3/12/21 Friday

Design and Analysis of Algorithms (DAA)

UNIT - 1

Algorithm: It is a step by step Procedure to Solve a particular task or problem.

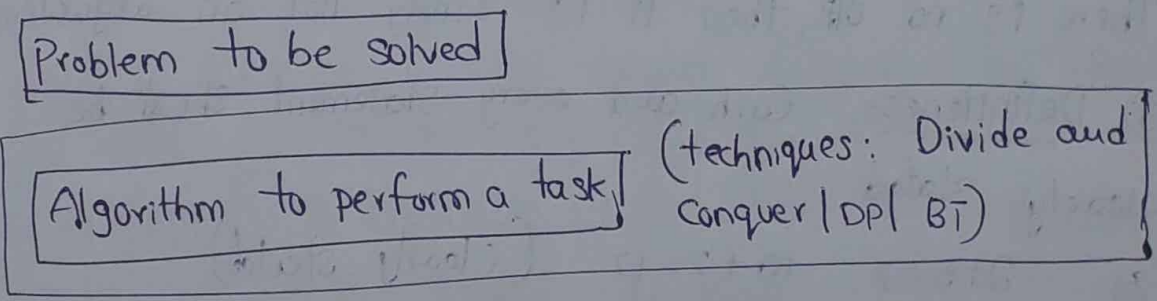
(or)

It is a finite number of steps which can be used to solve a particular task or problem in step by step Procedure. It must be terminated in a finite number of steps.

→ Algorithm term was first introduced by persian Author

→ Algorithm → Abu Jafar Mohammed Ibn Musaal Khwarizmi in 780 DC

Notation of an algorithm



Properties of an algorithm:

① Input ② output ③ Definiteness ④ Effectiveness ⑤ finiteness

1. **Input:** Algorithm may take zero or more number of inputs.

Eg: **Zero input:**

```
#include <stdio.h>
Void main (){
  Printf("WELCOME");}
```

o/p :- WELCOME.

Eg:- 1 (or) more input :-

```
#include <stdio.h>
Void main(){
Int a,b,c;
Printf("Enter a,b values:");
Scanf("%d%d",&a&b);
c=a+b;
Printf("c=%d",c);
}
```

o/p :- Enter a, b values: 2 3
c=5

② Output :- An algorithm should produce at least one output. Without O/p there would be no algorithm. If there is no O/p, then it is simply not an algorithm.

③ Definiteness: Each and every statement should be clearly stated.

Eg:- 2+5=7, 10+2=12 (clearly stated)

1/0 undefined - wrong

④ Effectiveness: Each and every statement should be effective. An algorithm doesn't contain unnecessary statements.

Eg:-
```
#include<stdio.h>
  Void main(){
    Int a,b,c;  //necessary Statement
    Printf("Enter a,b values:"); // may or may not
  Scanf("%d%d",&a,&b); necessary
```

```
c=a+b; // necessary statement
scanf("%d",&c); //unnecessary statement.
printf("c=%d",c); // necessary statement
}
```

⑤ Finiteness:- An algorithm must be terminated at any particular point. If there is no termination point, simply it is not an algorithm. It must be terminated in a finite (or) countable no. of steps.

eg:- for(int i=0;i<=5;i++) Printf("%d",i);

o/p: 0 1 2 3 4 5

4/12/21
~~Mon~~ Sat

Pseudo code for expressing algorithm:-

→ Pseudo code is an artificial and informal language used to develop an algorithm.:

→ Pseudo code is a text based language that can be used to develop an algorithm.

→ Algorithm is basically a sequence of instructions written in simple English language.

There are two ways to implement an algorithm.

→) flowchart - graphically represented by an algorithm.

→) pseudo code - its is a simple text based representation of an algorithm by using some programming constructs.

Writing an algorithm using Pseudo code

1) Algorithm contain heading and body.

algorithm:- algorithm name (Parameter1, Parameter2)

Syntax.

2) The beginning and ending of block should be indicated

ending "}" → ending ...

3. Every statement is ending with (;) delimeter.
4. Single line comments are written as // beginning of comment.
5. Identifiers are starting with letters but not with digits
6. If your assigning a value to the variable we are using an assignment operator (:=)
   ex: variable := value; ex:- a := 10;
7) There are some operators are used to implement an algorithm. <, <=, >, >=.
   Ex: a<b, a>=b, a<=b, c>b.
8) The conditional statement such as if-then or if-then-else are written as

```
if (condition)
begin:

ending;
then else (condition)
begin:

ending;
```

9) While statement can be written as

```
While (condition)
begin:
Statement 1
Statement 2
:
Statement n
Ending:
```

10) For loop can be written as using pseudo code

```
For (int i=0; i<=10; i++)   ← Program way
{
Statement;
}
```

Algorithm way:- fd variable := value1 to value2 step do

```
Begin
  Statement1
  Statement 2
Ending.
```

Algorithm:- //

```
Switch (value)
begin:
case1: st; break;
case 2: st; break;
ending:
```

11) Switch statement can be written as

Program:-

```
Switch statement (expression){
Case 1: Statement; break;
case 2: Statement; break;
Case 3: statement; break;
}
```

12. Input and o/p statements can be written as
Scanf → read
printf → write.

13. functions can be written as

Algorithm: datatype functionname(parameters)
Begin:
Statement 1;
Statement 2;
ending:

Implementing addition of two numbers algorithm using pseudo code

Eg:1) Addition of two numbers:- Pseudo code:-

Algorithm addition.(a,b,c)
begin
write("enter a,b values:");
read("read a,b values?");
c:a+b;
write("display c value");
ending.

Program;-

```
#include<stdio.h>
void main(){
int a,b,c;
printf("enter a, b values:");
Scanf("%d%d",&a,&b);
C=a+b;
Printf("c=%d",c);
}
```

2) Implementing an algorithm to check wheather the given number is even or odd using pseudo code.

Algorithm even or odd (a)
begin
write("Enter a value");
Read("read a value")
If (condition)
begin
Write("Given number is even");
ending
Then else
begin
write("Given number is odd");
ending
ending

Program:-

```
#include <stdio.h>
Void main(){
int a;
Printf("Enter a value:");
Scanf("%d",&a);
If (a%2==0)
Printf("Given number is
even \n");
else Printf("Given number
is odd");
}
```

Approaches to the time complexity

1. Priori analysis
2. Posterior analysis

1. Priori analysis:- Before executing the algorithm we can calculate the performance of an algorithm. It is not giving accurate results.

2. Posterior analysis:- After executing the algorithm we can calculating the Performance and execution time of an algorithm. It is giving accurate results.

⇒Best case:
⇒Worst case:
⇒Average case:

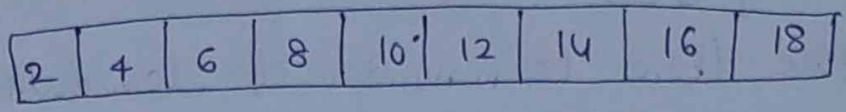
Best case:- Which Problem will takes minimum numbel of steps to complete the task or problem that is called best case. Which algorithm will takes minimum amount of time to complete the given task of Problem that is called best algorithm.

Average case:- Which Problem will takes average number of steps to complete the task or problem that is called average case. Which algorithm will takes avg amount of time to complete the task that is called as avg algorithm.

Worst case:- Which Problem will takes maximum number of steps to complete the task or problem that is called worst case. Which algorithm will takes maximum amount of time to complete the

task of Problem, that is called worst algorithm.

Ex: linear search (sequential search)



Problem: Searching the element 2 in an array

Sol: Searching starting from index zero (best case)

A[0] ----> 2 our searching element is 2 so both are matching.

Time complexity O(1)

Problem: Searching the element 10 in an array.

Sol: (avg case)

A[0] ----> 2 not matching (2 # 10)

A[1] ----> 4 not matching (4 # 10)

A[2] ----> 6 not matching (6 # 10)

A[3] ----> 8 not matching (8 # 10)

A[4] ----> 10 both are matching (10 = 10)

Note: If there is number of elements are there then the avg time complexity is O(n/2)

the time complexity O(n/2)

Problem: Searching the element 18 in array.

Sol: A[0] ----> 2 not matching (2 # 18)

A[1] ----> 4 not matching (4 # 18)

A[2] ----> 6 not matching (6 # 18)

A[3] ----> 8 not matching (8 # 18)

A[4] ----> 10 " " (10 # 18)

A[5] ----> 12 " " (12 # 18)

A[6] ----> 14 " " (14 # 18)

A[7] ----> 16 " " (16 # 18)

A(8) ----> 18 both are matching (18 = 18)

} (worst case)

Means elements is 18 is found at index 4.

Time Complexity O(n)

Note: Suppose there is n number of elements are there then the time complexity is O(n).

Performance of an algorithm:

The efficiency of an algorithm can be decided by measuring the performance of an algorithm.

We can measure the performance of an algorithm by using two factors.

1. Time complexity. 2. Space Complexity.

1. Space Complexity:- Space complexity means Space to store only data values but not the space to store the algorithm itself. Space complexity can be defined as amount of memory taken by an

→ Space Complexity can be calculated by the equation

$S(p) = c + sp$

or

→ Where c is constant vari fixed variables.

→ Sp means instance variables used in the algorithm.

Ex: c=a+b algorithm

```
Algorithm add(a,b,c)
Begin
  Write ("read a,b values");
  Read ("a,b values");
  c = add a,b values;
  Write ("display c value");
Ending
```

Space complexity $S(P) = C + sp$

a --- occupy 1 memory
b --- occupy 1 memory
c --- occupy 1 memory

there is no instance variables in an algorithm sp=0

Space complexity $S(P) = C + sp$

$= 3 + 0$

$S(P) = 3.$

Ex: What is the space complexity of an algorithm

```
Algorithm display()
Begin
Write ("Welcome to ogukt-iiitongole");
Ending
```

Space Complexity $S(P) = C + SP$

There is no variables in an algorithm So C=0.

And instance variable SP=0

$S(P) = C + SP \longrightarrow 0 + 0 \longrightarrow 0 + 0 = 0$

Ex:
```
Begin
  for (int i=0 ; i<=n; i++)
    {
    Write ("Welcome");}
  ending
```

```
Algorithm display()
Begin
 for i:=0 to n do
   begin
   Write ("welcome");
    ending
Ending.
```

In this algorithm n is fixed variable it occu

memory so, C=1

i is instance variable (changing variable) so Sp occupies 1 memory SP=1

Now, Space complexity S(P) = C+SP
= 1+1

SP = 2

Ex: Finding the space complexity of an algorithm.

```
Algorithm add(x,n)                  1 Space for x
Begin                               1 space for Sum   } C=3
  Sum := 0.0;                       1 space for n
  for i := 1 to n do
  Begin                             1 Space for n
    Sum := Sum + x[i];              1 space for i
  Return Sum                        P, Sum = instance
  ending                            n = fixed.
Ending
```

Space complexity S(P) = C+SP
= 1+2 = 3

Time complexity:

Howmuch amount of time taken by the algorith to complete the task or problem is called time complexity. The time complexity can be measure in terms of frequency count. Frequency count means howmany number of times that the statement are to be executed.

Calculating time complexity of display algorithm:

S.No	algorithm	frequency count
1.	algorithm display ()	0
2.	begin	0
3.	write ("Welcome");	1
4.	end	0

Total frequency count = 0+0+1+0 = 1

Time complexity = O(1).

Ex: calculating the time complexity of an algorithm.

So no	Algorithm	frequency count
1.	Algorithm display (i,n)	0
2.	begin	0
3.	for i:=1 to n do	n+1
4	begin	0
5	write ("welcome");	n
6.	ending	0
7.	ending	0

Total frequency count = 0+0+n+1+0+n+0+0
= 2n+1

Time complexity = O(2n+1)

Note: When you are calculating time complexity we are eliminating constants.

Time complexity = O(2n)
= O(n)

Ex: Calculating the time complexity of an algorithm.

So. No	Algorithm	frequency count
1.	Algorithm add()	0
2	begin	0
3	for i=1 to n do	n+1
4.	Begin	0
5.	for j=1 to n do	n(n+1)
6.	C[i][j] = a[i][j] + b[i][j]	n^2
7.	ending	0
8	Ending	0

frequency count = 0+0+n+1+0+n^2+n+n^2+0+0
= $2n^2$+2n+1

Time complexity = O($2n^2$+2n+1)

Note:- When you are calculating the time Complexity first we should eliminate the constants. Next we can take highest degree of variable.

∴ Time complexity = $O(n^2+n)$

$= O(n^2)$ ⟶

16/12/21 Friday

Ex:1

```
Void main( )
{
int a,b,c;
Printf("enter a, b values");
Scanf("%d%d",&a,&b);
c=a+b;
Printf("c=",&c);
}
```

Frequency Count = $O(1000000)$ --- $O(1)$

Ex:2

```
Void main()
{
s=0;
For(i=0;i<n;i++)
{
s=s+a[i];
}
}
```

Frequency count is = 0+0+1+n+1+0+n+0+0

= 2n+2

Time complexity = $O(2n+2)$ → eliminating constant

$= O(n)$

13/12/21

Asymptotic Notations

To choose the best algorithm we need to check the efficiency of an algorithm. The efficiency of an algorithm is measured by the time complexity.

Using asymptotic notations we need to calculated time complexity of an algorithm. Based on the time complexity we will measure best, worst & average.

1. Big oh notation
2. Omega notation
3. Theta notation
4. Little oh notation
5. Little omega notation.

1. Big oh notation :-

It is denoted by "O". It is a method of representing upper (worst time) bound of an algorithm running time.

Using big oh notation we can give which algorithm will takes maximum amount of time taken algorithm to complete.

Definition:- Let $f(n)$ and $g(n)$ are two non-negative functions and there exists constants c and integer n, then $c>0$ and $n>n_0$

Such that $f(n) <= c * g(n)$ then

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

Problem: 1

Let $f(n)$ is $2n+2$ and $g(n)$ is n^2 then satisfy the big oh notation $(f(n)) <= c * g(n))$.

Sol: $F(n) = 2n+2$ and

$g(n) = n^2$

$F(n) <= c * g(n)$

$F(n)$ and $g(n)$ are substitute in condition

$2n+2 <= c * n^2$

$N=1$ and and $c=1$ then

$2(1)+2 <= 1*(1)^2$

$4 <= 01$ ---- false.

$N=2$

$2(2)+2 <= 1*(2)^2$

$6 <= 4$ --- false

$N=3$

$2(3)+2 <= 1*(3)^2$

$8 <= 9$ ---- true

$N=4$

$2(4)+2 <= 1*(4)^2$

$10 <= 16$ --- true

Where $c=1$ and $n: 3(n>2)$ the big oh notation is satisfied.

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

$= O(n^2)$

Problem: 2

Let $f(n) = 12n^{\wedge 2} + 6n$ and $g(n)$ is $O(n^3)$

Sol: Big oh notation condition is $f(n) <= c * g(n)$

$F(n) = 12n^2 + 6n$

$g(n) = n^3$

$12n^2 + 6n <= c * n^3$

$n=1$ and $c=4$ then

Substitute c, n values

$12(1)^2 + 6(1) <= 4 * (1)^3$

$12+6 \leq 4*1$

$18 \leq 4$ --- false

$n=2$ and $c=4$

$12(2)^2 + 6(2) <= 4 * (2)^3$

$12(4) + 12 <= 4 * 8$

$48 + 12 \leq 32$

$60 \leq 32$ --- false.

$n=3$ and $c=4$

$12(3)^2 + 6(3) <= 4 * (3)^3$

$12(9) + 18 \leq 4 * 27$

$108 + 18 \leq 108$

$126 \leq 108$ --- false.

$n=4$ & $c=4$

$12(4)^2 + 6(4) <= 4 * (4)^3$

$12(16) + 24 <= 4 * 64$.

$192 + 24 <= 256$

$216 <= 256$ --- True.

$n = 5$

$12(5)^2 + 6(5) <= 4 * (5)^3$

$12(25) + 30 <= 4 * 125$

$300 + 30 <= 500$

$330 <= 500$ --- True.

Where $c = 4$ and $n > 3$

i.e $(n = 4)$

The big oh notation is satisfied.

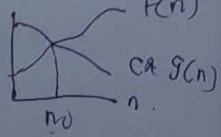
Such that $f(n) = O(g(n))$

$f(n) = O(n^3)$.

2. Omega notation:-

It is denoted by 'Ω'. It is the method of representing lower bound of an algorithm running time. Using Omega notation we denote which algorithm will takes less amount of time taken by the algorithm to complete.

Definition: Let $f(n)$ and $g(n)$ are two non-negative functions and there exists constant c and integer n such that $c > 0$ and $n > n_0$ then $f(n) >= c * g(n)$ F



Problem: Let $f(n) = 3n + 3$ and $g(n) = 2n + 5$ then satisfy the omega notation.

Sol: $F(n) >= c * g(n)$

$F(n) = 3n + 3$

$g(n) = 2n + 5$

Substitute $f(n)$ and $g(n)$

$3n + 3 >= c * (2n + 5)$

$n = 1$ and $c = 1$

$3(1) + 3 >= 1 * (2(1) + 5)$

$6 >= 1 * (7)$

$6 >= 7$ --- false.

$n = 2$

$3(2) + 3 >= 1 * (2(2) + 5)$

$6 + 3 >= 1 * (9)$

$9 >= 9$ --- true.

$n = 3$

$3(3) + 3 >= 1 * (2(3) + 5)$

$12 >= 1 * 11$

$12 >= 11$ --- true.

Where $c = 1$ and $n > 1$ condition is satisfied

Such that

Time Complexity is $= \Omega(g(n))$

$= \Omega(2n + 5)$

When we are calculating time complexity we are eleminating the constants so

Time Complexity $= \Omega(n)$.

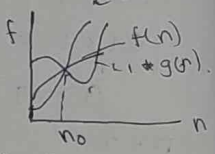
3 Theta Notation :-

It is denoted by "Θ". It is method of representing average bound of an algorithm running time. It can be used to denote which algorithm will take average amount time taken by the algorithm to complete.

Definition: Let f(n) and g(n), are two non-negative functions and there exists constants c_1, c_2 and integer n such that $c_1 < c_2$ and $n > n_0$ then

$c_1 * g(n) <= f(n) <= c_2 * g(n)$ then

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



Problem :- Let f(n) = 2n+8 and g(n) = On

Satisfy the Theta Notation.

Soln Theta notation is $c_1 * g(n) <= f(n) <= c^2 * g(n)$

F(n) = 2n+8 and g(n) = On

$c_1 = 2$, $c_2 = 7$ and n=1 then

Substitute c_1, c_2 and n values

$c_1 * n <= 2n+8 <= c_2 * n$

$2 * 1 <= 2(1)+8 <= 7 * 1$

$2 <= 10 <= 07$ ---- false

n = 2

$2 * 2 <= 2(2) + 8 <= 7 * 2$

$4 <= 4 + 8 <= 14$

$4 <= 12 <= 14$ --- True.

n = 3

$2 * 3 <= 2(3) + 8 <= 7 * 3$

$6 <= 14 <= 21$ --- True.

Where $c_1 = 2$ and $c_2 = 7$ and $n > 1$ Conditions satisfied the theta notation such that time complexity

$= \Theta(g(n))$

$= \Theta(n)$.

Problem :- Show that

$f(n) = 4n^2 - 64n + 288 = \Omega(n^2)$.

Soln $F(n) = \Omega(n^2)$

Omega notation $f(n) >= c * g(n)$

n=1 C=5.

LHS :- $f(n) = 4n^2 - 64n + 288$

$= 4(1)^2 - 64(1) + 288$

$= 4 - 64 + 288 = -60 + 288 = 228$.

RHS : $g(n) = n^2$

$= (1)^2$

$= 1$

LHS >= RHS

228 >= 1 --- True.

Where $n=1$ and $c=5$ satisfied the Omega notation Condition.

Such that $f(n) = \Omega(g(n))$

$f(n) = \Omega(n^2)$

4. Little oh notation:

Let $f(n)$ and $g(n)$ are two non-negative function then

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

Such that

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

5. Little Omega notation:-

Let $f(n)$ and $g(n)$ are two non-negative functions then

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

Such that

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

Select the element 32 from unsorted Sub array compare with all the elements in sorted sub array.

$32 > 31$

again compare 32 with 25

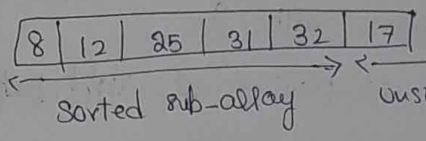
$32 > 25$

Again compare 32 with 12

$32 > 12$

again compare 32 with 8.

$32 > 8$

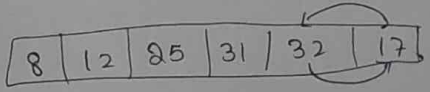


Compare 17 with all the elements in Sorted subarray.

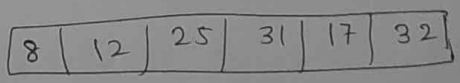
Now 17 compare 32

$17 < 32$

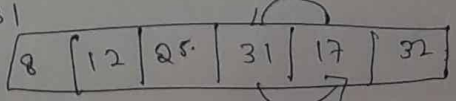
Swap 17 with 32



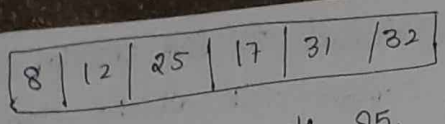
=)



Compare 17 with 31, $17 < 31$ Swap 17 with 31

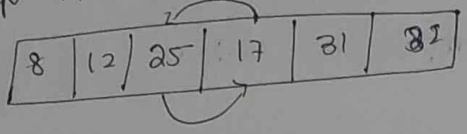


⇒

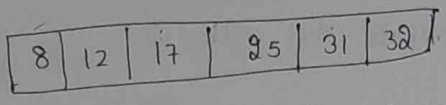


Now compare 17 with 25, 17 < 25;

Sweep 17 with 25



⇒

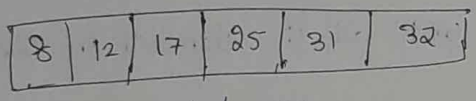


17 is compare with element 12

17 > 12 12 is placing at right position

Now 17 is compare with 8.

17 > 8 8 is placing at right position



← Sorted array → //

14/12/21

Analysis of insertion sort

Insertion sort works similar to the sorting of play cards. It is assumed that the first card is already sorted. Then we can select an unsorted card then compare with sorted card.

If it is greter than sorted card then we can place at right hand side. If it is less than sorted card then we can place at left hand side.

Algorithm:

1. assume the first element that is already sorted order.
2. pickup unsorted element and sorted it separately [compare with sorted sub-array elements]
3. If it is greater than sorted sub array element then placed at right hand side.
4. If it is less than sorted sub array element then placed at left hand side.
5. Repeate the steps 2 to 4 until all the elements in the is in sorted order.

Ex:1

12 31 25 8 32 17

--Sorted-- ←--- unsorted sub-array ---->

Initially we assume first element is in sorted order.

Pickup 31 from unsorted sub-array and compare with 12, now 31 is greater than 12, that

means 31 is placed at right hand side 12 is already in correct position.

12 31 25 8 32 17

<-sorted----> <----unsorted sub array-->

Now Pickup the next element from unsorted sub-array. Now compare with sorted sub-array elements.

Now 25 is less than 31 then swapping 25 with 31.

12 25 31 8 32 17

Next 25 is comparing with 12, now 25 is greater than 12. that means 12 is placing at correct position.

Now Sorted Order elements are

12 25 31 8 32 17

<--sorted---><--unsorted-->

Now select the element 8 from unsorted sub array. Then compare with all the elements in Sorted sub-array.

Now 8 is compare with 31, 8 is less than 31 that means 31 is not placing at right hand side position now we can swap 8 with 31

16/12/21

12 25 31 8 32 17

After swapping

12 25 8 31 32 17

Again element 8 is compare with 25. 8 is less than 25 (8 < 25). Means we can place at left hand side. 8 is swapping with 25.

12	25	8	31	32	17

After swapping elements are.

12	8	25	31	32	17

Step 6:- Now again 8 is compare with element 12. now 8 is less than 12 (8<12) now 8 is swapping with 12.

8	12	25	31	32	17

<- sorted sub array -> <- unsorted subarray ->

Con — Page 10

16/12/21

Algorithm:-

```
Algorithm insertion (i, n, j, a[])
{
For(i=1; i<n; i++)
{
Temp := a[i]
J = i-1
While(J>=0 && a[i]> temp)
{
A[J+1] = a[j]. ----- swapping the elements
J=j-1
}
A[j+1] = temp
}
}.
```

Ex: Sorting the elements in the array using insertion sort algorithm.

5	4	10	1	6	2

Sol: Step: Assume first element in the array is already in sorted order. Now we can select the next element from unsorted sub-array and compare with sorted sub-array.

For (i=1; i<n; i++)
For (i=1; 1<6; i++) ---- true.
{
Temp = a[i]
Temp = a[1]
Temp = 4
J = i-1
J = 1-1
J = 0
While (j>=0 && a[j] > temp)
While (0 >= 0 && a[0] > 4)
While (0 >= 0 && 5 > 4) ---- true.
{
A[j+1] = a[j]
A[0+1] = a[0]
A[1] = a[0] means a[1] is swapping with a[0]

5	4	10	1	6	2

After swapping

4	5	10	1	6	2

J = j-1
J = 0-1
J = -1
}
Again while loop is executed
While (j>=0 && a[j] > temp)
While (-1 >= 0 && a[-1] > 4) --- false.
Out of while loop is executed.
A[j+1] = temp
A[-1+1] = 4
A[0] = 4.

4	5	10	1	6	2

< sorted >< unsorted

Step-2: Now i value is incremented
i = i+1 = 1+1 = 2.
For (i=2; i<n; i++)
For (i=2; 2<6; i++) ---- true.
{
Temp = a[i]
Temp = a[2]
Temp = 10
J = i-1
J = 2-1
J = 1
While (j>=0 && a[j] > temp)
While (1 >= 0 && a[1] > 10)
While (1 >= 0 && 5 > 10) -- false.
Out of while loop is executed.
A[j+1] = temp
A[1+1] = 10
A[2] = 10 means element 10 is stored in array a[2]

4	5	10	1	6	2

< Sorted sub-array >< unsorted sub-array >

Step-3 Now i values is incremented

i = I + 1 = 2 + 1 = 3

for(i = 3; i<n; i++)

for(i = 3; 3<6; i++) ---- true

{

Temp = a[i]

Temp = a[3]

Temp = 1

J = i-1

J = 3-1

J = 2.

While (j >= 0 && a[j] > temp)

While (2 >= 0 && a[2] > 1)

While (2 >= 0 && 10 > 1) ---- true

{

A[j+1] = a[j]

A[2+1] = a[2]

A[3] = a[2] means a[2] is swapping with a[3]

4	5	10	1	6	2

After swapping

4	5	1	10	6	2

J = j - 1

J = 2 - 1

J = 1

}

Again while loop is executed until condition becomes false.

While (j >= 0 && a[j] > temp))

While (1 >= 0 && a[1] > temp)

While (1 >= 0 && 5 > 1) ---- true.

{

A[j+1] = a[j]

A[1+1] = a[1]

A[2] = a[1] means a[1] is swapping with A[2]

Now the array is

4	1	5	10	6	2

J = j - 1

J = 1 - 1

J = 0

}

again while loop is executed until condition is false.

While (j >= 0 && a[j] > temp)

While (0 >= 0 && a[0] > temp)

While (0 >= 0 && 4 > 1) ---- true.

{

A[j+1] = a[j]

A[0+1] = a[0]

A[i] = a[0] means a[0] is swapping with A[i]

Now array after swapping is

1	4	5	10	6	2

J = j - 1

J = 0 - 1

J = -1

}

again while loop is executed
while (-1>=0 && a[-1]> temp)---- false.
out of while loop is executed.

A[j+1] = temp
A[-1+1]= 1
A[0] = 1 means 1 is sorted at array of a[0].

Step4: Now i value is increment i=i+1 = 3+1=4

for(i=4 ; i<n; i++)
for(i=4; 4<6; 4++)--- true
{
temp = a[i]
temp = a[4]
Temp = 6
J= i-1
J= 4-1
J= 3
}
While (J >=0 && a[j] >temp)
While (3>= 0 && a[3] > 6)
While(3 >=0 && 10>6) --- true
{
A[j+1] = a[j]
A[3+1] = a[3]
A[4] = a[3] means a[3] is swapping with A[4]

Now array is

1	4	5	6	10	8

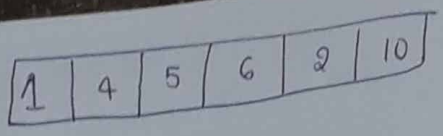
J = j-1
J= 3-1
J= 2
}
Again while loop is executed
While (j>=0 && a[j]>temp)
While(2>=0 && a[2]>6)
While (2>=0 && 5>6)--- false.
Outof while loop is executed.

A[j+1]= temp
A[2+1]=6
A[3]=6 means element 6 is placed at a[3]

Step:5 :- i value is increment i=i+1=4+1=5

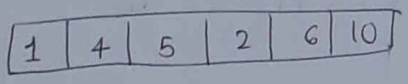
for(i=5; i<6; 5++)--- true
{
temp = a[i]
temp=a[5]
temp= 8
J=i-1
J=5-1
J=4
While (j>=0 && a[j]> temp)
While(4>=0 && a[4]>8)
While(4>=0 && 10>8)---true
{
A[j+1]=a[j]
A[4+1]=a[4]
A[5]=a[4] means a[4] is swapping with A[5]

Now array is

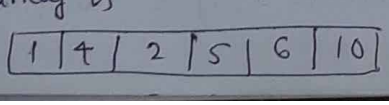


J = j-1
J = 4-1 = 3
}
Again while loop is executed
While (a >=0 && a[3] > a)
While (a >=0 && 5 > a) ---> true
{
A[j+1] = a[j]
A[3+1] = a[3]
A[4] = a[3] means a[3] is swapping with A[4]

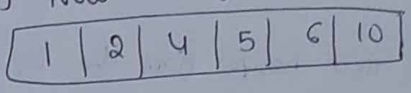
Now array is



J = j-1
J = 3-1 = 2
}
Again while loop is executed
While (a >=0 && a[2] > a)
While (a >=0 && 5 > a) ----> true
{
A[j+1] = a[j]
A[2+1] = a[2]
A[3] = a[2] means a[2] is swapping with A[3]
Now array is

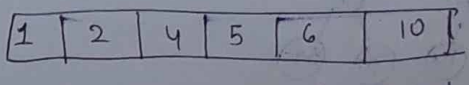


J = j-1
J = 2-1 = 1
}
Again while loop is executed
While (a >=0 && a[1] > a)
While (a >=0 && 4 > a) ---> true
{
A[j+1] = a[j]
A[1+1] = a[1]
A[2] = a[1] means a[1] is swapping with A[2] Now array is



J = j-1
J = 1-1 = 0
}
Again while loop is executed
While (a >=0 && a[0] > a)
While (a >=0 && 1 > a) ---> false
Outof while loop is executed
A[j+1] = temp
A[0+1] = a
A[1] = a means element a is placed at a[1].

step 6:- i value is incremented i = i+1 = 5+1 = 6
for (i=6; 6<6; i++) ---> false



Now all the elements are placing in sorted order (insertion sort)

17/12/21 Analysis of Heap sort:

Heap sort is a complete binary tree, a complete binary tree in which each node should contain at most two childrens. It follows left justified.

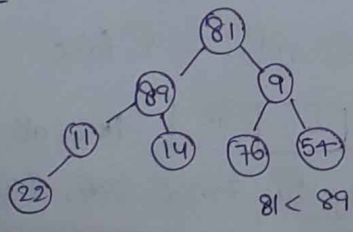
The concept of heap sort is eleminate the elements one by one from heap of the list and then insert them into the sorted array.

Algorithm:-

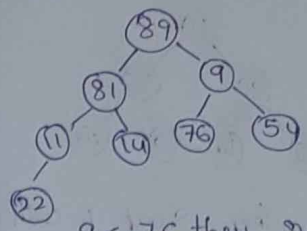
1. Construct the max heap from the input data. At this point the largest element is placed at the root of the heap.
2. Delete the root of the element from the heap. The deleted node is replaced with last node in the heap. And deleted element is placed at sorted array.
3. Repeat the step2 until all the elements are placing in sorted order.

Ex:- 81 89 9 11 14 76 54 22

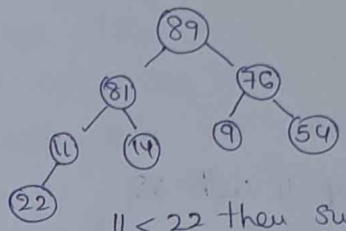
Step:-1 Construct Heap.



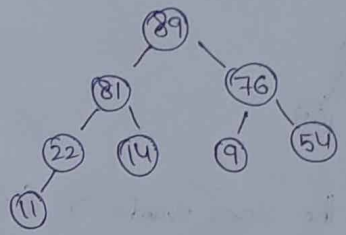
81 < 89 then 81 is swapping with 89.



9 < 76 then swapping 9 with 76.

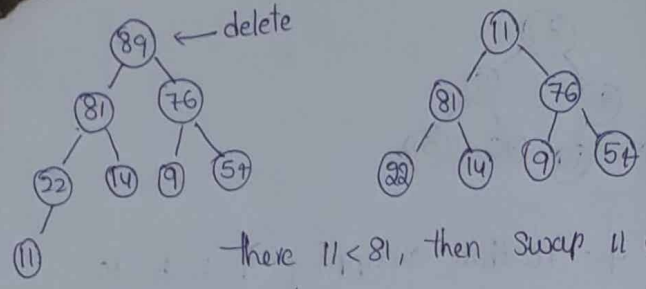


11 < 22 then swapping 22 with 11.

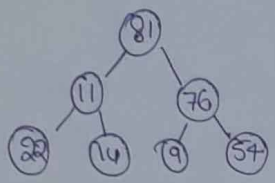


Then the array is 89 81 76 22 14 9 54 11.

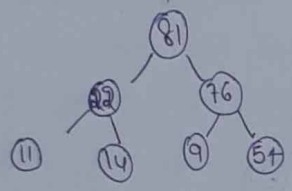
Step -2 Delete the root node from the max heap and placed at sorted array. Now the deleted node is replaced with the last node in the max heap.



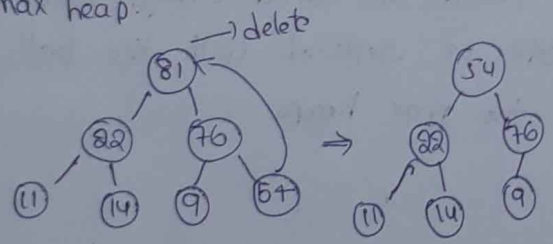
there $11 < 81$, then swap 11 with 81



$11 < 22$, then swap 11 with 22.

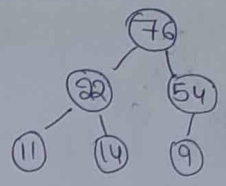


Step 3: Now delete the root node 81 from the max heap that is placed at sorted array and the deleted node is replaced with the last node in the max heap.



81	89

$54 < 76$ then swap 54 with 76.



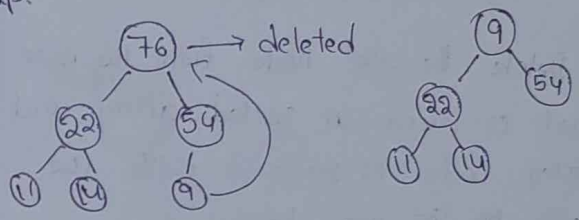
Max heap.

Then the array is 76 22 54 11 14 9.

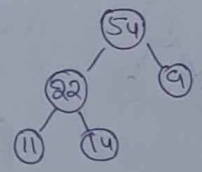
Step 4: The root node is deleted 76 is deleted from the max heap that is placed in sorted array and the deleted node is replaced with last node in the max heap.

76	81	89

Sorted array



$9 < 54$ then swapping 9 with 54.

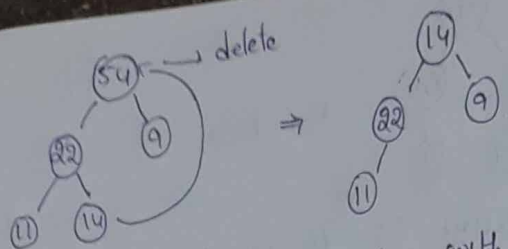


Then the array is 54 22 9 11 14.

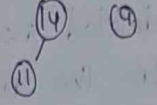
Step 5: Delete the root node from the max heap that is placed in sorted array and the delete node is replaced with the last node in the max heap.

54	76	81	89

Sorted heap.



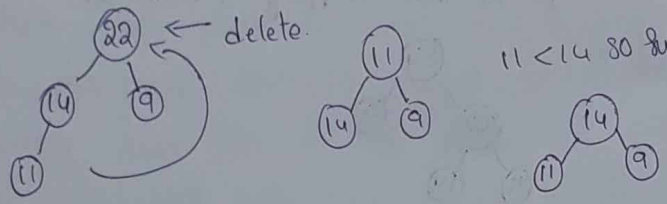
$14 < 22$ then 14 is swapping with 22



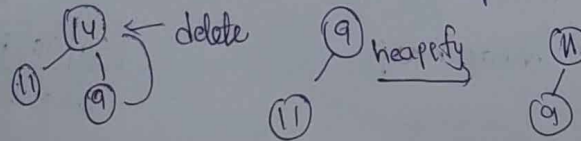
max heap.

Then the array is 22, 14, 9, 11.

Step:6 Delete the root node from the max heap that is placed in sorted array and the deleted node is replaced with the last node in the max heap.



Step-7:- Delete the root node from the max heap that is placed in sorted array and the deleted node is replaced with the last node in the max heap.

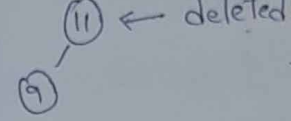


Then the sorted array is

14 22 54 70 81 89.

Step:8 Deleted root node from the max heap and it is placing in the sorted array.

11	14	22	54	70	81	8



max heap.

Then the sorted array is

9, 11, 14, 22, 54, 70, 81, 89.

8/12/21

Algorithm:

```
Void heapify (a[], n, i)
    {
Int largest = i;
Int l = 2*i;
Int r = 2*i+1;
// left child is larger than root.
While (l<=n && a[l] > a[largest]){
        largest = l}
// right child is larger than root
While (r<=n && a[r]>a[largest]){
    largest = r}
If (largest != i){
    Int temp = a[i]
```

```
a[i] = a[largest]
a[largest]= temp;
Heapify (a,n, largest)}
// function for implementing heap sort
Void heapsort (a[], n)
{
for (i= n/2-1 ; i>=1; i--) {
        Heapify(a, n, i) }
// one by one delete from the heap.
for (i= n; i>=1; i++)
   Int temp= a[0];
    a[0]= a[i]
    a[i] = temp
    Heapify (a, i, 1) } }.
```

Frequency Count;

```
Algorithm Sum (a,n){ ---  0
   s=0 ------------  1
for (i=0; i<n; i++){ ---  1+n+1  0+n= 2n+2
   s = s+a[i] } ---  n
Returns ---------  1
} ---------  0
```

Frequency Count $f(n) = 1+n+1+n+1$

$= 2n+3.$

Time Complexity $= O(2n+3)$

$= O(n)$

Space Complexity $S(P) = Sp+c$

Sp - i-1 memory $S(P) = n+3$

s-1 memory $S(P) = O(0)$

c- a-n memory

n-1 memory

```
Algorithm add (a, b, n, i, j) --- 0
{ ---------- 0
for (i=0; i<n; i++) { ------ n+1
for (j=0; j<n; j++){ ------ n(n+1)
   c[i,j] = a[i,j] +b[i,j]--- n*n
} ---------- 0
} ---------- 0
} ---------- 0
```

Frequency Count $= n+1+n^2+n+n^2$

$= 2n^2+2n+1$

Time complexity $= O(n^2)$

Space Complexity = ~~O(n)~~ $S(P) = Sp+c.$

i - 1 memory

j - 1 memory

c - n^2 memory

a - n^2 memory

b - n^2 memory

n - 1 memory

$S(P) = 3n^2+3$

$S(P) = O(n^2).$

Algorithm

```
multiply (a, b, c, i, j, n) {---- 0
for (i=0; i<n; i++) {---- n+1
for (j=0; j<n; j++) {---- n(n+1)
c[i, j] = 0 ;---------- n*n
for (k=0; k<n; k++) {---- n(n)(n+1)
c[i, j] = c[i, j] + a[i, k] * b[k, j])--- n*n...
} ---- 0
} ---- 0
} ---- 0
} ---- 0
```

frequency count = $n+1+n^2+n+n^2+n^3+n^3+n^3$

$= 2n^3+3n^2+2n+1$

Time complexity = $O(n^3)$

Space Complexity $S(P) = Sp+C$

A = n^2

B = n^2

C = n^2

i - 1

j - 1

n - 1

k - 1

$Sp) = 3n^2+4$

$S(P) = O(n^2)$.

9/12/21 Ex:4

Increment for loop

```
For (i=0; i<n; i++)------ n+1
{
Stat; ---------- n
}
```

Frequency Count $f(n) = n+1+n = 2n+1$

Time Complexity $= O(2n+1)$

$= O(n)$

Ex:5

Decrement for loop

```
For (i=n; i>0; i--)---- n+1
{
Stat; ---- n.
}
```

→ frequency count $f(n) = 2n+1$

Time complexity $= O(2n+1)$

$= O(n)$

Explaination

N = 6

I = 6; 6>0 ---- i = i-1 = 6-1 = 5

I = 5; 5>0 ---- i = i-1 = 5-1 = 4

I = 4; 4>0 ---- i = i-1 = 4-1 = 3

I = 3; 3>0 ---- i = i-1 = 3-1 = 2

I = 2; 2>0 ---- i = i-1 = 2-1 = 1

I = 1; 1>0 ---- i = i-1 = 1-1 = 0

I = 0; 0>0 ---- false.

6+1 = 7 times if it is n time = n+1,

Ex:6

```
For(i=1; i<n; i=i+2)   ---------- n/2+1
{
stat;                  ---------- n/2
}
```

frequency count
$f(n) = n/2 + n/2 + 1$
$= 2n/2 + 1$
$= n+1$

Time complexity
$= O(n+1)$
$= O(n)$

Explanation:

Suppose n=4

I = 1; 1<4 ---- i = i+2 = 1+2 = 3
I = 3; 3<4 ---- i = i+2 = 3+2 = 5
I = 5; 5<4 ---- false.

Suppose n=8

I = 1; 1<8 ---- i = i+2 = 1+2 = 3
I = 3; 3<8 ---- i = i+2 = 3+2 = 5
I = 5; 5<8 ---- i = i+2 = 5+2 = 7
I = 7; 7<8 ---- i = i+2 = 7+2 = 9
I = 9; 9<8 ---- false.

Exp 7:

```
For(i=1; i<n; i=i*2)
{
stat;
}
```

Time complexity

Explanation;

I = 1 ---- i = i*2 = 1*2 = 2
I = 2 ---- i = i*2 = 2*2 = 4
I = 4 ---- i = i*2 = 4*2 = 8

I = 8 ---- i = i*2 = 8*2 = 16

I = k ---- 2 pow k.

When it stops the condition.

i > n
i = n
$i = 2^{pow\ k} = 2^k$

Substitute i value

$i = n$
$2^k = n \Rightarrow k = \log_2 n$

Time complexity

$T = 8 = O(\log_2 n)$
$= O\log$
$= O(\log 2^3)$
$= O(3\log_2 2)$
$= O(3*1)$
$= O(3)$ [$\because \log_a a^b = b\log_a a$, $\log_a a = 1$]

Ex:8

```
For(i=0; i*i<n; i++)
{
stat;
}
```

i*i<n
i*i>n it stops the condition. (terminates).

n = 8

i=0; 0*0<8 ⟶ i=0+1 = 1
i=1; 1*1<8 ⟶ i = 1+1 = 2

i = 2 ; 2*2 < 8 → i = 2+1 = 3

i = 3 ; 3*3 < 8

9 < 8 → false

9 < 8 ✗

9 > 8 ✓

∴ Time complexity = O(log√n)

9) for (i=n; i>=1 ; i = i/2)
{
stat ;
}

Explanation :-

I = 1 - - - - i = n/2 pow 1

i = 2 - - - - i = n/2 pow 2 = $(n/2)^2$

i = 3 - - - - i = n/2 pow 3 = $(n/3)^3$

i = k - - - - i = n/2 pow k = $(n/2)^k$

When it stops the condition

$i = n$

$i = \frac{n}{2^k}$

Substitute i value

$\frac{n}{2^k} = 3$

$n = a^k$

$k = (\log_2 n)$

Time complexity = $O(\log_2 n)$

10) I = 1

While (i < n)
{
stat ;
i = i*2
}

for (i = 1 ; i < n ; i = i*2)
{
stat ;
}

Both are same here.

Time complexity = $O(\log_2 n)$.

11) i = n ;
While (i > 1)
{
stat ; { i = i/2 ;
}

Time complexity = $O(\log_2 n)$

for (i=0; i<n; i++) - - - - O(n)

for (i=0; i<n; i=i+2) - - - O(n)

for (i=n; i>1; i - -) - - - - O(n)

for (i=1; i<n; i = i*2) - - - $O(\log_2 n)$

for (i=1; i<n; i = i*3) - - - $O(\log_3 n)$

for (i=1; i<n; i = i*7) - - - $O(\log_7 n)$

for (i=n; i>1; i = i/2) - - - $O(\log_2 n)$

Constant time complexity :- without looping concept

Algorithm add (a, b, c) - - - - 0
{
printf (enter a, b values) - - - - 1
Scanf (read a, b values) - - - - 1
c = a+b ; - - - - - - - - - - 1
printf (display c values) - - - - 1
} - - - - - - - - - - - - 0

Frequency count = 0+1+1+1+1+0

= 4

Time complexity = O(1).