Insertion Sort Algorithm:

→ Consider the given array below:

0	1	2	3	4
7	2	91	77	3

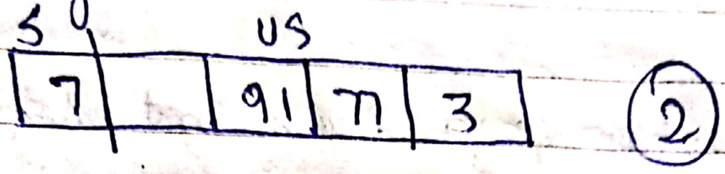
REMEMBER: An array of ~~s~~ a single element is always sorted. So, what we have now is an array of length 5 with a subarray of length 1 already sorted.

S		US		
7		91	77	3

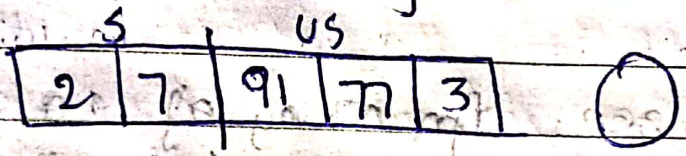
②

→ Moving from the left to the right, we will pluck the first element from the unsorted part, and ~~s~~ insert it in the sorted subarray. This way at each insertion, our sorted subarray length would ↑se by 1 and unsorted subarray length decreases by 1. Let's call each of these insertions and the traversal

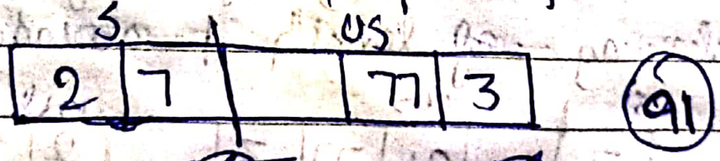
of the sorted subarray to find the best position, a pass.
So, let's start with pass 1, which is to insert 2 in the sorted array of length 1.

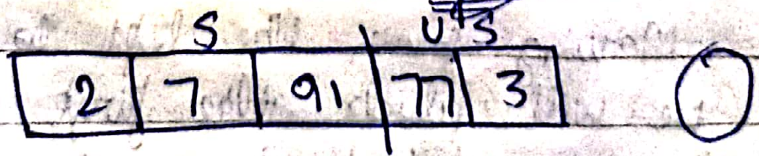


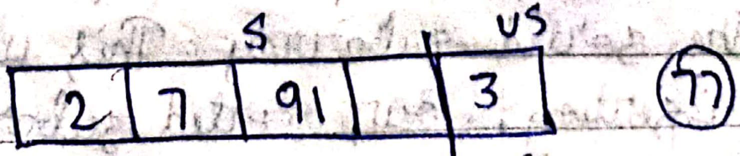
So, we plucked the first element from the unsorted part. Let's insert element 2 at its correct position, which is before 7. And this ↑ses the size of our sorted array.

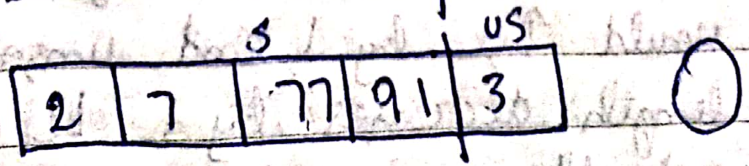


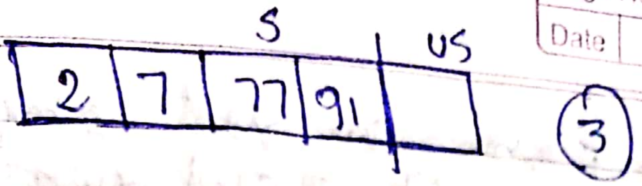
→ Repeat the same process:

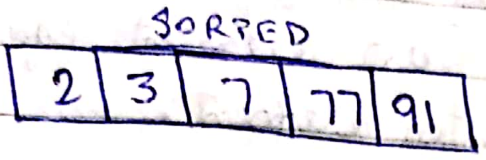












* Analysis:
Conclusively, we had to have 4 passes to sort an array of length 5. And in the first pass, we had to compare the to-be inserted element with just one single element 7. So, only one comparison, and one possible swap. Similarly for ith pass, we would have i number of comparisons, and i possible swaps.

1) Time Complexity of Insertion Sort Algorithm
→ Let's now calculate the time ~~co~~ complexity of the algorithm. We made 4 passes for this array of length 5, and for ith pass, we mad i number of comparisons. So, the total number of comparisons is 1+2+3+4. Similarly, for an array of length n, the total number of comparison / possible swaps would be $1+2+3+4+ \dots +(n-1)$ which is $n(n-1)/2$, which ultimately is $O(n^2)$.

2) Insertion sort algorithm is a stable

Date

algorithm, since we start comparing from the back of the sorted subarray, and never cross an element equal to the to be inserted element.

3) Insertion sort algorithm is an adaptive algorithm. When our array is already sorted, we just make (n-1) passes, and don't make any actual comparison b/w the elements. Hence, we accomplish the job in O(n).

Note: At each pass, we get a sorted subarray at the left, but this intermediate state of the array has no real significance, unlike the bubble sort algorithm where at each pass, we get the largest element having its position fixed at the end.

Scanned with CamScanner