CS5800 Assignment 3

For each problem, please present intuitions of your algorithm, write pseudocode, and state its running time. Each problem is 10 points unless specified.

Problem 1

Given a sorted array of distinct integers ***A[1,...,n]***, you want to find out whether there is an index i for which ***A[i] = i***. Design an O(logn) time algorithm, and show your analysis.

Approach:

Since the Array is sorted, we can use Binary search to do this, which will delete half of the remaining array, and because the array is filled with distinct integers, so the binary search approach will lead to a time complexity of O(logn) time.

Code:

```
public int searchIndex(int[] A) {
        int left = 0;
        int right = A.length - 1;
        while(left <= right) {
            int mid = left + (right - left) / 2;  //avoid overflow.
            if(A[mid] == mid) return mid;
            else if(A[mid] < mid) left = mid + 1;
            else right = mid - 1;
        }
        return -1;
    }
```

Running Time:

O(logn)

Problem 2

Shuffling a linked list. Design a divide and conquer algorithm that randomly shuffles a linked list in O(nlog(n)) time and logarithmic extra space.

Approach:

This is like merge sort a linked list, while instead of sorting in each part of the divided list, we randomly shuffle it and perform the divide and conquer method continuously.

Code:

```
public ListNode shuffleList(ListNode head) {
    if(head == null || head.next == null) return head;

    //step 1. cut the list into 2 halves
    ListNode prev = null, fast = head, slow = head;
    while(fast != null && fast.next != null) {
        prev = slow;
        slow = slow.next;
        fast = fast.next.next;
    }

    //step 2. random arrange each half
    prev.next = null;
    ListNode l1 = shuffleList(head);
    ListNode l2 = shuffleList(slow);

    //step 3. merge l1 and l2
    return shuffleMerge(l1, l2);
}

private ListNode shuffleMerge(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(0);
    ListNode temp = dummy;
    Random rand = new Random();
    while(l1 != null && l2 != null) {
        int flipCoin = rand.nextInt(2); // use random function in Java to generate 0 or 1
        if(flipCoin == 0) {
            temp.next = l1;
            l1 = l1.next;
        } else {
            temp.next = l2;
            l2 = l2.next;
        }
        temp = temp.next;
    }
    if(l1 != null) temp.next = l1;
    if(l2 != null) temp.next = l2;
    return dummy.next;

}

public static class ListNode{
        int val;
        ListNode next;
        ListNode(int x) {
            val = x;
            next = null;
        }
    }
```

Running Time:

O(nlog(n))

Problem 3

There are two sorted arrays ***A[1..n]*** and ***B[1..m]*** of size n and m respectively. Design an efficient algorithm to find the kth smallest number of the two sorted arrays.

Approach:

By using a divide and conquer approach, similar to the binary search method, we divide 2 arrays both by k / 2 and each time we will dump the small part of one array and only compare the larger part of this array and the other total array, do this action recursively and we will get our Kth smallest element.

Code:

```
public int findKthElement(int[] A, int n, int[] B, int m, int k) {
        if(k > m + n || k < 1) return -1;
        if(m > n) return findKthElement(B, m, A, n, k); //make n always larger than m.
        if (m == 0) return A[k - 1];
        if (k == 1) return Math.min(A[0], B[0]);

        int i = Math.min(n, k / 2);
        int j = Math.min(m, k / 2);

        if (A[i - 1] > B[j - 1]) {
            // we only need to compare the larger part of B and whole A.
            int temp[] = Arrays.copyOfRange(B, j, m);
            return findKthElement(A, n, temp, m - j, k - j);
        }
            // we only need to compare the larger part of A and whole B.
            int temp[] = Arrays.copyOfRange(A, i, n);
            return findKthElement(temp, n - i, B, m, k - i);
    }
```

Running Time:

O(log k)

Problem 4

You are given an infinite array A in which the first n cells contain integers in sorted order and the rest of the cells are filled with ∞. You are not given the value of n. Design an algorithm that takes an integer x as input and finds a position in the array containing x, if such a position exists. Your algorithm should run in O(logn) time. Argue the correctness of your algorithm and give analysis why your algorithm runs in O(logn) time.

Approach:

When I saw the time complexity should be O(logn) time, the first algorithm came to my mind is binary search, while binary-search needs a lower bound and a higher bound, since the array is infinite, we don't know the size of the array, thus we can not locate the higher bound.

So, the first thing we should do is to locate the appropriate higher bound, and then we can apply our beautiful binary search algorithm.

The way to do this is: first we let or lower bound pointing to 1st element and higher bound pointing to 2nd element of array and compare our target x with current higher bound. if x is greater than our higher bound then make lower bound index equals to our higher bound index and double the higher bound index. Continue do this until we find that our current higher bound is larger than x then apply binary search on current higher and lower bounds.

Code:

```
private int[] findHigherBound(int[] A, int x) {
        int low = 0, high = 1;
        int val = A[high];
        int[] res = new int[2]; // 2-position array to store higher and lower bound.
        while(val < x) {
            low = high;
            if(high * 2 < A.length - 1) {
                high = high * 2;
            } else {
                high = A.length - 1;
            }
            val = A[high];
        }
        res[0] = low;
        res[1] = high;
        return res;
    }

    public int findElementInInfiniteArray(int[] A, int x) {
        int[] bounds = findHigherBound(A, x);
        int left = bounds[0]; // lower bound found using helper function.
        int right = bounds[1]; // higher bound found using helper function.
        while(left <= right) {
            int mid = left + (right - left) / 2;
            if(A[mid] == x) return mid;
            else if(A[mid] > x) right = mid - 1;
            else left = mid + 1;
        }
        return -1;
    }
```

Running Time:

O(logn)

Problem 5

Design an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

- Integers in each row are sorted in ascending from left to right.
- Integers in each column are sorted in ascending from top to bottom.

Approach:

Consider the following matrix, we can easily figure out, if we start from the top right point, which is the 15, then for each row, numbers tend to decrease from right to left, and for each column, numbers tend to increase from top to bottom. Thus, we can build an algorithm such that starting from right top position, if element at this position is larger than our target, we move left, if smaller, we move down. Continuously do this, we will reach the position where our target locates.

```
[ ←----decrease----
  [1,   4,  7, 11, 15],
  [2,   5,  8, 12, 19],
  [3,   6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]
]
```

---increase------↓

Code:

```
public boolean searchMatrix(int[][] matrix, int target) {
        if(matrix == null || matrix.length == 0 || matrix[0].length == 0) return false;
        int row = 0, col = matrix[0].length - 1; // start from the top right point.
        while(row <= matrix.length - 1 && col >= 0) {
            if(matrix[row][col] == target) return true;
            else if(matrix[row][col] > target) col--;
            else row++;
        }
        return false;
    }
```

Running Time:

Since in the worst case, the 2 pointers will walk through the row and column length of the matrix, so the time complexity is O(m+n), where m and n are the size of the row and column.

Problem 6

Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, design an efficient algorithm to find the area of the largest rectangle in the histogram.

Approach:

In this approach, we maintain a stack. Initially, we push a -1 onto the stack to mark the end. We start with the leftmost bar and keep pushing the current bar's index onto the stack until we get two successive numbers in descending order, i.e. until we get a[i]<a[i-1]. Now, we start popping the numbers from the stack until we hit a number stack[j] on the stack such that a[[stack[j]] ⩽a[i]. Every time we pop, we find out the area of rectangle formed using the current element as the height of the rectangle and the difference between the current element's index pointed to in the original array and the element stack[top-1] -1 as the width i.e. if we pop an element stack[top] and i is the current index to which we are pointing in the original array, the current area of the rectangle will be considered as: (i−stack[top−1]−1)×a[stack[top]].

Further, if we reach the end of the array, we pop all the elements of the stack and at every pop, this time we use the following equation to find the area: (stack[top]−stack[top−1])×a[stack[top]], where stack[top] refers to the element just popped. Thus, we can get the area of the of the largest rectangle by comparing the new area found every time.

Code:

```
public int largestRectangleArea(int[] heights) {
        Stack<Integer> stack = new Stack<>();
        stack.push(-1);
        int max_Area = 0;
        for(int i = 0; i < heights.length; i++) {
            while(stack.peek() != -1 && heights[stack.peek()] >= heights[i]) {
                max_Area = Math.max(max_Area, heights[stack.pop()] * (i - stack.peek() - 1));
            }
            stack.push(i);
        }

        while(stack.peek() != -1) {
            max_Area = Math.max(max_Area, heights[stack.pop()] * (heights.length - stack.peek() -1));
        }

        return max_Area;
    }
```

Running Time:

Time complexity : O(n). n numbers are pushed and popped.

Problem 7 [20 pts]

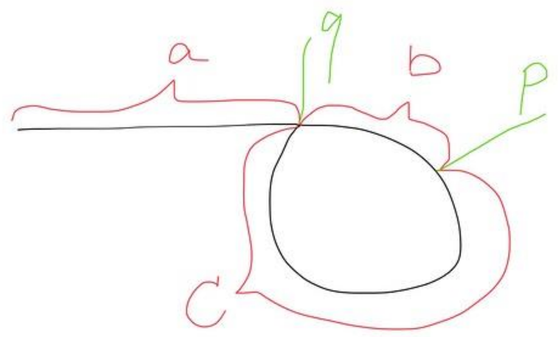
Given an array nums containing n + 1 integers where each integer is between 1 and n (inclusive), it is easy to see that at least one duplicate number must exist. Assume that there is only one duplicate number, find the duplicate one.

- You must not modify the array (assume the array is read only).
- You must use only constant, O(1) extra space.
- Your running time should be better than quadratic.
- There is only one duplicate number in the array, but it could be repeated more than once.

Approach:

Use two pointers the runner and the walker. The runner one goes forward two steps each time, while the walker one goes only step each time. They must meet the same item when walker==runner. In fact, they meet in a circle, the duplicate number must be the entry point of the circle when visiting the array from nums[0]. Next we just need to find the entry point. We use a point(we can use the runner one before) to visit form beginning with one step each time, do the same job to walker. When runner==walker, they meet at the entry point of the circle.

The supporting of this approach is the Floyd–Warshall algorithm: see picture below
When runner and walker meet at point p, the length they have run are 'a+2b+c' and 'a+b'.
Since the runner is 2 times runner than the walker. So a+2b+c == 2(a+b), then we get 'a==c'.
So when another walker2 pointer run from head to 'q', at the same time, previous walker pointer will run from 'p' to 'q', so they meet at the pointer 'q' together.



Code:

```
public int findDuplicate(int[] nums) {
        if(nums == null || nums.length <= 1) return -1;
        int walker = nums[0];
        int runner = nums[nums[0]];

        while(walker != runner) {
           walker = nums[walker];
           runner = nums[nums[runner]];
        }

        runner = 0;
        while(runner != walker) {
           walker = nums[walker];
           runner = nums[runner];
        }

        return walker;
    }
```

Running Time:

O(n) time and O(1) space

Problem 8 [20 pts]

The square of a matrix A is its product with itself, AA.

1. Show that five multiplications are sufficient to compute the square of a 2 by 2 matrix.
2. From (1), can you say the running time for computing the square of an n by n matrix is O($n^{log_2 5}$)? Why or why not?

1. Consider a 2 * 2 matrix A = $\begin{matrix} a11 & a12 \\ a21 & a22 \end{matrix}$, we compute it's square using standard multiplication to get:

AA = $\begin{matrix} a11 & a12 \\ a21 & a22 \end{matrix} * \begin{matrix} a11 & a12 \\ a21 & a22 \end{matrix} = \begin{matrix} a11 * a11 + a12 * a21 & a11 * a12 + a12 * a22 \\ a21 * a11 + a22 * a21 & a21 * a12 + a22 * a22 \end{matrix}$

Therefore, we only need to make the five multiplications appearing here: a11^2, a22^2, a12 * a21, a12*(a11 + a22) and a21*(a11+a22).

2. From(1), the way to get O($n^{log_2 5}$) running time is that: use a divide-and-conquer approach as in Strassen's algorithm except that instead of getting 7 subproblems of size n/2, we now get 5 subproblems of size n/2 by using our observation in part a the problem is that this trick only works at the top level of recursion.
But, some of the recursive calls will need to compute solutions to matrix multiplications that are not themselves problems of squaring a matrix, to which this trick cannot be applied. There is also a problem in that part a relied on the fact that regular multiplication is commutative, whereas matrix multiplication is not. Either is a fine answer.