Selection Algorithms

Main Focus: The Partitioning Method

---

k-smallest elements in an array S of n elements using the partitioning method

Brute Force Method:

We scan through the numbers k times to have the desired element. This method is the one used in bubble sort and selections sort. We try to find out the smallest element in the entire sequence by comparing every element.

Since, the sequence has to to be traversed k times, we get the complexity to be $O(n \times k)$.

Sorting Approach:

Based on the previous file, we know very well we can work on improving the time complexity by just using sorting methods. So, we essentially-

1. Sort the numbers.
2. Pick the first k numbers.

The time complexity we get will be $O(nlogn + k) = O(nlogn)$.

So, we now try to implement **tree sorting**. So, we essentially-

1. Insert all elements in a binary search tree.
2. Do an InOrder traversal and print k elements which are the smallest ones.

This gives us the same time complexity as the sorting method above i.e. $O(nlogn)$.

However, this has a major **disadvantage**. If we sort the numbers in descending order, we will get a tree which is skewed completely to the left. In such a situation, the constructed tree will be:

$0 + 1 + 2 + \ldots + (n-1) = n(n-1)/2$

which is $O(n^2)$.

Now, before we move on to paritioning, we discuss a simple solution to fix the above problem of the tree sorting method.

Resolution Solution: We just use a *smaller tree* to give the same result. Simple as it sounds, we essentially-

1. Take the first k elements of the sequence to create a balanced tree of k nodes. [Cost = $klogk$]
2. Now, we take the remaining numbers. We take them one by one and-

   a. If the number is larger than the largest element of the tree, return.
   b. If the number is smaller than the largest element of the tree, remove the largest element of the tree and add the new element. This step is to ensure that the smaller element always replaces a larger element from the tree.

The cost of the above operation is $logk$ since the tree is a balanced tree of k elements.

The time complexity is a bit complicated to calculate. But it is:

1. For the first k elements,we make the tree with cost $klogk$.
2. For the rest $n-k$ elements, the complexity is $O(logk)$.

Total Cost: $klogk+(n-k)logk=nlogk$. That is $O(nlogk)$. This is significantly better than the ones above.

Partitioning Technique:

The Algorithm:

1. Choose a pivot from the array.
2. We Partition the array such that: A[low...pivotpoint-1] $\leq$ pivotpoint $\leq$ A[pivotpoint+1...high].
3. If $k<pivotpoint$ then it must be on the left side of the pivot. Recursively do the same method on the left.
4. If $k=pivotpoint$ then it must be the pivot and print all elements from low to $pivotpoint$.
5. If $k>pivotpoint$ then it must be on the right side of the pivot. We then do the same recursion process on the right part.

```
int Selection(int low, int high, int k){
    int pivotpoint;
    if(low==high)
        return S[low];
    else{
        pivotpoint = Partition(low,high);
        if(k==pivotpoint)
            return S[pivotpoint];
        else if(k<pivotpoint)
            return Selection(low,pivotpoint-1,k);
        else
            return Selection(pivotpoint+1,high,k);
    }
}
```

Time Complexity: $O(n^2)$ in worst case which is same as QuickSort. This method performs better than on the average case: $O(nlogk)$.
