1. (30 Points) What is the asymptotic complexity of the following methods, in terms of the Big-O notation.

a. `void methodA(int n)
   {
   for (int i=n; i>1; i/=2) {
       System.out.println(i);
   }
   }

Analysis:
Here, in the loop i is decreased by 2 (from n to 2) in each iteration, so the loop runs n/2 times. Hence the complexity is O(n).

b. `void methodB(int n)
   {
   for (int i=1; i<n; i++){
       for (int j=n; j>n-i; j--){
           System.out.println(j);
       }
   }
   }

Analysis:
In this case there are two nested loops, in the outer loop i is incremented by 1 in each iteration, in the inner loop, j is decreased by 1. The complexity of the outer loop is O(n) and the inner loop is O(n). Hence the complexity of the code block is O(n^2).

c. `void methodC(int n)
   {
   for (int i=n; i>0; i--){
       for (int j=0; j<i; j++){
           fun(n);
       }
   }
   `
void fun(int n) {
    for (int i=1; i<n; i=i*3) {
        System.out.println(i);
    }
}

Analysis:
In the outer loop i is decreased by 1 in each iteration, in the inner loop, j is incremented by 1. The complexity of the outer loop is O(n) and the inner loop again is O(n). Now the inner loop calls a method whose complexity in turn is O(logn). In all, the complexity becomes O(n^2 logn).

d.
void methodD(int n) {
    for (int i=n; i<=n; i++) {
        for (int j=n; j>1; j=j/2) {
            System.out.println(j);
        }
    }
}

Analysis:
In this case there are two nested loops, in the outer loop i is incremented by 1 in each iteration, while in the inner loop, j is divided by 2, so you might think that the complexity is of O(nlogn). But look closely, in the first loop, i is initialized to n, so in effect it is executed only once due to which the complexity of the code block becomes O(logn).

e.
void methodE(int n, char ch) {
    if (ch == 'a') {
        for (int s=1; s<n; s++)
            for (int t=1; t<n; t=t*4) {
                System.out.println(t);
            }
    } else if (ch == 'b') {
        for (int i=0; i<n; i++)
            for (int j=0; j<n; j=j*4) {
                System.out.println(j);
            }
    }
}
System.out.println(i);
}
} else if (ch == 'c')
{
    for (int i=0; i<n; i++)
    {
        System.out.println(i);
    }
}
} else
{
    for (int k=n; k>1; k--)
    {
        for (int m=1; m<k; m++)
        {
            System.out.println(m);
        }
    }
}

Analysis:
Here the first loop has a complexity of O(nlogn); the second loop has a complexity of O(n); the third loop has a complexity of O(n); the fourth loop has a complexity of O(n^2). But we are considering the worst case and in the worst case the complexity is O(n^2).

2. (15 points) Prove that the summation \( \sum_{i=1}^{n} \log_2 i \) is \( O(n \log_2 n) \).

Analysis:
For any \( i < n \), we have \( \log_2 i < \log_2 n \). Then \( \sum_{i=1}^{n} \log_2 i < n \log_2 n \).

So \( \sum_{i=1}^{n} \log_2 i = O(n \log_2 n) \).

3. (15 points) Rank the following functions by order of growth from the slowest to the fastest (\( \log n \) means \( \log_{10} n \)).

\[
\begin{align*}
\text{n+1000} & \quad 2n^2+100 & \quad 2^n+n^2+10n & \quad 4^n & \quad n\cdot3^n & \quad (5/2)^n \\
8n^2 & \quad n! & \quad 2^{\log n} & \quad 4^{\log n} & \quad (n+1)! & \quad n^{2/3}
\end{align*}
\]

Analysis:

\( n^{2/3} < 2^{\log n} = n+1000 < 4^{\log n} = 2n^2 + 100 = 8n^2 < 2^n + n^2 + 10n < (5/2)^n < n\cdot3^n < 4^n < n! < (n+1)! \)
Given two arrays (the first array and the second array) of the same size, each consisting of n positive integers, write a Java program to determine how many unique integers the second array have. Here a unique integer means that it shows only once in the second array but does not show in the first array. For example, if the first array is [1, 5, 3, 6, 8] and the second array is [2, 4, 2, 5, 9] (with n=5), the second array has two unique integers (4 and 9). What is the complexity of the program? Measure the runtime of the program for array sizes of 10, 100, 1000 and 10000. Use `System.nanoTime()` to get time measurements. Include screenshots of the program running and submit your .java code in a separate file.

**Solution:**

The code is as follows:

```java
import java.util.Random;

public class Question4 {
    public static void findUniqueIntegers(int[] arr1, int[] arr2) {
        int count = 0;
        boolean uniqueflag = true;
        for(int i=0; i<arr2.length; i++) {
            uniqueflag = true;
            for(int j=0; j<arr1.length; j++) {
                if(arr2[i] == arr1[j]) //existing in array1
                {
                    uniqueflag = false;
                    break;
                }
            }
            for(int p=0; p<arr2.length; p++) {
                if(p != i && arr2[i] == arr2[p]) //show more than once in array2
                {
                    uniqueflag = false;
                    break;
                }
            }
            if(uniqueflag)
            {
                count++;
                System.out.println(arr2[i]);
            }
        }
        System.out.println("Total number of unique integers is "+ count);
    }
}
```
public static void main(String[] args) {

    int size = 10;
    int[] array1 = new int[size];
    int[] array2 = new int[size];

    int randomInt = 0;
    Random randomGenerator = new Random();  //Random generator to get random number
    for (int i=0; i<size; i++){
        randomInt = randomGenerator.nextInt(size)+1;
        array1[i] = randomInt;  //Populate the array with random numbers
        randomInt = randomGenerator.nextInt(size)+1;
        array2[i] = randomInt;  //Populate the array with random numbers
    }

    System.out.println("the contents of the array are ");
    for (int i=0; i<size; i++){
        System.out.print(" "+array1[i]);
    }
    System.out.println(");
    for (int i=0; i<size; i++){
        System.out.print(" "+array2[i]);
    }
    System.out.println(");

    long start = System.nanoTime();  //Start the timer
    for(int k=0; k<100; k++) { //Run the program multiple times to get an average runtime
        findUniqueIntegers(array1, array2);
    }

    long end = System.nanoTime();  //stop the timer
    //Calculate and print the average runtime of the program
    System.out.println("The average running time is "+(end-

start)/100 + "ns.");
}

Complexity:

To do what the question requires, we need two layers of loops, each layer individually having a complexity of O(n), so the complexity of the program is O(n²).
Time measurements:

<table>
<thead>
<tr>
<th>n</th>
<th>runtime(ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>98743</td>
</tr>
<tr>
<td>100</td>
<td>1504160</td>
</tr>
<tr>
<td>1000</td>
<td>5824026</td>
</tr>
<tr>
<td>10000</td>
<td>188218175</td>
</tr>
</tbody>
</table>

The screenshot of the system while running the program for n = 10.
5. **(20 points)** Write a Java program to determine all positive integers, c, such that $c \leq n$ and c can be the length of the longest side (i.e., hypotenuse) of a right integer triangle. A right integer triangle is a right triangle all of whose sides have lengths that are integers. For example, with $n=10$, we have $\{5, 10\}$. What is the complexity of the program? Measure the runtime of the program and plot it for values of $n=10, 100, 1000$ and $10000$. Use `System.nanoTime()` to get time measurements. Include screenshots of the program running and submit your .java code in a separate file.

**Solution:**

The code is as follows:
```java
/*
 * Question 5 of Homework 2
 */
public class Question5 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int a, b, c, n;
        n = 10;
        long start = System.nanoTime(); //Start the timer
        //Run the program multiple times to get an average runtime
        for(int i=0;i<100;i++)
        {
            boolean[] existing = new boolean[n+1]; //for checking duplication
            for(int j=0; j<=n; j++)
            {
                existing[j] = false;
            }
            for(a=1; a<n; a++)
            for(b=a+1; b<n; b++)
            for(c=b+1; c<=n; c++)
            {
                if(a*a + b*b == c*c && existing[c] == false)
                //Pythagorean theorem and check duplication
                {
                    existing[c] = true;
                    System.out.println(c);
                    //print the the length of the longest side of a right integer triangle
                }
            }
        }
        long end = System.nanoTime(); //Stop the timer
        //Calculate and print the average runtime of the program
        System.out.println("The average running time for n = " + n + " is " + (end-start)/100 + "ns.");
    }
}
```

Complexity:

To do what the question requires, we need three nested of loops, which individually has a complexity of $O(n)$, so the complexity of the program is $O(n^3)$.

Time measurements:

<table>
<thead>
<tr>
<th>n</th>
<th>runtime(ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>173476</td>
</tr>
<tr>
<td>100</td>
<td>565431</td>
</tr>
<tr>
<td>1000</td>
<td>588433455</td>
</tr>
<tr>
<td>10000</td>
<td>582841972378</td>
</tr>
</tbody>
</table>

The screenshot of the system while running the program for $n = 10$. 

```
public class Question5 {
    public static void main(String[] args) {
        // TODO: Auto-generated method stub
        int a, b, c, n; 
        n = 10;
        long start = System.nanoTime(); //Start the timer:
        //Run the program multiple times to get an average runtime
        for(int i=0;i<100;i++)
        {
            boolean[] existing = new boolean[n+1]; //For checking duplication
            for(int j=0; j<=n; j++)
            {
                existing[j] = false;
            }
            for(a=0; a<n; a++)
            for(b=a; b<n; b++)
            for(c=n-1; c>b; c--)
            {
                if(a*a + b*b == c*c && existing[c] == false) //Fermat's theorem and check duplication
                existing[c] = true;
            }
            System.out.print(c); //Print the the length of the longest side of a right integer triangle
        }
        long end = System.nanoTime(); //Stop the timer;
        //Calculate and print the average runtime of the program
        System.out.println("The average running time for n = " + n + " is " + (end-start)/1000 + "ms.");
    }
}
```