Algorithm

Algorithm :

A formula or set of steps for solving a particular problem. To be an algorithm, a set of rules must be unambiguous and have a clear stopping point.

Characteristics of an algorithm :

Donald Ervin Knuth has given a list of five properties for Algorithm, these properties are:

1. **Input :**

The Input is the data to be transformed during the Computation to produce the output. An Algorithm should have 0 or more well-defined inputs. Input precision requires that we know that what kind of data, how much and what form the data should be

2. **Output :**

One always expects output/result (Expected value /Quantities) in terms of output from an algorithm. The result may be obtained at different stages of the algorithm. If some result is from the intermediate stage of the Operation then it is known as intermediate result and result obtained at the end of algorithm is known as end result. The Output is expected value/Quantities always have a specified relation to the inputs.

3. finiteness :

An algorithm must always terminate after a finite number of steps or instructions. It Means after every step one reach closer to solution of the problem and after a finite number of steps algorithm reaches to an end point.

4. Definiteness :

Each step of an algorithm must be precisely defined. It is done by well thought actions to be performed at each step of the algorithm. Also the actions are defined unambiguosly for each activity in the algorithm.

5. Effectiveness :

Algorithms to be developed / written using basic operations. Actually operations should be basic so that even they can in principle be done exactly and in a finite amount of time by a person, by using paper and pencil only.

difference between Algorithm and Program :

Algorithm	program.
1. step by step procedure is known as Algorithm	1. Step by step Instruction is known as program.
2. Domain knowledge is needed	2. Knowledge of programming Language is required.
3. It is Software independent	3. It is Software dependent
4. Syntax free	4. It follows Syntax.
5. Design level	5. Implementation level.

Algorithm	program.
6. Algorithm is for human reader to easy understand	6. program is for the Computere to Execute directly.
7. easy to understand	7. It is difficult to understand

pseudo Code for Expressing the algorithm :

pseudo Code :

It is simply an implementation of an algorithm in the form of annotations and informative text written in plain English. It has no syntax like any of the programming Language and thus can't be Compiled or interpreted by the Computer.

```
//This program calculates the lowest Common multiple
// for excessively long input values.
import java.util.*;
public class LowestCommonMultiple {

    private static long
    lcmNaive (long numberOne, long numberTwo)
    {
     long lowestCommonMultiple;
     lowestCommonMultiple
            = (numberOne * numberTwo)
              /greatestCommonDivisor (numberOne,
                                      numberTwo);
     return lowestCommonMultiple;
    }
    private static long
    greatestCommonDivisor (long numberOne, long numberTwo)
    {
        if (numberTwo = 0)
```

```
        return numberOne;
    return greatestCommonDivisor(numberTwo
```

```
    public static void main(String args[])
    {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter the Inputs");
        long numberOne = scanner.nextInt();
        long numberTwo = scanner.nextInt();
        System.out.println(lcmNaive(numberOne, numberTwo));
    }
}
```

And here's the pseudo code for the same

This program calculates the lowest common multiple for excessively long input values

```
function lcmNaive(ArgumentOne, ArgumentTwo) {
    Calculate the lowest common variable of Argument
    1 and Argument 2 by dividing their product
    by their Greatest Common divisor product.
```

Performance Analysis:

To measure the preformance of the algorithm we have two factors those time and space. Complexity.

1. time Complexity :

The amount of time is needed to run an algorithm in Computer is called time Complexity.

2. space Complexity :

The amount of memory is needed to store the algorithm in the system is called space Complexity.

⇒ for many algorithms we well calculate the time Complexity and not bother about the space Complexity.

⇒ In order to find the time Complexity of an algorithm we need to find the frequency Count of each instruction in the algorithm.

⇒ To excute each instruction in the program it requires 1 unit of time.

Time Complexity :

```
Algorithm, Sum (a,n)                    ------> 0
{                                       ------> 0
 int s = 0;                             ------> 1
 for i = 1 to n do                      ------> n+1
 s = s+a[i];                            ------> n
 return s;                              ------> 1
}                                       ------> 0
```

2n+3 is the time complexity for the Algorithm.

Time Complexity :

Time Complexity is the sum of the all frequency Count of each executable instructions is called time Complexity of the algorithm.

time Complexity $O(n)$

$n^2 + 2n + 1 \longrightarrow O(n^2)$

$n + 300 \longrightarrow O(n)$

Example 2 : Algorithm for matrix addition.

```
Algorithm matrix ( )                    ------ 0
{                                       ------ 0
for i=1 to n do                         ------ n+1
    for j=1 to m do                     ------ n(m+1)
    S[i][j] = a[i][j] + b[i][j];        ------ n.m
}                                       ------ 0
```

$nm + nm + n + n + 1 = 2mn + 2n + 1$

Time Complexity $O(mn)$

Example 3 : Algorithm for even or odd.

		true	false
Algorithm evenlodd(n)	0	0	0
{	0	0	0
read n;	1	1	1
if (n%.2 == 0) then	1	1	1
write ("even");	1	1	0
else	0	0	0
write ("odd");	0	0	1
}	0	0	0

Constant time Complexity $O(1)$

fC = 3

100

Example 4 : Algorithm for fibonacci series.

	Step/Execution	Frequency Count true	Frequency Count false
Algorithm fib(n)	0	0	0
{	0	0	0
if (n<=1) then	1	1	1
write n;	1	1	0
else	0	0	0
{	0	0	0
f1=0;	1	0	1
f2=1;	1	0	1
for i=2 to n do	1	0	n
{	0	0	0
f3=f1+f2;	1	0	n-1
f1=f2;	1	0	n-1
f2=f3;	1	0	n-1
write f3;	1	0	n-1
}	0	0	0
}	0	0	0
}	0	0	0
		2	5n-1

time Complexity in true case $O(1)$

time Complexity in false case $5n-1 = O(n)$

2. Space Complexity :

The amount of memory needed by the algorithm to store is said to be "space Complexity"

There are two types of space Complexity.

1. Constant space Complexity.
2. linear space Complexity.

Ex 1 : algorithm Sum (a,b,c)

```
{
    return a+b+c;
}
```

no variable parts

Space complexity of this algorithm is nothing but space occupied by 3 variables i.e. a, b, c.

integer takes 3 2 bytes.

a → 2 b → 2 c → 2

2+2+2 = 6 bytes.

→ int a, b, c → 6 bytes.

float a, b, c → 12 bytes.

→ In Constant space Complexity, space Complexity is independent of input size.

Ex2:

```
Algorithm Sum (a, n)
{
    s = 0;
    for i = 1 to n do
    s = s + a[i];
    return s;
}
```

→ here space Complexity is nothing but space occupied by variables a, n, s, i (4 variables)

→ Space needed to store a, n, s, i (variables)

↓

a[i]

→ if we take n → 10, we have a[10] → 26 memory locations n → 100, we have a[100] → 206

It is called linear Complexity, here space occupation is vary. with input size

here one dependent variable is there

i.e, a[i], on n

i. Constant space Complexity :

space requirment doesn't vary with the input variable input size.

ii. Linear Space Complexity :

. It vary with the variation of input size.

* $S(p) = C + S_p$

$S(p)$: space Requirement of the program.

C : Constant part (space requirement of Constant)

S_p : space requirement of the variable part.

* In our algorithm, Constant parts.

S, n, i

Variable part → a[i] (∵ it depends on n)

Asymptotic notations :

Asymptotic notations is used to represent the time Complexity of an algorithm

There are 5 types of Asymptotic notations.

1. Big oh notation (O)
2. big Omega notation (Ω)
3. theta notation (Θ)
4. little oh notation (o)
5. Little Omega notation. (ω)

There are 3 Cases of time Complexities

1. best Case time Complexity.
2. Average Case time Complexity.
3. Worst Case time Complexity.

Ex: Linear Search.

10 elements

2 5 6 4 3 1 7 8 9 10

5 --- 2 Comparisions ---- best case
1 --- 6 Comparisions ---- Average Case
10 --- 10 Comparisions --- worst Case

1. Big oh notation (O) :

Big oh notation is used to represent upper bound running time or Largest amount of time or max amount of time taken by the algorithm.

Formal definition :

Let $f(n)$ and $g(n)$ be two non negative functions. If there exists an integer $n0$ and a Constant c such that $f(n) <= c.g(n)$, $c>0$ and for all integers $n>=n0$ then $f(n) = O(g(n))$

Ex.1: $f(n)=3n+2$ $g(n)=n$, $f(n)=O(g(n))$?

$f(n) <= C.g(n)$

put $c=1$ and $n=1$ $f(n)=5$, $g(n)=1$

put $c=2$, $n=1$, $f(n)=5$, $C.g(n)=2$

put $c=3$, $n=1$, $f(n)=5$, $C.g(n)=3$

$C=4$, $n=1$ $f(n)=5$, $C.g(n)=4$

$C=4$, $n=2$ $f(n)=8$, $C.g(n)=8$

$C=4$ and $n>=2$ $f(n) <= C.g(n)$

$f(n) = O(g(n))$

Ex 2 : $f(n) = 100n+6$, $g(n) = n$, $f(n) = O(g(n))$?

$f(n) <= C \cdot g(n)$ where

$C = 101$, $n >= 6$

hence $f(n) = O(g(n))$

Ex 3 : $f(n) = 10n^2 + 4n + 2$, $g(n) = n$ $f(n) = O(g(n))$?

$f(n) <= C \cdot g(n)$ where.

2. Big omega notation (Ω) :

It is used to represent the minimum time taken by the algorithm.

(or)

It is used to represent the mintime or smallest amount or lower bound running time of an algorithm.

Formal defination :

Let $f(n)$ and $g(n)$ be the two non negative functions if there exist an integer n_0 and a Constant C and for all integers $n \geq n_0$ if $f(n) >= C \cdot g(n)$ then $f(n) = \Omega(g(n))$

Ex 1 : $f(n) = 5n+10$, $g(n) = 2n$ Show that $f(n) = \Omega(g(n))$

Sol : Here

$f(n) = C \cdot g(n)$ where $C = 1$, $n >= 1$

hence $f(n) \geq \Omega(g(n))$

Ex 2 : $f(n) = n+1)$, $g(n) = n^2$ Show that $f(n) = \Omega(g(n))$

Sol : In Ω notation $f(n)$ degree should be greater than or equal to $g(n)$. But In given Example $f(n)$ is less than $g(n)$. Hence $f(n)$ is not equal to Ω of $g(n)$

Ex 3 : $f(n) = 2n+5$, $g(n) = 2n$ Show that $f(n) = \Omega(g(n))$

Sol : Here $f(n) >= C \cdot g(n)$ where $C=1$ and $n>=1$

hence $f(n) = \Omega g(n)$.

3. theta Notation (Θ) :

It is used for representing the Average time taken by the algorithm (Average case time Complexity)

Formal definition :

Let $f(n)$ and $g(n)$ be two non negative functions Then there exist an Integer n and Constants C_1, C_2 such that $C_1 > 0$ and $C_2 > 0$ for all integers $n >= n_0$

if $C_1 \cdot g(n) <= f(n) <= C_2 \cdot g(n)$ Then

$f(n) = \Theta(g(n))$

Ex 1 : $f(n) = 2n+8$, $g(n) = n$ Show that $f(n) = \Theta(g(n))$

Sol : $f(n) >= C_1 \cdot g(n)$

$C_1 = 1$, $n >= 1$

$f(n) <= C_2 \cdot g(n)$

$C_2 = 10$, $n >= 2$

We Can Say that $C_1 \cdot g(n) <= f(n) <= C_2 g(n)$

When $C_1 = 1$, $C_2 = 10$, $n >= 1$

Hence $f(n) = \Theta(g(n))$

Ex 2 : $f(n) = n^2 + n$, $g(n) = 5n^2$ Show that $f(n) = \theta(g(n))$

Sol : $f(n) >= C_1 \cdot g(n)$

$C_1 = \frac{1}{5}$, $n >= 1$

$f(n) <= C_2 \cdot g(n)$

$C_2 = 1$, $n >= 1$

We can say that $C_1 \cdot g(n) <= f(n) <= C_2 (g(n))$ where

$C_1 = \frac{1}{5}$, $C_2 = 1$, $n >= 1$

Hence $f(n) = \theta(g(n))$

4. little oh notation (o) : (for all cases)

Formal Definition :

Let $f(n)$ and $g(n)$ be two non-negative functions if $\lim_{n \to \infty} f(n)/g(n) = 0$ then $f(n) = o(g(n))$

Ex : $f(n) = n$. $g(n) = n^2$, check whether $f(n) = o(g(n))$

Sol : $\lim_{n \to \infty} f(n)/g(n) = 0$

$\lim_{n \to \infty} n/n^2 = \lim_{n \to \infty} \frac{1}{n} = 0$

Here $\lim_{n \to \infty} f(n)/g(n) = 0$

Hence $f(n) = o(g(n))$

5. Little omega notation (ω) :

Formal definition :

Let $f(n)$ and $g(n)$ be two non-negative function If $\lim_{n \to \infty} f(n)/g(n) = \infty$ or $\lim_{n \to \infty} g(n)/f(n) = 0$ then

$f(n) = \omega(g(n))$

Ex : $f(n) = 3^n$, $g(n) = 2^n$, $f(n) = \omega(g(n))$?

$\lim_{n \to \infty} f(n)/g(n) = \lim_{n \to \infty} (3/2)^n = \infty$

hence $f(n) = \omega(g(n))$

Note :

Any notation is used for representing any Case time Complexity

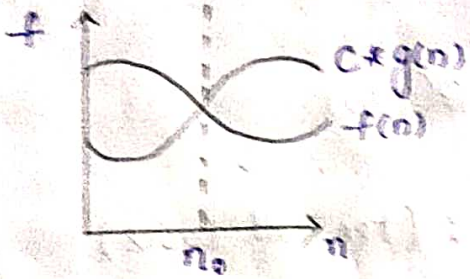
Diagrams for Asymptotic notations :

1. Big Oh notation :

$f(n) = 3n + 2 \quad g(n) = n$

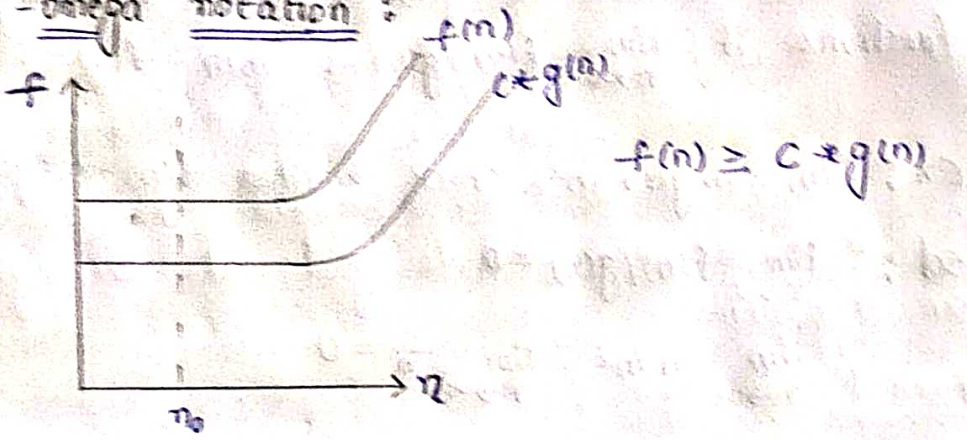
$n >= n_0$

$n >= 3$



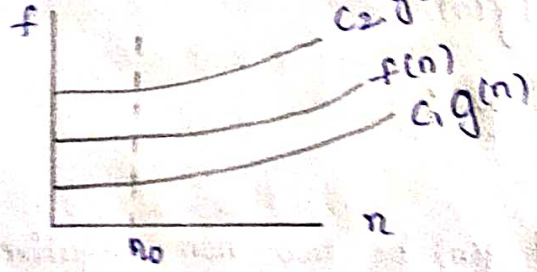
3

2. Big - omega notation :



$f(n) \geq c * g(n)$

3. Theta notation :



$c_1 g(n) \leq f(n) \leq c_2 g(n)$

Note :-

$O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) <$

$O(n^3) < \ldots < O(2^n) < O(3^n) < \ldots < O(n^n)$

$f(n) = n^2$

$<$

$>$

$=$

None

$g(n) = n(\log n)$

$f(n) > g(n)$ 1

$f(n) >= c \cdot g(n)$ 2

3

4.

Insertion Sort :

- Arrange the elements from Ascending (or) descending order is also called insertion sort.

Ex :

0	1	2	3	4	→ Index
30	10	40	20	50	→ Element

30 10 40 20 50

↓

temp = 10

Compare 10 with 30 10<30

place 30 in index 1 and place 10 in index 0

0	1	2	3	4	→ index
10	30	40	20	50	→ Elements.

temp = 40

40>30

temp = 20

20 < 40

10 30 40 40 50

20<30.

10 30 30 40 50

20 <10 false

0	1	2	3	4
10	20	30	40	50

Temp = 50

50 > 40

10 20 30 40 50

Sorted elements

Ex:

0	1	2	3	4	5	→ Index
6	3	8	5	1	10	→ Elements.

temp = 3

Compare 3 with 6 $3<6$

$3<6$

place 6 in index 1 and 3 in index 0

3 6 8 5 1 10

temp = 8

$8<6$ → false

3 6 8 5 1 10

Temp = 5

$5<8$

3 6 8 8 1 10

$5<6$

3 6 6 8 1 10

$5<3$ false

3 5 6 8 1 10

temp = 1

$1<8$

3 5 6 8 8 10

$1<6$

3 5 6 6 8 10

$1<5$

3 5 5 6 8 10

$1<3$

3 3 5 6 8 10

After sorting

1 3 5 6 8 10 → $10<8 \Rightarrow$ false

1 3 5 6 8 10 → Complete sorting.

n = no of elements

a ( ) → storing the elements in array

i → index no.

Algorithm :

```
for (i=1 ; i<n; i++)
{
    temp = a[i];
for (j=i  j>0 && 
    temp < a(j-1) ; j--)
    {
        a[j] = a[j-1];
    }
    a[j] =temp;
}
```

0	1	2	3	4
10	30	40	5	6

temp = 3

j=1

j-1 = 1-10

j = 0

Insertion Sort :

6 3 8 5 1 10 → elements.

Given ; 6 3 8 5 1 10

1st iteration :

i=1 ; i<6 → true

temp = a[1] = 3

j=i , 1>0 && 3<6 true

a[1] = 6.

j=0 , 0>0 && 3<a[-1] & false

a[0] = 3

∴ | 3 | 6 | 8 | 5 | 1 | 10 |

2nd iteration : i=2 ; i<6 → true

temp = a[2] = 8

j=2 ; 2>0 && 8<6 → false

a[2]=8

∴ | 3 | 6 | 8 | 5 | 1 | 10 |

3rd iteration :

i=3 , 3<6 → true

temp = 5

j=3; 3>0 && 5<8 → True

a[3] = 8

j = 2 ; 2>0 && 5<6 → True

a[2] = 6

j=1 ; 1>0 && 5<3 → false

a[1] = 5

∴ | 3 | 5 | 6 | 8 | 1 | 10 |

4th iteration :

i=4 , 4<6 → true

temp=1

j=4; 4>0 && 1<8 → True

a[4]=8

j=3; 3>0 && 1<6 → True

a[3] = 6

j=2 ; 2>0 && 1<5 → True

a[2] = 5

j=1 ; 1>0 && 1<3 → True

a[1] = 3

j=0; 0>0 && 1<a[0] → false

a[0] = 1

∴ | 1 | 3 | 5 | 6 | 8 | 10 |

5th iteration :

i=5 ; 5<6 → true

temp = 10

j=5 ; 5>0 && 10<8 → false

a[5] = 10

∴ | 1 | 3 | 5 | 6 | 8 | 10 |

6th iteration :

i = 6 , 6 ≮ 6 → false

After Sorting

∴ | 1 | 3 | 5 | 6 | 8 | 10 |

Algorithm and time Complexity :

```
for (i=1, i<n, i++)          → n times
                             → 0
{                            → (n-1) times
    temp = a[i];
    for (j=1, j>0 && temp < a[j-1]; j--)  → n(n+1)/2
    {
        a[j] = a[j-1];       → n(n+1)/2 - 1      n²
    }
    a[j] = temp;             → (n-1) time.
}                            total → n² + 4n - 3
```

i =	j =	no. of times
1	1	1
2	2	2
3	3	3
4	4	4
n	n	n

$1+2+3+4+n$

$= \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$

$= n^2$

Time Complexity $= O(n^2)$ (This is for average & Worst Case)

Best case :

Time Complexity $= O(n)$

* If the elements are in sorted, then we have Best Case time Complexity.

Worst / Average :

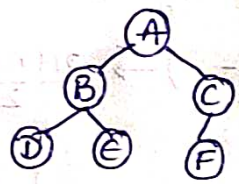
If elements are not in sorting order, then we have Worst / Average Case time Complexity

Heap sort :

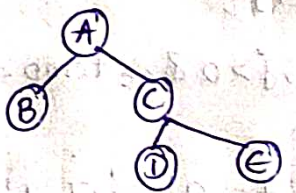
Heap is a Complete binary tree.

→ All the levels are Completely filled except last Level.

→ Root, Left, Right, A, B, C, D, E, F



Completely filled



Incomplete.

There are 2 types :

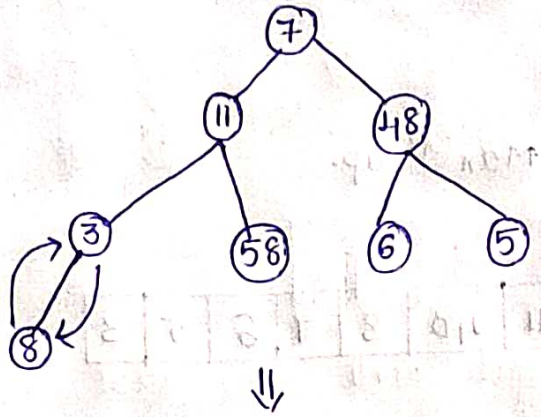
1. Max heap : Every parent node is greater than or equal to its child node.

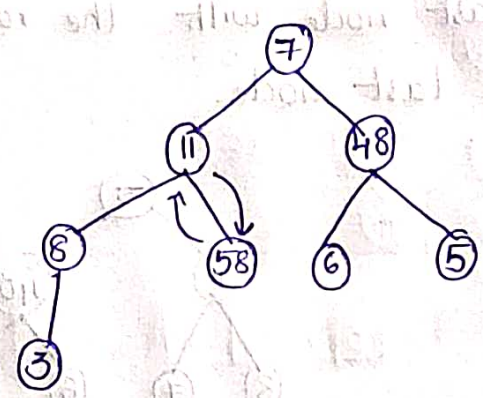
2. Min heap : Every parent node is Less than or equal to its child node.

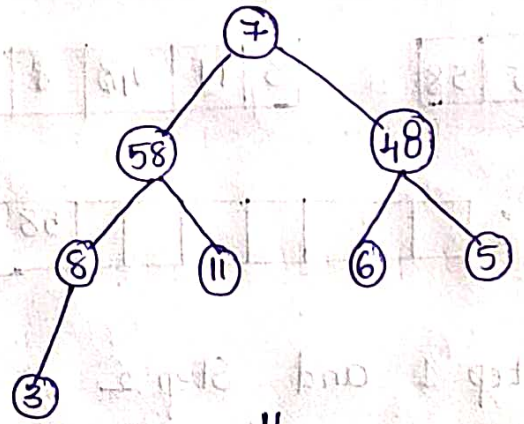
Example :

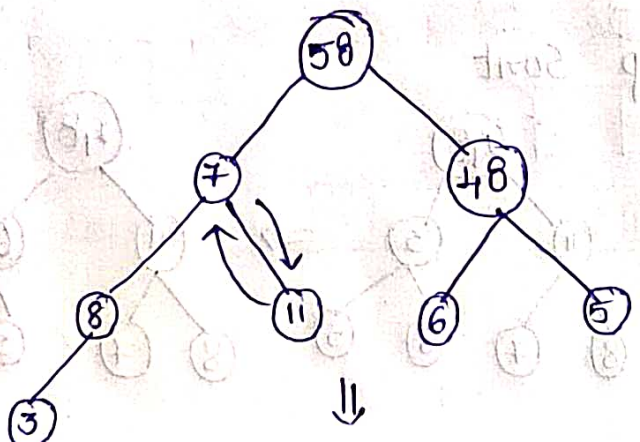
7 11 48 3 58 6 5 8

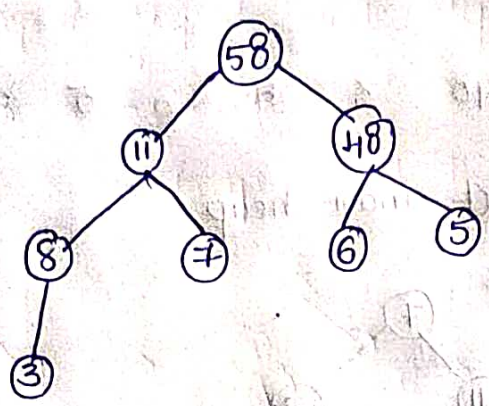
Step 1 : Construct max heap.







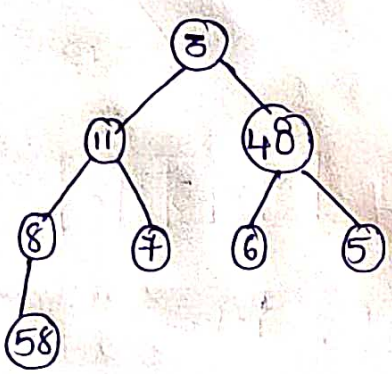


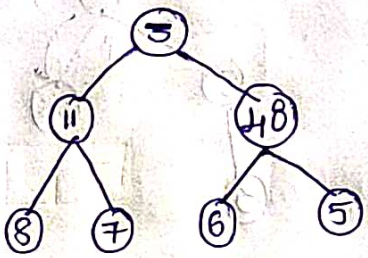


Max heap.

58	11	48	8	7	6	5	3

Step-2 : Replace last node with the root node and delete the last node.





3	11	48	8	7	6	5	58

3	11	48	8	7	6	5

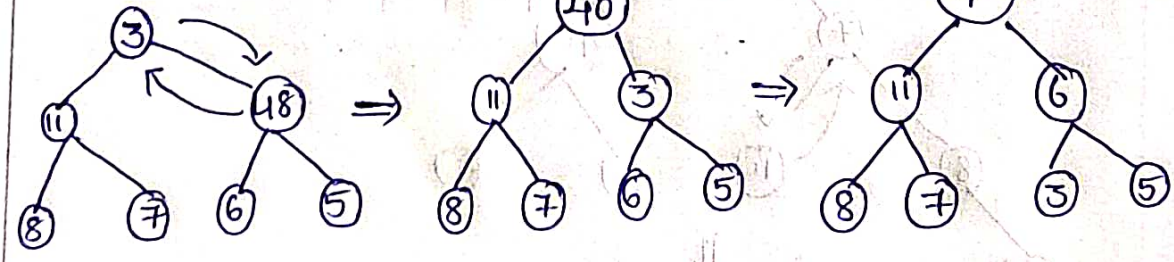
sorted :

							58

Step 3 : Repeat Step 1 and Step 2

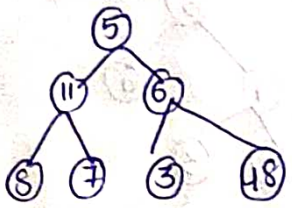
3, 11, 48, 8, 7, 6, 5.

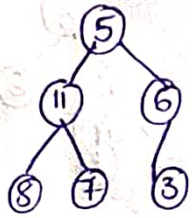
a. Construct heap sort



48	11	6	8	7	3	5

b. Replace root node with the last node and delete it





5	11	6	8	7	3	48

5	11	6	8	7	3

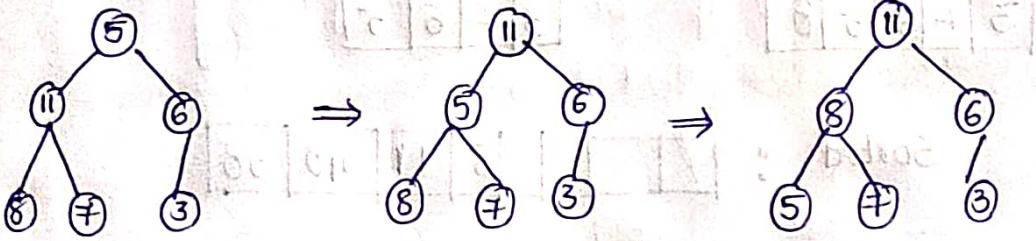
Sorted :

						48	58

Step 4 : Repeat Step 1 and Step 2 for below elements

5, 11, 6, 8, 7, 3.

a. Construct max heap.



11	8	6	5	7	3

b. Replace root node with last node and delete it.

3	8	6	5	7	11

3	8	6	5	7

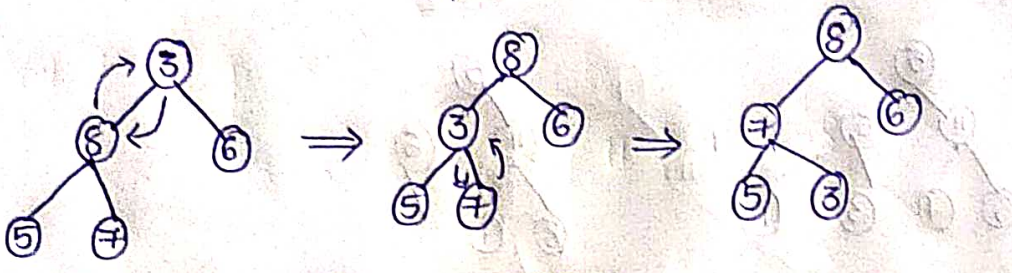
Sorted :

					11	48	58

Step 5 : Repeat step 1 and step 2 for below elements

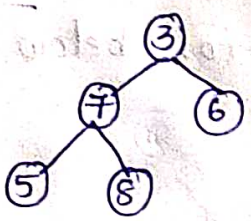
3, 8, 6, 5, 7

a. Construct max heap

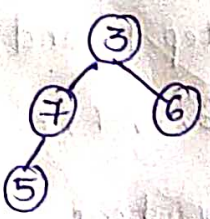


8	7	6	5	3

b. Replace root node with the last node and delete it



3	7	6	5	8



3	7	6	5

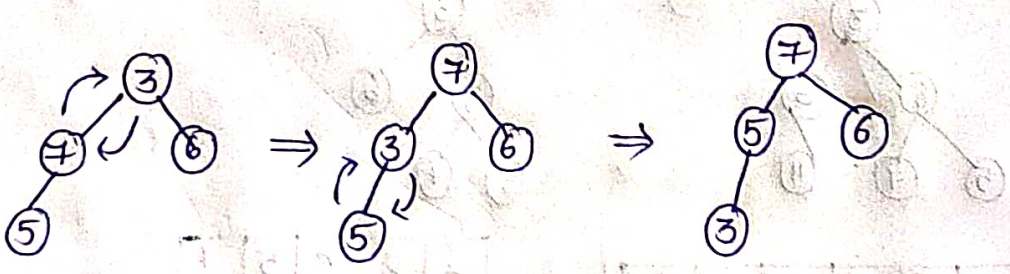
Sorted :

				8	11	48	58

Step 6 : Repeat Step 1 and step 2 for below elements

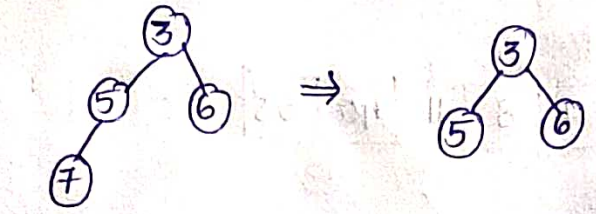
3, 7, 6, 5

a. Construct max heap.



7	5	6	3

b. Replace root node with last node and delete it.



3	5	6	7

3	5	6

Sorted :

			7	8	11	48	58

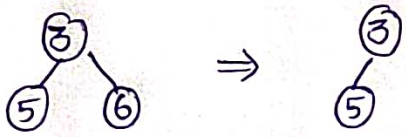
Step 7 : Repeat step 1 and step 2 for below

3,5,6

a. Construct max heap.

6	5	3

b. Replace root node with last node and delete it.



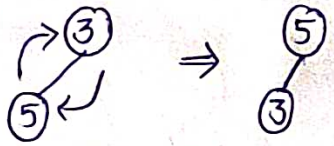
Sorted :

		6	7	8	11	48	58

Step 8 : Repeat step 1 and step 2 for below elements

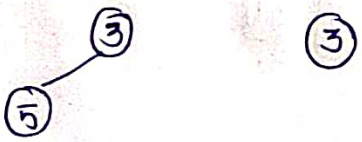
3,5

a. Construct max heap



5	3

b. Replace root node with last node and delete it.



3	5

3

Sorted :

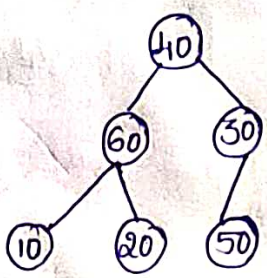
3	5	6	7	8	11	48	58

Sorted heap is

3	5	6	7	8	11	48	58

Max heap :

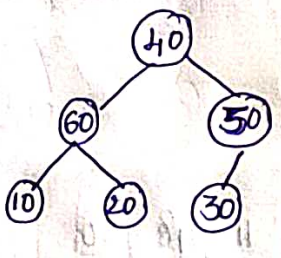
40, 60, 30, 10, 20, 50



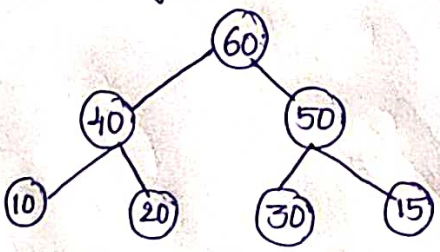
It is Complete Binary tree

Here max heap omit its property

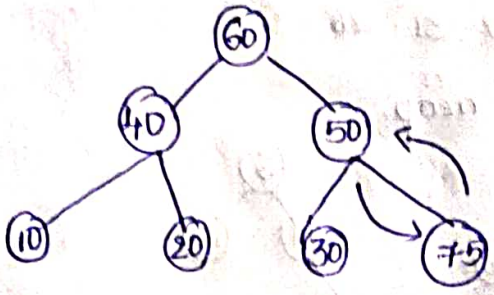
We convert into max heap.

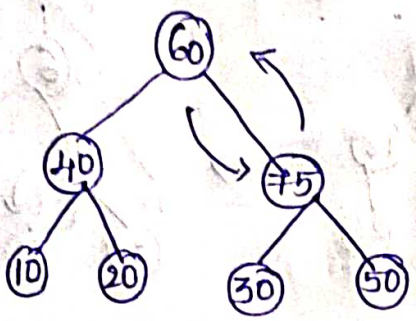


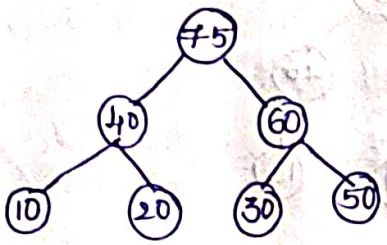
Insert the newnode is as last leaf from left to right



Insert the newnode '75' as leaf node from left to right







75	40	60	10	20	30	50
0	1	2	3	4	5	6

i → parent

2i+1 → left

2i+2 → right

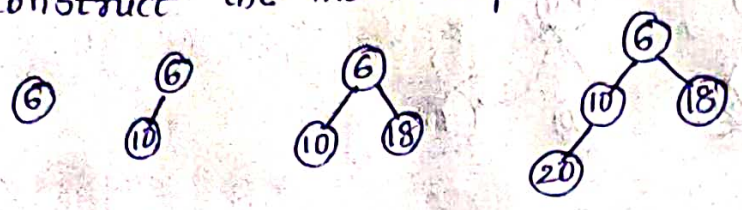
How sorting takes place:

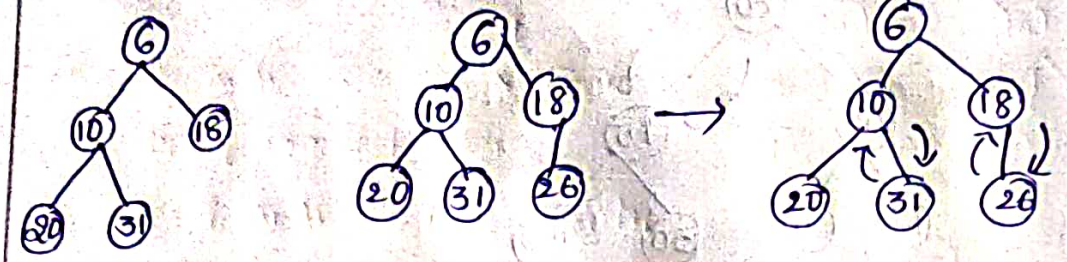
In this heap sort elements are given we sort the elements

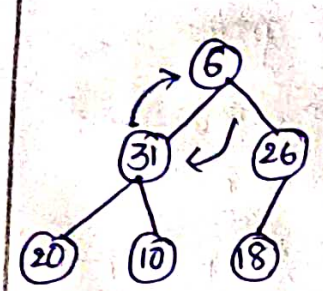
1. Max heap.
2. Delete the replace the root node with the last node.
3. deleted node will be placed at the last position.

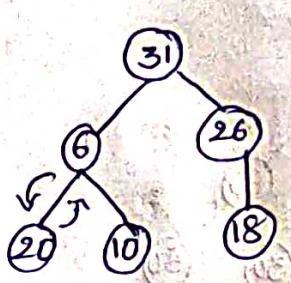
Ex: 6 10 18 20 31 26

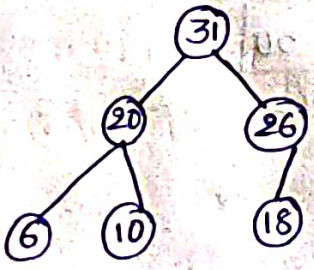
Construct the max heap.

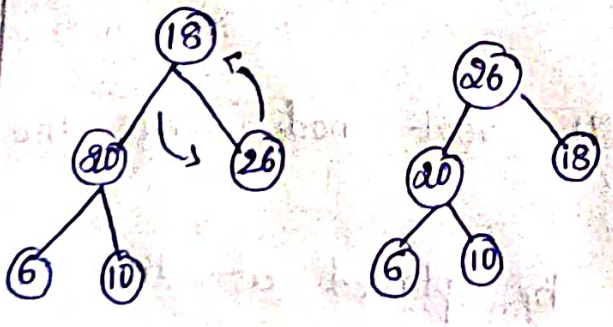












a

31	20	26	6	10	18

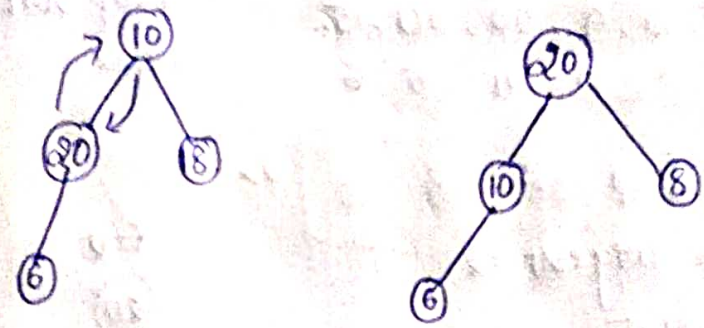
a

18	20	26	6	10	31

a =

26	20	18	6	10

10	20	18	6	26	31



a = | 20 | 10 | 18 | 6 | 26 | 31 |

a = | | 10 | 18 | 20 | 26 | 31 |

a = | 18 | 10 | 6 | 20 | 26 | 31 |

a = | 6 | 10 | 18 | 20 | 26 | 31 |

It is the list the elements are in sorting order.

6	10	18	20	26	31

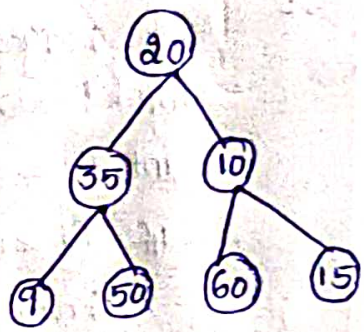
Algorithm for heap sort:

```
void heapify (int arr[], int n, int i)
{
    int Largest = i;
    int l = 2*i+1;
    int r = 2*i+2;
    if (l < n && arr[l] > arr[largest])
        Largest = l;
    if (r<n && arr[r] > arr[largest])
        Largest = r;
    if (largest != i) // If Largest is not root
    {
        Swap (arr[i], arr[Largest]);
        // Recursively heapify the affected sub-tree.
        heapify (arr, n, Largest);
    }
}
// main function to do heap sort.
void heapsort (int arr[], int n)
{
    for (int i = n/2 - 1; i >= 0; i--)
        heapify (arr, n, i);
    for (int i = n-1; i>0; i--)
    {
        Swap (arr[0], arr[i]);
        heapify (arr, i, 0);
    }
}
```

Ex : 20, 35, 10, 9, 50, 60, 15

20	35	10	9	50	60	15
0	1	2	3	4	5	6

$i = \frac{7}{2} - 1 = \frac{7}{2} - 1 = 2$ $\boxed{i=2}$ $2 >= 0$



heapify (arr, 7, 2)

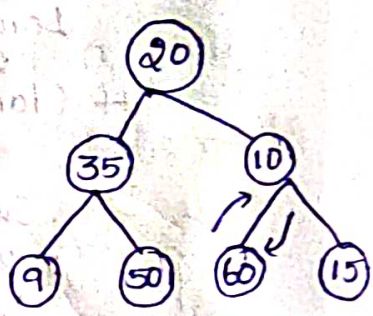
Largest = 2

l = 2 * 2 + 1 = 5

r = 6

(5 < 7) && (60 > 10) ⟶ True

Largest = 5



(6 < 7) && (15 > 60) ⟶ false

(5 != 2)

swap (10, 60)

20	35	60	9	50	10	15
0	1	2	3	4	5	6

heapify (arr, 7, 5) [recursive function]

formal parameter i = 5

Largest = 5

l = 11

r = 12

(11 < 7) → false

(12 < 7) → false now go, for loop i decrement

actual parameter i = 2 decreased by 1

i = 1

1 >= 0 (True)

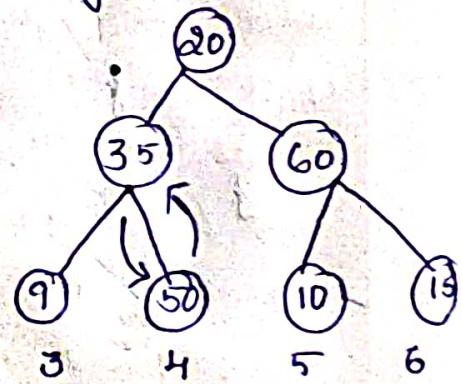
l = 3

r = 4

(3 < 7) && (9 > 35) false

(4 < 7) && (50 > 35) True

Largest = 4



(n1 = 1) → true

swap (55, 50)

| 90 | 50 | 60 | 7 | 55 | 10 | 15 |

heapify (arr, 1, 11) [recursive function]

- formal parameter i = 1

largest = 1

l = 2, r = 10

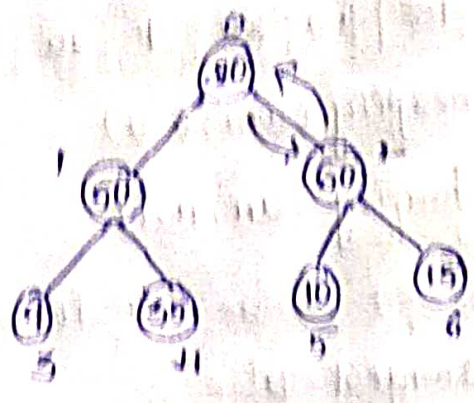
(2 < 11) → false

(10 < 11) → false

go to for statement

actual parameter i = 1, i = 0

0 >= 0



0 = 1

r = 2

(1 < 11) && (50 > 20) true.

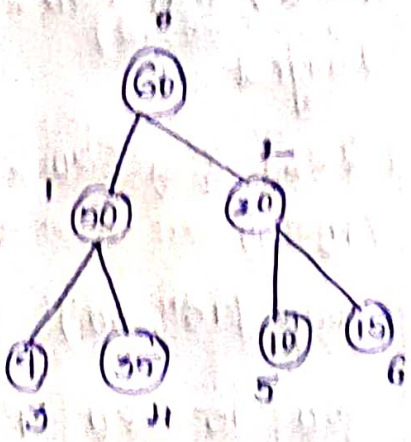
largest = 1

(2 < 11) && (60 > 50)

largest = 2

(2 != 0)

swap (50, 60)



| 60 | 50 | 20 | 7 | 55 | 10 | 15 |

heapify (arr, 1, 2) [recursive function]

largest = 2

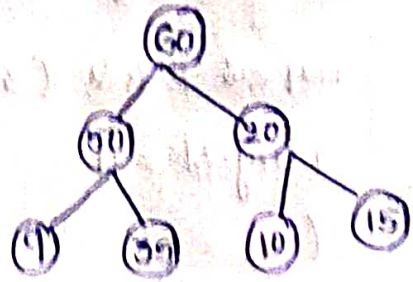
l = 5

r = 6

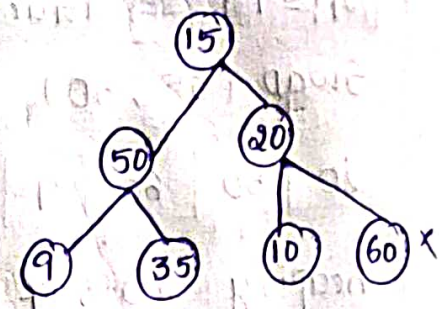
(5 < 11 && 10 > 20) → false

(6 < 11 && 15 > 20) → false

go to for loop



i = i-1 = 0-1 = -1 >= 0 → false



go to next for loop

i = 7-1 = 6

6 > 0

Swap (60, 15)

15	50	20	9	35	10	60

heapify (arr, 6, 0) (Calling function)

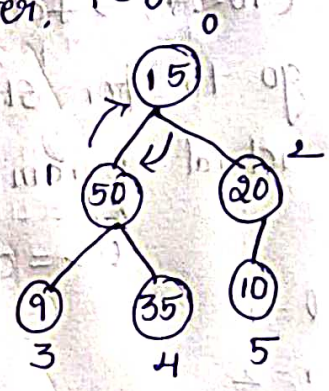
heapify (arr, n, i) (Called function)

∴ formal parameter, i = 0

Largest = 0

l = 1

r = 2



(1 < 6) && (50 > 15) → True

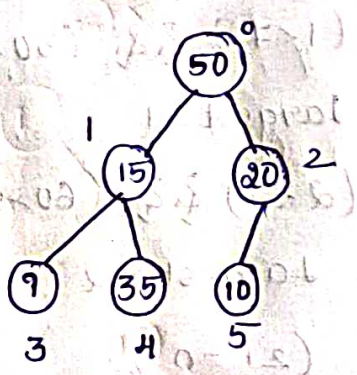
Largest = 1

(2 < 6) && (20 > 50) → false

(1 != 0)

Swap (15, 50)

50	15	20	9	35	10	60



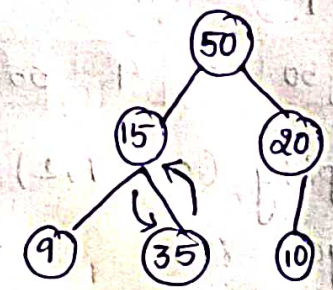
heapify (arr, 6, 1) (recursive function)

Now i = 1

Largest = 1 ;

l = 3

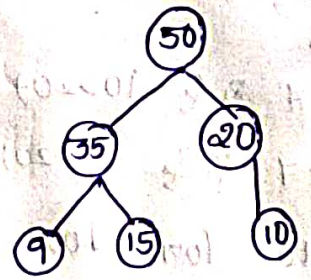
r = 4



(3 < 6) && (9 > 15) false

(4 < 6) && (35 > 15) True

Largest = 4



(4 != 1)

Swap (15, 35)

50	35	20	9	15	10	60

heapify (arr, 6, 4)

i = 4 (formal parameter)

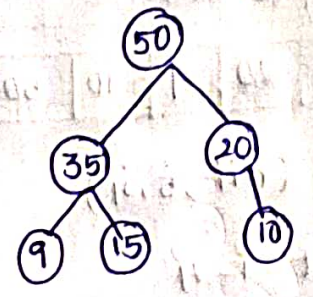
Largest = 4

l = 9

r = 10

(9<6) false

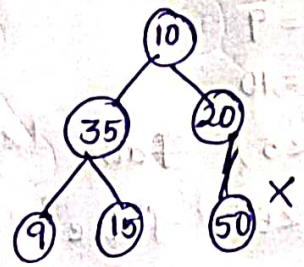
(10<6) false



go to for loop (2nd for Loop

i- - = 6-1 = 5

Swap (50, 10)

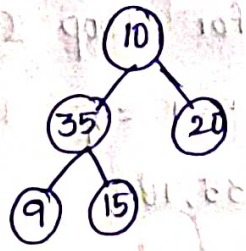


10	35	20	9	15	50	60

heapify (arr, 5, 0) (Calling function)

heapify (arr, n, i) (called function)

n=5, i=0 (formal parameter)



Largest = i = 0

l = 1

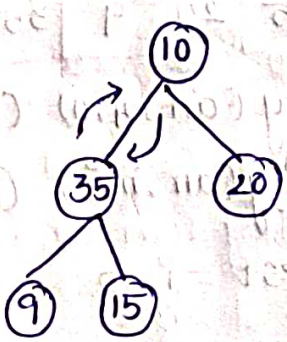
r = 2

(1<5) && (35>10) → True

Largest = 1

(2<5) && (20>35) → false

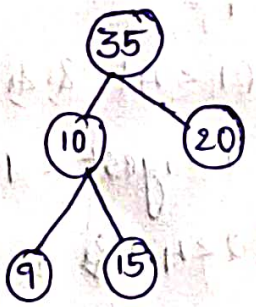
(21 != 0)



Swap (10, 35)

35	10	20	9	15	50	60

heapify (arr, 5, 1) (recursive function)



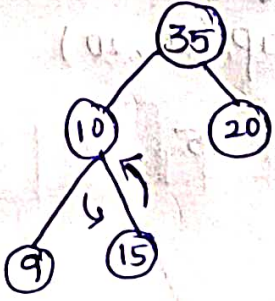
Largest = i = 1

l = 3

r = 4

(3<5) && (9>10) → false

(4<5) && (15>10) → True.

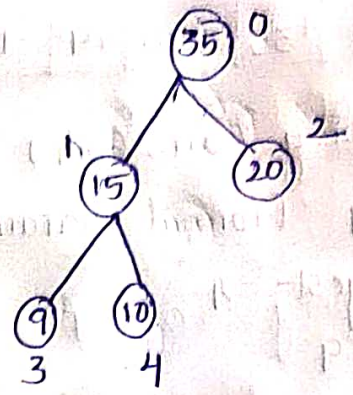


Largest = 4

(4 != 1)

swap (10, 15)

35	15	20	9	10	50	60



heapify (arr, 5, 4) (recursive function)

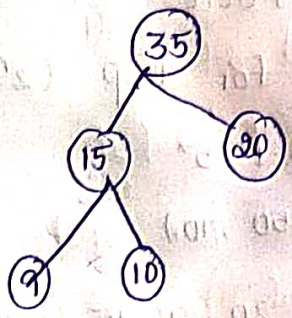
Largest = i = 4

l = 9

r = 10

(9 < 5) false

(10 < 5) false

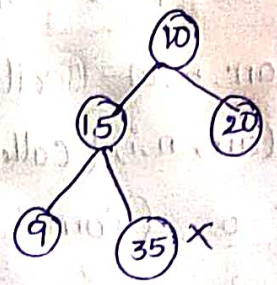


go to for loop (2nd loop)

i-- = 5 - 1 = 4

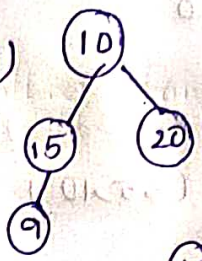
Swap (35, 10)

10	15	20	9	35	50	60



heapify (arr, 4, 0) (Calling - function)

heapify (arr, n, i) (Called - function)

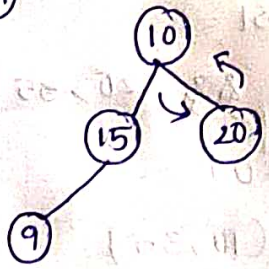


Largest = i = 0

l = 1

r = 2

(1 < 4) && (15 > 10) → True



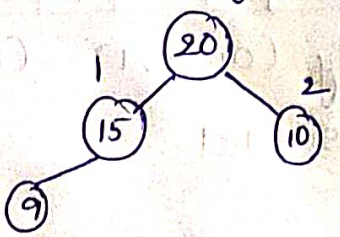
Largest = 1

(2 < 4) && (20 > 15) True

Largest = 2

Swap (10, 20)

20	15	10	9	35	50	60



heapify (arr, 4, 2) (recursive function)

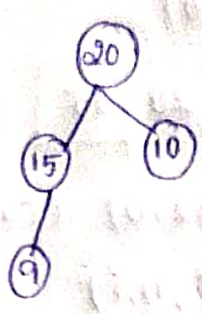
Largest = 2

l = 5

r = 6

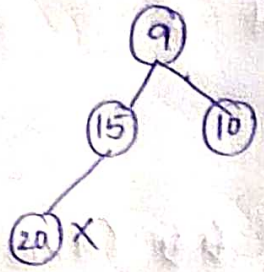
(5 < 4) false

(6 < 4) false



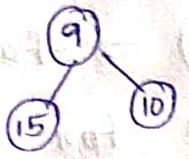
go to for loop (2nd for Loop)

i-- = 4-1 = 3 > 0



Swap (20, 9)

9	15	10	9	35	50	60



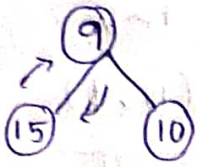
heapify (arr, 3, 0) (Calling function)

heapify (arr, n, i) (Called function)

Largest = i = 0

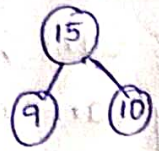
l = 1

r = 2



(1 < 3) && (15 > 9) True

Largest = 1



(2 < 3) && (10 > 15) false

(i = 0)

Swap (9, 15)

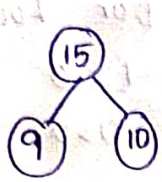
15	9	10	20	35	50	60

heapify (arr, 3, 1) (recursive function)

Largest = 1

l = 3

r = 4

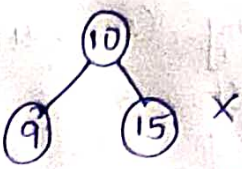


(3 < 3) false

(4 < 3) False

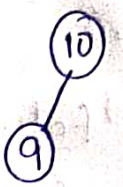
go to A for loop (2nd Loop)

i-- = 3-1 = 2



Swap (15, 10)

10	9	15	20	35	50	60



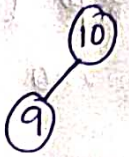
heapify (arr, 2, 0) (calling function)

heapify (arr, n, i)

largest = 0

l = 1

r = 2



(1 < 2) && (9 > 10) False

(2 < 2) → false

go to for loop

i-- = 2 - 1 = 1

1 > 0

Swap (10, 9)

9	10	15	20	35	50	60

heapify (arr, 1, 0) (calling function)

heapify (arr, n, i)

largest = 0

l = 1

r = 2

(1 < 1) → false

(2 < 1) → false

go to for loop

i = 1 - 1 = 0

0 > 0 → false

So, list is

9	10	15	20	35	50	60

Time Complexity for heap Sort :

```
void heapSort (int arr[], int n)
{
    for (int i = n/2 - 1 ; i >= 0 ; i--) ------ n/2
        heapify (arr, n, i) ; ---------- n/2 (log n)
    for (int i = n-1 ; i > 0 ; i--) ------- n
    {
        swap (arr[0], arr[i]) ; ---------- n-1
        heapify (arr, i, 0) ; ---------- n (log2 n)
    }
}
```

Time Complexity :

$O(n \log_2 n)$ for Best, Average and Worst Cases.