**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.

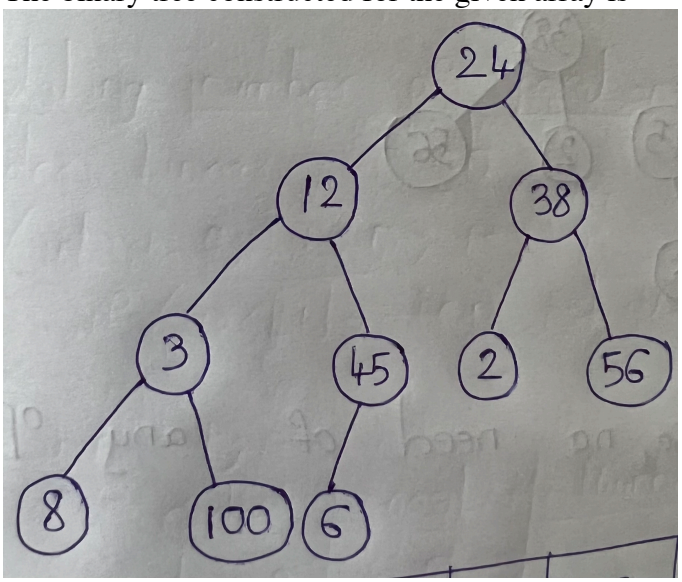
Answer)

(a)

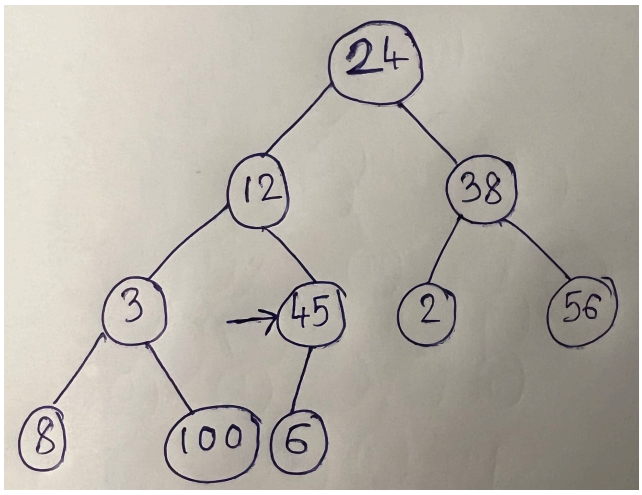
The algorithm to construct a binary tree from a given array is

```
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;
}
```

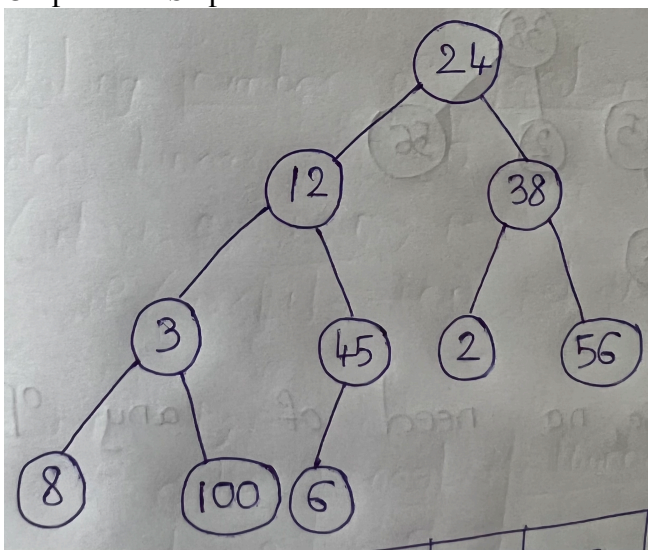
The binary tree constructed for the given array is



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

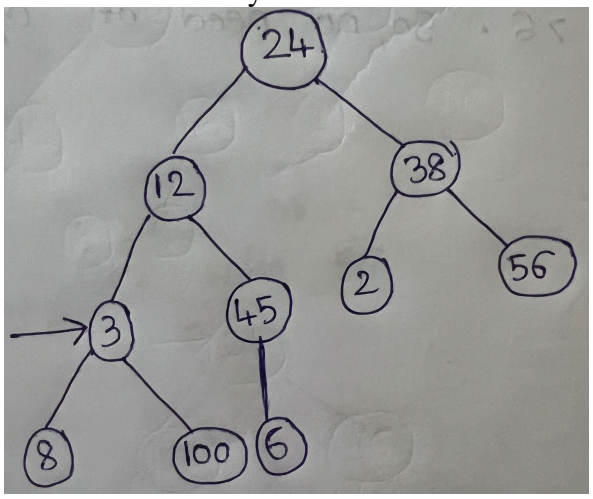



45 > 6. So, no need of any operations.
Output after Step-I is

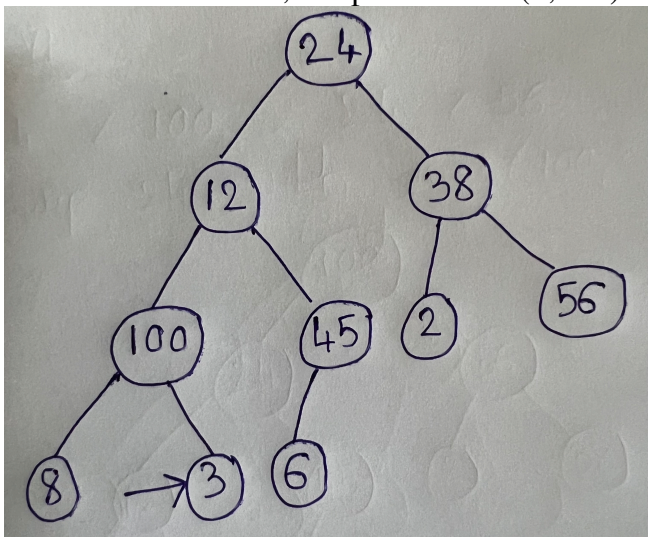



| 24 | 12 | 38 | 3 | 45 | 2 | 56 | 8 | 100 | 6 |
|---|---|---|---|---|---|---|---|---|---|

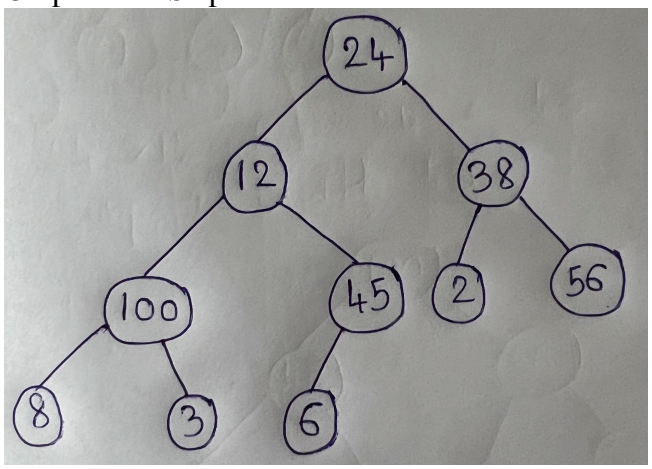
Step-II:
Decrement index by 1. Index = 3. Element at index 3 is 3



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

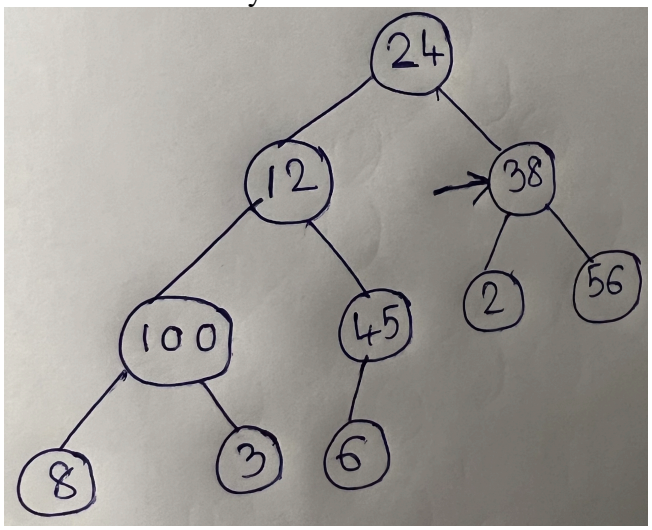



3 is a Leaf Node. So, no need of any operations.
Output after Step-II is

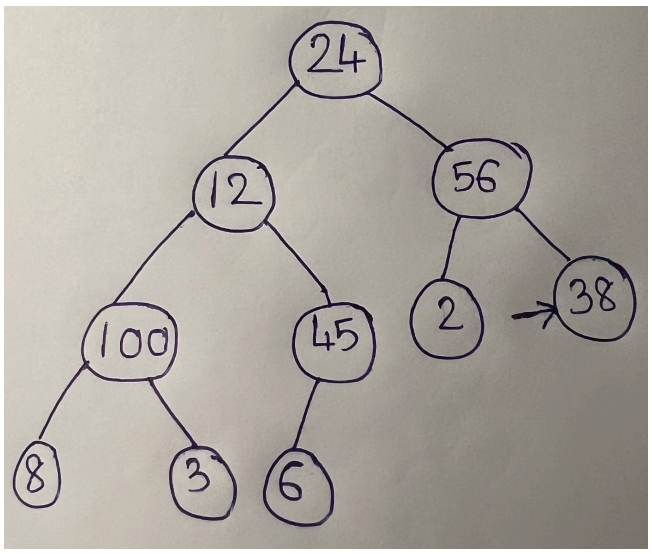



| 24 | 12 | 38 | 100 | 45 | 2 | 56 | 8 | 3 | 6 |
|---|---|---|---|---|---|---|---|---|---|

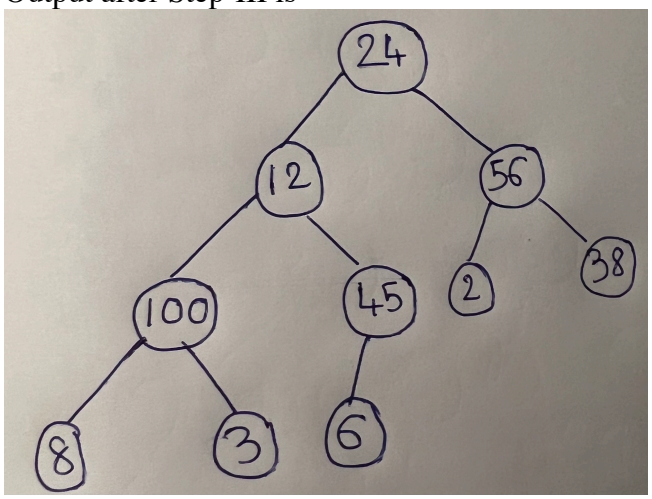
Step-III:
Decrement index by 1. Index = 2. Element at index 2 is 38




38 < 56. Swap 38 with 56.

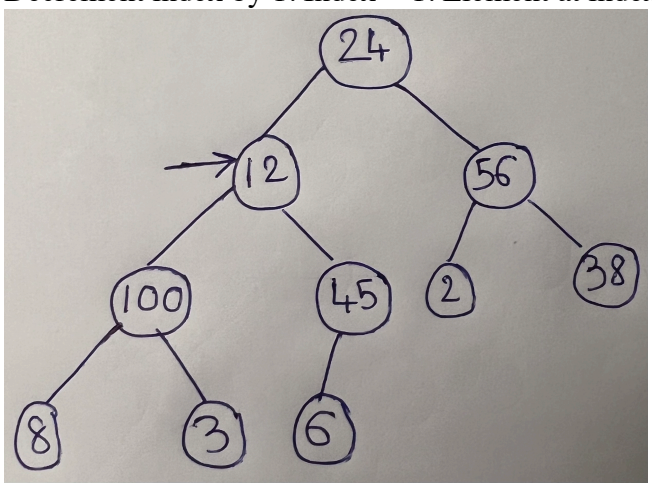



38 is a leaf node. So, no need of any operations.
Output after Step-III is

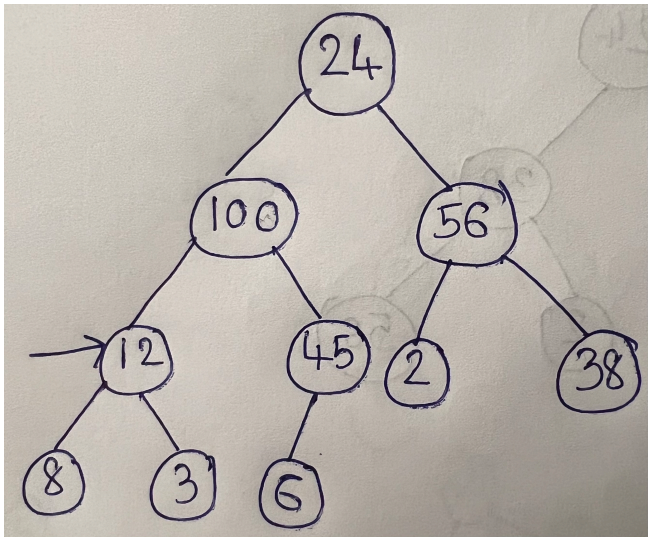



| 24 | 12 | 56 | 100 | 45 | 2 | 38 | 8 | 3 | 6 |
|---|---|---|---|---|---|---|---|---|---|

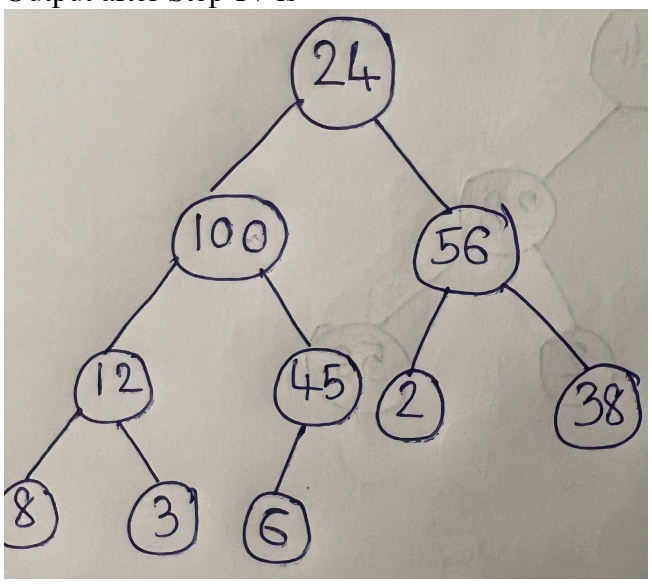
Step-IV:
Decrement index by 1. Index = 1. Element at index 1 is 12




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

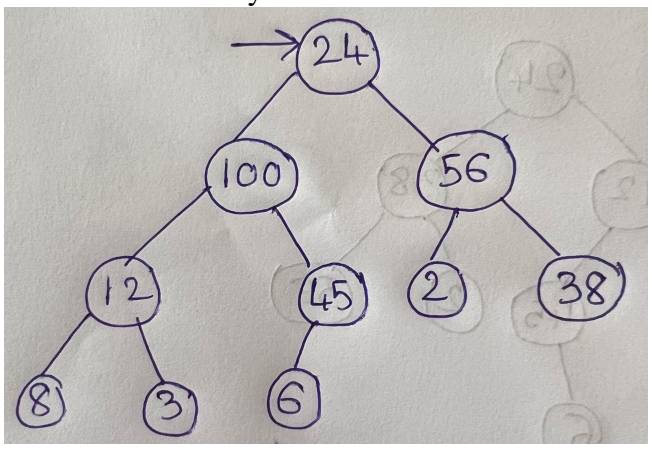



12 > 8 and 12 > 3. So, no need of any operations.
Output after Step-IV is

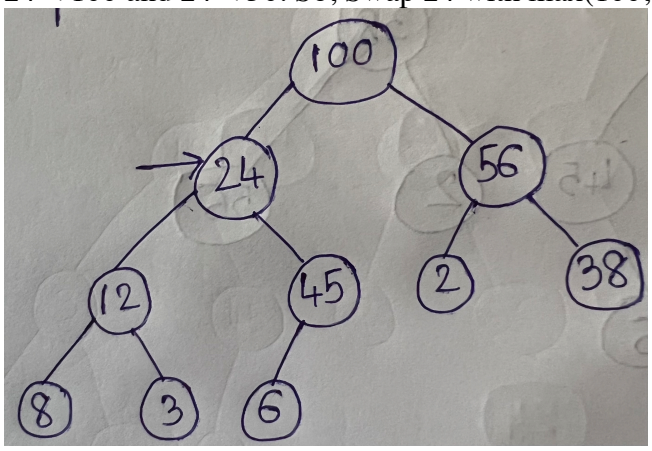



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 24 | 100 | 56 | 12 | 45 | 2 | 38 | 8 | 3 | 6 |

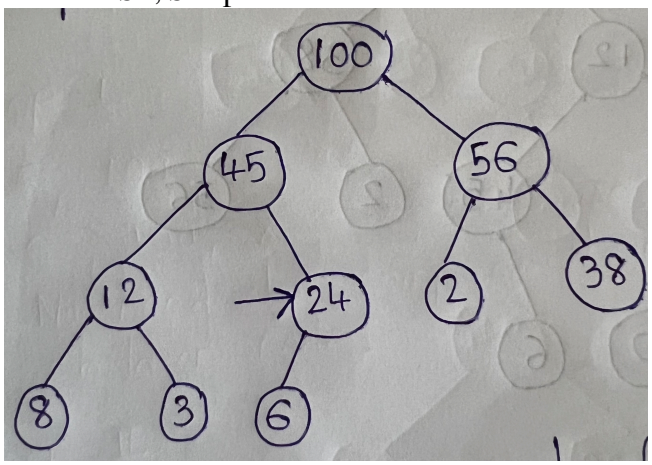
Step-V:
Decrement index by 1. Index = 0. Element at index 1 is 24



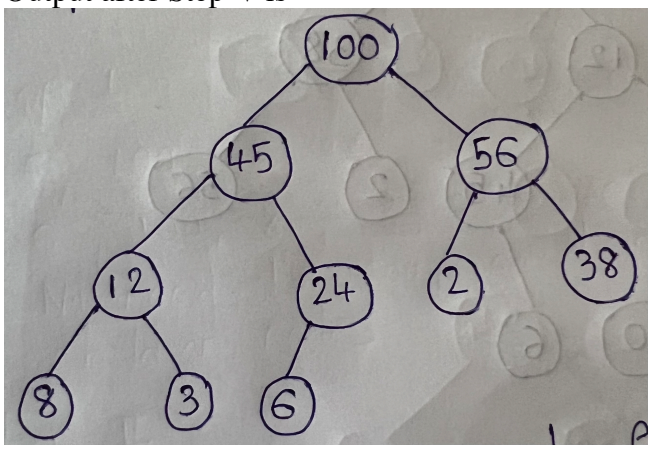
24 < 100 and 24 < 56. So, Swap 24 with max(100, 56) = 100




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 = array.length / 2 - 1 | C1 | 1 |
| i >= 0 | C2 | n / 2 + 1 |
| i-=1 | C3 | n / 2 |
| heapify(array, i) | C4 | n / 2 |

f(n) = ( C2 / 2 + C3 / 2 + C4 / 2) * n + C1 + C2

Let C > 10 * (C1 + C2 + C3 + C4)

Then, f(n) will always be <= C * n for all >= 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) 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:

Consider,

f1(n) < f5(n)

n < nlogn

Divide both sides by n

1 < logn

2 < n [Since 2^x = n === x = logn]

As we can see for n >= 3, 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:

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

Consider,

f5(n) < f6(n)

nlogn < n^2logn

Divide both sides by nlogn

1 < n

So, f5(n) will be < f6(n) for all n > 1

At n = 1, logn will be 0. So, both functions will be equal to 0

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:

Consider,

f6(n) < f2(n)

n^2logn < n^10

Divide both sides by n^2

logn < n^8

We know this will be true for all n > 1. Because logn 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 >= 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:

Consider,

f2(n) < f3(n)

n^10 < 2^n

Apply log() on both functions

$\log_n(10) < \log_2(n)$

10logn < nlog2

10log(n) < n

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

f3(n) = 2^n = 2 * 2 * …. 2.

f8(n) = n! = 1 * 2 * …. n

As we can see both have exactly n multiplications. But in f3(n) we multiply by only 2, n number of times. Whereas in f8(n) we multiply from 1 to n. So, as n increases

f8(n) increases more rapidly because if we change n to n + 1, f3(n) will have only 1 more multiplication by 2. But in f8(n) we will multiply by n.

Let n = 3. Then f3(n) = 8 and f8(n) = 6

But if n = 4, f3(n) = 16 and f8(n) = 24 and it will keep on increasing for increase in n value.

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:

Consider,

f4(n) < f7(n)

100^n < n^n

Apply nth root on both sides

100 < n

So, f4(n) will be < f7(n) for all n > 100

As we can see for n >= 101, f4(n) is always <= C * f7(n) where C = 1

Therefore, f4(n) = O(f7(n)) = O($n^n$)