

Lab Report

EXERCISE 2: HISTOGRAM

	Name	Titel des Kurses	Datum
Juri Wiechmann	571085	Prof. Dr.	03.11.2019
Pascal Radtke	571625	Weber-Wulff	
		Info 2 Group 2	

Index

Introduction

Pre-lab

1. Programs A and B are analysed and are found to have worst-case running...
2. An algorithm takes 0.5ms for input size 100. How long will it take for input size...
3. An algorithm take 0.5ms for input size 100. How large a problem can be solved...
4. Order the following functions by growth rate and indicate which, if any, grow...

Assignments

1. For each of the following eight program fragments, do the following:...
2. A **prime number** has no factors besides 1 and itself. Do the following:...
3. Try the sequence that is generated by the following method:...
4. The Sieve of Eratosthenes is a method used to compute all primes less...

Reflections

Pascal
Juri

Introduction

This week's exercise was about algorithms and the complexity they can get. Which was introduced to us in the lectures. This week many questions were hard to understand in the right way. We asked around and hoped we got the right understanding. This week was not that much about coding more about understanding an algorithm and analyse it.

PRE-LAB

1. Programs A and B are analyzed and are found to have worst-case running times no greater than $150 N \log N$ and N^2 , respectively. Answer the following questions, if possible:

- a. Which program has the better guarantee on the running time for large values of N ($N > 10\,000$)?

You just put 10.000 into both functions and look which one is worse(higher) for.

For $N = 10.000$:

$$\begin{aligned} 150 N \log N &= \sim 20.000.000 \\ N^2 &= 100.000.000 \end{aligned}$$

So $150 N \log N$ is better.

- b. Which program has the better guarantee on the running time for small values of N ($N < 100$)?

For $N = 100$:

$$\begin{aligned} 150 N \log N &= \sim 100.000 \\ N^2 &= 10.000 \end{aligned}$$

So N^2 is better.

- c. Which program will run faster on average for $N = 1000$?

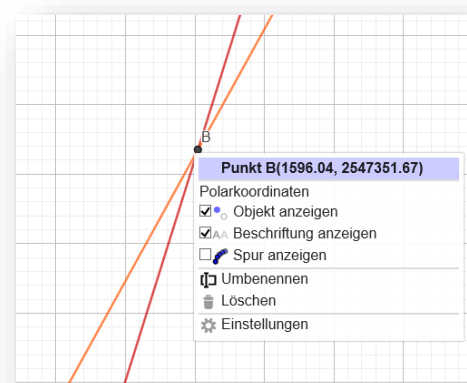
For $N = 1.000$

$$\begin{aligned} 150 N \log N &= \sim 1.500.000 \\ N^2 &= 1.000.000 \end{aligned}$$

So N^2 is better.

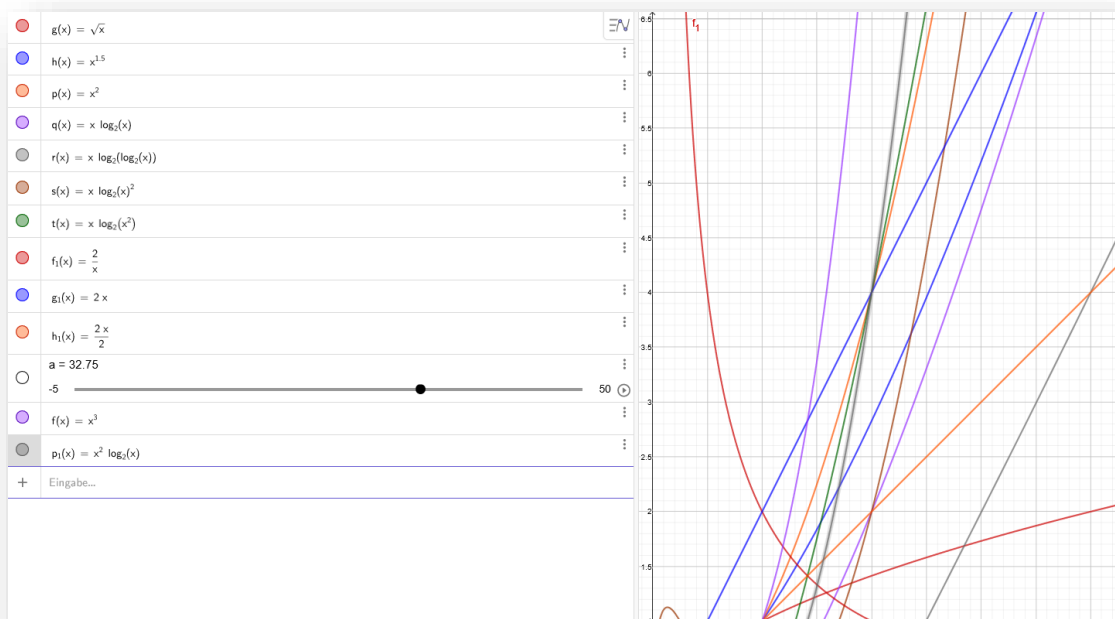
- d. Is it possible that program B will run faster than program A on all possible inputs?

In the picture you can see at which point $150 N \log N$ gets better than N^2 . The only way to say that N^2 is better on all possible inputs is too set a precondition that all inputs needs to be smaller than ~ 1596 .



2. An algorithm takes 0.5ms for input size 100. How long will it take for input size 500 if the running time is the following:
 - a. $O(N)$ (linear)
 $2.5\text{ms} = 500/100 * 0.5\text{ms}.$
 - b. $O(N \log N)$
 $3.37\text{ms} = 500 \log 500 / 100 \log 100 * 0.5\text{ms}.$
 - c. $O(N^2)$ Quadratic
 $12.5\text{ms} = 500^2 / 100^2 * 0.5\text{ms}.$
 - d. $O(N^3)$ cubic
 $62.5\text{ms} = 500^3 / 100^3 * 0.5\text{ms}.$
3. An algorithm takes 0.5ms for input size 100. How large a problem can be solved in 1 min if the running time is the following:
 - a. $O(N)$ (linear)
 $12.000.000 = 60.000\text{ms} * 100 / 0.5$
 - b. $O(N \log N)$
 $= e^{(60.000\text{ms} * 100 \log(100) / 0.5)}$
 - c. $O(N^2)$ Quadratic
 $34.641 = \sqrt{60.000\text{ms} * 100^2 / 0.5}$
 - d. $O(N^3)$ cubic
 $4.932 = \sqrt[3]{60.000\text{ms} * 100^3 / 0.5}$
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 , $N \log N$, $N \log \log N$, $N \log^2 N$, $N \log(N^2)$, $2/N$, $2N$, $2N/2$, $37N^3$, $N^2 \log N$

We will answer this question by showing everything in a graph:



Assignments

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.
 2. Implement the code in a simple main class and run it for several values of N, including 10, 100, 1.000, 10.000, and 100.000.
 3. Compare your analysis with the actual number of steps(i.e. the value of sum after the loop) for your report.

First we implemented all Fragments in our Code and run them. For each fragment we created a method and at the end of each method we added a call to the next method. Thanks to that we don't need eight different method calls in our loop and its looks more clean.

```
for(int i = 10; i<=100000; i*=10) {
    Fragment1(i);
}
```

We also changed the datatype from sum to BigInteger. It has no effect on the Fragments. We just wanted to be safe not to get to high sums that can't be saved in Integer. Then it looks like this:

```
public static void Fragment1(int n) {
    BigInteger sum = BigInteger.ZERO;
    for ( int i = 0; i < n; i ++ )
        sum = sum.add(BigInteger.ONE);
    System.out.println("#1 " + n + ":" + sum.toString());
    Fragment2(n);
}
```

So we took a look at the Fragments and tried to guess what our complexity could be. By printing the sum at the end of each Fragment we got the values, $O(N)$ for each N. All in all we put our solutions into this table:

Fragmente	Erwartung	Sum(N)						Realität
		10	100	1.000	10.000	100.000		
#1	$O(N)$	10	100	1000				$O(N)$
#2	$O(N^2-N)$	100	10000	1.000.000				$O(N^2)$
#3	$O(N \log N)$	55	5050	500500				$O((N^2+N)/2)$
#4	$O(N^2)$	10	100	1000				$O(N)$
#5	$O(N^2 \log N)$	1000	1000000	1000000000				$O(N^3)$
#6	$O(N \log N)$	45	4950	499500				$O((N^2-N)/2)$
#7	$O(2^N \log N)$	14002	258845742					
#8	$O(\log N)$	3	6	10	13	17		$O(\log N)$

- #1 Fragment This one was pretty easy because it's linear.
- #2 Fragment Our first guess was $O(N^2)$ too but there was #4 which confused us and we thought this must be $O(N^2)$. So now #2 needed to be lower than this and we just subtract linear.

- #3 Fragment For every step in our outer loop we need one less in the inner so we thought this could be $O(N \log N)$. It turns out, that it's some more complicated stuff.
- #4 Fragment Now we get to the tricky one. As we said we thought it's quadratic but missing brackets tricked us.
- #5 Fragment To be honest we can't remember why we said $O(N^2 \log N)$ it's obviously $O(N^3)$
- #6 Fragment This looks the same like #3. At the end it was slightly different.
- #7 Fragment It needed to be higher than $O(2^N)$. We also couldn't see any sign of Linear or exponential increase so we multiplied it by $\log N$. We talked to other classmates but no one was sure what the right answer could be.
- #8 Fragment So we know everything that is divided by 2 has something to do with $\log N$ so we guessed $O(\log N)$.

We couldn't get all the values for each N because it took way too long.

2. A Prime Number has no factors besides 1 and itself. Do the following:

a. Write a simple method:

Public static bool isPrime(int n){...}
to determine if a positive integer N is prime.

So we knew we were not the first one who are programming such a method. We knew we could code such method but out there are very more efficient algorithms and because this lab is about complexity we searched for the fastest way to test a number if it is prime(S1). But we wanted to be sure not to get any problems with Integers-max-values so we changed everything to a BigInteger to be safe:

```
//https://stackoverflow.com/questions/2385909/what-would-be-the-fastest-method-to-
test-for-primality-in-java
public static boolean isPrime (BigInteger n) {

    BigInteger THREE = BigInteger.TWO.add(BigInteger.ONE);
    if(0>n.compareTo(BigInteger.TWO))
        return false;
    if(0 == n.compareTo(BigInteger.TWO) || 0 == n.compareTo(THREE))
        return true;
    //https://www.tutorialspoint.com/java/math/biginteger_mod.htm
    boolean first = n.mod(BigInteger.TWO).compareTo(BigInteger.ZERO) == 0;
    boolean second = n.mod(THREE).compareTo(BigInteger.ZERO) == 0;
    if(first || second)
        return false;

    //https://www.geeksforgeeks.org/biginteger-sqrt-method-in-java/
    BigInteger sqrtN = n.sqrt().add(BigInteger.ONE);

    BigInteger SIX = new BigInteger("6");
    BigInteger negativOne = new BigInteger("-1");

    for(BigInteger i = SIX; sqrtN.compareTo(i) != -1; i = i.add(SIX)) {
        sum++;
        first = n.mod(SIX.add(negativOne)).compareTo(BigInteger.ZERO) == 0;
        second = n.mod(SIX.add(BigInteger.ONE)).compareTo(BigInteger.ZERO)==0;
        if(first||second) {
            return false;
        }
    }
    return true;
}
```

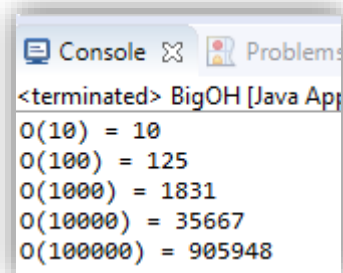
We needed a bit help to understand how to work with BigInteger(S₂), (S₃).

b. In terms of N, what is the worst-case running time of your program?

So first we wrote a method that lets us measure how long it takes. To test all numbers below a specific input to measure how the BigOh could look like:

```
public static BigInteger GetBigOH_isPrime(int n) {  
    BigInteger N = new BigInteger(Integer.toString(n));  
    BigInteger total0 = BigInteger.ONE;  
    for(BigInteger Index = BigInteger.ONE ; Index.compareTo(N) == -1;  
        Index = Index.add(BigInteger.ONE)){  
        sum = 1;  
        if(isPrime(Index)) {}  
        total0 = total0.add(new BigInteger(Integer.toString(sum)));  
    }  
    return total0;  
}  
  
for(int i = 10; i <= 100000; i *= 10) {  
    System.out.println("O(" + i + ") = " +  
        GetBigOH_isPrime(i).toString());  
}
```

We tested a lot numbers like the Fragments before and it turns out that it only need 1 or 2 steps with the parameter isn't a prime. If the parameter is a prime he start to count up sum.

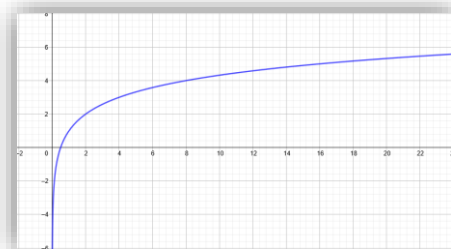


```
<terminated> BigOH [Java App  
O(10) = 10  
O(100) = 125  
O(1000) = 1831  
O(10000) = 35667  
O(100000) = 905948
```

c. Let B equal the number of bits in the binary representation of N. What is relationship between B and N?

$\log(N) + 1$ is the function we need to talk about here. Let N be on the horizontal axis and B on the vertical axis and it will look like this:

Every more bit(B) the value of N doubles.



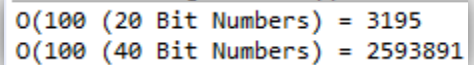
- d. In terms of B, what is the worst-case running time of your program?

Only if we find a Prime Number, B matters to us and the higher B is, the more time it consumes. Actually we tested a 100 decimal digits long number and it still took way under one second.

- e. 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.

So first we wrote a method that let us measure how many steps it would take. In a loop we went through 100 random Numbers with the right Bit-value:

```
//20 Bits numbers:  
BitPrime(20);  
//40 Bits numbers:  
BitPrime(40);  
...
```



```
O(100 (20 Bit Numbers) = 3195  
O(100 (40 Bit Numbers) = 2593891
```

```
public static void BitPrime (int Bits) {  
  
    BigInteger total0 = BigInteger.ONE;  
  
    for(int i = 0; i<100;i++) {  
        BigInteger Twenty_Bits = new BigInteger(Bits, new Random());  
        sum = 1;  
        isPrime(Twenty_Bits);  
        total0 = total0.add(new BigInteger(Integer.toString(sum)));  
    }  
    System.out.println("O(100 (" + Bits + " Bit Numbers) = " +  
        total0.toString());  
}
```

3. Try the sequence that is generated by the following method:

```
public int sequence (int n)
{
    int sum = 0;
    int i = n;
    while (i > 1) {
        if (i%2 == 0) {
            i = i/2;
        } else {
            i = 3*i + 1;
        }
        sum++;
    }
    return sum;
}
```

Can you determine the complexity of the number of steps needed for the loop to terminate for various values of n? Can you find out more about this sequence?

So we implemented it in our code and tested the results that we will get for different N:

```
for(int i = 10; i<=10000; i++) {
    System.out.println(sequence(i));
}
```

It is really weird. It turns out that it will always reach 1 and the steps it takes is independent from the input size and the Bit length, if you can divide the value by 2 it will be else it gets multiply it by 3 and add one to it. This way it will always reach one after a random looking amount of steps.

4. The *Sieve of Eratosthenes* (S₄) is a method used to compute all primes less than N. Begin by making a table of integers 2 to N. Find the smallest integer i that is not crossed out. Print i (it is prime!) and cross out i, 2i, 3i, Terminate when i is greater than the square root of N. The running time has been shown to be $O(N \log \log N)$. Write a program to implement the Sieve and verify that the running time is as claimed. If you are really bored, animate this with a GUI like on the Wikipedia!

So for this one we used the Wikipedia link and read about it. There was also a Pseudocode which we used to code our method:

73
73
73
166
42
91
166
166
91
91
91
166
179
73
179
65
179
179
91
91
29

37
41
43
47
53
59
61
67
71
73
79
83
89
97
101
103
107
109
113
120

```
boolean[] Primes = Eratosthenes(120);
for(int i = 2;i<Primes.length;i++) {
    if(Primes[i]) {
        System.out.println(i);
    }
}
...
public static boolean[] Eratosthenes (int n){
    sum = 1;
    if(n<1) {
        return null;
    }
    boolean[] Primes = new boolean[n+1];
    Arrays.fill(Primes, true);
    for(int i = 2;i<Math.sqrt((double)n);i++){
        sum++;
        if(Primes[i]) {
            for(int j = (int) Math.pow(i, 2), k = 1; j<n; k++, j =
                (int) Math.pow(i, 2)+k*i) {
                Primes[j] = false;
                sum++;
            }
        }
    }
    return Primes;
}
```

We ran the method and counted the sum. It worked properly and showed us all primes. For some reason our outcome didn't represent a runtime of $O(N \log \log N)$. Even if we played around where to put our "sum++" we never reproduced our expected value. But also the last output that we got was wrong (120). The last output was all time N itself. We took a look at our code and the Pseudocode and we weren't be able to see a difference. Maybe the Pseudocode itself is not right or we understood something wrong.

Reflections

JURI

In this weeks' lab we got trained to analyse algorithm about their complexity. We were talking a lot about it in lecture. First time we worked a lot with BigInteger. Its simply to understand but still confusing to see it in code. I now at the after some people ask for a bit help and hints I saw there are more simple ways to organise BigInteger. Like you want to see if a value is lower than another. We used methods to compare but to use the equal method is more elegant.

PASCAL

This lab taught me how important the efficiency of your algorithm is. There are huge gaps between the execution times of our tested algorithm. I got used to JavaAPI because I had to find out how BigInteger gets implemented correctly. I still have problems analysing and rating algorithm by just looking at them. Sources:

S1: <https://stackoverflow.com/questions/2385909/what-would-be-the-fastest-method-to-test-for-primality-in-java>

S2: https://www.tutorialspoint.com/java/math/biginteger_mod.htm

S3: <https://www.geeksforgeeks.org/biginteger-sqrt-method-in-java/>

S4: https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

Appendix:

Java Code:

```
package Übung3;
```

```
import java.math.BigInteger;
```

```
import java.util.Arrays;
```

```
import java.util.Random;
```

```
public class BigOH {
```

```
    static BigInteger n;
```

```
    static int sum = 1;
```

```
    public static void main(String[] args) {
```

```

/*for(int i = 10;i<=100000;i*=10) {

    Fragment1(i);

}

for(int i = 10;i<=100000;i*=10) {

    System.out.println("O(" + i + ") = " +
GetBigOH_isPrime(i).toString());

}

//20 Bits numbers:

BitPrime(20);

//40 Bits numbers:

BitPrime(40);

for(int i = 10;i<=10000;i++) {

    System.out.println(sequence(i));

}

/*

*/

boolean[] Primes = Eratosthenes(120);

for(int i = 2;i<Primes.length;i++) {

    if(Primes[i]) {

        System.out.println(i);

    }

}

System.out.println("O(120) = " + sum);

```

```
}
```

```
//https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
```

```
public static boolean[] Eratosthenes (int n){  
    sum = 1;  
    if(n<1) {  
        return null;  
    }  
    //o = 2  
    boolean[] Primes = new boolean[n+1];  
    Arrays.fill(Primes, true);  
    for(int i = 2;i<Math.sqrt((double)n);i++){  
        sum++;  
        if(Primes[i]) {  
            for(int j = (int) Math.pow(i, 2), k = 1; j<n; k++, j = (int)  
Math.pow(i, 2)+k*i) {  
                Primes[j] = false;  
                sum++;  
            }  
        }  
    }  
    return Primes;  
}
```

```
public static int sequence(int n) {
```

```
    int sum = 0;
```

```
    int i = n;
```

```
    while (i > 1) {
```

```

        if (i%2 == 0) {
            i = i/2;
        }
        else {
            i = 3*i + 1;
        }
        sum++;
    }
    return sum;
}

```

```

public static void BitPrime (int Bits) {

```

```

    BigInteger totalo = BigInteger.ONE;

```

```

    for(int i = 0; i<100;i++) {

```

```

        BigInteger Twenty_Bits = new BigInteger(Bits, new Random());

```

```

        sum = 1;

```

```

        isPrime(Twenty_Bits);

```

```

        totalo = totalo.add(new BigInteger(Integer.toString(sum)));

```

```

    }

```

```

    System.out.println("O(100 (" + Bits + " Bit Numbers) = " +
totalo.toString());

```

```

}

```

```

public static BigInteger GetBigOH_isPrime(int n) {

```

```

    BigInteger N = new BigInteger(Integer.toString(n));

```

```

        BigInteger totalo = BigInteger.ONE;

        for(BigInteger Index = BigInteger.ONE ; Index.compareTo(N) == -1;
Index = Index.add(BigInteger.ONE)){

            sum = 1;

            if(isPrime(Index)) {}

            totalo = totalo.add(new BigInteger(Integer.toString(sum)));

        }

        return totalo;
    }

```

[//https://stackoverflow.com/questions/2385909/what-would-be-the-fastest-method-to-test-for-primality-in-java](https://stackoverflow.com/questions/2385909/what-would-be-the-fastest-method-to-test-for-primality-in-java)

```

public static boolean isPrime (BigInteger n) {

```

```

    BigInteger THREE = BigInteger.TWO.add(BigInteger.ONE);

```

```

    if(o>n.compareTo(BigInteger.TWO))

```

```

        return false;

```

```

    if(o == n.compareTo(BigInteger.TWO) || o == n.compareTo(THREE))

```

```

        return true;

```

```

//https://www.tutorialspoint.com/java/math/biginteger_mod.htm
boolean first = n.mod(BigInteger.TWO).compareTo(BigInteger.ZERO)
== 0;

boolean second = n.mod(THREE).compareTo(BigInteger.ZERO) == 0;
if(first || second)
    return false;

//https://www.geeksforgeeks.org/biginteger-sqrt-method-in-java/
BigInteger sqrtN = n.sqrt().add(BigInteger.ONE);

BigInteger SIX = new BigInteger("6");
BigInteger negativOne = new BigInteger("-1");

for(BigInteger i = SIX; sqrtN.compareTo(i) != -1; i = i.add(SIX)) {
    sum++;
    first =
n.mod(SIX.add(negativOne)).compareTo(BigInteger.ZERO) == 0;
    second =
n.mod(SIX.add(BigInteger.ONE)).compareTo(BigInteger.ZERO) == 0;
    if(first||second ) {
        return false;
    }
}
return true;
}

```

```

public static void Fragment1(int n) {
    BigInteger sum = BigInteger.ZERO;

```

```

        for ( int i = 0; i < n; i ++)
            sum = sum.add(BigInteger.ONE);

        System.out.println("#1 " + n + ":" + sum.toString());

        Fragment2(n);
    }

    public static void Fragment2(int n) {

        BigInteger sum = BigInteger.ZERO;

        for ( int i = 0; i < n; i ++)
            for ( int j = 0; j < n; j ++)
                sum = sum.add(BigInteger.ONE);

        System.out.println("#2 " + n + ":" + sum);

        Fragment3(n);
    }

    public static void Fragment3(int n) {

        BigInteger sum = BigInteger.ZERO;

        for ( int i = 0; i < n; i ++)
            for ( int j = i; j < n; j ++)
                sum = sum.add(BigInteger.ONE);

        System.out.println("#3 " + n + ":" + sum);

        Fragment4(n);
    }

    public static void Fragment4(int n) {

        BigInteger sum = BigInteger.ZERO;

        for ( int i = 0; i < n; i ++)
            sum = sum.add(BigInteger.ONE);

            for ( int j = 0; j < n; j ++)
                sum.add(BigInteger.ONE);

        System.out.println("#4 " + n + ":" + sum);
    }

```



```

        Fragment5(n);
    }

    public static void Fragment5(int n) {
        BigInteger sum = BigInteger.ZERO;
        for ( int i = 0; i < n; i ++)
            for ( int j = 0; j < n*n; j ++)
                sum = sum.add(BigInteger.ONE);
        System.out.println("#5 " + n + ":" + sum);
        Fragment6(n);
    }

    public static void Fragment6(int n) {
        BigInteger sum = BigInteger.ZERO;
        for ( int i = 0; i < n; i ++)
            for ( int j = 0; j < i; j ++)
                sum = sum.add(BigInteger.ONE);
        System.out.println("#6 " + n + ":" + sum);
        Fragment7(n);
    }

    public static void Fragment7(int n) {
        BigInteger sum = BigInteger.ZERO;
        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 = sum.add(BigInteger.ONE);
        System.out.println("#7 " + n + ":" + sum);
        Fragment8(n);
    }
}

```

```
public static void Fragment8(int n) {  
    BigInteger sum = BigInteger.ZERO;  
    int i = n;  
    while (i > 1) {  
        i = i / 2;  
        sum = sum.add(BigInteger.ONE);  
    }  
    System.out.println("#8 " + n + ":" + sum);  
}  
}
```