PROBLEM 1:  (Short Answer)

A. In order to use the class `Point` containing fields `x` and `y` in a `HashSet`, you are considering multiple hash functions. Of these hash functions, which one would give the best performance in a `HashSet`? Assume that your points are likely to be between (0, 0) and (1280, 1024) (the size of the average computer monitor).

   I. `public int hashCode () { return super.hashCode(); }`
   II. `public int hashCode () { return 42; }`
   III. `public int hashCode () { return x; }`
   IV. `public int hashCode () { return x + y; }`
   V. `public int hashCode () { return x * 3 + y; }`
   VI. `public int hashCode () { return x * 1000 + y; }`

B. Compare and contrast

   I. Abstract Class & Interface
   II. Class & Object
   III. LinkedList & ArrayList

C. Bill shows you an algorithm to optimally choose classes for all students at Duke. You, being a good Computer Science student, notice that the big-Oh complexity of the algorithm is $O(2^n)$ where $n$ is the number of students. However, Bill demonstrates the program for a sample of 100 students and it returns the schedules almost immediately.

   Bill says that his algorithm is good enough for Duke. He mentions something about Moore’s Law and states that “Computers are getting faster at an exponential rate. That is, every 18 months, they double in speed. Even if the program is not fast enough now, it will be soon.”

   Bill is off a little bit on what Moore’s law means. However, given that computers continue doubling in speed every 18 months, will the $O(2^n)$ algorithm ever be practical? Explain why or why not?

D. Draw what root and next points to after executing the following code. Indicate if there is some kind of error.

```java
TreeNode root = new TreeNode(8, new TreeNode(3, new TreeNode(2, null, null),
                                     new TreeNode(7, null, null)),
                              new TreeNode(10, null, null));
TreeNode next = root.right.right;
```
PROBLEM 2:  (Reverse (9 points))
Each of the Java functions on the left take a string s as input, and returns its reverse. For each of the following, state the recurrence (if applicable) and give the big-Oh complexity bound.
Recall that concatenating two strings in Java takes time proportional to the sum of their lengths, and extracting a substring takes constant time.

A. public static String reverse1(String s) {
    int N = s.length();
    String reverse = "",
    for (int i = 0; i < N; i++)
        reverse = s.charAt(i) + reverse;
    return reverse;
}

B. public static String reverse2(String s) {
    int N = s.length();
    if (N <= 1) return s;
    String left = s.substring(0, N/2);
    String right = s.substring(N/2, N);
    return reverse2(right) + reverse2(left);
}

C. public static String reverse3(String s) {
    int N = s.length();
    char[] a = new char[N];
    for (int i = 0; i < N; i++)
        a[i] = s.charAt(N-i-1);
    return new String(a);
}

PROBLEM 3:  (Interval)
From Sedgewick: Consider the following data type, for intervals on the line:

public class Interval implements Comparable<Interval> {
    public int left;
    public int right;

    public Interval(int l, int r) {
        left = l; right = r;
    }

    public int compareTo(Interval b) {
        return left - b.left;
    }

    /**
    * Returns true if and only if this Interval overlaps with b
    */
    public boolean overlap(Interval b) {
        // TODO: complete in part A
public static int countIntervals(Interval[] a) {
    Arrays.sort(a);
    int count = 1;
    int max = a[0].right;
    for (int i = 1; i < a.length; i++) {
        // TODO: complete missing line of code for part B
        if (a[i].right > max)
            max = a[i].right;
    }
    return count;
}

For a particular application, clusters of intervals are of importance. To find clusters, replace any pair of intervals that intersects (by even an endpoint) by the union of the two intervals, continuing until all intervals do not intersect. For example, the following set of intervals has 3 clusters:

Note that you are guaranteed to have Intervals with non-negative numbers. Given an array of intervals, how many clusters are there? The brute-force algorithm is quadratic, but an enterprising COS226 student figured out a way to find the number of clusters in an array of intervals in linearithmic time, with the following code to be added to Interval.

A. Complete overlaps above.

B. Complete countIntervals above.

PROBLEM 4: (BST redux (16 points))

For this problem binary trees, note that store integer values instead of strings and no duplicate values appear in the tree.

By definition a binary search tree is a binary tree if each of the following properties hold for every node in the tree.

1. All values in the left subtree of a node are less than the node's value.
2. All values in the right subtree of a node are greater than the node's value
3. Both the left and right subtree of a node are binary search trees.

A. The methods getMin and getMax below return the smallest and largest values in a tree, respectively. They are used in writing isSearchTree that returns a boolean value indicating whether a tree is a search tree using the definition of a search tree given above.
public int getMax(TreeNode root){
    if (root == null) return Integer.MIN_VALUE;
    return Math.max(root.info,
                    Math.max(getMax(root.left), getMax(root.right)));
}

public int getMin(TreeNode root){
    if (root == null) return Integer.MAX_VALUE;
    return Math.min(root.info,
                    Math.min(getMin(root.left), getMin(root.right)));
}

public boolean isSearchTree(TreeNode root){
    if (root == null) return true;
    return
       isSearchTree(root.left) &&
       isSearchTree(root.right) &&
       root.info > getMax(root.left) &&
       root.info < getMin(root.right);
}

What is the runtime of isSearchTree in both the average and the worst case for a tree of n nodes? You must provide two recurrence relations for isSearchTree: one for the average case in which trees are balanced, and one in the worst case in which trees are completely unbalanced. Your answer should include justifications for the runtime for other methods that are called (i.e. getMin and getMax).

B. A student suggests different implementations of getMax and getMin because in their use in the code above they are only called when it is known that the tree is a search tree. The alternate implementations are below.

    public String getMax(TreeNode root){
        while (root.right != null){
            root = root.right;
        }
        return root.info;
    }

    public String getMin(TreeNode root){
        while (root.left != null){
            root = root.left;
        }
        return root.info;
    }

Explain what the recurrences would be in the average and worst case for isSearchTree using these alternative implementations of getMin and getMax. You do not need to solve the recurrences, you just need to give them.

C. Another student says that the alternative implementations can be used even when the body of isSearchTree is below, where the recursive calls are swapped with the calls to getMin and getMin. In a few words indicate whether you think this code will work correctly with the alternative implementations and why you think this.

    public boolean isSearchTree(TreeNode root){
        if (root == null) return true;
        return
           root.info > getMax(root.left) &&
           ...
Test 2 Review Questions

```
root.info < getMin(root.right) &&
isSearchTree(root.left) &&
isSearchTree(root.right) &&
}
```

PROBLEM 5: (Grids (18 points))

Consider a \(N\)-by-\(N\) grid in which some squares are occupied by black circles. Two squares belong to the same group if they share a common edge.

In the picture to the right, there are

- 1 group of 4 occupied squares
- 1 group of 3 occupied squares
- 2 groups of 2 occupied squares
- 2 groups of individually occupied squares

Given that grid is an two-dimensional array where grid[i][j] == true if and only if grid cell (i,j) is occupied, the following questions ask you to find the groups.

Given a grid and a grid cell location (row, col), groupSize should compute the size of the group including that square. For example, in the example above groupSize(grid, 1, 3) should return 2.

```java
public int groupSize(boolean[][] grid, int row, int col)
{
```

PROBLEM 6: (Puzzle Hunt)

You are given a matrix of positive integers to represent a game board, where the (0, 0) entry is the upper left corner. The number in each location is the number of squares you can advance in any of the four primary compass directions, provided that move does not take you off the board. You are interested in the total number of distinct ways one could travel from the upper left corner to the lower right corner, given the constraint that no single path should ever visit the same location twice.

Consider the initial game board to the left, and notice that the upper left corner is occupied by a 2. That means you can take either two steps to the right, or two steps down (but not two steps to the left or above, because that would carry you off the board). Suppose you opt to go right so that you find yourself in the configuration to the right.
After that, you could continue along as follows:

This series of moves illustrates just one of potentially several paths you could take from upper left to lower right. Your task is to write a method called `numPaths`, which takes a 2-d array of integers and computes the total number of ways to travel to the lower right corner of the board. Note that you never want to count the same path twice, but two paths are considered to be distinct even if they share a common sub-path. And because you want to prevent cycles, you should change the value at any given location to a zero as a way of marking that you’ve been there. Just be sure to restore the original value as you exit the recursive call. You may want to write a helper function to handle the recursion and a utility function to decide if you are on the board or not.

A. Write `numPaths` below.

```cpp
/**
 * Calculates total number of distinct ways one could travel from the
 * upper left corner of grid to the lower right corner, given the
 * constraint that no single path should ever visit the same location twice.
 * @param board square matrix board[i][j] is the number of squares
 * one can advance vertically or horizontally from (i,j)
 * @return the number of possible paths from (0,0) to the lower
```
* right corner of board (board.length-1, board[0].length - 1)
*/
public static int numPaths(int[][] board)
{
}

// HELPER FUNCTIONS
/**
 * @return true if (row,col) is within the bounds of the board
 * (i.e. 0 <= row < board.length and 0 <= col < board[0].length)
 * false otherwise
 */
public static boolean onBoard(int[][] board, int row, int col)
{
}

/**
 * @return the number of possible paths from (row,col) to the lower
 * right corner of board (board.length-1, board[0].length - 1)
 */
public static int numPaths(int[][] board, int row, int col)
{
}

B. Give a recurrence for your solution. You do not need to solve the recurrence.