For problems 1 and 2, draw a recursive tree diagram for each method call. Also, show what is returned to the method call. If something is printed, show the print output as well.

1. (6 points)

```
public int backwards(int a, int b)
{
    if (a % 3 == 0)
        return b;
    return a + b + backwards(b - 1, a - 1);
}
```

```
backwards(20, 10);
```

2. (6 points)

```
public void printNames(String[] elements, int k)
{
    if (k > 1)
    {
        printNames(elements, k - 1);
        System.out.print(elements[k] + "   ");
        printNames(elements, k - 2);
    }
}
```

```
String [] students = {"Amy", "Jung", "Joe",
                      "Neha", "Nancy"};
printNames(students, students.length - 1);
```

3. (6 points) True/False

_____ a) If a routine calls itself, then it is recursive.

_____ b) Accessing an **ArrayList** can cause an **ArrayIndexOutOfBoundsException**.

_____ c) Dividing by 0 causes a **MathematicalException** to occur.

_____ d) Exception handling only occurs during the execution of a program.

_____ e) The APCS course assumes that all two-dimensional arrays are ordered in column-major fashion.

_____ f) **IllegalArgumentException** does not require an **import** statement.

4. (6 points) Consider the following list of **String**s, sorted in order. Perform a Binary Search for the indicated value, showing your work clearly by describing the variables. Determine the number of steps (number of times mid is calculated) and the value returned.
Search for **"HELLO"**.

0	1	2	3	4	5	6	7	8	9	10	11	12
ACE	**AN**	**BE**	**CAN**	**DO**	**HOW**	**LIE**	**MINE**	**NOT**	**SO**	**THAT**	**THE**	**WHO**

5. (6 points) Consider the following list of **int**s, sorted in order. Perform a Binary Search for the indicated value, showing your work clearly by describing the variables. Determine the number of steps (number of times mid is calculated) and the value returned.
Search for **39**.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	**8**	**12**	**15**	**23**	**34**	**39**	**42**	**61**	**77**	**80**	**94**	**106**	**121**	**144**	**176**

6. (4 points) Consider the following code segment.

```
String str = "abcdef";
for(int rep = 0; rep < str.length() - 1; rep++)
{
        System.out.print(str.substring(rep, rep + 2));
}
```

What is printed as a result of executing the code segment?

(A) abcdef
(B) aabbccddeeff
(C) abbccddeef
(D) abcbcdcdedef
(E) an **IndexOutOfBoundsException** is thrown.

7. (4 points) Consider the following class:

```
public class TestSample
{
        private ArrayList<Integer> samples;

        public TestSample (int n)
        {
                samples = new ArrayList<Integer>();
                for(int k = 1; k < n; k++)
                {
                        samples.add(k);
                }
        }

        public double getBestRatio ( )
        {
                double maxRatio = samples.get(1).intValue() / samples.get(0).intValue();

                for(int k = 0; k < samples.size(); k++)
                {
                        double ratio = samples.get(k+1).intValue() / samples.get(k).intValue();
                        if (ratio > maxRatio)
                        {
                                maxRatio = ratio;
                        }
                }
                return maxRatio;
        }
}
```

What is the result of the following code segment?

```
TestSample test = new TestSample(3);        //  notice that's a 3 there . . .
System.out.println(test.getBestRatio());
```

(A) **NullPointerException**
(B) **ArithmeticException**
(C) **IndexOutOfBoundsException**
(D) **2.0** is printed
(E) **3.0** is printed

8. (2 points) Consider an ArrayList of 1200 (sorted) **Objects**. Assume that we apply a binary search algorithm to this array, looking for an element that does not exist in the array. What is the maximum number of steps that it can take to determine that this element does not exist in the list?

9. (2 points) Consider an array of 5500 (sorted) **Integers**. Assume that we apply binary search algorithm to this array, looking for an element that exists in the array. What is the maximum number of steps that it can take to find this element?

10. (4 points) Consider the following code segment.

```
int [][] mat = new int [3][4];
for(int row = 0; row < mat.length; row++)
{
        for(int col = 0; col < mat[0].length; col++)
        {
                if (row < col || col == 0)
                {
                        mat[row][col] = 1;
                }
                else if (row % col == 0)
                {
                        mat[row][col] = 2;
                }
                if(row - 1 == col)
                {
                        mat[row][col] = 3;
                }
        }
}
```

What are the contents of mat after the code segment has been executed?

(A)
```
{{1, 3, 1, 1},
 {1, 2, 1, 1},
 {1, 3, 2, 1}}
```

(B)
```
{{1, 1, 1, 1},
 {1, 2, 1, 1},
 {1, 3, 2, 1}}
```

(C)
```
{{2, 1, 1, 1},
 {3, 2, 1, 1},
 {3, 3, 2, 1}}
```

(D)
```
{{1, 1, 1, 1},
 {3, 2, 1, 1},
 {1, 3, 2, 1}}
```

(E)
```
{{1, 1, 1, 1},
 {3, 2, 1, 1},
 {3, 3, 2, 1}}
```

11. (4 points) Consider the following two methods that appear within a single class.

```
public void changeIt (Integer [] list, int num)
{
        list = new Integer [5];
        num = 0;
        for(int x = 0; x < list.length; x++)
        {
                list[x] = x + 1;
        }
}

public void start()
{
        Integer [] nums = new Integer [5];
        int value = 6;
        changeIt(nums, value);
        for(int k = 0; k < nums.length; k++)
        {
                System.out.print(nums[k].intValue() + "  ");
        }
        System.out.print(value);
}
```

What is printed as a result of the call **start()**?

(A) 0 0 0 0 0 0
(B) 0 0 0 0 0 6
(C) 1 2 3 4 5 6
(D) 1 2 3 4 5 0
(E) a **NullPointerException**. is thrown.

12. (4 points) Consider the following code segment.

```
int [] oldArray = {1, 2, 3, 4, 5, 6, 7, 8, 9};
int [][] newArray = new int [3][3];

int row = 0;
int col = 0;
for(int value : oldArray)
{
        newArray[row][col] = value;
        row++;
        if((row % 3) == 0)
        {
                col++;
                row = 0;
        }
}
System.out.print(newArray[0][2]);
```

What is printed as a result of executing the code segment?

(A) **3** (B) **4** (C) **5** (D) **7** (E) **8**

13. Consider the following incomplete class that stores information about a client, which includes a name and unique ID (a positive integer). To facilitate sorting, clients are ordered alphabetically by name. If two or more clients have the same name, they are further ordered by ID number. A particular client is "greater than" another client if that particular client appears later in the ordering than the other client.

```
public class Client
{
        // constructs a Client with given name and ID number
        public Client (String name, int idNum)
        { /* implementation not shown */ }

        // returns the client's name
        public String getName()
        { /* implementation not shown */ }

        // returns the client's id
        public int getID()
        { /* implementation not shown */ }

        // returns 0 when this client is equal to the other;
        // a positive integer when this client is greater than other;
        // a negative integer when this client is less than other
        public int compareClient(Client other)
        { /* to be implemented in part (a) */ }

        // There may be fields, constructors, and methods that are not shown.
}
```

(a) (6 points) Write the **Client** method **compareClient**, which compares this client to a given client, **other**. Clients are ordered alphabetically by name, using the **compareTo** method of the **String** class. If the names of the two clients are the same, then the clients are ordered by ID number. Method **compareClient** should return a positive integer if this client is greater than **other**, a negative integer if this client is less than **other**, and 0 if they are the same.

For example, suppose we have the following **Client** objects.

```
Client c1 = new Client ("Smith", 1001);
Client c2 = new Client ("Anderson", 1002);
Client c3 = new Client ("Smith", 1003);
```

The following table shows the result of several calls to **compareClient**.

Method Call	Result
c1. compareClient (c1)	0
c1. compareClient (c2)	a positive integer
c1. compareClient (c3)	a negative integer

(6 points) complete method **compareClient** below.

```
// returns 0 when this client is equal to the other;
// a positive integer when this client is greater than other;
// a negative integer when this client is less than other
public int compareClient (Client other)
```

(b) (16 points) A company maintains client lists where each list is a sorted array of clients stored in ascending order by client. A client may appear in more than one list, but will not appear more than once in the same list.

Write method **mergeLists**, which takes three array parameters. The first two arrays, **list1** and **list2**, represent existing client lists. It is possible that some clients are in both arrays. The third array, **result**, has been instantiated to a length that is no longer than either of the other two arrays and initially contains **null** values. Method **mergeLists** uses an algorithm similar to the merge step of a Mergesort to fill the array **result**. Clients are copied into **result** from the beginning of **list1** and **list2**, merging them in ascending order until all positions of **result** have been filled. Clients who appear in both **list1** and **list2** will appear at most once in **result**.

For example, assume that three arrays have been initialized as shown below.

list1

Arthur 4920	Burton 3911	Burton 4944	Franz 1692	Horton 9221	Jones 5554	Miller 9360	Nguyen 4339
0	1	2	3	4	5	6	7

list2

Aaron 1729	Baker 2921	Burton 3911	Dillard 6552	Jones 5554	Miller 9360	Noble 3335
0	1	2	3	4	5	6

result

null	null	null	null	null	null
0	1	2	3	4	5

In this example, the array **result** must contain the following values after the call **mergeLists (list1, list2, result)**.

result

Aaron 1729	Arthur 4920	Baker 2921	Burton 3911	Burton 4944	Dillard 6552
0	1	2	3	4	5

In writing **mergeLists**, you may assume that **compareClient** works as specified, regardless of what you wrote in part (a). Solutions that create any additional data structures holding multiple objects (e.g., arrays, ArrayLists, etc.) will not receive full credit.

(16 points)Complete method **mergeLists** below.

```
// fills result with clients merged from the beginning of list1 and list2;
// result contains no duplicates and is sorted in ascending order by client
// precondition:   result.length > 0;  list1.length >= result.length;
//                 list1 contains no duplicates;  list2.length >= result.length;
//                 list2 contains no duplicates;  list1 and list2 are sorted in ascending order by client
// postcondition:  list1, list2 are not modified
public static void mergeLists (Client [] list1, Client [] list2, Client [] result)
```