Lab 6

Problem 1:
Show all steps of QuickSort in sorting the array [1, 6, 2, 4, 3, 5]. Use leftmost values as pivots at each step.

Answer:
Array {1, 6, 2, 4, 3, 5}
Pivot = 1, left = 0, right = 5
Start i = 1; j = 5
Start scanning from right to left to find smaller than pivot=1, i = 1, j starts from 5 and decreases to 1

i	array[i]	j	array[j]	pivot	smaller than pivot
1	6	5	5	1	FALSE
1	6	4	3	1	FALSE
1	6	3	4	1	FALSE
1	6	2	2	1	FALSE
1	6	1	6	1	FALSE

Swap left_index = 0 with right_index = 0
New array = {1, 6, 2, 4, 3, 5}

Sort partition (left_index = 0, right_index = -1) → stop

Sort partition (left_index = 1, right_index = 5)
Start scanning from left to right to find a greater than pivot=6, i starts from 2, increases to 5

i	array[i]	j	array[j]	pivot	greater than pivot
2	2	5	5	6	FALSE
3	4	5	5	6	FALSE
4	3	5	5	6	FALSE
5	5	5	5	6	FALSE

Swap left_index = 1, right_index = 5
New array = {1, 5, 2, 4, 3, 6}

Sort partition (left_index = 1, right_index = 4)
Start scanning from left to right to find a greater than pivot=5, i starts from 2, increases to 4

i	array[i]	J	array[j]	pivot	greater than pivot
2	2	4	3	5	FALSE
3	4	4	3	5	FALSE
4	3	4	3	5	FALSE

Swap left_index = 1, right_index = 4
New array = {1, 3, 2, 4, 5, 6}

Sort partition (left_index = 1, right_index = 3)
Start scanning from left to right to find a greater than pivot=3, i starts from 2, increases to 3

i	array[i]	J	array[j]	pivot	greater than pivot
2	2	3	4	3	FALSE
3	4	3	4	3	TRUE

Swap left_index = 1, right_index = 2
New array = {1, 2, 3, 4, 5, 6}

Sort partition (left_index = 1, right_index = 1) → Stop
Sort partition (left_index = 3, right_index = 3) → Stop
Sort partition (left_index = 5, right_index = 4) → Stop
Sort partition (left_index = 6, right_index = 5) → Stop

Sorted Array = {1, 2, 3, 4, 5, 6}

Problem 2:
In our average case analysis of QuickSort, we defined a good self-call to be one in which the pivot x is chosen so that number of elements < x is less than 3n/4, and also the number of elements > x is less than 3n/4. We call an x with these properties a good pivot. When n is a power of 2, it is not hard to see that at least half of the elements in an n-element array could be used as a good pivot (exactly half if there are no duplicates). For this exercise, you will verify this property for the array A = [5, 1, 4, 3, 6, 2, 7, 1, 3] (here, n = 9). Note: For this analysis, use the version of QuickSort in which partitioning produces 3 subsequences L, E, R of the input sequence S.

a. Which x in A are good pivots? In other words, which values x in A satisfy:
    i. the number of elements < x is less than 3n/4, and also
    ii. the number of elements > x is less than 3n/4
b. b. Is it true that at least half the elements of A are good pivots?

Answer:
A good pivot should be x = 3.
The condition to satisfy lets us know that the array should be partitioned into 3 subsequences, which is likely {L=n/4, E=n/2, G=n/4}, and the pivot should be in the E subsequence.
A good pivot is defined which has neither partition less than n/4 or greater than 3n/4. That means there's 50% chance of having a good partition, and the recursion depth at least 2 times the depth of the partitioning. In other word, the recursion depth would be 2*log(4/3) of n. That means we have O(n) steps per each level of partitioning, and overall we have O(n log n) complexity.
By doing so, we can randomly choose not one, but three elements from the subsequences, and then take median of the three as the pivot. By the median, we will have the elements in the middle-half, which is E subsequence. In our case, we have as following.

- L subarray = {5, 1}, median is 3
- E = {4, 3, 6, 2}, median is 3.7
- G = {7, 1, 3}, median is 3.5

When we combine these three numbers and find a median of them, the median is 3, which we can see as a good pivot.

Problem 3:

Explain in your own words what it means to say that “QuickSort runs in O(nlog n) time with high probability”. Then explain in two or three sentences the steps for proving this is true (no need to include any math – just a high level discussion).

Answer:
During the QuickSort process, every iteration the QuickSort picks a pivot, makes the existing array into 2 partitions, and continuously it does on both partitions in the same way.
If we trace the sequences, steps of the inputs, we can see the iteration ends when there’s a single element itself. Therefore, to see the QuickSort takes O(n log n) time, we just need to see every element from the array in the partitioning process and the comparison, swapping process.
Indeed, the partitioning process actually splits the given array into 2 sub-arrays. Since any number between n/4 and 3n/4 could be a pivot, so we need half the time on average. That means we need only 4 log n to complete the recursion for the QuickSort on the partitions. Beside, at each partition we need O(n) for the comparison. Therefore, the overall average time is O(n log n).

Problem 4:
Devise a fast algorithm for determining whether a sorted array A of distinct integers contains an element m for which A[m] = m. In this case “fast” means that the algorithm runs faster than Θ(n) time (in other words, it runs in O(n) time but not Θ(n) time). You must prove that your algorithm is fast in this sense.

Answer:
This problem is about searching an element equally to its index. There are two ways of doing, linear search or binary search.

In linear search, we just simply go thru all elements of the array and check if array[i] = i. That means the complexity is O(n), not Θ(n), because we just go thru maximum n times in the array given. Below is the implementation of the linear search to illustrate the algorithm.

```
int linearSearch(int[] array) {
      for (int i = 0; i < array.length; i++) {
            if (array[i] == i) {
                  return i;
            }
      }
      return -1;
}
```

With binary search, we need to check:

- First, if the middle element has its index equal to the value.
- If not, check if the index of the middle element is more than the value
    - If more, go right side of the middle element
    - If less, go left side
- Recur the algorithm
- The complexity of the binarySearch is O(log n), which is faster than the O(n) linear search above.

Below is the implementation of the binary search.

```
int binarySearch(int[] array, int left, int right) {
      if(right >= left) {
            int middle = (left + right) / 2;
            if(middle == array[middle])
                  return middle;
            if(middle > array [middle])
                  return binarySearch(array, (middle + 1), right);
            else
                  return binarySearch(array, left, (middle - 1));
      }
      return -1;
}
```

Problem 5:
In the slides several pivot-selection strategies were mentioned: random selection, selecting leftmost (or rightmost) element, or “median-of-three.” Consider the following alternative pivot-selection strategy: Whenever another pivot is needed, use QuickSelect to locate the median value for the array being considered (the median value is the value x for which the number of values in the array that are less than x is (approximately) equal to the number of values greater than x, For instance, 4 is the median value for {1, 3, 4, 7, 8} and also for {1, 4, 5, 6}.) What would the expected running time be for QuickSort if this new pivot selection strategy is used? Explain.

Answer:
As we know, the QuickSelect basically is similar to QuickSort, however, instead of running on both partitions after the pivot found, it runs on only the part contains the smaller elements. So, that would reduce the complexity from O(n log n) to O(n) only.
Being said, if we use the median as the pivot for the QuickSort, the complexity will reduce from O(n^2) to O(n log n), since the median finding process costs only in linear time O(n).

Problem 6:
Show the steps performed by QuickSelect as it attempts to find the median of the array [1, 12, 8, 7, -2, -3, 6]. (The median is the element that is less than or equal to n/2 of the elements in the array. Since n is odd in this case, it is the element whose position lies exactly in the middle. Hint: The median is 6.) For pivots, always use the leftmost element of the current array.

Answer:
The array = {1, 12, 8, 7, -2, -3, 6}
Length = 7

Partition from next element of pivot to the rest of array.
Run 2 pointers:

- Left pointer starts right after the pivot from left-most index, to the right-most index element
- Right pointer from right-most index to the left-pointer
- Comparing with pivot value. If element smaller than pivot, swap the 2 elements

Position to swap for the pivot starts with 0.

Loop:
Pivot = 1, left-most index = 1, right-most index = 6

left	array[left]	pivot	smaller than pivot
1	12	1	FALSE
2	8	1	FALSE
3	7	1	FALSE
4	-2	1	TRUE

Swap 12 with -2, new array = {1, -2, 8, 7, 12, -3, 6}
Increase left-most index by 1, decrease right-most index by 1.
Position to swap for the pivot increases from 0 to 1

Continue the loop, pivot = 1, left-most index =5, right-most index = 5

left	array[left]	pivot	smaller than pivot
5	-3	1	TRUE

Swap 8 with -3, new array = {1, -2, -3, 7, 12, 8, 6}
Increase left-most index by 1, decrease right-most index by 1.
Position to swap for the pivot increases from 1 to 2

Continue loop, left-most index =6, right-most index = 5

left	array[left]	pivot	smaller than pivot
6	6	1	FALSE

End loop, swap the current pivot with the position for the pivot updated above (= 2)
New array = {-3, -2, 1, 7, 12, 8, 6}

New pivot is 7 (the next of the updated pivot: 1).
Start loop on the partition from next element after the new pivot.

left	array[left]	pivot	smaller than pivot
4	12	7	FALSE
5	8	7	FALSE
6	6	7	TRUE

Swap 12 with 6, new array = {-3, -2, 1, 7, 6, 8, 12}
Updated the new position for the pivot from 2 to 3
End loop, swap the current pivot with the updated position for new pivot (=3).
New array = {-3, -2, 1, 6, 7, 8, 12}
No more swapping since the array sorted.
Return the median = 6 with position 3