① Pseudo-code for insertion-sort algorithm

```
input: array -> arr
def insertionsort (arr)
    for i in range(0, length of (arr))
        put j = i
        While j>0 and arr[j-1] > arr j
        ~~swap~~ swap arr[j-1] and arr[j]
            j = j-1
```

Following the above algorithm we get

		Unsatisfied condition
pass 1:	< 31, 41, 59, 26, 41, 58 >	j = 0
pass 2:	< 31, 41, 59, 26, 41, 58 >	arr[j-1] > arr[j]
pass 3:	< 31, 41, 59, 26, 41, 58 >	arr[j-1] > arr[j]
pass 4:	< 26, 31, 41, 59, 41, 58 >	
pass 5:	< 26, 31, 41, 41, 59, 58 >	
pass 6:	< 26, 31, 41, 41, 58, 59 >	

In pass 4 we shift: 31, 41, 59, 26, 41, 58

In pass 5 we shift: 26, 31, 41, 59, 41, 58

In pass 6 we shift: 26, 31, 41, 41, 59, 58

Final output: < 26, 31, 41, 41, 58, 59 >

(2) Given the algorithm :-

Input A of N numbers :-

unknown (A)

for j = 1 to N-1

    if (A[N] < A[j])

        swap A[j] and A[N]

output A[N].

$\left.\begin{array}{l}\text{for } j=1 \text{ to } N-1 \\ \quad \left.\begin{array}{l}\text{if } (A[N] < A[j]) \\ \quad \text{swap } A[j] \text{ and } A[N]\end{array}\right\} O(1)\end{array}\right\} O(n)$

(1) The output of this algorithm is always going to be ~~a~~ a ~~sorted~~ array so A[N] will always give the largest element in the array since the largest element is always swapped to the right of the array, leaving the last element as the largest element.

(2) The time complexity of this algorithm is O(n)

③ let there be K arrays each having N elements
Input:- array 1 = [~~2~~ 1 ___ N]
array 2 = [1 ___ N]
,
,
array k = [1 ___ N]

define function count_func (~~1~~ array)
count = 0
for i in Range (array length)
count ++ ;
print count

$\}$ O(n)

count_func (array 1), count_func (array 2) ___ count_fun (array k)

$\}$ O(K)

Total time complexity = O(KN).

The above algorithm can also be written as:

for i in range (0 to k) :
for j in range (0 to N)
count ++ ;
print (count)

$\}$ O(N) $\}$ O(K)

Total time complexity = O(K*N)
= O(KN)

4

Consider the following Algorithm.

```
input : array <- arr , step = 0 , counter = 0

for i 0 to N-2
{
    if (arr[i] - i == step)
    {
        // do nothing
    }
    else
    {
        print ("Missing : " + i);
        counter++;
        step = arr[i] - i
        if (counter == 2)
        {
            exit()
        }
    }
}
```

O(N) time complexity

(5) Pseudo-code for finding inversions

```
for i in range (array_length)
    for j in range (i, array_length)
        if i<j and arr[i] > arr[j]
            print (i, j)
```

(1) lets follow the above algorithm and get the inversions of <2, 3, 8, 6, 1>

pass 1: (0, 4)
pass 2: (1, 4)
pass 3: (2, 3)
pass 4: (2, 4)
pass 5 (3, 4)

They all ~~as~~ satisfy the inversion conditions.

(2) An array will ~~al~~ always have the most inversions when they are sorted in a descending order ie-- starting from maximum and going to minimum so the set ~~to~~ {n, n-1, n-2 -- 1} will have the most inversions

(3) The more inversions more is the runtime of selection sort algorithm as it will have to get each element from n to 1; n-1 to 2, n-2 to 3 etc. This will become the worst case run-time of algorithm $O(n^2)$. The number of operations for shifting each element will be 'n' for the ~~worst~~ first iteration. The total would be $n+n-1+n-2$ --- till 1. This adds upto $O(n^2)$

(6) Consider the following pseudo-code for the given problem:-

Consider the following Algorithm:-
input: ~~arry~~ 2 sorted ~~arrays~~ arr1, arr2.

```
define small(start1, end1, start2, end2, k)
{
    if (start1 == end1)  return arr2[k]
    if (start2 == end2]  return arr1[k]
    else
    {
        mid1 = (start1 + end1) / 2
        mid2 = (start2 + end2) / 2
        if (Mid1 + Mid2 < k)
        {
            if (arr1[Mid1] > arr2[Mid2]))
            {
                return small(Start1 * end1,
                Start2+Mid2+1, end2, k - Mid2 - 1)
            }
            else
            {
                return small(start1+Mid1+1, end1, start2,
                end2, k - Mid1 - 1)
            }
        }
        else {
            if(arr1[Mid1] > arr2[mid2]{
                return small(start1, start1+Mid, start2, end2, k)
            }
```

FOR EDUCATIONAL USE

```
else
{
    return small ( start1, end1, start2, start2 + mid2, k )
}

}

}
```

Total time complexity => $O(\log m + \log n)$
where m => total No. of elements in sorted array arr1
n => total No. of elements in sorted array arr2

(7) Consider the following pseudo-code for the given problem statement:-

```
input: array → arr of length 'n'
def function_permute(arr, n)

    for i in range (1, n)
        j = pick any random no. between 0 and i

        swap (arr[i], arr[j])
return arr
```

- Here the probability of any element going to any other position would be:-

= probability that element was not picked previously × probability that element is picked now

$$= \left(1 - \frac{1}{n}\right) \times \left(\frac{1}{n-1}\right)$$

$$= \frac{n-1}{n} \times \frac{1}{n-1} = \frac{1}{n}$$ -- this holds true for all the positions

hence its of equal probability.