Informatik 2 Group 2

Lab 3 Execution Times

By Bartholomäus Berresheim (s0568624) & Stepan Burlachenko (s0571718)

Index

Introduction

In the first half of this week's lab, we looked at algorithms and their execution time and in the second half we worked on prime numbers, the bit-size of numbers and the running time of numbers for different bit-sizes.

Pre-lab

Bartholomäus

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:

> **1.Which program has the better guarantee on the running time for large values of N (N > 10 000)?**
>
> To find the answer, we take a number N = 10 500, and calculate:
> A: 150*10500*log(10500) = 14.583.131
> B: 10500*10500 = 110.250.000
> We see that for N>10 000, A has a better running time than B.

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

To find the answer to that question, we take N = 50, and calculate:
A: 150*50*log(50) = 12.742
B: 50*50 = 2.500
We see that for N<100, B has a better running time than A.

3. Which program will run faster on average for N = 1000?
This cannot be determined.

4. Is it possible that program B will run faster than program A on all possible inputs?
It is not possible, as we have seen in the previous exercise, B is slower than A when it comes to large values.

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: 500/100*0.5 = 2.5 ms
2. O (N log N): 500 log(500)/100log(100)*0.5 = 3.4 ms
3. quadratic: $500^2/100^2*0.5$ = 12.5 ms
4. cubic: $500^3/100^3*0.5$ = 62.5 ms

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: 60000*100/0.5 = 12.000.000
2. O (N log N): e^(60000*100log(100)/0.5) = ?
3. quadratic: sqrt(60000*100^2/0.5) = 34.641
4. cubic: cbrt (60000*100^3/0.5) = 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, 37,N^3, N^2 log N

Ordered from lowest to highest growth rate, we get the following result:
2/N, 37, square root of N, [N, 2N/2], 2N, N log log N, N log N, N^1.5, N log (N^2), N $\log^2$ (N), N^2, N^2 log (N), N^3

(the square braces indicate that they grow at the same rate.)

Stepan

Stepan Burlachenko
28.10.2019

Exercise 3: Execution Times

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:

1. Which program has the better guarantee on the running time for large values of N (N > 10 000)? - **A**

2. Which program has the better guarantee on the running time for small values of N (N < 100)? = **B**

3. Which program will run faster on average for N = 1000? - **A**

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 – **2.5 ms**
2. O (N log N) – **3.36 ms**
3. quadratic – **12.5 ms**
4. Cubic – **62.5 ms**

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 – **12000000**
2. O (N log N) - ?
3. quadratic - **34641**
4. cubic - **10062**

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, 37,N^3, N^2 log N

2/N	N 2N/2	2N	N log log N	log N	N	$N^{1.5}$	N^1.5	N log(N^2)	N$\log^2$N	N^2	N^2 log N	N^3

Bartholomäus Berresheim 568624 02.11.19
Stepan Burlachenko 571718 Informatik 2 Group 2

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, 1000, 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 estimated the Big-Oh`s for each fragment. Then, in Eclipse, we created a class with a method for each fragment that prints out the sum after each calculation to the console. Whe then called the fragment methods each with the values 10, 100, 1000, 10000 and 100000 as parameters. We put the output from the console into tables, based on which we then calculated the running time for each fragment and later compared them with our estimations.

// Fragment #1

1. O(N)
2.

Steps	10	100	1.000	10.000	100.000
Fragment #1	10	100	1.000	10.000	100.000

3. As we can see, our analysis of the running time was right.

// Fragment #2

1. O(N^2)
2.

Steps	10	100	1.000	10.000	100.000
Fragment #2	100	10.000	1.000.000	100.000.000	10.000.000.000

3. As we can see, the results are as expected.

// Fragment #3

1. O(N)
2.

Steps	10	100	1.000	10.000	100.000
Fragment #3	55	5.050	500.500	50.005.000	5.000.050.000

3. The end result looks like it has a running time of (N^2+N)/2, which means our first assessment was wrong.

// Fragment #4

1. O(2N)
2.

Steps	10	100	1.000	10.000	100.000
Fragment #4	20	200	2.000	20.000	200.000

3. The end results show that our analysis was correct.

// Fragment #5

1. O(N³)
2.

Steps	10	100	1.000	10.000	100.000
Fragment #5	10^3	10^6	10^9	10^12	takes too long

3. As expected, the running time is N³.

// Fragment #6

1. O(N)
2.

Steps	10	100	1.000	10.000	100.000
Fragment #6	45	4.950	499.500	49.995.000	4.999.950.000

3. The end result shows a running time of (N²+N)/2-N, which means our first assessment was wrong.

// Fragment #7

1. O(N^5)
2.

Steps	10	100	1.000	10.000	100.000
Fragment #7	14.002	258.845.742	3742257028683	takes too long	takes too long

3. The running time is less than N^5 but it is difficult to pin-point it exactly, it is at least more than N^4.

// Fragment #8

1. O(log N)
2.

Steps	10	100	1.000	10.000	100.000
Fragment #8	3	6	9	13	16

3. The running time is as predicted.

[100 min]

1. 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.

To make a method isPrime() that determines if a positive integer is prime, we needed to implement 2 things. First of all, we needed to check if n is positive. We did that by implementing an if statement that checks if n is smaller than 0 and if that's the case, it returns false.

If n is positive, nothing happens and we continue with the for loop. The loop has the "i" parameter starting at 2 since we want to exclude 0 and 1, which aren't considered prime numbers.
Since a prime number is only divisible by itself, we added an if statement into the loop that checks if n%i is equal to 0, if that is true, it means that n is divisible by another smaller number and thus not a prime number.
If the loop ends without any intervention of the if-statement, it means that n is a prime number, so we return true.

```
    public static boolean isPrime (int n) {

        if (n<0) {
            System.out.println("n is not a positive number");
            return false;}

        for (int i =2; i<n; i++){

            if (n%i == 0)
            {
                return false;
            }
        }
        return true;
    }

}
```

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

The worst case running time would be if the for-loop runs without any interference from the if-statement, which corresponds to the running time of N.

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

B determines the range of values N can have.

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

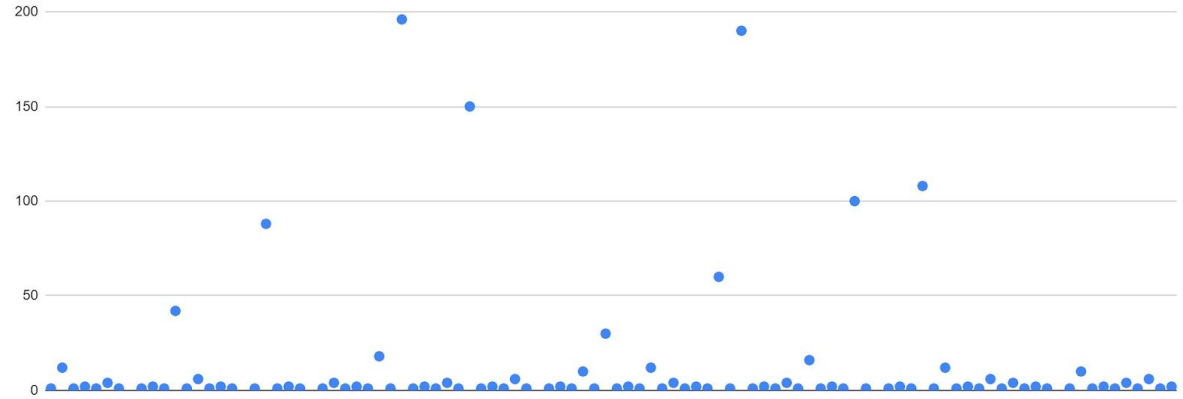
The bigger B the longer the running time, which means if B is too big it is possible for the running time to be indefinitely long.

e. Compare the running times needed to determine if a 20-bit number and a 40-bit number are prime by running 100 examples

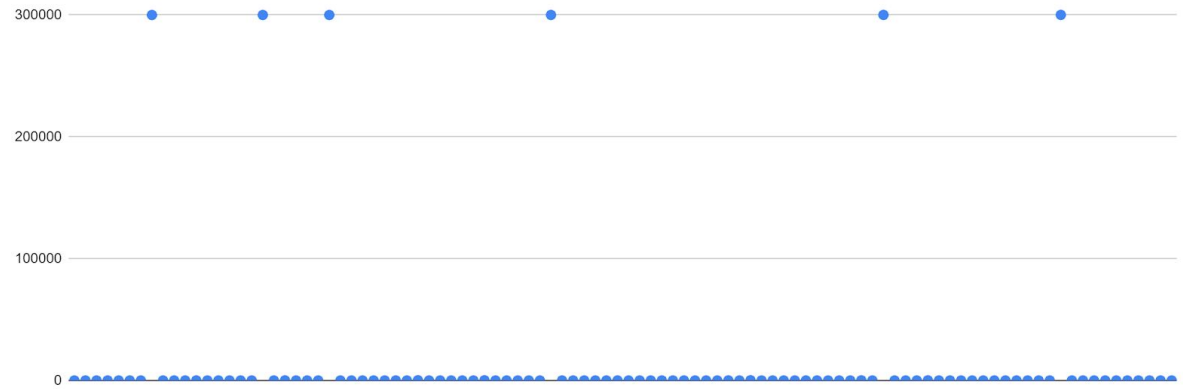
of each through your program. Report on the results in your lab report. You can use Excel to make some diagrams if you wish.

For this task we created 2 array lists, which we then filled with 20-bit and 40-bit numbers respectively with the use of for-loops.
We then proceeded by adding another for-loop in which we called the isPrime() method with one of the Arrays as parameters, then we added a step counter to the isPrime() method which would allow us to see the running time of the called isPrime() method from the before mentioned for-loop. After the loop was done, we took the values we got and created diagrams with them.

For the 20-bit numbers:
1. close-up (0-200 steps)



2. all values (0-310000)

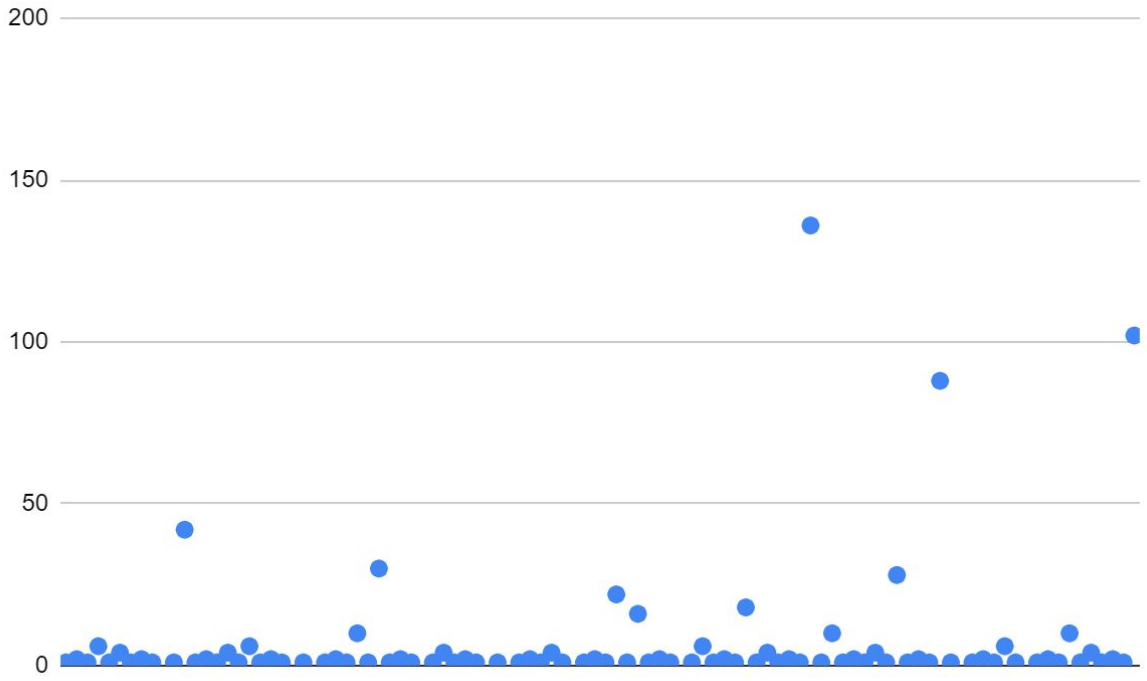


As we can see in the first diagram, most of the running times are below 20 and with a few exceptions going up to 200. In the second diagram we can see that we had 6 prime numbers in our ArrayList, which took over 300000 steps to confirm.

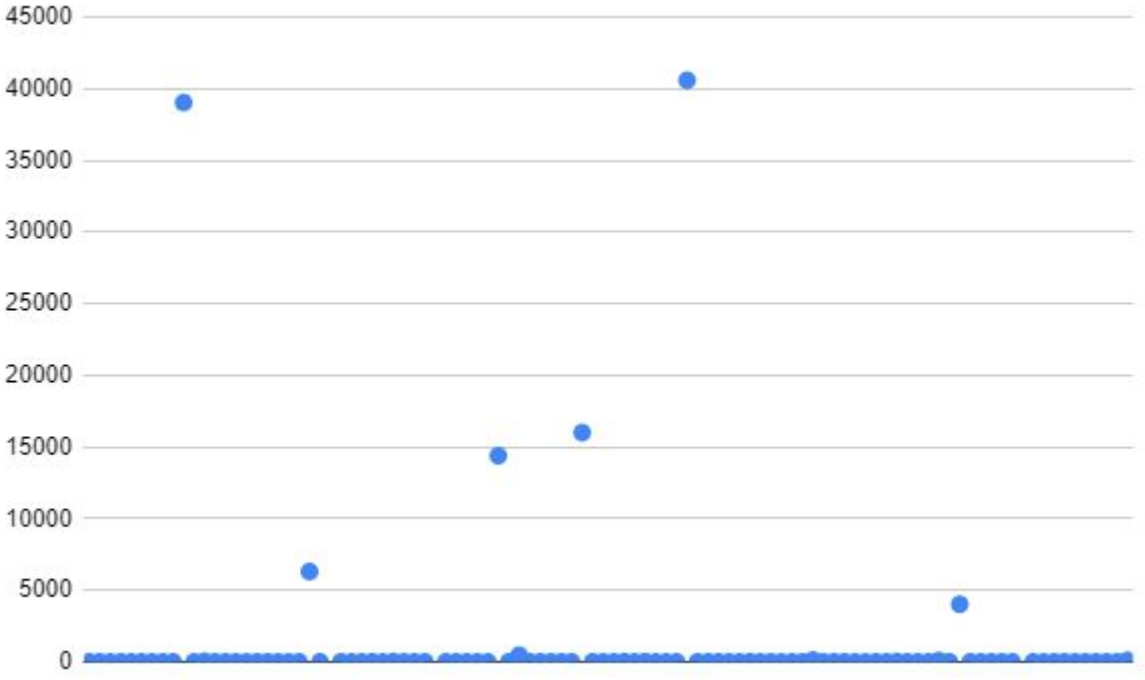
For the 40-bit numbers:

After running through all the 100 values, we noticed that we our 3 highest step counters all had a value of 4.29 billion, which would mean that we had the same prime number thrice in our Array. After some thinking, we noticed that it corresponds to 2^32, which is the max value of an unsigned Integer. It turned out that our counter was still an integer and not a long and that it had counted to its max value of 4.29 and then stopped the loop, instead of going up to 500 billion. We fixed that quickly and started the program again but the running time was way too long, now that the counter went up to 500 billion. Thankfully we were able to deduce the prime numbers from the position of the faulty entries in our table and to correct them.

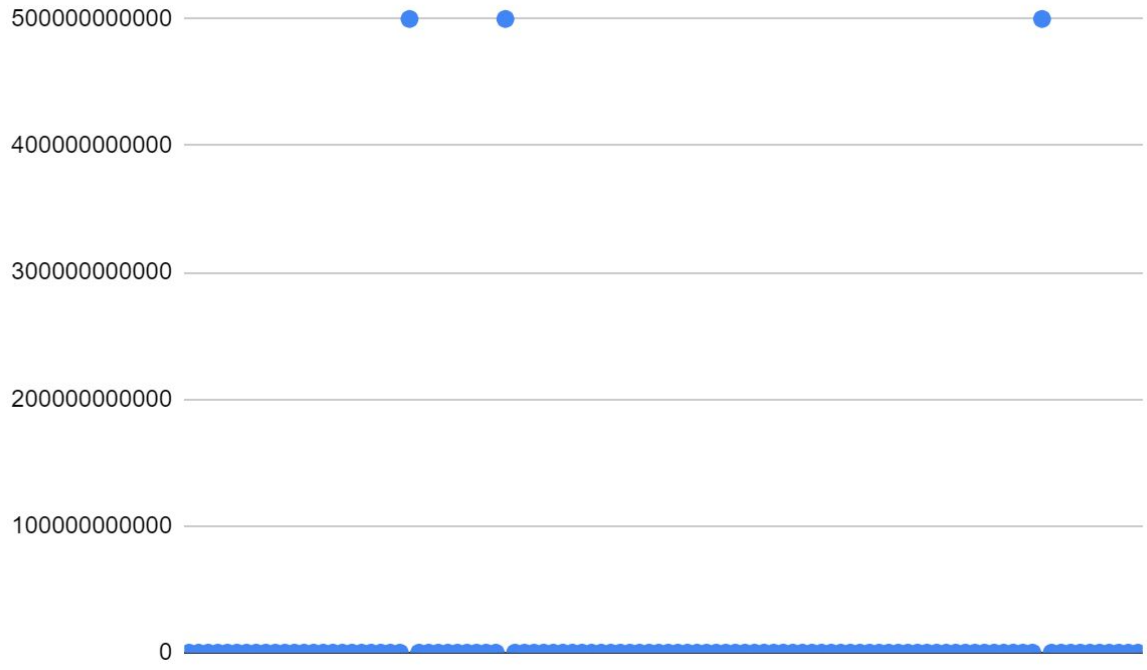
1. close-up (0-200 steps)



2. another close-up (0-45000)



3. all values (0-510 000 000 000 steps)



In the first diagram, we can see that once again, most running times stay below 20 with a couple going up to around 150. In the second diagram, we see that there are 6 dots spread between 45000 and 4000. And in the third diagram, we see that our ArrayList contained 3 prime numbers in between the 500 billion and 510 billion range.

Although we found more prime numbers in the 20-bit number Array than in the 40-bit one, it took longer to find 1 40-bit prime number than to go through 100 20-bit numbers.

(The link to our execution-time tables : https://docs.google.com/spreadsheets/d/1EuttRcUg-mkp_cOylKUlYkz1oQhIdN76QkjAziZAVEk/edit?usp=sharing)

[180 min]

Reflections

Stepan:
This lab was quite complicated; even the pre-lab tasks took me quite some time to solve since I had to do some extra reading. Finding the right way to calculate tasks 2 & 3 from there also required some time.
The actual lab tasks were not easy either. The first task was alright because the code for the fragments was available; it was the calculation part which required more attention, we had to do the calculations several times to make sure we got the numbers right.
The second task, especially the b)-e) parts were more challenging. But I think I managed to grasp the idea of Algorithms and their Complexity.
What surprised me is the inability of my computer to finish some fragments with large values even though the fragments seemed to be quite simple.

Bartholomäus:
This weeks lab started off a bit bumpy, since I wasn't able to attend classes for a couple days and I wasn't in any condition to work properly. After I had gotten better, I started working on the lab. The pre lab wasn't a problem apart from tasks 2 and 3, which confused me at first but I later got my head around it. The assignment 1 also had its ups and downs, some of the Big-Oh analysis were easily made, while others weren't easily made without running the program first. The second assignment wasn't a problem, although I'm not sure about the c and d questions.
In my opinion, the most important thing of this lab was to see examples of the different running times, so that we can get a feel of how we should

go on about programming in the future, when it comes to running time efficiency.

Code

```
package Ubung3;

public class Execution_Times {

    public static void main(String[] args) {

        int a = 10;
        int b = 100;
        int c = 1000;
        int d = 10000;
        int e = 100000;

        fragment_1 (a);
        fragment_1 (b);
        fragment_1 (c);
        fragment_1 (d);
        fragment_1 (e);
        }

    public static void fragment_1(int n) {
        long sum = 0;
        for ( int i = 0; i < n; i ++)
            sum++;
        System.out.println(sum); }


    public static void fragment_2(int n) {
        long sum = 0;
        for ( int i = 0; i < n; i ++)
            for ( int j = 0; j < n; j ++)
                sum++;
        System.out.println(sum);}


    public static void fragment_3(int n) {
        long sum = 0;
        for ( int i = 0; i < n; i ++)
            for ( int j = i; j < n; j ++)
                sum++;
        System.out.println(sum); }


    public static void fragment_4(int n) {
        long sum = 0;
        for ( int i = 0; i < n; i ++)
            sum ++;
            for ( int j = 0; j < n; j ++)
                sum ++;
            System.out.println(sum);}


    public static void fragment_5(int n) {
        long sum = 0;
        for ( int i = 0; i < n; i ++)
            for ( int j = 0; j < n*n; j ++)
            sum++;
        System.out.println(sum);}
```

```
    public static void fragment_6(int n) {
        long sum = 0;
        for ( int i = 0; i < n; i ++)
            for ( int j = 0; j < i; j ++)
                sum++;
        System.out.println(sum);}


    public static void fragment_7(int n) {
        long sum = 0;
        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++;
        System.out.println(sum);}


    public static void fragment_8(int n) {
        long sum = 0;
        int i = n;
        while (i > 1) {
            i = i / 2;
            sum++;
        }
        System.out.println(sum);
    }
}
```

```java
package Ubung3;

import java.util.*;

public class Prime_Numbers {

    public static void main(String[] args) {
        List<Integer> Twenty_Bits = new ArrayList<Integer>();
        List<Long> Fourty_Bits = new ArrayList<Long>();

        for (int i = 300000; i<300100; i++)
        {
            Twenty_Bits.add(i);
        }


        for (long i = 500_000_000_000L; i<500_000_000_100L; i++)
        {
            Fourty_Bits.add(i);
        }

        for (int i=0;i<100; i++) {
        isPrime(Twenty_Bits.get(i));
        }

        for (int i=0;i<100; i++) {
        isPrime(Fourty_Bits.get(i));
        }

    }


    public static boolean isPrime (long n) {
        long sum =0;
        if (n<0) {
            System.out.println("n is not a positive number");
            sum++;
            System.out.println(sum);
            return false;}

        for (long i =2; i<n; i++){
            sum++;
            if (n%i == 0)
            {
                System.out.println(sum);
                return false;
            }
        }
        System.out.println(sum);
        System.out.println("is prime");
        return true;
    }
}
```