Polynomial → $O(n^2)$, $O(n)$, $O(n^3)$

Exponential → $O(2^n)$, $O(b^n)$, $O(n!)$.

-: Space Complexity :-

↳ max space (worst case) that is utilized at any point during running the algorithm.

→ we calculate this also in Big-Oh.

```
func(int N) { // 4 bytes
    int x; // 4 bytes
    int y; // 4 bytes
    long z; // 8 bytes
}
```

not counted.

→ 20 bytes.

i/p & o/p space is not counted.

so, rest space is 16 bytes.

↓

Constant

Hence S.C → O(1).

```
func(int N) { // 4 bytes
    int arr[10]; // 40 Bytes
    int x; // 4 bytes
    int y; // 4 bytes
    long z; // 8 bytes
    int[] arr = new int[N]; // 4 * N bytes
}
```

Space → ~~56 + 4N~~

S.C → O(N).

```
func(int N) { // 4 bytes
    int x = N; // 4 bytes
    int y = x * x; // 4 bytes
    long z = x + y; // 8 bytes
    int[] arr = new int[N]; // 4 * N bytes
    long[][] l = new long[N][N]; // 8 * N * N bytes
}
```

Total Space => ~~16+4N+8N²~~ $8N^2$ → $O(N^2)$.

Q) S.C of below Code ?.

```
int maxArr(int arr[], int N) {
    int ans = arr[0];
    for(i from 1 to N-1) {
        ans = max(ans, arr[i]);
    }
    return ans;
}
```

int i → 4 Bytes

S.C → O(1).

```
int maxArr(int arr[], int N) {
    int ans = arr[0];
    for(i from 1 to N-1) {
        ans = max(ans, arr[i]);
    }
    return ans;
}
```

int x = 5;

int i = 1.

int x = 5

ans = max(ans, arr[i]);

8 → 8 Bytes

S.C → O(1).

-: Arrays :-

Ordered collection of same type of data.

```
int arr[n];
```

int → data type, arr → variable name.

arr[0], arr[1], arr[2] . . . . arr[n-1];

Print all elements of array :-

```
void print_array(int arr[], n){
    for(i=0; i<n; i++){
        print(arr[i]);
    }
}
```

T. C → O(n)

S. C → O(1).

Ques) Given an array of size n, reverse the entire array.

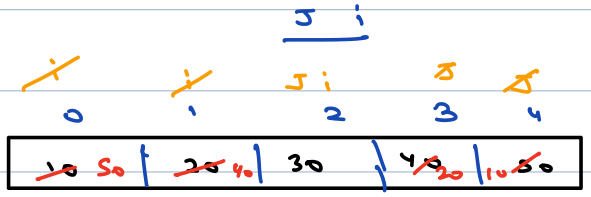
arr[] = {1, 2, 3, 4, 5}

↳ arr[] = {5 4 3 2 1}

arr[6]

i < j

i < j.



i <= j

Keep on doing till (i < j).

```
function reverse ( int arr[], int n) {
    i = 0, j = n-1;
    while (i<j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
        i++, j--
    }
}
```

swap.

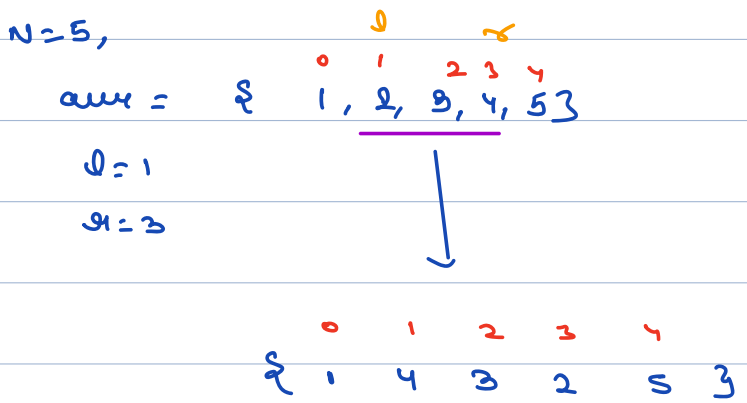
iteration = $n/2$.

T.C → O(n).

S.C → O(1).

Ques) Given an array of size n, & 'l' & 'r', reverse the array from l to r.

N = 5,




```
function reverse ( int arr[], int l, int r ) {
    i = l, j = r ;
    while (i<j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
        i++, j--
    }
}
```

swap.

T.C → O(n) (worst case)

S.C → O(1).

Break 10:01pm – 10:10pm.

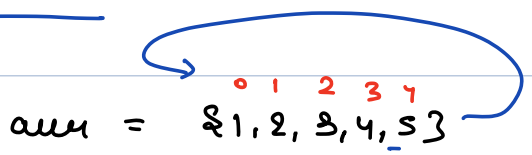
Ques) Given an array, Rotate it right to left K times.

N = 5, arr = {1, 2, 3, 4, 5}

K = 1 → {5, 1, 2, 3, 4}

K = 2 → {4, 5, 1, 2, 3}

← Brute force →



1 time, arr = 5 1 2 3 4 (indices 0 1 2 3 4)

int temp = arr[n-1]; (50)

for(j = n-1; j >= 1; j--) {

arr[j] = arr[j-1];

}

arr[0] = temp;

n = 5

arr = 10 20 30 40 50 (indices 0 1 2 3 4)

50 10 20 30 40

5 * 8 2 * 0

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

T.C → O(n*k)

S.C → O(1)

arr[] = {10, 20, 30, 40, 50} (0 1 2 3 4)

k = 2 = {40, 50, 10, 20, 30} (0 1 2 3 4)

arr[] = {1, 2, 3, 4, 5, 6, 7}

k = 1, {7, 1, 2, 3, 4, 5, 6}

k = 2, {6, 7, 1, 2, 3, 4, 5}

k = 3, {5, 6, 7, 1, 2, 3, 4}

reverse

{7, 6, 5, 4, 3, 2, 1}

↓

{5, 6, 7, 1, 2, 3, 4}

arr[] = {1, 2, 3, 4, 5, 6, 7} k = 2

① reverse whole array,

{7, 6, 5, 4, 3, 2, 1}

② reverse first k elements

{6, 7, 5, 4, 3, 2, 1}

③ reverse rest n-k elements

{6, 7, 1, 2, 3, 4, 5}

function reverseK(int[] arr, int n, int k){

k = k%n;

① reverse whole array
reverse(arr, n, 0, n-1);

② reverse first k elements
reverse(arr, n, 0, k-1);

③ reverse rest elements
reverse(arr, n, k, n-1);

n/2 | k/2 | (n-k)/2 | 3

n - k - k + k

Total iterations = $\frac{n}{2} + \frac{k}{2} + \frac{n-k}{2}$

$\Rightarrow \frac{n+k+n-k}{2} \Rightarrow n$.

T.C → O(n).

S.C → O(1).

arr[] = {1, 2, 3, 4, 5, 6, 7} k = 22

⊙ ↓

{7, 6, 5, 4, 3, 2, 1}

arr[] = {1, 2, 3, 4}

k = 0, {1, 2, 3, 4} k = 4, k = 8

k = 1, {4, 1, 2, 3} k = 5, k = 9

k = 2, {3, 4, 1, 2} k = 6, k = 10

k = 3, {2, 3, 4, 1} k = 7, k = 11

k = 20, k = 9

k = k % n k = k % n

⇒ 0. ⇒ 1.

Dynamic Arrays | Array

↓ size is always increasing.

↓ size is fixed.

Arrays only.

lookups → O(1) time.

Dynamic Array

↳ Array → 10 size.

↳ 10 elements.

↓

11 element.

Behind → 20 size

Insertion

↳ O(1).

Some insertion → O(n).

ArrayList

↓ amortized.

Insertion → O(1).

Next class

Prefix Sum

```
for(i=0; i<2^n; i++) {
    j=i
    while(j>0) {
        j--
    }
}
```

outer	inner	iterations
0	—	1
1	1	2
2	2	3
3		⋮
⋮		
2^n-1		2^n-1

$$\underbrace{1 + 2 + 3 + \cdots + (2^n-1)}_{a \quad\quad b}$$

$$\frac{2^n-1}{2}(1+2^n-1) \qquad b-a+1$$

$$2^n-1-1+1$$

$$2^n-1$$

$$\frac{2^n \times 2^n - 2^n}{2}$$

$$\Rightarrow \frac{2^{2n}-2^n}{2}$$

$$\frac{(2^2)^n - 2^n}{2}$$

$$\frac{4^n-2^n}{2}$$

```
for (i=1; i<=n; i++) {
    for (j=1; j<=n; j=j*i)
        {
        3
        }
}
```

i	j	iteration
1	[1 n]	n
2	[1 n]	n/2
3	[1 n]	n/3
⋮	⋮	
n	[1 n]	1

$$n + \frac{n}{2} + \frac{n}{3} + \cdots 1$$

$O(n \log n)$