Decrement index by 1. Index = 1. Element at index 1 is 12

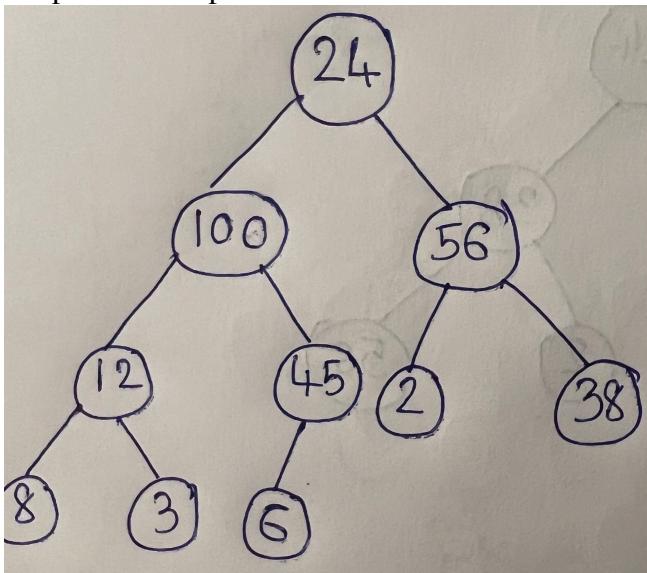


$12 < 100$  and  $12 < 45$ . So, Swap 12 with  $\max(100, 45) = 100$



$12 > 8$  and  $12 > 3$ . So, no need of any operations.

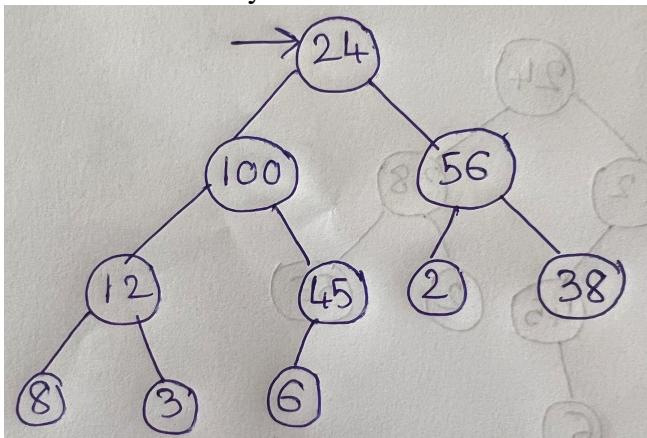
Output after Step-IV is



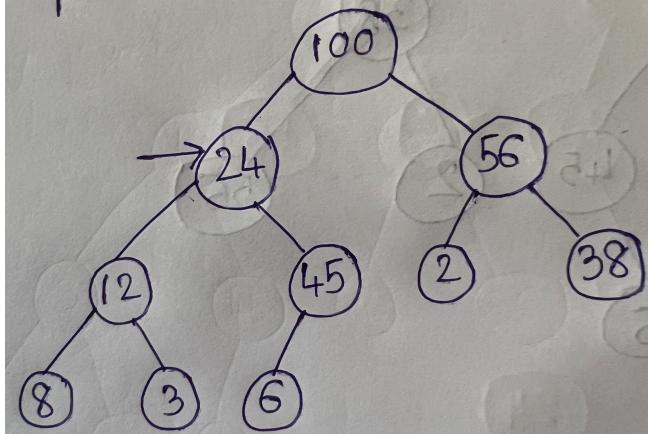
24	100	56	12	45	2	38	8	3	6
----	-----	----	----	----	---	----	---	---	---

Step-V:

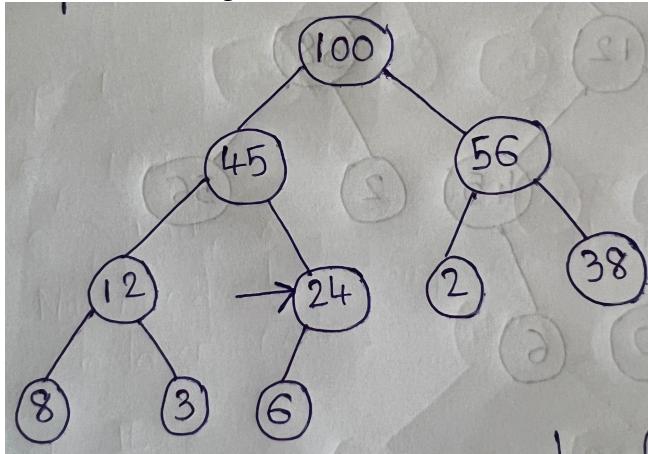
Decrement index by 1. Index = 0. Element at index 1 is 24



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

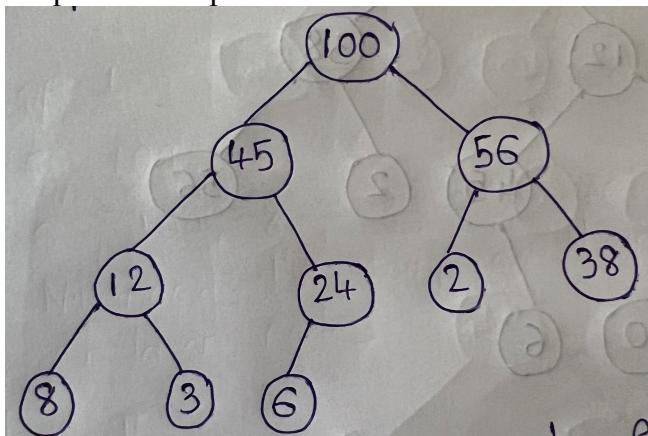


$24 < 45$ . So, Swap 24 with 45.



$24 > 6$ . So, no need of any operations.

Output after Step-V is



100	45	56	12	24	2	38	8	3	6
-----	----	----	----	----	---	----	---	---	---

(c)

Algorithm I used to convert Binary Tree to Max Heap is

```
static void convertToMaxHeap(int[] array){  
    for(int i = array.length / 2 - 1; i >= 0; i--){  
        heapify(array, i);  
    }  
}
```

```
static void heapify(int[] array, int i){  
    int largest = i;
```

```

int leftChild = 2 * i + 1;
int rightChild = 2 * i + 2;

if(leftChild < array.length && array[leftChild] > array[largest]){
    largest = leftChild;
}
if(rightChild < array.length && array[rightChild] > array[largest]){
    largest = rightChild;
}

if(largest == i) return;
int value = array[i];
array[i] = array[largest];
array[largest] = value;
heapify(array, largest);
}

```

### Analysis of Algorithm:

Let  $n$  be the size of array

Number of basic operations for convertToMaxHeap()

Code	Cost	Number of time it runs
int $i = \text{array.length} / 2 - 1$	$C_1$	1
$i \geq 0$	$C_2$	$n / 2 + 1$
$i = i - 1$	$C_3$	$n / 2$
heapify(array, $i$ )	$C_4$	$n / 2$

$$f(n) = (C_2 / 2 + C_3 / 2 + C_4 / 2) * n + C_1 + C_2$$

$$\text{Let } C > 10 * (C_1 + C_2 + C_3 + C_4)$$

Then,  $f(n)$  will always be  $\leq C * n$  for all  $n \geq 1$

Therefore according to Upper Bound rule in Asymptotic Notations,  $f(n) = O(n)$

But, heapify() is a function call. So, the overall time complexity of  $f(n)$  depends on the time complexity of heapify() as well.

In heapify() we swap values of current index and its child indices to maintain heap constraints. This algorithm at any level of node takes a maximum of height of that subtree steps 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 heapify() at worst case scenario will be  $\log(n)$ .

Since we call heapify() every time we run convertToMaxHeap(), 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  $f(n) = O(g(n))$ .

$$f_1(n) = n$$

$$f_2(n) = n^{10}$$

$$f_3(n) = 2^n$$

$$f_4(n) = 100^n$$

$$\begin{aligned}f5(n) &= n \log n \\f6(n) &= n^2 \log n \\f7(n) &= n^n \\f8(n) &= n!\end{aligned}$$

(b) In your arrangement in Q3 part (a), if you have  $f_i(n) \leq f_j(n)$  that means  $f_i(n) = O(f_j(n))$ . Pick **each** of such **consecutive** pair of functions in your arrangement (let's call them  $f_i(n)$  and  $f_j(n)$ , where  $f_i(n) \leq f_j(n)$  in the arrangement) and prove **formally** why you believe  $f_i(n) = O(f_j(n))$ .

### Answer)

(a)

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

$$\begin{aligned}f1(n) &= n \\f5(n) &= n \log n \\f6(n) &= n^2 \log n \\f2(n) &= n^{10} \\f3(n) &= 2^n \\f8(n) &= n! \\f4(n) &= 100^n \\f7(n) &= n^n\end{aligned}$$

(b)

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

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

Explanation:

Consider,

$$\begin{aligned}f1(n) &< f5(n) \\n &< n \log n \\&\text{Divide both sides by } n \\1 &< \log n \\2 &< n \text{ [Since } 2^x = n \implies x = \log n\text{]}\end{aligned}$$

As we can see for  $n \geq 3$ ,  $f1(n)$  is always  $\leq C * f5(n)$  where  $C = 1$

Therefore,  $f1(n) = O(f5(n)) = O(n \log n)$

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

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

Explanation:

Both functions have  $n \log n$  in common. So, which is larger is dependent on the remaining  $n$  in  $f6(n)$ .

Consider,

$$\begin{aligned}f5(n) &< f6(n) \\n \log n &< n^2 \log n \\&\text{Divide both sides by } n \log n \\1 &< n \\&\text{So, } f5(n) \text{ will be } < f6(n) \text{ for all } n > 1 \\&\text{At } n = 1, \log n \text{ will be } 0. \text{ So, both functions will be equal to } 0\end{aligned}$$

As we can see for  $n \geq 1$ ,  $f_5(n)$  is always  $\leq C * f_6(n)$  where  $C = 1$

Therefore,  $f_5(n) = O(f_6(n)) = O(n^2 \log n)$

(iii)  $f_6(n) = n^2 \log n$ ,  $f_2(n) = n^{10}$

$f_6(n) = O(f_2(n))$

Explanation:

Consider,

$$f_6(n) < f_2(n)$$

$$n^2 \log n < n^{10}$$

Divide both sides by  $n^2$

$$\log n < n^8$$

We know this will be true for all  $n > 1$ . Because  $\log n$  will be always  $< n$  and  $n^8$  is  $n$  multiplied by  $n$  8 times. So,  $n^8$  will always be  $> n$ .

As we can see for  $n \geq 1$ ,  $f_6(n)$  is always  $\leq C * f_2(n)$  where  $C = 1$

Therefore,  $f_6(n) = O(f_2(n)) = O(n^{10})$

(iv)  $f_2(n) = n^{10}$ ,  $f_3(n) = 2^n$

$f_2(n) = O(f_3(n))$

Explanation:

Consider,

$$f_2(n) < f_3(n)$$

$$n^{10} < 2^n$$

Apply  $\log()$  on both functions

$$\log(n^{10}) < \log(2^n)$$

$$10\log n < n\log 2$$

$$10\log(n) < n$$

We know that  $\log n$  is always less than  $n$ . So, the inequality depends on number where  $10\log n$  will become  $< n$

Now, consider  $n = 64$

Then,  $10\log(64)$  will be 60 because  $64 = 2^6$ .

So, value we get from  $f_2(n)$  is 60 which is less than  $f_3(n)$  which is 64.

We can say that for  $n \geq 64$ ,  $f_2(n)$  is always  $\leq f_3(n)$  where  $C = 1$

Therefore,  $f_2(n) = O(f_3(n)) = O(2^n)$

(v)  $f_3(n) = 2^n$ ,  $f_8(n) = n!$

$f_3(n) = O(f_8(n))$

Explanation:

$$f_3(n) = 2^n = 2 * 2 * \dots * 2$$

$$f_8(n) = n! = 1 * 2 * \dots * n$$

As we can see both have exactly  $n$  multiplications. But in  $f_3(n)$  we multiply by only  $2$ ,  $n$  number of times. Whereas in  $f_8(n)$  we multiply from  $1$  to  $n$ . So, as  $n$  increases

$f_8(n)$  increases more rapidly because if we change  $n$  to  $n + 1$ ,  $f_3(n)$  will have only 1 more multiplication by 2. But in  $f_8(n)$  we will multiply by  $n$ .

Let  $n = 3$ . Then  $f_3(n) = 8$  and  $f_8(n) = 6$

But if  $n = 4$ ,  $f_3(n) = 16$  and  $f_8(n) = 24$  and it will keep on increasing for increase in  $n$  value.

As we can see for  $n \geq 4$ ,  $f_3(n)$  is always  $\leq C * f_8(n)$  where  $C = 1$   
 Therefore,  $f_3(n) = O(f_8(n)) = O(n!)$

(vi)  $f_8(n) = n!$ ,  $f_4(n) = 100^n$

$$f_8(n) = O(f_4(n))$$

Explanation:

$$f(n) = O(g(n)) \text{ if } f(n) \leq C * g(n) \text{ for all } n \geq n_0$$

Let  $C = 1$

$n$	$f_8(n) = n!$	$C * f_4(n) = 100^n$
1	1	100
2	4	10000
3	6	1000000
4	24	100000000
5	120	10000000000

As we can see for  $n \geq 1$ ,  $f_8(n)$  is always  $\leq C * f_4(n)$  where  $C = 1$

$$\text{Therefore, } f_8(n) = O(f_4(n)) = O(100^n)$$

(vii)  $f_4(n) = 100^n$ ,  $f_7(n) = n^n$

$$f_4(n) = O(f_7(n))$$

Explanation:

Consider,

$$f_4(n) < f_7(n)$$

$$100^n < n^n$$

Apply  $n$ th root on both sides

$$100 < n$$

So,  $f_4(n)$  will be  $< f_7(n)$  for all  $n > 100$

As we can see for  $n \geq 101$ ,  $f_4(n)$  is always  $\leq C * f_7(n)$  where  $C = 1$

$$\text{Therefore, } f_4(n) = O(f_7(n)) = O(n^n)$$