Time & Space Complexity-1

i/p patterns

mostly covered.

→ rest is practice.

→

input (typ)

t → x.

sizeofarr → n:

items → 1 2 3 4 .. n.

→ n.

→ x

→ y

t → x

size → n.

items → 1

2

3

4

5

6.

t → x

#of query → q.

1 y

2

3

query type:

1 y → insert item y in stack.

2 → pop item

3 → print item at top of stack.

Grammar :-

⇒ Tense (Verb) (Object) (Subject) , (Predicate)

Rules

2

2

1 2.

4

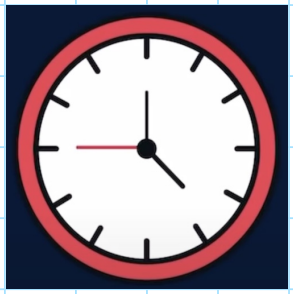
1 2 3 4.

i/p

to take

how such i/p in python

⇒ Go to Python Documentation on input taking.



Time

&

Space

Complexity

100000 /s.

20000/s

70000/s

P1 → A1 → 70000/s

P2 → A2 → 30000/s

P3 →

⇒ Let's understand it using real life example :

OS → Phone → Android / IOS

Laptop → Window / Linux / IOS ..

Processor
- core.
- GPU.

→ processing speed.

RAM

Inbuilt memory → to store your data

→ Multitasking.

Camera.
- MPixels

Phone A (50MP) → pic

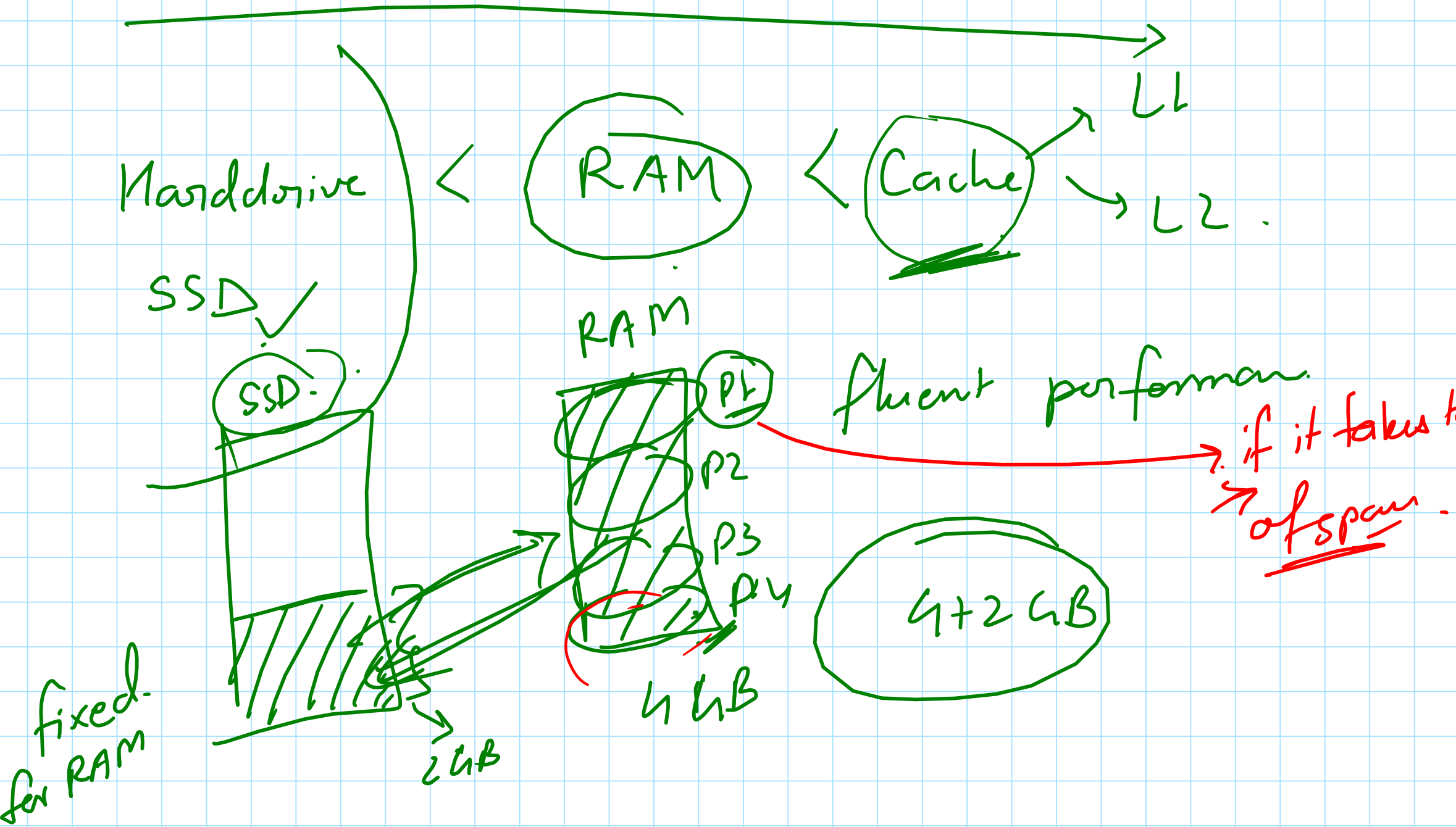
Phone B (50MP) → pic

→ compare..

Software / program.
- algo is not fast then alone processor can't do much
- space taken by that prog.

Google Pixel. photos.

Iphone photos


Harddrive
RAM
Cache
L1
L2.
SSD
SSD
RAM
P1
P2
P3
P4
4 GB
2 GB
fixed
for RAM
fluent performan
if it takes too much
offspace.
4+2 GB

money making depends

on usability → if slow & takes more space
↳ becomes less usable.

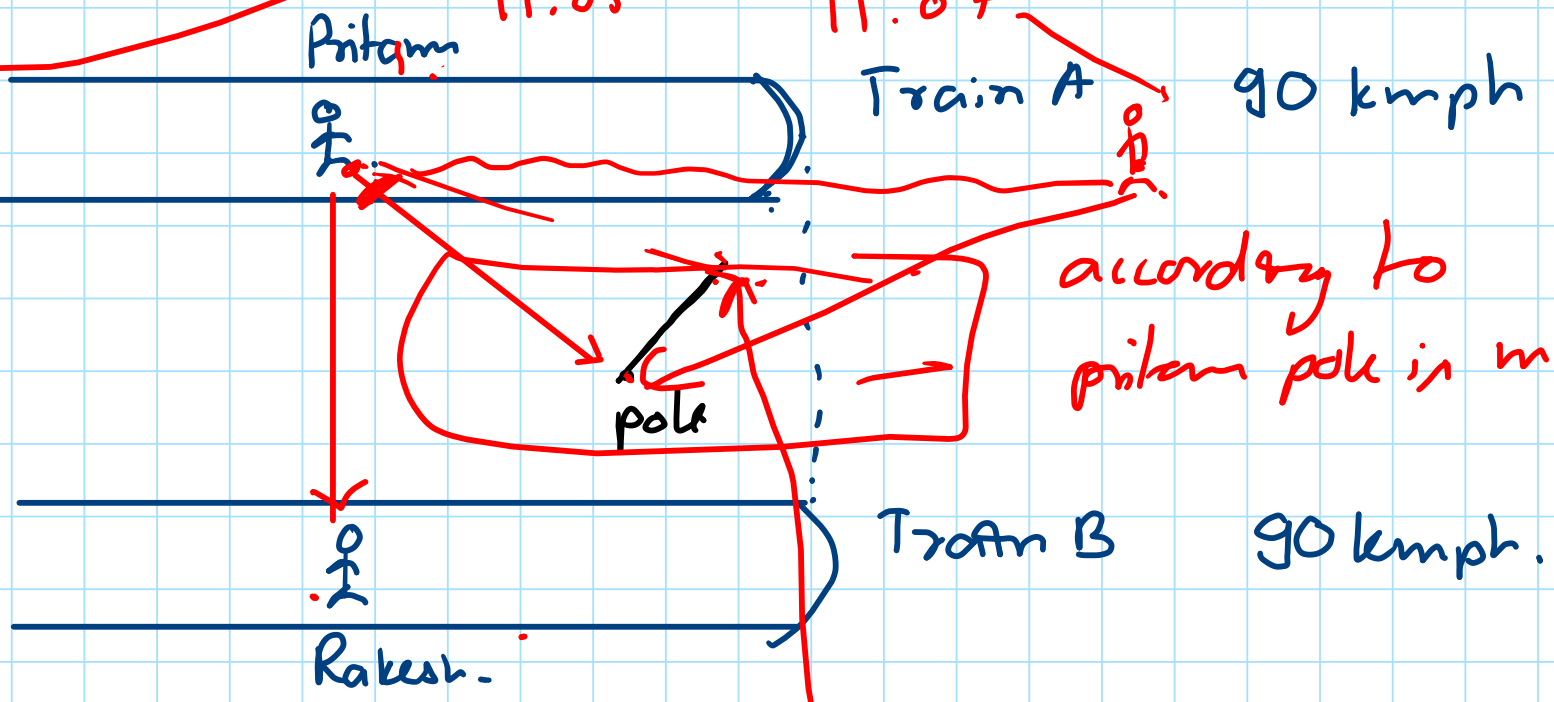
→ if fast & takes less space
↳ becomes more usable.
↓
ultimately leads to more sales.

In CS, the cost of Algorithms are calculated in terms of T.C. and S.C.

⇒ " A Program would be as good as Less space it takes and as Faster it works. "

⇒ That's why the final product should be efficient to take less space & time.

Frame of Reference :-



according to pritam pole is moving.

according to pritam Rakesh is stationary.

Time Complexity :

-> A technique to characterize the time taken by an Algorithm wrt to the Input Size.

if i/p size in ↑

if i/p size in ↓.

-> completely independent to the machine, language and compiler

-> It is used for evaluating the variations of execution time on different Algorithms.

Algo → 10^6 compt.

10^3 compt^n ⟶ 1 sec.

1 compt^n ⟶ $\frac{1}{10^3}$ sec

10^6 compt^n. ⟶ $10^6 \times \frac{1}{10^3} = 10^6 * 10^{-3} = 10^{6+(-3)} = 10^3$ seconds

→ P1 → 10^3 compt/s. → 10^3 seconds

→ P2 → 10^6 compt/s. → 1 second.

Now, the Question :

↳ How to find the Time & Space Complexity ?

Time Complexity :→ It's defined by the relation between i/p size and the Running Time (number of Operations) (RT).

for example :-

# of operations

① print("hello") → 1

② print("hello")
print("hello") → 2 operations.

Time Complexity : It's defined by the relation between i/p size and the Running Time (number of Operations)

$10^{18} \neq 2 \times 10^9$ (2 billion)

for example :-

# of operations

(1)
```
print("hello")
```
→ 1

(2)
```
print("hello")
print("hello")
```
→ 2

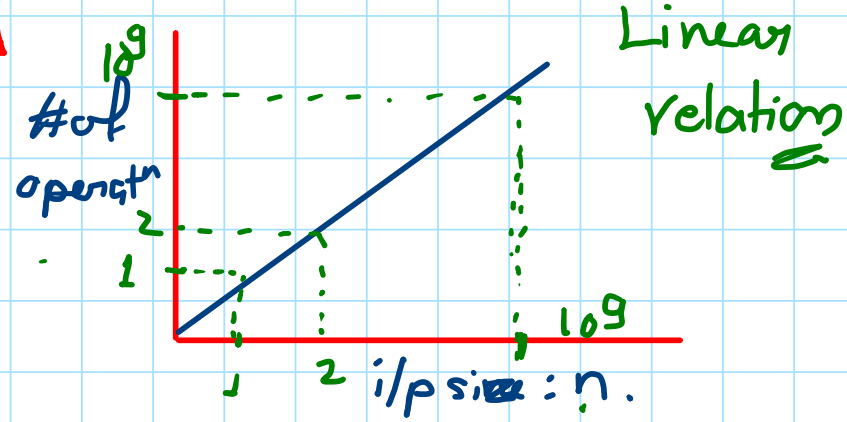
(3)
```
n = input()
for (i in range(n)) :
    print("hello")
```
→ n

for n = 6 → 6

$n = 10^9$

$n \simeq 10^9$ (1 billion)

$n = 10^{18} \rightarrow 10^{18}$



So, in T.C. we are finding how i/p size is related to the running time or operations.

→ the relation can be —

↳ Linear

if n becomes → 10 *n

R.T becomes → 10 * RT

↳ Quadratic (square)

if n ⟶ 2*n.

RT ⟶ $(2)^2$ *R.T

↳ Cubic relation.

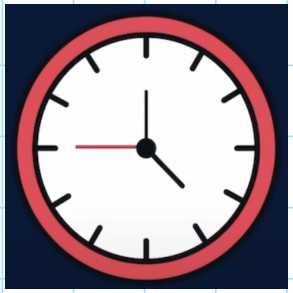
if n ⟶ 2*n.

R.T. ⟶ $(2)^3$ * R.T.

$2^3 = 8$.

↳ similarly for logn, sqrt, so on. . . .

So, "Lesser the Relation better the code would be." (in terms of T.C.)



Time Complexity :

Searching :

① Best Case :- $\Omega(\ )$

→ found the item at 0^{th} index.

→ $\Omega(1)$ operation.

---

② Average Case :- $\Theta(\ )$

→ average time/operations to find an item.

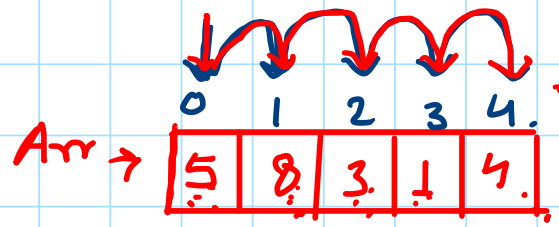
→ $\Theta(3)$ or $\left(\frac{n+1}{2}\right)$

---

③ Worst Case :- $O(\ )$ ← termed as upper limit.

→ max time/operations to find an item

→ $O(5)$ or $O(n)$



# of operations.

Key = 1000

key = 5	→	1 ✓
= 8	→	2 ✓
= 3	→	3 ✓
= 1	→	4 ✓
= 4	→	5
= 6	→	5

→ worst case : n operation

---

avg time to search any item

$$= \frac{1+2+3+4+5}{5} = \frac{15}{5} = 3$$

$$= \frac{(1+2+3+4+n)}{n} = \frac{1}{n} * (\text{sum of 1 to n})$$

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

Time Complexity :

→ In Interviews we talk about the Worst Case

i.e., "O( )" → Big O notation or simply call it "Order of"

Examples :-

n = 10.

```
n = input()
for(i in range(n)):
    for(j in range(n)):
        print("hello")
```

How many operations ?

10 times

i = 0 , j = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 → 10 times

i = 1 , j = 0, 1, 2, 3, 4, - - - 9 → 10 times

i = 2 , j = 0, 1, 2, 3, 4, - - - - 9 → 10 times

⋮

i = 9 , j = 0, 1, 2, 3, 4, - - - 9 → 10 times

T.C. → $O(n^2)$

Worst case scenario.

Total # of operations $= 10 \times 10 = 100$ or $n * n = n^2$.

i = 0 1 2 3 4 5 6 7 8 9

j = 10 + 10 + 10 + 10 + 10 + 10 + 10 + 10 + 10 + 10

$= 10(1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1)$

$= 10(10) = 10 * 10 = 100 \simeq n * n = n^2$.

5 min. break.

meet at 1:10 pm)

Examples :-

```
n = input()
m = input()
for(i in range(n)):
    for(j in range(m)):
        print("hello")
```

n times

i = 0 , j = 0, 1, 2, ... m-1 → m times

i = 1 , j = 0, 1, 2, - - - m-1 → m times

⋮ ⋮

i = n-1 , j : 0, 1, 2, - - - m-1 → m tim.

T.C. → O(n * m)

↳ worst case.

Constraint

$n \in [0, 10^9]$ → worst case 10^9

$m \in [0, 10^9]$ → 10^9.

Then, ∴ T.C. → O(nm)

= $O(n^2)$ or $O(m^2)$.

i = 0, 1, 2, ... n-1

j = m + m + m . . . . . + m.

= m (1 + 1 + 1 + ... 1)

= m (n) = m * n ≃ T.C. = O(mn)

Examples:-

```
n = input()
m = input()

for(i in range(n)):
        print("hello")

for(j in range(m));
        print("hello")
```

→ $O(n)$

→ $O(m)$

Total # of operations = $n + m$.

T.C → $O(n) + O(m) = O(n)$

not actual addition.

it will be narrowed down to the dominant term

constraints.

$n \in [0, 10^9]$

$m \in [0, 10^8]$

$O(n) + O(m)$ $O(m)$ ✓

$O(n+m)$

in some program

→ for (i in range(n)): → O(n)
    print ("hello")

→ for (j in range(m)): → O(m*p)
    for (k in range(p)):
        print ("world")

T.C. → O(n) + O(m*p)
↳ O(n) + O(m²)
↓ 10⁹ ; ↓ 10¹⁸ dominating.

$(10^9)^2 = 10^{9*2} = 10^{18}$.

ultimately → T.C. → O(m²) or O(n²)

Constraints:

n ∈ [1, 10^9]
m ∈ [1, 10^9]
p ∈ [1, 10^9]

m can be replaced by p & vice-versa in worst case

Compare :-

better

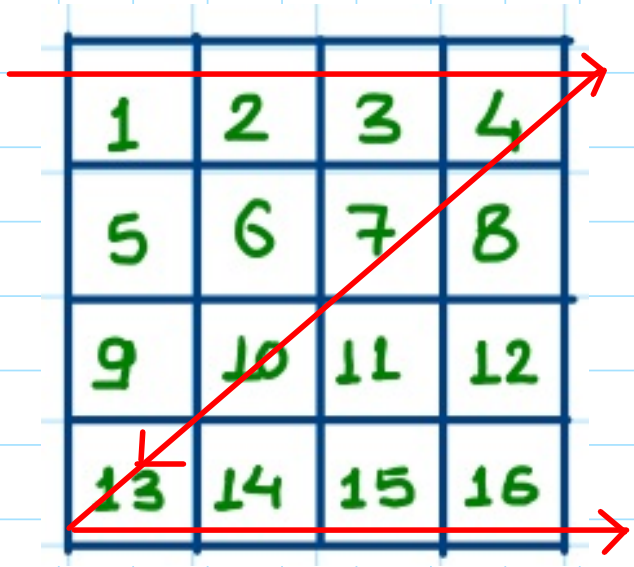
Size \ T.C.	$O(n)$	$O(n^2)$	$O(n^3)$
$n=1$	1	1	1
$n=2$	2	4	8
$n=3$	3	9	27
⋮			
$n=10^9$	10^9	10^{18}	10^{27}

↓

1 Billon. $< 10^{18} < 10^{27}$

$10^9 * 10^9 = 10^{(9+9)} = 10^{18}$

Z Traversal :



o/p → 1 2 3 4 7 10 13 14 15 16

Step 1 : Print Top row from L to R.

Step 2 : Print Right diagonal excluding 1st and last item.

Step 3 : Print the Bottom row from L to R.

Pseudo Code :

```
// Print the top row from left to right
    for (i = 0; i < N; i++) {
        print(matrix[0][i])
    }

// Print the diagonal (excluding the first and last element) from top-right to bottom-left
    for (i = 1; i < N - 1; i++) {
        print(matrix[i][N - 1 - i])
    }

// Print the bottom row from left to right
    if (N > 1) { // Ensure the bottom row is not the same as the top row for N=1
        for (i = 0; i < N; i++) {
            print(matrix[N - 1][i])
        }
    }
```

O(n) → O(n)

O(n) → O(n)

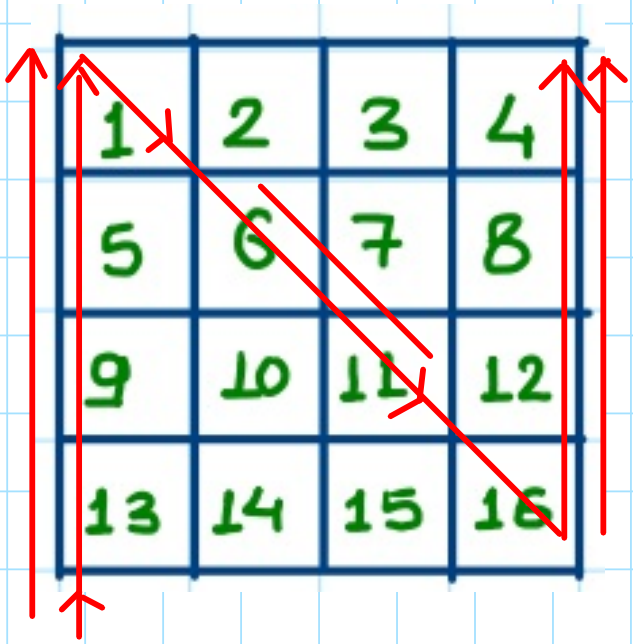
O(n) → O(n)

O(n) O(n²) 3n O(n³)

T.C. → O(n) + O(n) + O(n)

≃ O(n)

N Traversal :



O/p → 13 9 5 1 6 11 16 12 8 4

Step 1 : Print Left most column from Bottom to Top.

Step 2 : Print Left diagonal except 1st and last row.

Step 3 : Print Right most column from Bottom to Top.

Pseudo Code :

```
//Print Left most column from bottom to top
for(i = N - 1; i >= 0; i--) {
    print(matrix[i][0])
}

//Print left diagonal except 1st and last row
for (i = 1; i < N - 1; i++) {
    print(matrix[i][i])
}

//Print Right most column from bottom to top
for( i = N-1; i >= 0; i--) {
    print(matrix[i][N-1])
}
```

→ O(n)

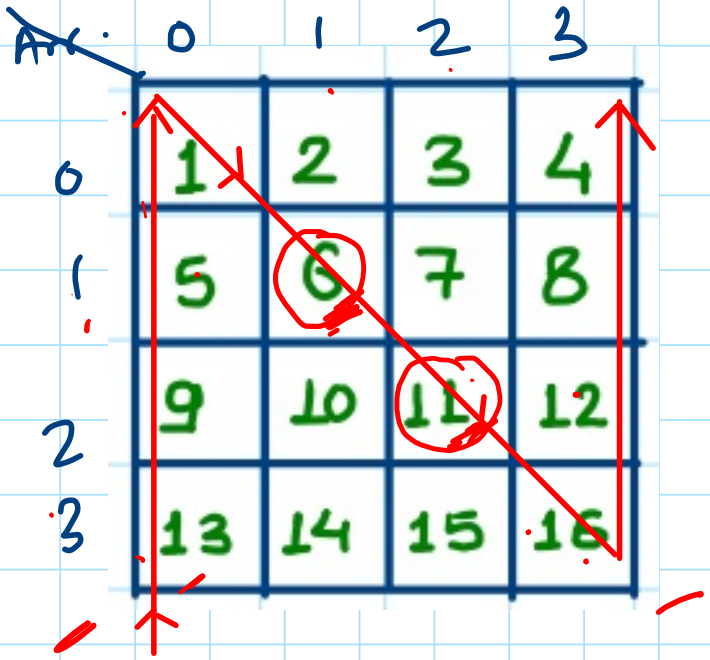
→ O(n)

→ O(n)

T.C. → O(n) + O(n) + O(n)

↳ O(n)

⇒ N Traversal :



T.C → O(n * n)

↳ O(n²)

o/p → 13 9 5 1 6 11 16 12 8 4

Print

- Step 1 → Right most column from Bottom to Top.
- Step 2 → Print Left diagonal.
- Step 3 → Print Right most column. from Bottom to Top.

reverse N Traw

correct

Step1 →

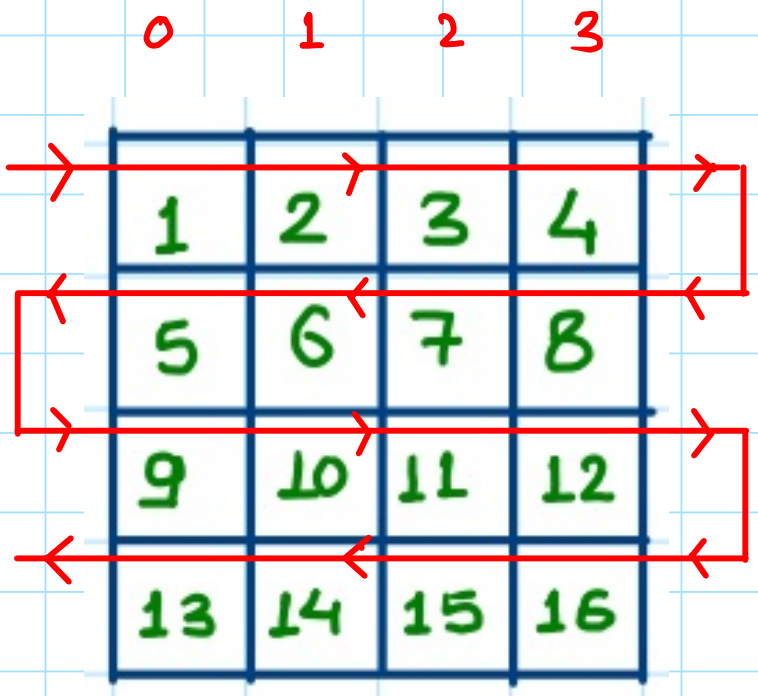
```
for(j=0 ; j<n ; j++)
{
  for(i=n-1 ; i>=0 ; i--)
  { if(j==0 || i==j || j==n-1)
      print ( A[i][j])
  }
}
```

j=0 , i=3 , A[3][0]
i=2 , A[2][0]
i=1 , A[1][0]
i=0 , A[0][0]

j=1 , i=3✓ , i=2✓ , i=1✗ , i=0✗

j=2 , i=3 , i=3 , i=2.

Snake Traversal or Zig-Zag Traversal :

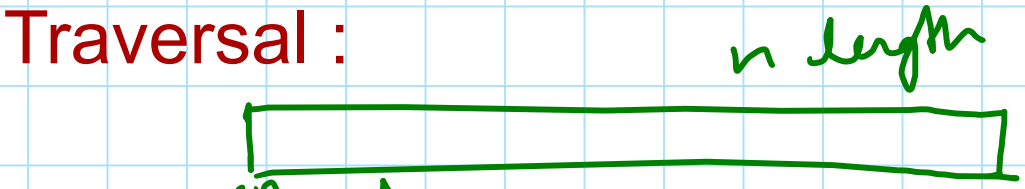


O/p → 1 2 3 4 8 7 6 5
9 10 11 12 16 15
14 13

Steps :

→ for every row# % 2 == 0
print items from L to R &

→ for every row# % 2 != 0
print items from R to L



Pseudo Code :

```
for(i = 0; i < n; i++) {
  if(i % 2 == 0){
      for(j = n - 1; j >= 0; j--) {
          print(array[i][j])
      }
  } else {
      for(j = 0; j < n; j++){
          print(array[i][j])
      }
  }
}
```

O(n) → n/2 → O(n)

O(n) → n/2 → O(n)

n/2 == n x

(A) O(n) (B) O(n²) (C) O(m x n)

(A) → 2 (B) → >20 (C) → 2

T.C. → O(n) (O(n) + O(n))
↳ O(n) . (O(n)) → O(n²)

Big-O Complexity Chart

Factorial
exponential
n^3
polynomial

Horrible | Bad | Fair | Good | Excellent

O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)

Operations

Elements

$O(\log n) < O(\sqrt{n}) < O(n)$

Comparison Table:

Time Complexity	Example	Description
O(1)	Accessing array element	Constant time, no impact on input size
O(log n)	Binary Search	Logarithmic time, efficient for large datasets
O(n)	Linear Search	Linear time, grows proportionally with input size
O(n log n)	Merge Sort	Linearithmic time, efficient sorting
O(n^2)	Bubble Sort	Quadratic time, slower for large datasets
O(n^k)	Matrix Multiplication	Polynomial time, common in matrix operations
O(2^n)	Recursive Fibonacci	Exponential time, grows rapidly
O(n!)	Brute-force permutation	Factorial time, extremely slow for large inputs