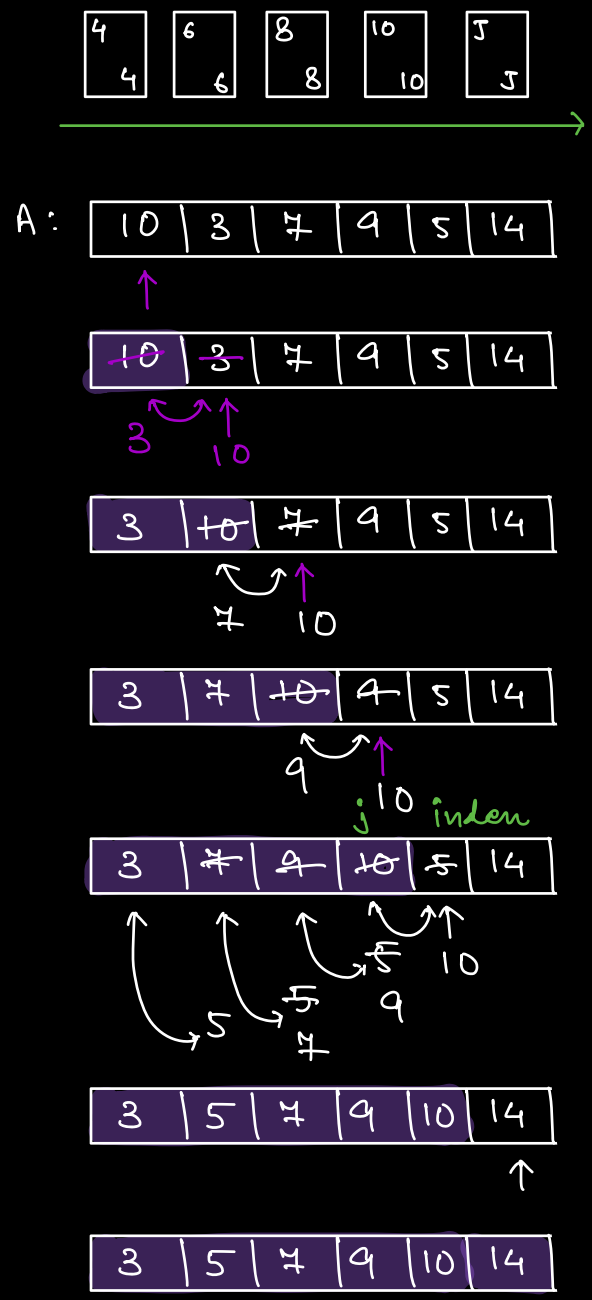
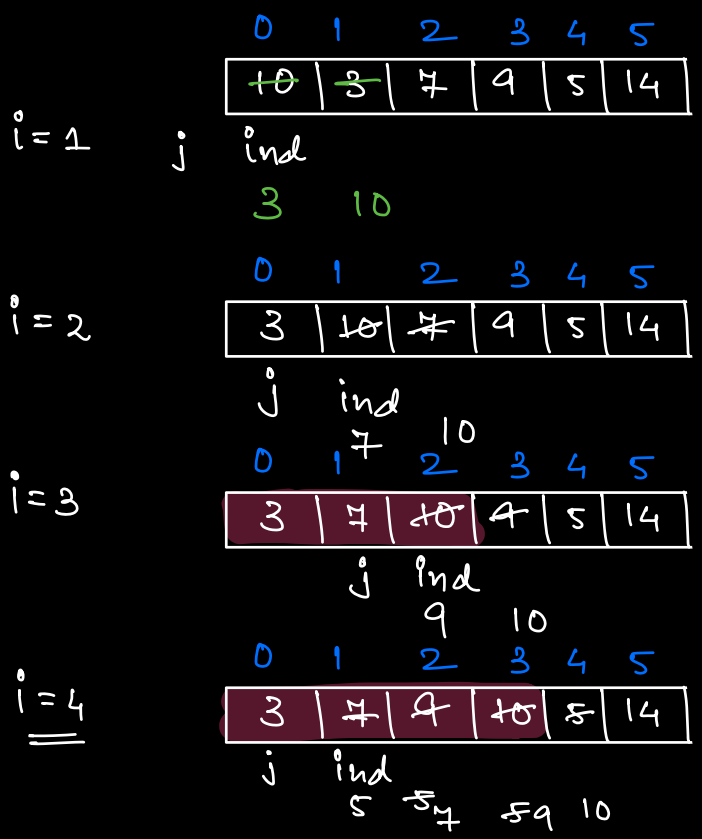
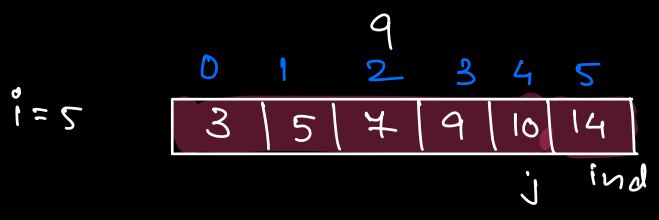


INSERTION : **Inserting** every element at its correct **position**.

```
Void insertionSort( A[], N ) {
    for( i = 1 ; i < N ; i++ ) {
        j = i-1;
        inden = i;
        while( j >= 0 ) {
            if( A[inden] < A[j] ) {
                Swap( A[inden], A[j] );
                inden = j;
                j--;
            }
            else { break; }
        }
    }
}
```





SC : O(1) ⇒ Inplace.

Stable ?

Yes.

A : { 3 2 1 2 5 }

{ 2 3 1 2 5 }

{ 1 2 3 2 5 }

{ 1 2 2 3 5 }

Best Case TC ⇒ O(N)

A : { 1 2 3 4 5 }

# of swaps :-

A : 8 7 5 4 3 2

① ② ③ .... (N-1)

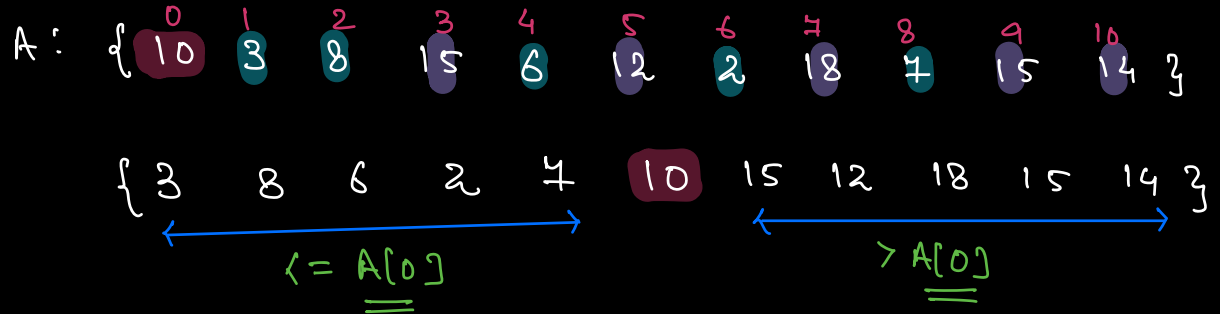
1 + 2 + 3 + ... (N-1)

$$\frac{N(N-1)}{2}$$

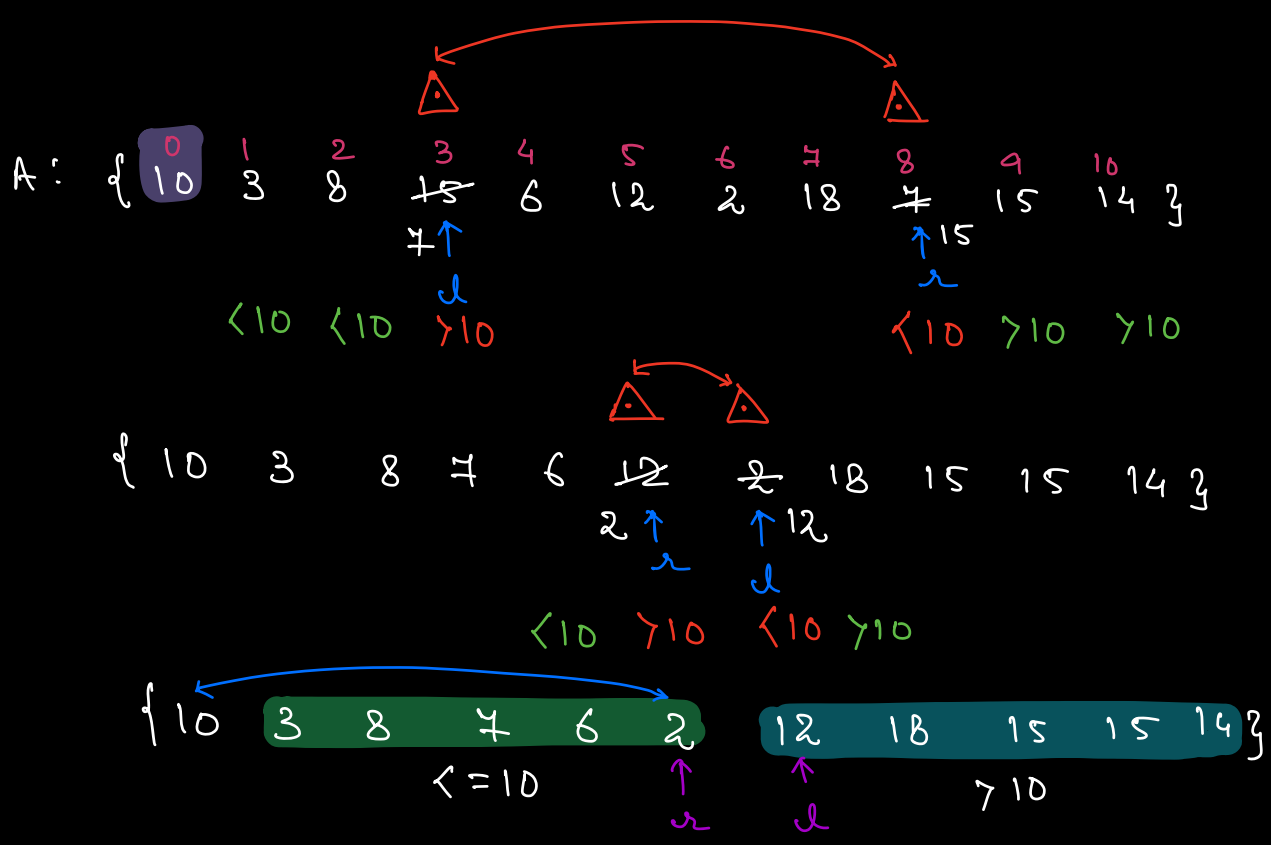
Q. Given an Array of size N, Rearrange the array such that

elements <= A[0] : Go to left

elements > A[0] : Go to right.

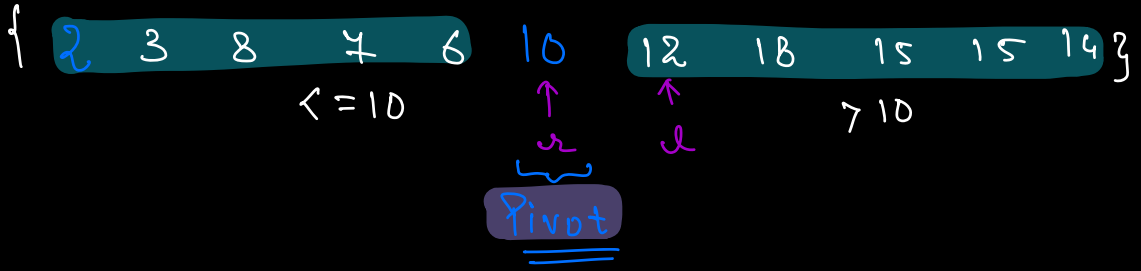


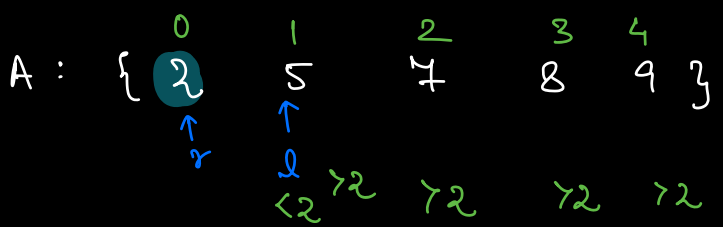
TC : O(N)
SC : O(1)



Swap(A[0], A[l]) ✗

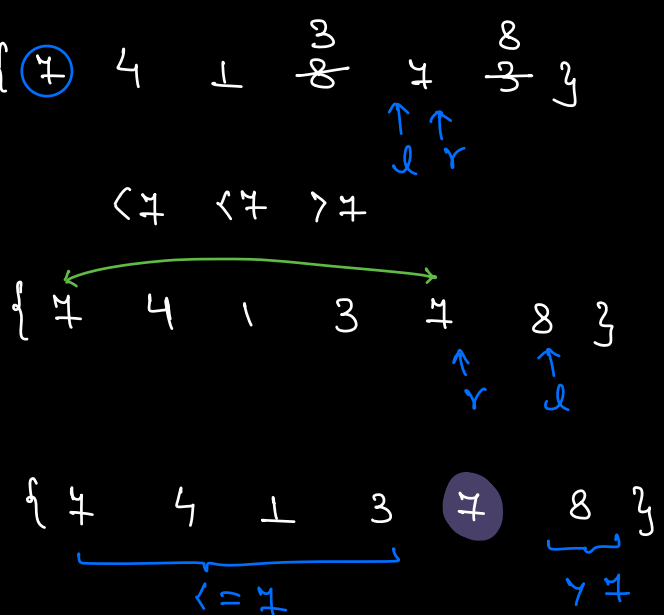
Swap(A[0], A[r]) ✓





```
Void partition( A[], N) {
    l = 1;
    r = N-1;
    while ( l <= r) {
        if( A[l] <= A[0] ) l++;
        else if ( A[r] > A[0] ) r--;
        else {
            swap( A[l], A[r])
            l++
            r--
        }
    }
    swap( A[0], A[r]);
}
```

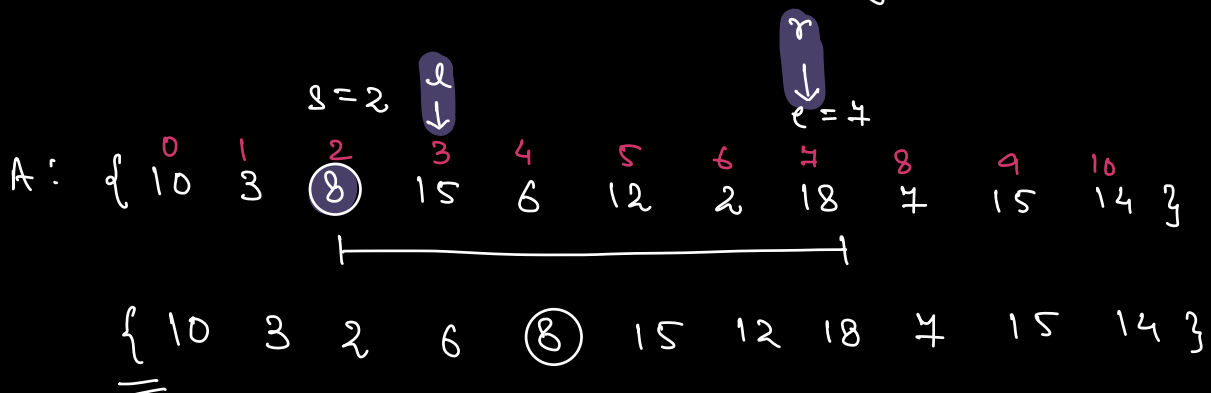
A: { 7 4 1 3 7 8 }



Q. Given an Array of size N, Rearrange the subarray from s to e such that

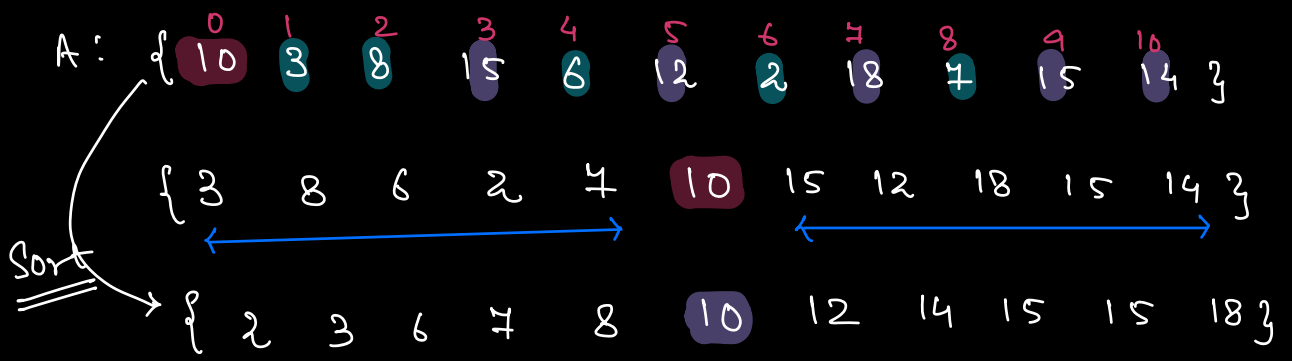
elements <= A[s] : Go to left

elements > A[s] : Go to right.

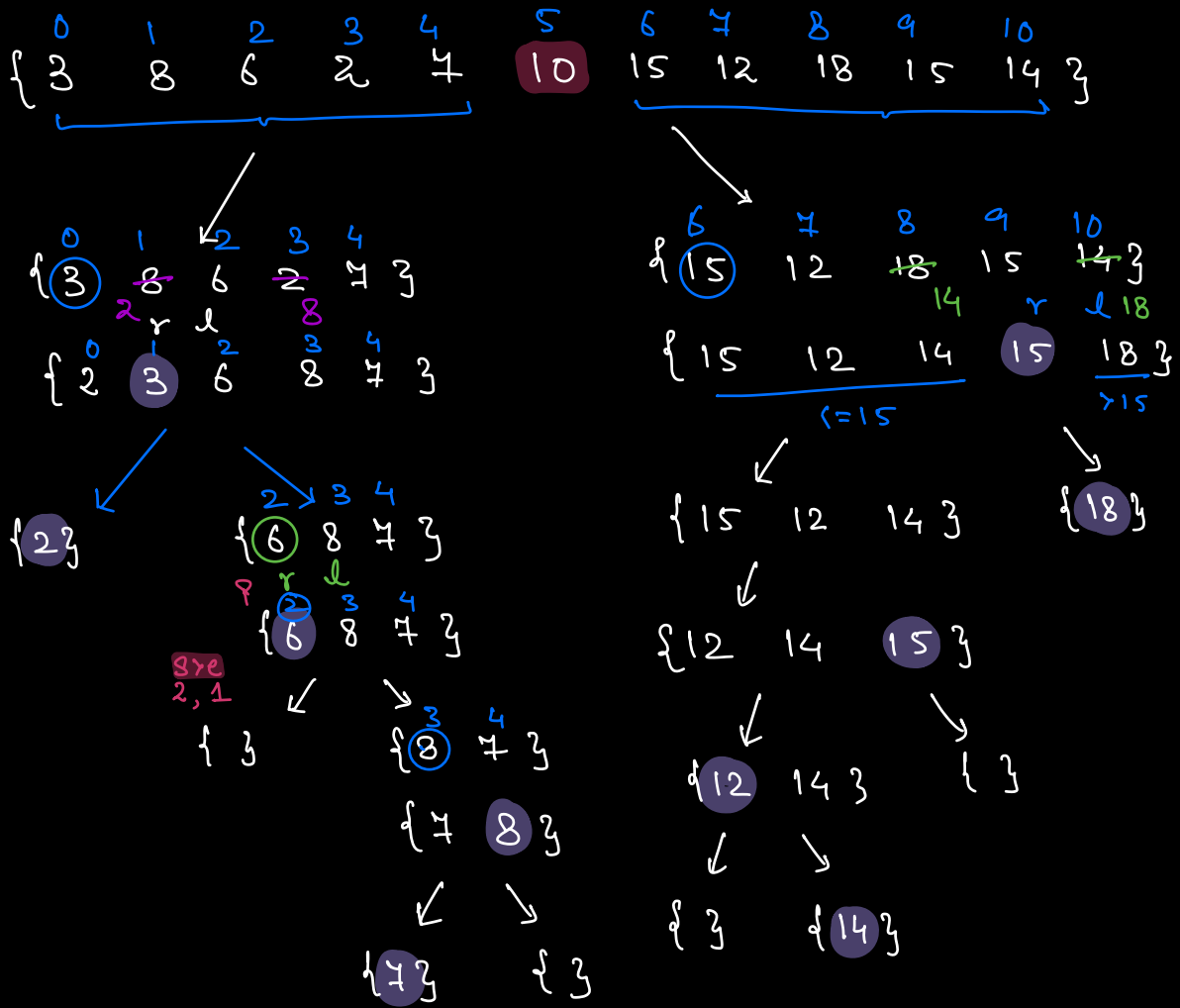


{ 10 3 2 6 8 15 12 18 7 15 14 }

```
Void partition( A[], N) {
    l = s+1;
    r = e;
    while ( l <= r) {
        if( A[l] <= A[s] ) l++;
        else if ( A[r] > A[s] ) r--;
        else {
            swap( A[l], A[r])
            l++
            r--
        }
    }
    swap( A[s], A[r]);
}
```



⇒ After partition, A[0] got it's correct position in sorted order.

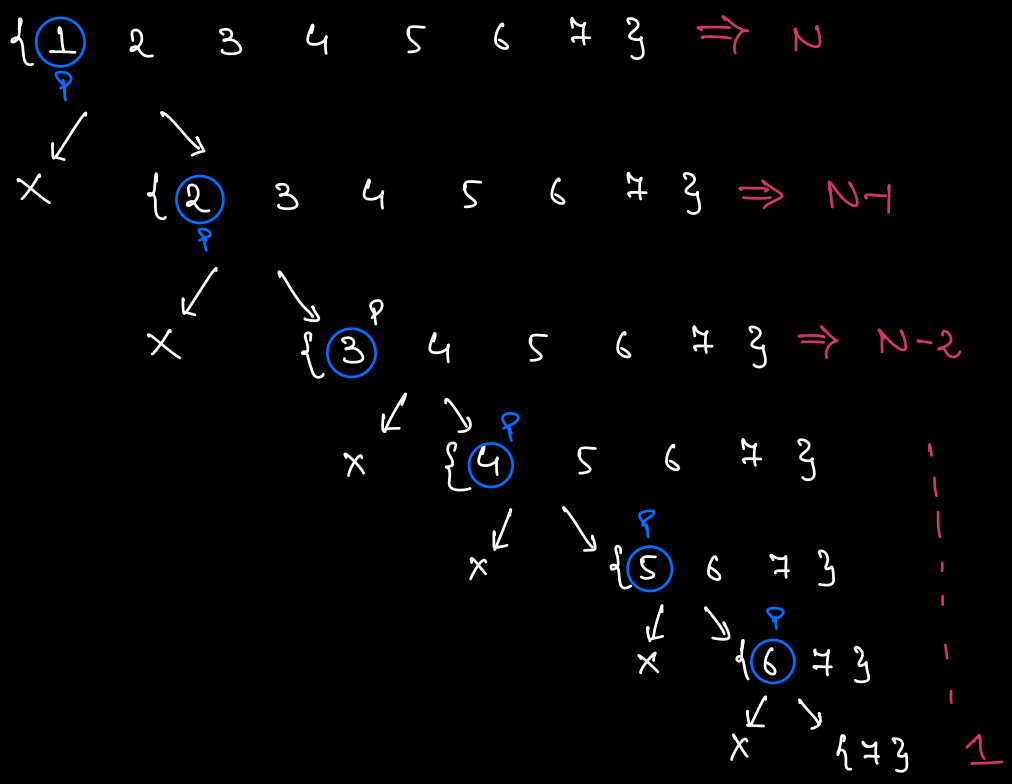


{2, 3, 6, 7, 8, 10, 12, 14, 15, 15, 18}

⇒ QUICK SORT

```
                                  0      N-1
                                  ↓      ↓
Void   quickSort(A[], s, e){
   // Assumption: quickSort(A,s,e) sorts the
   Array from index s to e.
   if(s>=e) return;
(Pivot index)  p = partition(A, s, e);
   quickSort(A, s, p-1);
   quickSort(A, p+1, e);
}
```

Best Case TC :

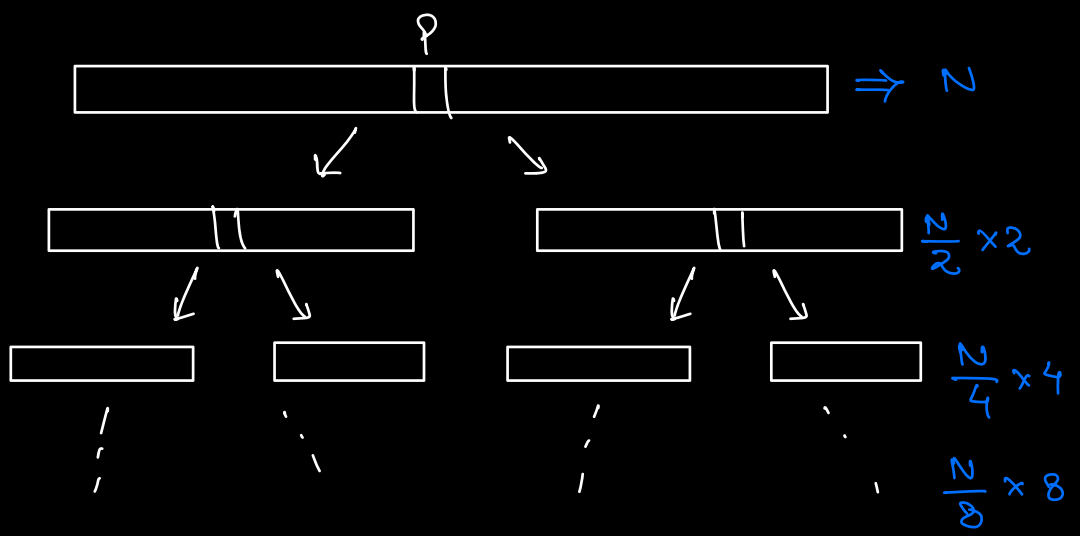


# of iterations

$\Rightarrow 1 + 2 + 3 + \cdots + N$

$\Rightarrow \frac{N(N+1)}{2}$

TC : $O(N^2)$ {Worst Case}



$N \times 1 + \frac{N}{2} \times 2 + \frac{N}{4} \times 4 + \cdots\cdots$

$\log N$

$\Rightarrow N \cdot \log N$

SC : $O(N)$ Worst Case

$O(\log N)$ Best Case.

⇒ Worst case Q.S : If the partition is happening across MIN element.

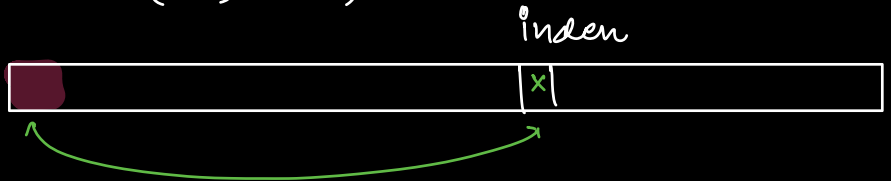
Probabibility of getting min element

$$\Rightarrow \frac{1}{N} * \frac{1}{N-1} * \frac{1}{N-2} * \cdots\cdots$$

≈ 0

⇒ Practically impossible.

⇒ rand(0, N-1)



⇒ Randomized Quick Sort

```
randomIndex = rand(s,e)
Swap(A[s], A[randomIndex]);
p = partition(A, s, e)
QS(A, s, p-1)
QS(A, p+1, e)
```

TC :

Avg Case ⇒ $O(N \log N)$

Best Case ⇒ $O(N \log N)$

Worst Case ⇒ $O(N^2)$ {Chances are negligible}

SC : $O(\log N)$ (Avg)

QS ⇒ Unstable.

Inplace ? ✓ ⇒ Because the Space is being used by recursive stack.

⇒ Comparison of Sorting Algorithms:

	Best	Worst	Avg
1) Selection	$O(N^2)$	$O(N^2)$	$O(N^2)$
2) Bubble	$O(N)$	$O(N^2)$	$O(N^2)$
3) Merge	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
4) Insertion	$O(N)$	$O(N^2)$	$O(N^2)$
5) Quick sort	$O(N \log N)$	$O(N^2)$	$O(N \log N)$

HW Compare **Stability** of all the above algo's.

Count sort ⇒ Intermedia Strings.

↳ Count the freq. of elements and store it **somewhere.**

$A[10^6] \Rightarrow 1 < A[i] < 100$

HW Explore your libarary sort method.