**CS 531 Homework3**

**Question1:**

a) Please write an algorithm to find the kth largest element in an unsorted array.

Logic: To find the kth largest element in an unsorted array, we can merge_sort the array first in increasing order and return Array[len(array)- k] (Assume Array starts from 0)

Here is the pseudo code:
First we construct merge sort:(breifly)

```
merge_sort(array,){

# Divide the array into halves: array1 and array2
# Recursively call merge sort on both sides :
merge_sort(array1)
merge_sort(array2)
# finally merge array1 and array2 together  where merge is a function that combine two arrays in sorted order.
merge(array1,array2)
}
```

The recurrence equation for this algorithm is:
$T(n) = 2(T(n/2)) + n$
As a result the time complexity is nlogn.

Apply merge sort in this question:

```
kth_largest(array){


sort_array = merge_sort(array)


return sort_array[len(array)-k]
}
```

For the sort algorithm, we can use the sort algorithms in class such as merge_sort.
Take merge_sort as an example, the time complexity is O(nlogn)
As a result, the running time for this algorithm is O(nlogn), since the time take to sort the array is O(nlogn) by merge sort and the time to access the Kth smallest element in the sorted array is O(1)

b) Can you optimize your algorithm to have expected runtime of O(n)?
If we want to optimize the algorithm above, we want to use something similar to randomized QuickSort. However, the optimized algorithm is slightly different from randomized QuickSort. Below is the pseudo code of this algorithm with explanations:

Example input : [1, 5, 7, 8, 2, 10, 11]
Suppose we want to find the second largest element:

1.Choose a random pivot from the array and perform partitions. (This can be done by modify the partition method in quick sort which takes the last element as pivot) Here, partitions means put everything **bigger** to the left side of the pivot and put everything **smaller** to the right side of the pivot. Then record the location of the pivot.

Suppose the pivot randomly picked is 7, the outcome of partition the step is:
[8, 10, 11, 7, 1, 5, 2]
Then, record the location of the pivot which is position=4 which means 7 is the fourth largest element in the list.

2. Now, we want to decide which side we want to do the partition again.
If position ==k:
    Return ## we are done since we have already found the solution

If position >k:
    Run the algorithm again on the left side of the array and update new k
If position <k:
    Run the algorithm again on the right side of the array and update new k

Since we want to find the second largest element, it is obvious that we have to run the algorithm on the left side of the pivot again. We calculate 4 - 2 = 2, then we find the second largest element on [ 8, 10 , 11]

3. Finally we recursively run step1 and step 2 and the final result will be achieved. find the kth largest element in an unsorted array.

This algorithm is almost the same as the randomized algorithm we learned in class to find the kth smallest element in the array. Follow the proof from the lecture, we know that the expected running time of this algorithm is O(n). The proof is on next page.

Following the proof from lecture notes:

$T(n)$ = the random variable for running the above algorithm.

For $k = 0, 1, \ldots, n-1$, define the indicator random variable.

$X_k = \begin{cases} 1 & \text{if partition generates a } k : n-k-1 \text{ split} \\ 0 & \text{otherwise.} \end{cases}$

$$T(n) = \begin{cases} T(\max\{0, n-1\}) + \Theta(n) & \text{if } 0 : n-1 \text{ split.} \\ T(\max\{1, n-2\}) + \Theta(n) & \text{if } 1 : n-2 \text{ split.} \\ \vdots \\ T(\max\{n-1, 0\}) + \Theta(n) & \text{if } n-1 : 0 \text{ split.} \end{cases}$$

To obtain the upper bound, we assume the $i$th element always falls in the larger side of the partition.

$$T(n) = \sum_{k=0}^{n-1} X_k \left( T(\max\{k, n-k-1\}) + \Theta(n) \right).$$

Take expectations of both sides.

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k \left(T(\max\{k, n-k-1\}) + \Theta(n)\right)\right].$$

$$= \sum_{k=0}^{n-1} E\left[X_k \left(T(\max\{k, n-k-1\}) + \Theta(n)\right)\right].$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E\left[T(\max\{k, n-k-1\}) + \Theta(n)\right]$$

$$= \frac{1}{n} \sum_{k=0}^{n-1} E\left[T(\max\{k, n-k-1\})\right] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n) \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + \Theta(n).$$

We use fact $\sum_{k=\lfloor n/2 \rfloor}^{n-1} k \leq \frac{3}{8}n^2 \Rightarrow E[T(n)] \leq \frac{2c}{n}\left(\frac{3}{8}n^2\right) + \Theta(n) = cn - \left(\frac{cn}{4} - \Theta(n)\right)$

from lecture notes

$\leq cn$ if $c$ is large enough.

As a result, the expectation of the algorithm is $O(n)$.