Day-2

=> For Nesting Concept Rules :- Time Complexity => O(n square)

Ex:- For (i=0; i<n; i++) ---> n+1 times

{

for (j=0; j<n; j++) -> n*(n+1)

{

statements :-> n*n ... n square

}

};

T.C => O(n square).

=> memories :-
1. Stack
2. Heap

=> Implement a 2D array & rotate the matrix 90°

C & Java -> premitive

- Python -> Non premitive

=> 2x2 matrix Rotation

=> Anti-clockwise (Left):

$\begin{bmatrix} 9 & 3 \\ 10 & 6 \end{bmatrix}$ Ans $\begin{bmatrix} 3 & 6 \\ 9 & 10 \end{bmatrix}$

-> $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ -> $\begin{bmatrix} 3 & 1 \\ 4 & 2 \end{bmatrix}$

=> clock wise (Right).

3x3 Matrix Rotation

$\begin{bmatrix} 10 & 15 & 20 \\ 100 & 200 & 300 \\ 1 & 2 & 3 \end{bmatrix}$ Ans ->

Anti clock wise.

$\begin{bmatrix} 20 & 300 & 3 \\ 15 & 200 & 2 \\ 10 & 100 & 1 \end{bmatrix}$

=> clock wise $\begin{bmatrix} 1 & 100 & 10 \\ 2 & 200 & 15 \\ 3 & 300 & 20 \end{bmatrix}$.

> $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ => Transpose of matrix

Ans => $\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$

Transpose

$$\Rightarrow \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \Rightarrow \begin{bmatrix} 3 & 1 \\ 4 & 2 \end{bmatrix}$$

reverse

2×2

Q:

$$\begin{bmatrix} 3 & 4 \\ 6 & 8 \end{bmatrix} \Rightarrow \begin{bmatrix} 3 & 6 \\ 4 & 8 \end{bmatrix} \Rightarrow \begin{bmatrix} 6 & 3 \\ 8 & 4 \end{bmatrix}$$

transpose

Reverse

ex ②: `for(i=0; i<n; i++)` ⇒ Tested for loop

```
{
for(j=0; j<i; j++)
{
statements;
}
};
```

ex: i=0 then 0<0 no

i	j
0	nothing ⇒ 0<0
1	0<1 ⇒ 0 will execute 1<1 ⇒ 1 will stop
2	0<2 ⇒ 0 will execute 1<2 ⇒ 1 will execute 2<2 ⇒ 2 will stop

⇒ 0, 1 will execute

⇒ so, when i 0 ---- 0 times
1 ---- 1 times
2 ---- 2 times

so 1+2+3+ ... n = n(n+1)/2

= (n square + n)/2

TC = O(n square)

⇒

```
P=0
for(i=1; p<=n; i++)
{
P=p+i;
}
```

i p

① 0 + 1 bcs p=p+i, when N is 0

② 1+2=3 --- N is 1

③ 1+2++3=6

⋮

k 1+2+ ---k.

=Not n times assuming

When will stop when p>n

p=k(k+1)/2 bcs 1+2+---k

p=k square ≥n

So n = square root of n

$(n = \sqrt{n})$

⇒ Time Complexity = O(sqrt(n))

```
for (i=1; i<n; i*2)
{
statements :
}
```

Analyze n=10, log 10 base 2

i=1 1 time

i=2 2 times (1*2)

i=3 (1*2)*2 = 2 power 2

i=4 (1*2)*2)*2 = 2 power 3

i=5 ((1*2)*2)*2)*2 = 2 power 4

So, when stopes i>=n

i=2 power k

2 power k >= n

k = log n base 2

So T.C => O(log n base 2)

8

```
x = int (input ("Enter starting value:"))
y = int (input ("Enter Ending value:"))
print ("The Even numbers").
for i in range (x, y+1):
    if i%2 == 0:
        print (i)
```

```
print ("The 2 powd value,")
for j in range (x, y+1):
    if (j & (j-1)) == 0 and
        j != 0:
    print (j).
```

=> Create an dynamic 1D array it should contains num's b/w 20 to 30

extract & print i) even numbers

ii) 2 power values.

=> for(i=n; i>=1; i=i/2)

{

statements;

}

}

n

n/2

n/2 power 2.

...

n/2 power k

Assume i<1 stops right?

n/2 power k<1

n/2 power k=1

n=2 power k

k=log n base 2

T.C => O(log n)

=> to n=10, i=10

=> 10=10; 10>=1 = i=10/2

n = 5 (- Float

n = int

=> 5>=1 = 5/2

n = 2

=> 2=2 => 2>=1, i=2/2

=1.

=> 1=1 => 1>=1 = i=1/2

n = .

Python

⇒ Derived Formulae

for(i=0; i<n; i++) ⟶ O(n)

for(i=0, i<n; i+2) ⟶ O(n)

for(i=n; i>1; i--) ⟶ O(n)

for(i=1; i<n; i=i×2) ⟶ O(logn base 2)

for(i=1; i<n; i=i×3) ⟶ O(logn base 3)

for(i=1; i>1; i=i/2) ⟶ O(logn base 2)

* Names of TC

1. Constant Time Complexity : O(1)...
2. Linear Time Complexity : O(n)...
3. Logarithmic TC : O(log n)...
4. Quadratic TC : $O(n^2)$
5. Exponential TC : $O(2^n)$.

=> i++ i-- 1+2 → O(1)

i*2 i/2 → log(n) base 2

* parallel Concept to Time Complexity → [Space Complexity]

* Array of size n, require O(n) space.

- Two dimensional array of size n*n: O(n^2) space.

Linear search O(1)

Merge sort O(n)

DFS O(n)

BFS O(n)

⇒ Dynamic programing O(n^2) (or) O(n*2)

* Constant Complexity = O(1)

Same amount of space regardless of the i/p size n it is called Constant Complexity.

ex:- (1) Sum of array elements & linear search

Q) why O(1)? why not O(n)?

⇒ space not depending on values

* Sum of array elements using function

```
def sum(arr):
    sum = 0
    for i in arr:
        sum = sum + i
    return(sum)
if __name__ == "__main__":
    arr = [12,3,4,15]
    n = len(arr)
    ans = sum(arr)
    print("Sum of the array is", ans)
```

```
=> def array_summer(arr):
        total = 0
        for item in arr:
            total += item
        return total
print(array_summer
        ([1,2,33,7]))
```

O/P: 16.

array => S.C => O(n)

sum of array => S.C => O(1)

linear search : O(n)

Summary = O(1)

=> ~~Space~~ Complexity:- Logarithmic Complexity:-

```
=> #include <stdio.h>
   struct a{
       double x;
       int y;
       char z;
   };
   int main()
   {
       struct a yes;
       printf("%d", sizeof(yes));
       return 0;
   }
```

O/P: 16

Linear Complexity: O(n)

when an alg takes space directly proportional to the i/p size

ex: i/p a num

-> Log linear Complexity: O(nlogn)

when the space Complexity of an alg grows proportionally to the i/p size & a logarithmic factor

ex: Merge sort

-> Log(n) => log linear Complexity

-> row [1-10] -> Nested list => polynomial complexity => O(n²)

```
def generate_lists_of_lists(n):
    table_list = []
    for num in range(n):
        row = []
        for i in range(n):
            row.append(i)
        table_list.append(row)
    return table_list
print(generate_lists_of_lists(10))
```

n = number
n = num of o/p
ex: 10
up to 9
-> 10 times

Type 1

~~double~~ → char
2 int - ~~24~~
2 double - 24

char 4,
int double 16
int double

int
double
char.

Type - 3
double
int
char
4,
16

double
char
int.

4,
char
double ⇒ 24
int

Sum of n numbers using Recursion

```
int add (int n);
  if (n<=0);
    return 0;
  }
    return n + add(n-1);
}
```

⇒ O(n) space Complexity.

⇒ Fibanocci ⇒ O(1)
⇒ Factorial
4,
O(n) = 5 · C
n x n! (n-1)