DAY-2 28/9/2023

* Nested for loop

```
for (i=0; i<n; i++).  ——— n+1 times
{
   for (j=0; j<n; j++)  ——— n*(n+1)
   {
      Statements   n*n ——— n square
   }
};
```

Time complexity : $O(n^2)$.

* Matrix

We use 2D arrays by using nested for loop for Implementing.

* Implement a 2D matrix and Rotate the
* C and Java uses primitive data type
* C and Java Supports no primitive data type
* Python Supports non primitive means we can use variables directly

* Pseudo code

=> Adding of numbers.

Set n_1, n_2
$n_3 : n_1 + n_2$
Print : n_3

} Algorithm.

* 2x2 matrix

* rotate 90°

$$\begin{bmatrix} 9 & 3 \\ 10 & 6 \end{bmatrix}^{90^\circ}$$

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

* $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^{90^\circ}$

$$\begin{bmatrix} 2 & 4 \\ 1 & 3 \end{bmatrix}$$ Add 9

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

3x3

$$\begin{bmatrix} 10 & 15 & 20 \\ 100 & 200 & 300 \\ 1 & 2 & 3 \end{bmatrix}^{90^\circ}$$

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

2x2 = 4 elements
3x3 = 9 elements

* $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^{90^\circ}$ clock wise, find transpose

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

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

transpose the matrix and reverse row wise for 90° clock wise direction.

matrix = a[2][2];

for range of (5:-1:-1)

o/p 4 3 2 10

* Nested for

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

i	j
0	nothing
1	0 will execute 1 it will stop so, 0, 1.
2.	2 it will stop.

i=0 the cn

0<0 is nothing

So, when i=0 — 0 times

1 — 1 time

2 — 2 times

so $1+2+3+ \cdots n = (n+1)/2$

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

time complexity = $O(n^2)$

*

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

0+1 — P = P+i n is 0

1+2=3 — N is 1

1+2+3 = n=6

k = 1+2+k -

not n times assuming

when will stop when $p \geq n$

$P = k(k+1)/2$ ——— $1 + 2 + k$

$P = k^2$ Sq $> n$

so $n = \sqrt{n}$ time complexity = $O(\sqrt{n})$

* for ($i=1$; $i < n$; $i * 2$)

{

Statements;

}

Analyse

$i=1$	1 time	
$i=2$	2 times	$(1*2)$
$i=3$	4	$(1*2)*2 = 2^2$
$i=4$	8	$((1*2)*2)*2 = 2^3$

So when $i >= n$

$i = 2^k$

$2^k >= n$

$k = \log n_2$

time complexity = $(O \log n_2)$

* Create Dynamic 1D Array A should Contains numbers between 10000 to 30

[10, 11, 25, 30, 16, 21, 29]

1. Extract and print even numbers
2. Extract 2 power values.

By using python.

```
x = int (input("enter initial value:"))
y = int (input("enter final value:"))
print("here the availasle even numbers are:"))
for i in range (x, y+1):
    if i%.2 == 0:
        print (i)
print("Here is 2 power values:")
for j in range (x, y+1):
    if (j+(j-1)) == 0 and j!=0:
        print (j).
```

Output

```
enter initial value: 10
enter final value: 40
here the availasle even numbers are:
10
12
14
16
18
20
22
24
26
28
30
32
36
38
40
the 2 power values are:
16, 32
```

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

{

statements;

}

n=10

log 2

10/2 >= 5 10 >= 1 true

10

5/2 >= 2 5 >= 1 - 5

2 >= 1 - 2

2/2 1 1 >= 1.

N

n/2

(n/2)^2

...

(n/2)^k

Assume i<1 it stops

n/2^k<1

n/2^k=1

n=2^k

k = log_2 n

time complexity = O(log n)

* Derivative formula

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(log n_2)

for (i=1; i<n; i=i*3) ——— O(log n_3)

for (i=n; i>1; i=i/2) ——— O(log n_10).

* Constant time complexity => O(1)
* Linear time complexity => O(n)
* Logarithmic time complexity => O(log n)
* Quadratic time complexity => O(n^2)
* Exponential time complexity => O(2^n)

* Space Complexity

* Parallel to time time Complexity
* array of size n, it require O(n) space.
* 2D array of size n*n : O(n^2)

* Linear Search — O(1)
* Merge Sort — O(n)
* Deapth First Search [DFS] — O(n)
* Breadth First Search [BFS] — O(n)
* Dynamic Programming O(n^2) or O(n*2
* Constant Complexity : O(1)
* Same amount of space of the input size 'n'. It is called Constant Complexity
* Sum of array elements using function

~~import java.util.*;~~

```
number_list = [1,5,7,7,5,4,3,8].
int a[] = {5,6,7,8};
int sum = 0;
for (int i=0; i<a.length; i++)
{
  sum = sum+i;
}
Printf(" " + sum);
}
```

* Linear Search

⇒ it is a Sequential Search.

⇒ it Searches ~~for~~ to Sequentially in the array.

⇒ It consists of the location of array

⇒ it Searches the element you want to be find.

* why it is O(1)

⇒ Space is not depending on values.

* Quick Sort

Space Complexity O(log n)

⇒ Quick Sort takes space proportional to log of input size.

⇒ Because Recursion takes extra space [stack]

Linear Complexity : O(n)

when an algorithm takes space directly Proportional to input size

Ex: input a number

Create an array with size n.

and fill it with all 0's and 1's.

used extra space and it depends on your input --

Space Complexity is O(n)

* log-linear Complexity O(n log n) → space.

when a space complexity of an algorithms grows proportionally to the input size and a logarithmic factor.

log( ) → it is called log-linear complexity

* 1 log n → one inner and outer loop.
→ divide and conquer / merge sort

* 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))

polynomial Complexity: O(n²)

* Structure - int, char, double

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

→ O/P: 16

double
double
char
} 24

int
int
char
} 12

char
double
double
} 24.

char
int
int
} 12

char
int
double
} 16

char
double
int
} 24

① take character wherever you want with int and double it will occupy 24 bytes.

Ex
char	double	double
~~int~~ double	char	double
double /	double ,	char

② take int double ~~wherever~~ you kept 'char' first/last

char	int	int
16 { int	double } 24.	char } *16.
double	char	double }

③ take double with char and int, double will be dominate

double	double	char
int } 16	char } 16	double } 24.
char	int	int

* Space Complexity of Recursion is O(n)
* Space complexity of factorial Recursion is O(n)
* Space complexity of fibonacci Recursion is. O(n)