

Students : Pavel Tsvyatkov
Cansu Ilhan

Email : s0559632@htw-berlin.de
Cansu.Ilhan@Student.HTW-Berlin.de

Instructor: Prof. Dr. Debora Weber-Wulff

Laboratory Report

Exercise 3: Execution times

Lab-plan

This lab was about analysing different fragments of code and testing their complexity. We had to run them for a number of times and compare the results with our guesses. Then we implemented a simple prime method and ran tests to determine different outcomes.

Assignment

1. For each of the following eight program fragments, do the following:
 1. Give a Big-Oh analysis of the running time for each fragment. (30 min)
 2. Implement the code in a simple main class and run it for several values of N, including 10, 100, 1000, 10.000, and 100.000. (1h)
 3. Compare your analysis with the actual number of steps (i.e. the value of sum after the loop) for your report. (25 min)

// Fragment #1

```
for ( int i = 0; i < n; i ++)  
    sum++;
```

// Fragment #2

```
for ( int i = 0; i < n; i ++)  
    for ( int j = 0; j < n; j ++)  
        sum++;
```

// Fragment #3

```
for ( int i = 0; i < n; i ++)  
    for ( int j = i; j < n; j ++)  
        sum++;
```

// Fragment #4

```
for ( int i = 0; i < n; i ++)  
    sum ++;  
    for ( int j = 0; j < n; j ++)  
        sum ++;
```

// Fragment #5

```
for ( int i = 0; i < n; i ++)  
    for ( int j = 0; j < n*n; j ++)  
        sum++;
```

// Fragment #6

```
for ( int i = 0; i < n; i ++)  
    for ( int j = 0; j < i; j ++)  
        sum++;
```

// Fragment #7

```
for ( int i = 1; i < n; i ++)  
    for ( int j = 0; j < n*n; j ++)  
        if (j % i == 0)  
            for (int k = 0; k < j; k++)  
                sum++;
```

// Fragment #8

```
int i = n;  
while (i > 1) {  
    i = i / 2;  
    sum++;  
}
```

The first thing we had to do was to write down our Big-Oh analysis about the given eight fragments. This took us a lot of time, because we wanted to understand where the difference between the fragments are exactly and what we should consider to determine the complexity. After a while our Big-Oh analysis in the lab looked like this:

Info 2 (ii) 28.10.19

Give Big-Oh Analysis

Fragment #1
The complexity of the algorithm is $O(N)$ linear
We have only one for loop and we increment at each step

Fragment #2
We have 2 for loops $\rightarrow O(N^2)$

Fragment #3
different from #2, because of 2nd for loop (int j = 1)
 $O(N^3)$

Fragment #4
 $O(N^2)$ quadratic

Fragment #5
 $O(N^3)$, we have $n*n$ so we thought the growth rate is getting high fast

Fragment #6

```

for (int i = 0; i < n; i++)
    for (int j = 0; j < i; j++)
        sum++;
    
```

$n=3$

1	1
1	1

i=0	i=1	i=2
j=0	j=1	j=2

$O(N^2)$

Fragment #7
 $O(N)$

Fragment #8
~~linear~~ $O(N \log N)$

When we were giving Big-Oh analysis, we were mostly looking at the for loops and the conditions regarding "n" in them. Talking about each of the fragments made us come up with different ideas and notice something that we didn't notice before.

Now that we were done writing our thoughts down, we started Eclipse and created a Java project with a class named ExecutionTimes. We had some problems creating the main method, because it was giving us an error when we tried to run the program. We asked Timo for some help and we were able to fix our problem. At first we thought that we need to check how much time in milliseconds the fragments take to execute for different values of **N**. Our initial program looked like this:



```
1 public class ExecutionTimes {
2
3     public static int sum;
4     public static int n;
5
6     public static void main(String[] args) {
7         ExecutionTimes test = new ExecutionTimes();
8         n = 0;
9         long sum = 0;
10        long startOfExecution = System.currentTimeMillis();
11        test.fragment1(n = 10);
12        long endOfExecution = System.currentTimeMillis();
13        System.out.print("The fragment took: ");
14        System.out.print(endOfExecution-startOfExecution);
15        System.out.print(" ms to execute.");
16    }
17    private int fragment1(int n) {
18        // Fragment 1
19        for (int i = 0; i < n; i++)
20            sum++;
21        return sum;
22    }
23 }
```

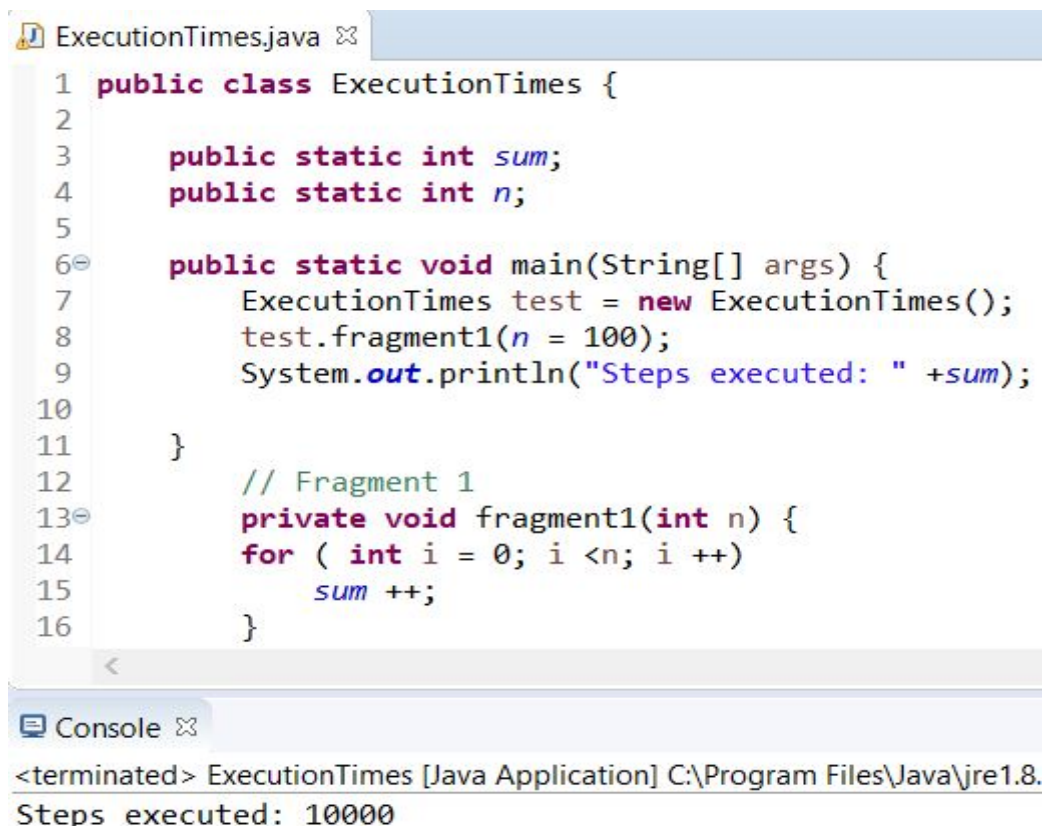
Console

<terminated> ExecutionTimes [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw
The fragment took: 0 ms to execute.

We tested the fragments and looked at the milliseconds it was taking to run the program. However, after a while we started noticing weird numbers when we were testing the fragments. Sometimes we were getting negative numbers for the milliseconds and sometimes for a higher value of **N** it took less time than for the lower value of the same fragment.

We were sure that our approach is not correct and we stopped for a minute to talk about the problem. At this moment we were not sure where we went wrong, but soon after that we heard Prof. Weber-Wulff talking to the other students in the lab, that this is not the way we should be testing the fragments. We realized that we made the same mistake by checking the milliseconds it takes for the execution of a fragment. We understood that we were supposed to count and return the steps it takes and not the time. Returning the time would be a very bad idea, because different machines would have different times and it wouldn't really help us with testing at all. At this time we didn't have much time left in the lab so we made sure to save everything and we both agreed to continue working from home.

When we started working on the program again, we first discussed exactly what we want to do and how we are going to change our program. We removed the previous counters for milliseconds and changed the program to the following:



The screenshot shows an IDE with a file named 'ExecutionTimes.java'. The code defines a class 'ExecutionTimes' with two static integers, 'sum' and 'n', and a 'main' method. The 'main' method creates an instance of 'ExecutionTimes', calls 'fragment1' with 'n = 100', and prints the value of 'sum'. The 'fragment1' method is a private void that takes an integer 'n' and increments 'sum' for each integer from 0 to n-1. Below the code, the console output shows the program terminated successfully and printed 'Steps executed: 10000'.

```
1 public class ExecutionTimes {
2
3     public static int sum;
4     public static int n;
5
6     public static void main(String[] args) {
7         ExecutionTimes test = new ExecutionTimes();
8         test.fragment1(n = 100);
9         System.out.println("Steps executed: " + sum);
10    }
11
12    // Fragment 1
13    private void fragment1(int n) {
14        for ( int i = 0; i < n; i ++ )
15            sum ++;
16    }
17 }
```

<terminated> ExecutionTimes [Java Application] C:\Program Files\Java\jre1.8.
Steps executed: 10000

After fixing our error and changing the code a bit, we were able to see how many steps were getting executed. We started testing the fragments and then wrote down the results to see how the growth rate changes for all fragments.

We had the problem that the last two values of Fragment5 and Fragment7 took a really long time. We decided to wait for a while and see if the program is going to respond.

The result of Fragment 5 for N = 10.000 needed 15 minutes.

The result of Fragment 7 for N = 1000 needed around 30 minutes.

We tried to test N = 100.000 for Fragment 5, but the process was taking really long that we had to terminate it. Same happened with Fragment 7 for N = 10.000 and N = 100.000.

When we tested Fragment6 for N = 100.000 we noticed that the result was exceeding the upper bound of int so we had to change from int to long to run that test. The results of Fragment 8 really surprised us, because we did not expect such values.

We decided to write the results into a table, so we could have a better overview.

Our results for every fragment:

N =	10	100	1000	10.000	100.000
Fragment 1	10	100	1000	10.000	100.000
Fragment 2	100	10.000	1.000.000	100.000.000	10.000.000.000
Fragment 3	55	5.050	500.500	50.005.000	5.000.050.000
Fragment 4	20	200	2.000	20.000	200.000
Fragment 5	1000	1.000.000	1.000.000.000	1.000.000.000.000	X
Fragment 6	45	4.950	499.500	49.995.000	4.999.950.000 ->1.000.000.000.000
Fragment 7	14002	258845742	3742257028683	X	X
Fragment 8	3	6	9	13	16

After looking at the table we were able to get a better overview of the growth rate for each fragment. We decided to compare our initial analysis with the actual result and we created another table for it.

Comparison of analysis and actual result:

Fragment#	Our guess	Actual result
Fragment 1	$O(N)$	$O(N)$
Fragment 2	$O(N^2)$	$O(N^2)$
Fragment 3	$O(N^2)$	$O(N)$
Fragment 4	$O(N^2)$	$O(N)$
Fragment 5	$O(N^3)$	$O(N^3)$
Fragment 6	$O(N^2)$	$O(N)$
Fragment 7	$O(2^N)$	$O(N!)$
Fragment 8	$O(\log N)$	$O(\log N)$

2. A **prime number** has no factors besides 1 and itself. Do the following:

2.1 Write a simple method `public static bool isPrime (int n) {...}` to determine if a positive integer N is prime. (45 min)

We remembered that In the first semester we already had an exercise in which we worked with prime numbers, so we made sure to revisit that and see what we did already. We then wrote a simple program in Eclipse to determine whether a given integer is prime or not.


```
*PrimeNumber.java
34  /* A method to check if an Integer is prime or not.
35  */
36  /* @param n Integer to check
37  /* @return true if n is prime, false otherwise with message
38  */
39  public static boolean isPrime(int n)
40  {
41      if (n <= 0)
42      {
43          System.out.println("You should use a positive number.");
44          return false;
45      }
46
47      int divisor = 2;
48      while(divisor < n )
49      {
50          if(n % divisor == 0)
51          {
52              System.out.println("Number "+ n + " is not prime.");
53              return false;
54          }
55
56          divisor = divisor + 1;
57      }
58      System.out.println("Number "+ n + " is prime.");
59      return true;
60  }
```

<terminated> PrimeNumber [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (N
Number 7 is prime.

We added some more text to make it more clear, when a given number is prime and when it isn't. We proceeded to test our method and it worked as we wanted it to, so we continued with the next task.

2.2 In terms of N , what is the worst-case running time of your program? (15 min)

When we read through the exercise we were a bit confused, because at first we couldn't understand the task well. In this exercise we were out of ideas, because we were only thinking about the while loop and what it does and we didn't think about the prime numbers themselves. Our initial idea was that since we are using int, at some point we are going to reach the upper bound and would not be able to execute further. Since we were not sure how to formulate our answer we decided to check

for more information about this online and found this site:

<https://poanchen.github.io/blog/2018/11/23/how-to-calculate-time-complexity-of-an-algorithm>

On that page we saw an example and an explanation about the complexity of our prime program. We got a much clearer idea why the **worst-case running time is $O(\sqrt{n})$** after reading through the information. In addition to that, we also thought that n should be prime to have the worst-case scenario.

2.3 Let B equal the number of bits in the binary representation of N . What is relationship between B and N ? (30 min)

For this exercise we discussed our approach and decided to create some additional methods that are going to help us find out if there is any relationship between B and N . We found about the method `toBinaryString()` in the Java API about `Integer` and we were sure that this is going to help us return the integer as a string representation as an unsigned integer in base 2. We created a method "toBinString" that takes a number "n" and returns it after calling the `toBinaryString` method on it.

```
25 private String toBinString(int n) {  
26     String first = Integer.toBinaryString(n);  
27     return first;  
28 }
```

Then we created another method "amountOfBits" to return the length of the bits needed to represent the `Integer`. We used the `length()` method for this after using the `toBinString()` method on the `Integer`. Our method looked like this.

```
private int amountOfBits(int n) {  
    String first = toBinString(n);  
    int length = first.length();  
    return length;  
}
```

After that we needed another method to print our results, so we wrote the following method. We made sure to add proper additional information so that after printing the results it's easily readable.

```
35 private void analyzeBinString(int n) {  
36     System.out.println("Binary representation: "+ toBinString(n));  
37     System.out.println("Amount of bits: "+ amountOfBits(n));  
38     System.out.println();  
39 }  
40 }
```

We then tested our method for different values and it was working well.

```
primeTest.java  
5 public static void main(String[] args) {  
6     primeTest prime = new primeTest();  
7     prime.analyzeBinString(10);  
8     prime.analyzeBinString(100);  
9     prime.analyzeBinString(1000);  
10    prime.analyzeBinString(10000);  
11    prime.analyzeBinString(100000);  
12    prime.analyzeBinString(1000000);  
13 }
```

We also had the following printed in the console after testing the method.

```
Binary representation: 1010
Amount of bits: 4

Binary representation: 1100100
Amount of bits: 7

Binary representation: 1111101000
Amount of bits: 10

Binary representation: 10011100010000
Amount of bits: 14

Binary representation: 11000011010100000
Amount of bits: 17

Binary representation: 11110100001001000000
Amount of bits: 20
```

After running some tests we came to the conclusion that there is a relationship between **B** and **N**. The number of bits when we turn the number into its binary representation, or also **B**, shows us the power of 2 of the number we are testing with. In our case B is the logarithm of N+1. We could also write the following:
 $B = O(\log N)$.

2.4 In terms of B, what is the worst-case running time of your program? (5min)

We both agreed that $O(N)$ has to be less than 2^B . In that case we thought the worst-case running time has to be **$O(2^B)$** .

2.5 Compare the running times needed to determine if a 20-bit number and a 40-bit number are prime by running 100 examples of each through your program. Report on the results in your lab report. You can use Excel to make some diagrams if you wish. (40 min)

To solve this task we needed some help from Katja's group. They gave us the hint to use a method and pass a BigInteger as a parameter and also about using Random to fill in the method with random 20 and 40-bit numbers. We discussed it for a while and after that we created a new method called "isPrimeBigInt" that takes something of type BigInteger. Our method at first looked like this.

```
45 public static boolean isPrimeBigInt(BigInteger bigInt)
46 {
47     if (bigInt.compareTo(BigInteger.valueOf(1)) == -1)
48     {
49         System.out.println("You should use a positive number.");
50         return false;
51     }
52
53     BigInteger divisor = BigInteger.valueOf(2);
54
55     while(divisor.compareTo(bigInt) == -1)
56     {
57         if(bigInt.mod(divisor) == BigInteger.valueOf(0))
58         {
59             return false;
60         }
61         divisor = divisor.add(BigInteger.valueOf(1));
62     }
63     return true;
64 }
65 }
```

Now that we had this method, we imported Random from the java.util and wrote a new field in our class.

```
1 import java.util.Random;
2
3
4
5 public class PrimeNumber {
6
7     private Random random;
8 }
```

We also looked at the Java API about BigInteger and found the following constructor.

```
BigInteger(int numBits, Random rnd)
```

Constructs a randomly generated BigInteger, uniformly distributed over the range 0 to $(2^{\text{numBits}} - 1)$, inclusive.

We decided to use it and wrote a new method called "bigIntegerAlgorithm" that takes the amount of bits as a parameter. We included messages in the system.out to properly see what exactly gets executed.

```
72 public void bigIntegerAlgorithm(int bit)
73 {
74     for (int i = 0; i < 100; i++)
75     {
76         BigInteger bigInt = new BigInteger(bit, random);
77         System.out.println("The number " + bigInt + " is prime: " + isPrimeBigInt(bigInt) + " !");
78     }
79 }
80 }
```

After that we created a new Random() in our main method and called the bigIntegerAlgorithm() method for 20 bits.

```
10 public static void main(String[] args) {
11     primeTest prime = new primeTest();
12
13     prime.random = new Random();
14     prime.bigIntegerAlgorithm(20);
15 }
```

After running our program we had the following result.

```
primeTest.java
1 import java.math.BigInteger;
2 import java.util.Random;
3
4 public class primeTest {
5
6     private Random random;
7
8
9     public static void main(String[] args) {
10         primeTest prime = new primeTest();
11
12         prime.random = new Random();
13         prime.bigIntegerAlgorithm(20);
14     }
15     public boolean isPrime(int x) {
16
17     }
18 }
```

```
Console
<terminated> primeTest [Java Application] C:\Program Files\Java\jre1.
The number 318734 is prime: false !
The number 875346 is prime: false !
The number 915905 is prime: false !
The number 254530 is prime: false !
The number 50791 is prime: false !
The number 728599 is prime: false !
The number 999919 is prime: false !
The number 56087 is prime: true !
The number 510427 is prime: false !
The number 563164 is prime: false !
The number 13542 is prime: false !
The number 218694 is prime: false !
The number 385636 is prime: false !
The number 939075 is prime: false !
The number 385513 is prime: false !
The number 807727 is prime: false !
The number 816282 is prime: false !
The number 323479 is prime: false !
The number 943966 is prime: false !
The number 710397 is prime: false !
The number 485004 is prime: false !
The number 570745 is prime: false !
The number 42099 is prime: false !
```

We also tested it with an input of 40 bits and got the following result.


```

9 public static void main(String[] args) {
10     primeTest prime = new primeTest();
11
12     prime.random = new Random();
13     prime.bigIntegerAlgorithm(40);
14 }

```

Console

primeTest [Java Application] C:\Program Files\Java\jre1.8.0_201\

```

The number 131070444395 is prime: false !
The number 1010980806859 is prime: false !
The number 417772134456 is prime: false !
The number 459584112331 is prime: false !
The number 388349747368 is prime: false !
The number 826937088667 is prime: false !
The number 160742792661 is prime: false !
The number 444137441760 is prime: false !
The number 823684010564 is prime: false !
The number 834268225405 is prime: false !
The number 940364541475 is prime: false !
The number 36897646972 is prime: false !
The number 1032109495762 is prime: false !
The number 538260889909 is prime: false !
The number 667340368804 is prime: false !
The number 971853013673 is prime: false !
The number 150011092453 is prime: false !
The number 196488846382 is prime: false !
The number 372802312223 is prime: false !

```

In the end we added some more lines to the main method to check how long it takes for the method to execute. We decided to implement it in the same way we tried for the first exercise and wrote the following.

```

9 public static void main(String[] args) {
10     primeTest prime = new primeTest();
11
12     long startOfExecution = System.currentTimeMillis();
13     prime.random = new Random();
14     prime.bigIntegerAlgorithm(20);
15     long endOfExecution = System.currentTimeMillis();
16
17     System.out.print("The method took ");
18     System.out.print(endOfExecution-startOfExecution);
19     System.out.print(" ms to execute.");
20 }

```


After changing the method we tested it again and tried it first for 20-bit Integers. We were getting different results, since we were using Random.

```
Console
<terminated> primeTest [Java Application] C:\Program Files\J
The number 866437 is prime: false !
The number 342128 is prime: false !
The number 834966 is prime: false !
The number 863580 is prime: false !
The number 639373 is prime: false !
The number 48095 is prime: false !
The number 451459 is prime: false !
The number 814859 is prime: true !
The number 202146 is prime: false !
The number 306834 is prime: false !
The number 37451 is prime: false !
The number 423202 is prime: false !
The number 1041909 is prime: false !
The number 524872 is prime: false !
The number 1030926 is prime: false !
The number 318166 is prime: false !
The number 363042 is prime: false !
The number 443541 is prime: false !
The number 353015 is prime: false !
The number 628150 is prime: false !
The number 188642 is prime: false !
The number 365988 is prime: false !

The method took 353 ms to execute.
```

```
Console
<terminated> primeTest [Java Application] C:\Program
The number 539232 is prime: false !
The number 301525 is prime: false !
The number 428398 is prime: false !
The number 216190 is prime: false !
The number 121027 is prime: false !
The number 958884 is prime: false !
The number 68210 is prime: false !
The number 95451 is prime: false !
The number 837925 is prime: false !
The number 252173 is prime: true !
The number 877362 is prime: false !
The number 232400 is prime: false !
The number 11170 is prime: false !
The number 154462 is prime: false !
The number 182803 is prime: true !
The number 1045312 is prime: false !
The number 398299 is prime: false !
The number 146681 is prime: true !
The number 173797 is prime: false !
The number 36569 is prime: false !
The number 804853 is prime: false !
The number 146367 is prime: false !

The method took 401 ms to execute.
```

After that we tried the same for 40-bit Integers. We waited for a while, but the program kept on running and we had to terminate the process.

Personal Reflection

Cansu Ilhan:

This exercise was really different with those kinda algorithms. We had little problems at the beginning, but after solving them there were a lot of fun. But still, it was very hard to understand. We needed a lot of time, for example during the first task. We just waited more than one 30 min for some fragments.

Pavel:

This exercise gave me a much better understanding about algorithms and analyzing their complexity. With additional research i learned more about prime numbers and some of their functions. I learned about BigInteger and the method toBinaryString, which helped us with the second exercise. It was interesting to see the different results after testing our program in the second exercise.

Appendix

Sources:

<https://poanchen.github.io/blog/2018/11/23/how-to-calculate-time-complexity-of-an-algorithm>

Pre-Lab

1. Which program has the better guarantee on the running time for large values of N ($N > 10\,000$)? $-150N \log N$
2. Which program has the better guarantee on the running time for small values of N ($N < 100$)? N^2
3. Which program will run faster on average for $N = 1000$?

4. Is it possible that program B will run faster than program A on all possible inputs?: no

2. An algorithm takes 0.5 ms for input size 100. How long will it take for input size 500 if the running time is the following:

1. linear = 2,5ms
2. $O(N \log N)$ = 3,4ms
3. quadratic = 12,5ms
4. cubic = 62,5ms

3. An algorithm takes 0.5 ms for input size 100. How large a problem can be solved in 1 min if the running time is the following:

1. linear = 12,000,000 ms
2. $O(N \log N)$ = 3,656,807 ms
3. quadratic = 34,641ms
4. cubic = 4,932.

4. Order the following functions by growth rate, and indicate which, if any, grow at the same rate.:

N , square root of N , $N^{1.5}$, N^2 , $N \log N$, $N \log \log N$, $N \log^2 N$, $N \log(N^2)$, $2/N$, $2N$, $2N/2$, $37N^3$, $N^2 \log N$.

$= 2/N < 37 < \sqrt{N} < N < N \log \log N < N \log N \leq N \log(N^2) < N \log 2N < N^{1.5} < N^2 < N^2 \log N < N^3 < 2N/2 < 2$

Code:

Task 1:

```
public class ExecutionTimes {

    public static long sum;
    public static int n;

    public static void main(String[] args) {
        ExecutionTimes test = new ExecutionTimes();
        test.fragment6(n = 100000);
        System.out.println("Steps executed: " + sum);
    }

    // Fragment 1
    private void fragment1(int n) {
        for (int i = 0; i < n; i++)
            sum++;
    }
}
```

```

// Fragment 2
private void fragment2(int n) {
for ( int i = 0; i < n; i ++)
    for ( int j = 0; j < n; j ++)
        sum++;
}
private long fragment3(int n) {
    // Fragment #3
    for ( int i = 0; i < n; i ++)
        for ( int j = i; j < n; j ++)
            sum++;
    return sum;
}

private long fragment4(int n) {
    // Fragment #3
    for ( int i = 0; i < n; i ++)
        for ( int j = i; j < n; j ++)
            sum++;
    return sum;
}

// Fragment #5
private void fragment5(int n) {
for ( int i = 0; i < n; i ++)
    for ( int j = 0; j < n*n; j ++)
        sum++;
};

// Fragment #6
private void fragment6(int n) {
for ( int i = 0; i < n; i ++)
    for ( int j = 0; j < i; j ++)
        sum++;
};
private long fragment7(int n) {
// Fragment #7
for ( int i = 1; i < n; i ++)
    for ( int j = 0; j < n*n; j ++)
        if (j % i == 0)
            for (int k = 0; k < j; k++)
                sum++;
return sum;
}
private void fragment8(int n) {
    int i = n;
    while (i > 1) {
        i = i / 2;
        sum++;
    }
}
}
}

```

Task 2:

```
import java.math.BigInteger;
import java.util.Random;

public class primeTest {

    private Random random;

    public static void main(String[] args) {
        primeTest prime = new primeTest();

        long startOfExecution = System.currentTimeMillis();
        prime.random = new Random();
        prime.bigIntegerAlgorithm(40);
        long endOfExecution = System.currentTimeMillis();
        System.out.println();
        System.out.print("The method took ");
        System.out.print(endOfExecution-startOfExecution);
        System.out.print(" ms to execute.");
    }

    public boolean isPrime(int x) {
        int begin = 2;
        while(begin<x) {
            if(x % begin == 0) {
                System.out.println("The number " + x + " is not prime.");
                return false;
            }
            begin++;
        }
        System.out.println("The number " + x + " is prime.");
        return true;
    }

    private String toBinString(int n) {
        String first = Integer.toBinaryString(n);
        return first;
    }

    private int amountOfBits(int n) {
        String first = toBinString(n);
        int length = first.length();
        return length;
    }

    private void analyzeBinString(int n) {
        System.out.println("Binary representation: "+ toBinString(n));
        System.out.println("Amount of bits: "+ amountOfBits(n));
        System.out.println();
    }

    public static boolean isPrimeBigInt(BigInteger bigInt)
    {
        if (bigInt.compareTo(BigInteger.valueOf(1)) == -1)
        {
```

```

        System.out.println("You should use a positive number.");
        return false;
    }

    BigInteger divisor = BigInteger.valueOf(2);

    while(divisor.compareTo(bigInt) == -1)
    {
        if(bigInt.mod(divisor) == BigInteger.valueOf(0))
        {
            return false;
        }
        divisor = divisor.add(BigInteger.valueOf(1));
    }
    return true;
}

public void bigIntegerAlgorithm(int bit)
{
    for (int i = 0; i < 100; i++)
    {
        BigInteger bigInt = new BigInteger(bit, random);
        System.out.println("The number " + bigInt + " is prime: " +
isPrimeBigInt(bigInt) + " !");
    }
}
}

```