Problem 1 – Arrays and Functions (25 points)

Suppose we have a 2D array which is used to record grocery item prices. Each row represents an item and each column represents a daily price. Notice that different rows may have different number of columns. Write a method to do statistics for these items. The method takes the aforementioned 2D array as input and returns another 2D array. Specifically, in the returned 2D array, each row represents the same item with the corresponding input 2D array and each row has exactly 3 columns where the first column is to store the average price, the second column is to store the maximum price and the last column is to store the minimum price. The method signature is given below:

```
public static double[][] stat(double[][] prices)
```

For example, if the method parameter is {{1,2,3}, {2,3}, {3,4,5,6}}, then the output would be a 3 by 3 2D array which has the following values:

```
2.0     3.0     1.0
2.5     3.0     2.0
4.5     6.0     3.0
```

Sol:

```
public static double[][] stat(double[][] prices){
    double[][] ret = new double[prices.length][3];
    for(int i=0; i<prices.length; i++){
        double sum = prices[i][0];
        double max = prices[i][0];
        double min = prices[i][0];
        for(int j=1; j<prices[i].length; j++){
            sum = sum + prices[i][j];
            if(max < prices[i][j])
                max = prices[i][j];
            if(min > prices[i][j])
                min = prices[i][j];
        }
        ret[i][0] = sum / prices[i].length;
        ret[i][1] = max;
        ret[i][2] = min;
    }
    return ret;
}
```

Grading:
4 pts for creating a 2-D, including 2 pts for prices.length.

4 pts for the outer loop to handle row of the parameter.

3 pts for initialization, 1 pt for sum, 1 pt for max and 1 pt for min.

4 pts for the inner loop to handle each column, 2 pts for the loop exit condition.

1 pt for computing sum

1 pt for computing max

1 pt for computing min

2 pt for computing average, including 1 pt for prices[i].length

1 pt for setting the average

1 pt for setting max

1 pt for setting min

2 pts for return

Problem 2 – Recursion (25 points)

1. **(4 points)** Select the output of the `RecursionProblem1` program when executed.

```
public class RecursionProblem1
{
    public static void recTest(int n)
    {
        if (n <= 0) return;
        recTest(n-4);
        recTest(n-2);
        StdOut.print(n + " ");
    }

    public static void main(String[] args)
    {
        recTest(6);
    }
}
```

a) 2 4
b) 2 2 4 8
c) 2 4 4 2
d) 2 2 4 6

Answer : D

2. **(9 points)** Given the RecursionProblem2 program below.

```
public class RecursionProblem2
{
    public static int recCheck(int x)
    {
        if (x <= 0)
        {
            return 0;
        }
        else
        {
            StdOut.print(x + ":");
            return (recCheck(x/2));
        }
    }

    public static void main(String[] args)
    {
        StdOut.print(recCheck(1));
    }
}
```

a) **(2 points)** What is the base case for `recCheck`?
X being less than or equal to zero

b) **(2 points)** What is the reduction/recursive step?
Cutting x in half or making the next call for x/2

c) **(4 points)** What is the output of `recCheck(300)`?
300:150:75:37:18:9:4:2:1:0

d) **(1 point)** What is the output of `recCheck(1)`?
1:0

3. **(12 points)** Given the following `Rabithole` program.

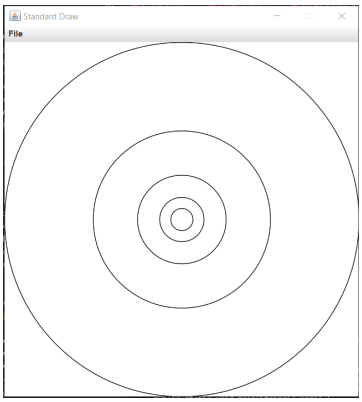
```
public class Rabbithole {

    // draw an order n Rabbithole, centered on (x, y)
    public static void draw(int n, double x, double y, double rad) {
        if (n == 0)
        {
            return;
        }
        else
        {
            // INSERT CODE HERE
        }
    }

    // reads an integer command-line argument n and creates concentric circles down.
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);

        double x = 0.5, y = 0.5;   // center of circle
        double size = 0.5;         // radius of circle
        draw(n, x, y, size);
    }
}
```

Insert code (where shown) to create a recursive program that draws smaller circles (circles with a radius half that of the circle outside of it) inside the larger ones. When the code is run for 5 levels ("`java Rabbithole 5`" at a command line) it looks like this:



Answer:

```
/******************************************************************************
 *  Compilation:  javac Rabbithole.java
 *  Execution:    java Rabbithole n
 *  Dependencies: StdDraw.java *
 *
 *  % java Rabbithole 5
 *
 ******************************************************************************/

public class Rabbithole {

    // plot an order n Rabbithole, centered on (x, y)
    public static void draw(int n, double x, double y, double rad) {
        if (n == 0)
        {
            return;
        }
        else
        {
            StdDraw.circle(x, y, rad);
            // call the next smaller circle, cutting the radius in half
            draw(n-1, x, y, rad/2);  // inner C
        }
    }

    // reads an integer command-line argument n and creates concentric circles down.
    Run | Debug
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);

        double x = 0.5, y = 0.5;   // center of circle
        double size = 0.5;         // radius of circle
        draw(n, x, y, size);
    }
}
```

Problem 3 – Object Oriented Programming (40 points)

A gardening supply center sells many varieties of plants and must keep an adequate amount of each variety available for customers to purchase. This problem involves writing methods in a `PurchaseOrder` class. A `PurchaseOrder` is an order that holds multiple `PlantOrders`.
Each `PlantOrder` has its own unique `orderID`. The IDs are assigned to each `PlantOrder` in the `PlantOrder` constructor. The first `orderID` assigned is 100. The second `orderID` assigned is 101, etc.

a. **(4 points) Refer to the `PlantOrder` class below.** Explain how you must modify the `PlantOrder` class so that when a `PlantOrder` is created, a unique `orderID` is assigned. This `orderID` is determined by incrementing the last assigned `orderID` by 1.

```
/* A PlantOrder is used to order a specific variety of plant.
   The PlantOrder specifies two pieces of informnation:the variety of plant to be ordered
   and the number of plants to order.
 */
public class PlantOrder{

   private int orderID;
   private String variety;
   private int numPlants;

   // Constructs a new PlantOrder; Precondition: numPlants>0
   public PlantOrder(String variety, int numPlants){
        this.variety = variety;
        this.numPlants = numPlants;
    }

    // Returns the variety of plants being ordered
    public String getVariety(){
        //implementation not shown
    }

    // Returns the orderID of this PlantOrder
    public int getOrderID(){
       //implementation not shown
   }

    // Returns the number of plants being ordered
    public int getNumPlants(){
        //implementation not shown
    }

    public String toString(){
        return (this.orderID + " " + this.variety + ": " + this.numPlants +  " ordered");
    }
}
```

4 points:

- **(2 points) Add a private static int variable orderNumber that is initialized to 100 to the variable declaration section of the class:**

  ```
  private static int orderNumber = 100; //variable could be different
  ```

- **(2 points) assign this variable to orderID and increment this variable in the constructor**

  ```
  this.orderID = orderNumber;
  orderNumber++;
  ```

PurchaseOrder class

```
import java.util.ArrayList;

public class PurchaseOrder{

    // The list of all plant orders
    private ArrayList<PlantOrder> masterOrder;

    //Constructs a new PuchaseOrder object.
    public PurchaseOrder() {
        //to be inoplemented in part b
    }

    // adds a new PlantOrder to the masterOrder
    public void addOrder(PlantOrder order) {
        masterOrder.add(order);
    }

    /* returns the sum of the number of plants of all of the plant orders
       on the master order
    */
    public int getTotalPlants() {
        //to be implemented in part c
    }

    /* returns total plants of specified variety in MasterOrder.
     */
    public int getPlants(String variety) {
        //to be implemented in part d
    }

    /** Removes all orders from the master order that have the same variety of
        plant as the parameter, variety, and returns the total number of plants
        that were removed.
    */
    public int removeVariety(String variety) {
        //to be implemented in part e
    }

    /*
      for appropriate printing of masterOrder contents
    */
    public String toString(){
        return masterOrder.toString();
    }
}
```

Consider the following statements in a client program.

```
PurchaseOrder plantDepot = new PurchaseOrder();
plantDepot.addOrder(new PlantOrder("tulips",1000));
plantDepot.addOrder(new PlantOrder("roses",100));
plantDepot.addOrder(new PlantOrder("lilies",200));
plantDepot.addOrder(new PlantOrder("daisies",100));
plantDepot.addOrder(new PlantOrder("roses",50));
plantDepot.addOrder(new PlantOrder("daffodils",60));
```

The resulting `PurchaseOrder` would hold the `PlantOrder`s below:

plantDepot

100 **tulips** **1000**	**101** **roses** **100**	**102** **lilies** **200**	**103** **daisies** **100**	**104** **roses** **50**	**105** **daffodils** **60**

Consider further, the following statements in a client program:

```
System.out.println(plantDepot);
System.out.println("Total plants ordered = " + plantDepot.getTotalPlants());
System.out.println("Total roses ordered = " + plantDepot.getPlants("roses"));
System.out.println("Total mums ordered = " + plantDepot.getPlants("mums"));
System.out.println(plantDepot.removeVariety("roses") + " roses removed." ) ;
System.out.println(plantDepot);
```

The result of executing the above would print:

[100 tulips: 1000 ordered, 101 roses: 100 ordered, 102 lilies: 200 ordered, 103 daisies: 100 ordered, 104 roses: 50 ordered, 105 daffodils: 60 ordered]
Total plants ordered = 1510
Total roses ordered = 150
Total mums ordered = 0
150 roses removed.
[100 tulips: 1000 ordered, 102 lilies: 200 ordered, 103 daisies: 100 ordered, 105 daffodils: 60 ordered]

b. **(6 points)** Implement `PurchaseOrder()`

```
//Constructs a new PuchaseOrder object.
public PurchaseOrder() {
    //to be inoplemented in part b
}
```

Solution

```
//Constructs a new PuchaseOrder object.
public PurchaseOrder() {
    //to be implemented in part a
     masterOrder = new ArrayList<PlantOrder>();
}
```

6 points
lose 2 points if missing ()
lose 2 points if new
lose 2 pt for no assignment to masterOrder – lose this if redeclaring type.

c. **(8 points)** Implement `getTotalPlants()`

```
/* returns the sum of the number of plants of all of the plant orders
   on the master order
*/
public int getTotalPlants() {
    //to be implemented in part c
}
```

Solution

```
public int getTotalPlants() {
    //to be implemented in part b
    int total = 0;
    for(PlantOrder p: masterOrder){
        total += p.getNumPlants();
    }
OR
    for (int i = 0; i < masterOrder.size(); i++){
        total += masterOrder.get(i).getNumPlants();
    }

    return total;
}
```

8 points :

2 pt init accumulator
2 pt return calculated value
2 pt correct loop limits
2 points correct call to getNumPlants()

d. **(10 points)** Implement `getPlants()`

```
/* returns total plants of specified value in MasterOrder.
 */
public int getPlants(String variety) {
    //to be implemented in part d
}
```

Solution

```
public int getPlants(String variety) {
    //to be implemented in part d
    int total = 0;
    for(PlantOrder p: masterOrder){
        if (p.getVariety().equals(variety)){
            total += p.getNumPlants();
        }
    }
OR>>>>
    for (int i = 0; i < masterOrder.size(); i++){
        if (masterOrder.get(i).getVariety().equals(variety)){
           total += masterOrder.get(i).getNumPlants();
        }
    }

    return total;
}
```

10 points :
2 pt init accumulator;
2 pt return calculated value
2 points for correct loop limits
2 points for correct if condition (lose this for ==)
2 points for accumulating numPlants correctly

e. **(12 points)** Implement `removeVariety()`

```
/** Removes all orders from the master order that have the same variety of
    plant as the parameter, variety, and returns the total number of plants
    that were removed.
*/
public int removeVariety(String variety) {
    //to be implemented in part e
}
```

```
public int removeVariety(String variety) {
    //to be implemented in part e
    int total = 0;
    for(int i = 0; i < masterOrder.size(); i++){
        if (masterOrder.get(i).getVariety().equals(variety)){
            total += masterOrder.get(i).getNumPlants();
            masterOrder.remove(i);
        }
    }
    return total;
}
```

12 points :
2 pt init accumulator;
2 pt return calculated value
2 points for correct loop limits—CAN NOT USE Enhanced FOR loop.
2 points for correct if condition (lose this for ==)
2 points for accumulating numPlants correctly
2 points for removing orders.

Problem 4 – Searching and Sorting (20 points)

a) **(12 points)** Search for the character **F** using the **binary search** algorithm on the following array of characters:

B	C	F	K	N	O	P	T	U
0	1	2	3	4	5	6	7	8

For each iteration of binary search use the table below to list: (a) the left index and (b) the right index of the array that denote the region of the array that is still being searched, (c) the middle point index of the array, and (d) the number of character-to-character comparisons made during the search.

1 point for each correct

iteration	left	right	middle	#compares
1	**0**	**8**	**4**	**2**
2	**0**	**3**	**1**	**2**
3	**2**	**3**	**2**	**1**

b) **(8 points)** Given an unsorted array of *n* elements. Using one of the sorting algorithms you have learned, describe the fastest algorithm that would modify the input array to have the smallest *k* elements in the array in the first *k* positions in sorted order. The remaining values can be in any order. Assume 0 < *k* <= *n*. The algorithm is expected to run in O(n).

For example:

- If the array is [**5, 12, 9, 8, 1, 7, 15**] and **k = 2** the algorithm would return [**1, 5**, 9, 8, 12, 7, 15], notice that the first 2 positions have the 2 smallest values in sorted order.
- If the array is [**5, 12, 9, 8, 1, 7, 15**] and **k = 3** the algorithm would return [**1, 5, 7**, 8, 12, 9, 15], notice that the first 3 positions have the 3 smallest values in sorted order.

Selection sort finds the minimum item in the array and swaps with the first item of the unsorted region.
Run selection sort k times in the array.

Problem 5 – Complexity Analysis (20 points)

1. (2 points) (True or False) In a doubling experiment we double the time it takes to run the experiment.

   False. You double the size of the data.

2. (2 points) When considering an algorithm's performance we consider time and ?
   a. cost
   b. memory
   c. power
   d. elegance

   Answer: memory

3. (2 points) In order to provide a performance guarantee for an algorithm we take a conservative view and examine the running time of :
   a. all cases
   b. the worst case
   c. the best case
   d. a random sample of cases

   Answer: b

4. (2 points) The Big O (order of growth) for the code fragment below is:

```
for (int i = 0; i < n; i++)
{
    for (int j = i+1; j < n; j++)
    {
        StdOut.println(i + " " + j);
    }
}
```

   a. $O(n^2)$
   b. $O(n)$
   c. $O(n^3)$
   d. $O(1)$
   e. $O(\log n)$
   f. $O(n \log n)$

   Answer: a

5. (2 points) The Big O (order of growth) for the code fragment below is:

```
StdOut.println(n);
```

a. $O(n^2)$
b. $O(n)$
c. $O(n^3)$
d. $O(1)$

e. O(log n)
f. O(n log n)

Answer: d

6. (2 points) The Big O (order of growth) for the code fragment below is:

```
for (int i = 0; i < n; i++)
{
    StdOut.println(i);
}
```

a. $O(n^2)$
b. O(n)
c. $O(n^3)$
d. O(1)
e. O(log n)
f. O(n log n)

Answer: b

7. (2 points) The Big O (order of growth) for the code fragment below is:

```
for (int i = n; i > 0; i /= 2)
{
    StdOut.println(i);
}
```

a. $O(n^2)$
b. O(n)
c. $O(n^3)$
d. O(1)
e. O(log n)
f. O(n log n)

Answer: e

8. (2 points) The Big O (order of growth) for the code fragment below is:

```
// a is an array of size n
int n = a.length;
    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            for (int k = j+1; k < n; k++) {
                if (a[i] + a[j] + a[k] == 0) {
                    StdOut.println(a[i] + " " + a[j] + " " + a[k]);
                }
            }
        }
    }
```

a. $O(n^2)$
b. O(n)
c. $O(n^3)$
d. O(1)
e. O(log n)
f. O(n log n)

Answer: c

9. (4 points) Consider the snippet of code below.

```
public class counting
{
    Run | Debug
    public static void main(String[] args)
    {

    int n = Integer.parseInt(args[0]);
    int a[] = new int[n];
    for (int i = 0; i < n; i++) a[i] = i;

    int b = 1;
    int clicker = 0;

    while (b < a.length)
    {
        for (int i = 0; i < b; i++)
        {
            clicker += 1;
        }
        b++;
    }

StdOut.println(clicker);
```

9.1 – (2 points) What is the Big O (order of growth) for the above snippet of code?

Answer: $O(n^2)$

9.2 – (2 points) What is the value of the variable "clicker" after the snippet is run for n=5?

10