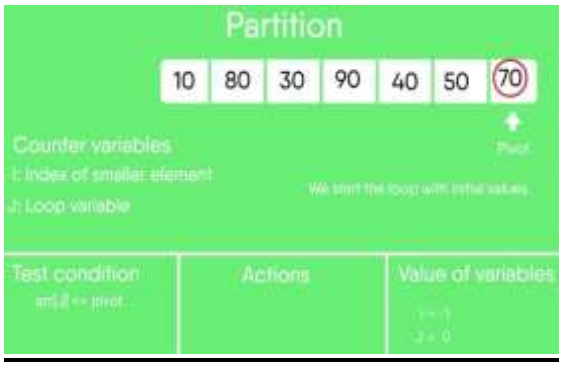
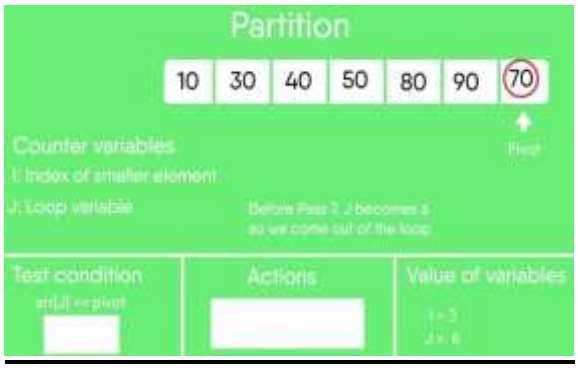
PRACTICE GUIDELINES

(WEEK 2)

- **Practice 1:**

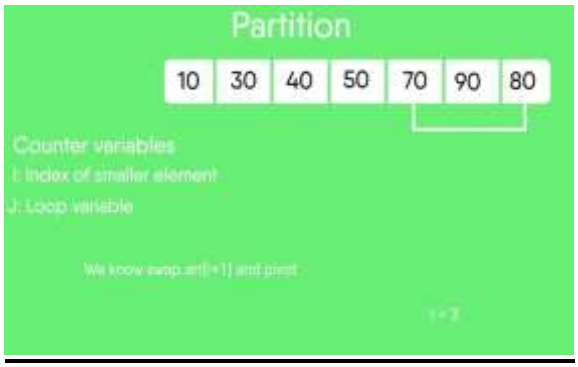


1

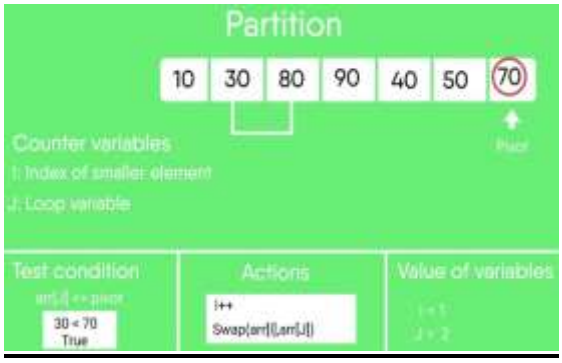


5

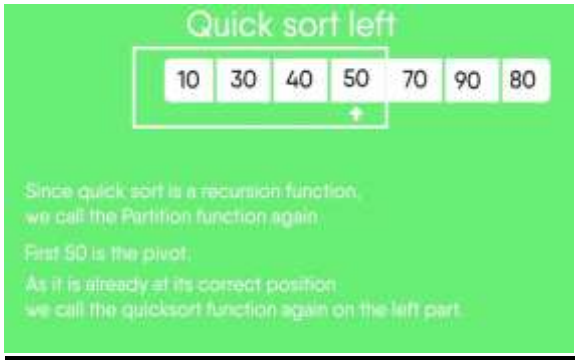
2



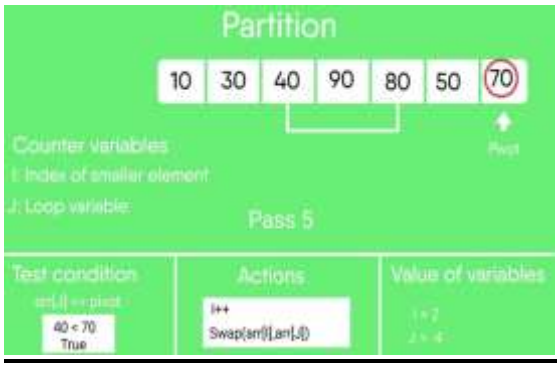
6



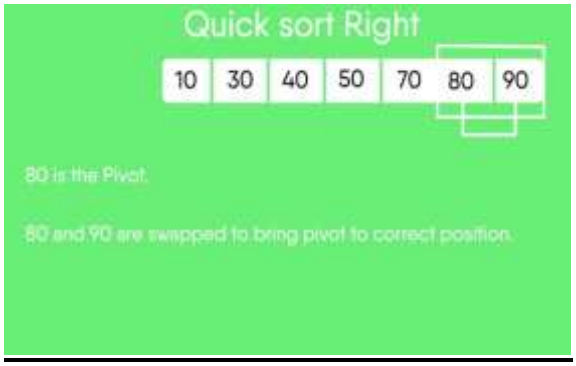
3



7



4



8

- **Practice 3:**

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide the array into two halves:
   middle m = (l+r)/2
2. Call mergeSort for first half:
   Call mergeSort(arr, l, m)
3. Call mergeSort for second half:
   Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
   Call merge(arr, l, m, r)

- **Practice 4:**

For simplicity, consider the data in the range 0 to 9.

Input data: 1, 4, 1, 2, 7, 5, 2

1. Take a count array to store the count of each unique object.
   Index: 0 1 2 3 4 5 6 7 8 9
   Count: 0 2 2 0 1 1 0 1 0 0
2. Modify the count array such that each element at each index stores the sum of previous counts.

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 4 4 5 6 6 7 7 7

The modified count array indicates the position of each object in the output sequence.

3. Output each object from the input sequence followed by decreasing its count by 1.

   Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.

   Put data 1 at index 2 in output. Decrease count by 1 to place next data 1 at an index 1 smaller than this index.

- **Practice 5:**

The problem with the previous counting sort was that we could not sort the elements if we have negative numbers in it. Because there are no negative array indices. So what we do is, we find the minimum element and we will store count of that minimum element at zero index.

Hint: Using *vector<int>* to store the array.

**max_element(arr.begin(), arr.end())* to get the index of max element

**min_element(arr.begin(), arr.end())* to get the index of min element

- **Practice 6:**

A **simple solution** *is to create an output array of size n*k and one by one copy all arrays to it. Finally, sort the output array using any O(n Log n) sorting algorithm. This approach takes O(nk Log nk) time.*

One efficient solution *is to first merge arrays into groups of 2. After first merging, we have k/2 arrays. We again merge arrays in groups, now we have k/4 arrays. We keep doing it unit we have one array left. The time complexity of this solution would O(nk Log k). How? Every merging in first iteration would take 2n time (merging two arrays of size n). Since there are total k/2 merging, total time in*

first iteration would be O(nk). Next iteration would also take O(nk). There will be total O(Log k) iterations, hence time complexity is O(nk Log k)

Another efficient solution *is to use Min Heap. This Min Heap based solution has same time complexity which is O(nk Log k). But for different sized arrays, this solution works much better.*

Following is detailed algorithm.

1. Create an output array of size n*k.

2. Create a min heap of size k and insert 1st element in all the arrays into the heap

3. Repeat following steps n*k times.

a) Get minimum element from heap (minimum is always at root) and store it in output array.

b) Replace heap root with next element from the array from which the element is extracted. If the array doesn't have any more elements, then replace root with infinite. After replacing the root, heapify the tree.

Time Complexity: The main step is 3rd step, the loop runs n*k times. In every iteration of loop, we call heapify which takes O(Logk) time. Therefore, the time complexity is O(nk Logk).