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

$a^k = b \Rightarrow k = \log_a b$

$\log_2 32 = 5$

$\log_3 81 = 4 \quad (3^4 = 81)$

$7^3 = 343 \Rightarrow \log_7 343 = 3$

$\log_2 n \longrightarrow O(\log n) < O(\sqrt{n})$

$\log_2 64 = 6$ $\qquad \sqrt{64} = 8$

$\log_2 1024 = 10$ $\qquad \sqrt{1024} = 32$

$\log_2 (2^{20}) = 20$ $\qquad \sqrt{2^{20}} = 1024$

~~$4n^2 + 3n + 1$~~

$\longrightarrow O(n^2)$

$1 <= n <= 10^5$. (1 sec time limit) (1 GHz processor)

$O(n^3) \longrightarrow (10^5)^3 = 10^{15} >> 10^8$ X

$O(n^2 \cdot \log n) \longrightarrow (10^5)^2 * \log(10^5) \approx 10^{10} * 17 = 1.7 * 10^{11} >> 10^8$ X

$O(n^2) \longrightarrow (10^5)^2 = 10^{10} > 10^8$ X

$O(n \cdot \log n) \longrightarrow 10^5 \cdot \log(10^5) = 10^5 \cdot 17 = 1.7 * 10^6 < 10^7$ ✓

$\log_2 10^5 \approx 16.\ldots \approx 17$

$2^{16} \approx 64k$

$2^{17} \approx 128k$

In 1 sec, a program can finish ~ 10^8 operations.

(10^7 to 10^8 iterations)

How to approach any problem?

→ Problem statement, constraints

→ Formulate an idea or logic

→ Verify the correctness of logic

→ Mentally develop an idea of the loops that you use

→ Determine the Time Complexity

→ Assess if it's less than 10^8 iterations for max input size and preferably less than 10^7 iterations. If not, change your approach. If within desired range, code it down.

Space complexity ⟶ Max space utilised at any point of time while running the algorithm.

```
func (int n) {                    // 4 bytes
    int x = 5;                    // 4 bytes
    int y = 10;                   // 4 bytes
    long z = 12;                  // 8 bytes
    ...
}
```

Don't count the I/P and O/P.

16 bytes. ⟶ O(1) S.C.

int w[] = new int[n]; ⟶ SC :- 4bytes * n

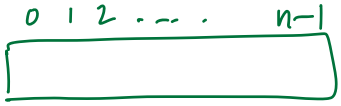
= 4n = O(n)

$n^2 + 3n + 15 = n^2 + 3*n^1 + 15*n^0$

$n^2 \rightarrow O(n^2)$

$f(n) = 34 = 34 * n^0 = O(n^0)$

$= O(1)$

n elements → 

O(n) S.C.

n elements, k at a time → 

O(k) S.C.

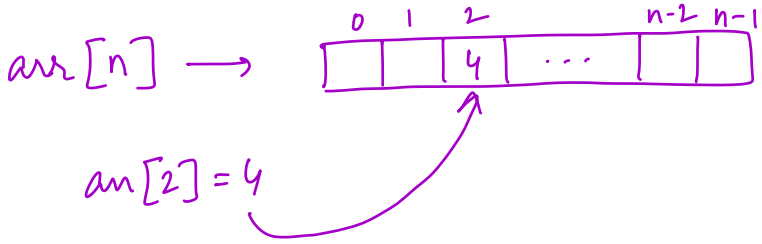
```
fn maxArr ( int arr[], int n){
    max = -10^9
    fn ( i → 0 to n-1) {
        max ← max(arr[i], max)
    }
    return max
}
```

S.C. → O(1).

arr[n] → Space should not counted in your space complexity

Additional space
Computational space
Auxiliary space.

[Break till 10:26 PM]



Print all elements of array :-

```
fn i → 0 to n-1 {
|    print(arr[i])
}
```

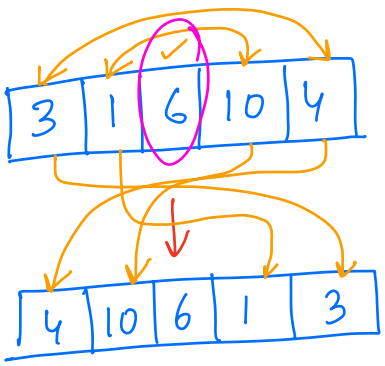
O(n) T.C. , O(1) S.C.

Array → O(1) T.C. of accessing any index.

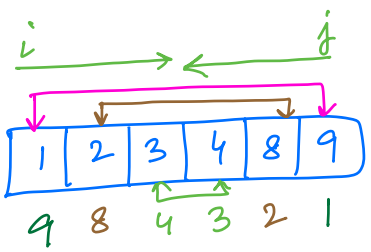
Q1) Given an array arr[n], reverse the array.

I/P → arr = {3, 1, 6, 10, 4}

O/P → arr → {4, 10, 6, 1, 3}



arr[::-1]

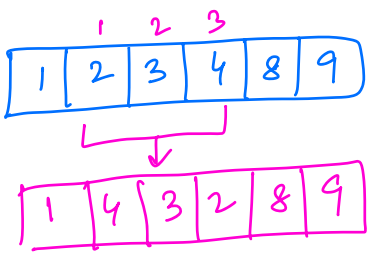


```
i = 0
j = n-1
while( i < j ){
    tmp = arr[i]
    arr[i] = arr[j]
    arr[j] = tmp
    i += 1
    j -= 1
}
```

$\frac{n}{2} \approx O(n)$ T.C.

$O(1)$ S.C.

Q) Reverse the part of the array from index i to index j $[0 \le i \le j \le n-1]$.

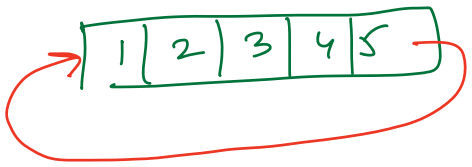


$i = 1 \quad j = 3$

```
fn reverse(arr, i, j, n){
    while( i < j ){
        tmp = arr[i]
        arr[i] = arr[j]
        arr[j] = tmp
        i += 1
        j -= 1
    }
}
```

8) Given arr[n] and an integer k, rotate arr from right to left k times.

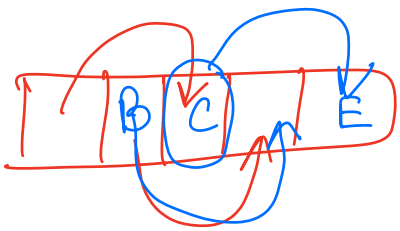
n = 5
arr = {1, 2, 3, 4, 5}
k = 2



5 1 2 3 4

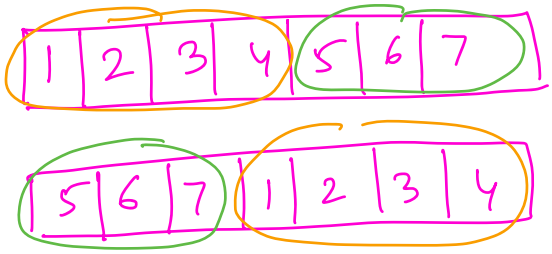
{1, 2, 3, 4, 5}

→ 4 5 1 2 3




```
fn (i → 1 to k) {  // k steps
    tmp = arr[n-1]
    fn (j = n-2 to 0) {
        arr[j+1] = arr[j]
    }                          → (n-1) steps
    arr[0] = tmp
}
```

k * (n-1) = O(n*k) T.C.
O(1) S.C.



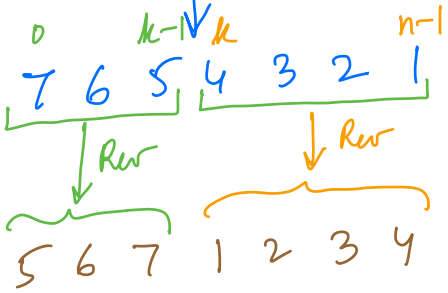
$k = 3$

start ⟶ ⟵ end

arr → 1 2 3 4 5 6 7 $\quad k = 3.$

Reverse

last k elements have come to the first.



$k = k \% n$

reverse(arr, 0, n-1, n)

reverse(arr, 0, k-1, n)

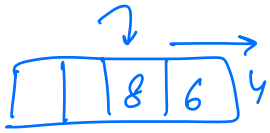
reverse(arr, k, n-1, n)

O(n) T.C.

O(1) S.C

arr = [2, 3, 6, 4]

arr.insert(2, 8) Not good



arr.append(16) ✓