Inversion Count and Selection Algorithms

Assignment Solutions

PW SKILLS

Q1. Given an integer array, find the kth largest element using the quick select algorithm.
input 1 : arr[] = {1,3,2,4,5,6,7} , k = 3
output 1 : 5

input 2: arr[] = {4,3,3,2,1} , k = 4
output 2: 2

Solution:

code : **ASS_Code1.java**
Output :

```
K-th largest element in array : 10
```

Approach :
How Does Quickselect Work?
Quickselect works identical to quicksort in that we:

- Pick a pivot
- Partition the data into two where:
- Numbers less than the pivot go to the left
- Numbers greater than the pivot go to the right

However, instead of recursing into both sides as in Quicksort, quickselect only recurs into one side; whichever one would have our kth largest element.
The main thing to note here is that our pivot at any given partition will always end up at the correct index. Therefore, we just need to check:

If our pivot is at our "kth largest" index, return the number at that index.
If our pivot comes before the "kth largest" index, perform quickselect on the right partition.
If our pivot comes after the "kth largest" index, perform quickselect on the left partition.
Performing quickselect only on one partition reduces our average-case complexity from O(nlogn) to O(n).

Note: quickselect (and quicksort) have a worst-case of O(n^2). This would occur whenever a pivot we choose is an extreme; the smallest or largest element.
In practice, however, quickselect and quicksort are incredibly fast algorithms and the worst-case occurrence is quite unnoticeable on large sets of data.

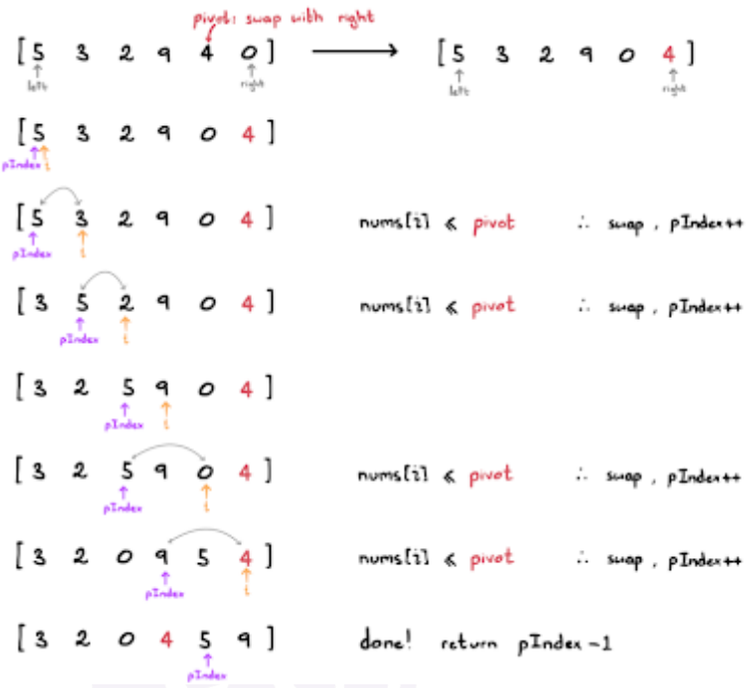
That's basically it regarding how it works!
How Would This Work In Code?
Our algorithm is quite simple:

```
function quickSelect(nums, left, right, k)
   if left = right
       return nums[left]   // base case

   pivotIndex = random element between left and right
   pivotIndex = partition(nums, left, right, pivotIndex)

   if k = pivotIndex
      return nums[k]
   else if k < pivotIndex
      return quickselect with: right = pivotIndex - 1
   else
      return quickselect with: left = pivotIndex + 1
```

Keep in mind that k above represents the kth smallest element; not the largest. We handle this in our main function as aforementioned.
All we need now are a couple of helper functions, namely partition() and swap().

Partition Function:
In our partition function, all we need to do is:

Swap our pivot and the right-most element
Move each element to less than the pivot to the left partition.
It can be a bit confusing to understand how swaps are made so let me try and explain it visually. Let the pivot here be 4.



As you can see, after partitioning, our pivot is at the exact index it's supposed to be in a sorted array. Not only that but all the elements less than 4 are to the left of it and all the elements to the right are greater than it. In Java, we have to use a custom swap function to swap two elements in an array.

Note: the reason to use a random pivot is to minimize the potential that our algorithm hits the worst-case time. In other words, with random pivoting, our algorithm's expected performance is equally good on all datasets.