Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

B. Sc. Internationale Medieninformatik
Prof. Dr. Weber-Wulff

Group 1
Tuesday, 29.10.2019

Group members:
Timo Schmidt 571252 (timo.schmidt@student.htw-berlin.de)
Anh Pham Viet 569867 (anh.phamviet@@student.htw-berlin.de)

Post-Lab Report

Lab 3: Execution Times

1.) For each of the following eight program fragments, do the following:

a) Give a Big-Oh analysis of the running time for each fragment.

Fragment 1: O(N)

Fragment 2: $O(N^2)$

Fragment 3: $O(N^2)$

Fragment 4: $O(N^2)$

Fragment 5: $O(N^2)$

Fragment 6: $O(N^2)$

Fragment 7: $O(N^3)$

Fragment 8: O(1)

b) 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.

We first created the two fields *int n* and *long sum* and initialized them to *0*. We went with type *long* since we expected the numbers to get too long. An int could possibly not be able to cover the whole range of sums (1c). We instantiated a *new ExecutionTimes()* object named *et*.

ExecutionTimes.java

```
public class ExecutionTimes {

    private static int n;
    private static long sum;

    public static void main(String[] args) {

        ExecutionTimes et = new ExecutionTimes();

        // Initialize all fields with 0
        n = 0;
        sum = 0;
```

After that we copied all the fragment codes into our *main class* and put them all in single methods. Therefore we added both a *return statement* (*return sum;*) and the corresponding *return type* (*long*) to the methods.

```
// Fragment #1
private long frag1(int n) {
    for (int i = 0; i < n; i++)
        sum++;
    return sum;
}
```

In order to execute them we created a method called *countTime(int n)*. We asked Tony in the Lab how their group is going about measuring the time and implemented two longs *start* and *end* with *System.currentTimeMillis()* as their value. Between them we called our fragment method. We also created a *System.out.println* to print the duration in ms (the duration is calculated by subtracting *start* from *end*).

```java
/**
 * Exercise 1b
 * Method to measure how long the method takes to execute (in ms)
 * @param n - Input number
 */
private void countTime(int n) {
    long start = System.currentTimeMillis();
    frag1(n);
    long end = System.currentTimeMillis();

    System.out.println("----------------------------------------------\n");
    System.out.println("Results for " + n);
    System.out.println("Duration: " + (end-start) + " ms" + "\n");
}
```

Eventually we called our method *countTime()* with 5 different values from *10* to *100,000* in our *main method* and wrote down our results.

```java
// Exercise 1b: Time in milliseconds
et.countTime( n: 10);
et.countTime( n: 100);
et.countTime( n: 1000);
et.countTime( n: 10000);
et.countTime( n: 100000);
```

```
Results for 10
Duration: 0 ms

----------------------------------------------

Results for 100
Duration: 0 ms

----------------------------------------------

Results for 1000
Duration: 0 ms

----------------------------------------------

Results for 10000
Duration: 0 ms

----------------------------------------------

Results for 100000
Duration: 2 ms
```

Fragment	N = 10	N = 100	N = 1,000	N = 10,000	N = 100,000
1	0 ms	0 ms	0 ms	0 ms	1 ms
2	0 ms	0 ms	3 ms	7 ms	365 ms
3	0 ms	0 ms	0 ms	5 ms	186 ms
4	0 ms	0 ms	0 ms	1 ms	1 ms
5	0 ms	3 ms	41 ms	39500 ms	5 ms
6	0 ms	0 ms	0 ms	5 ms	187 ms
7	1 ms	21 ms	155401 ms	too long	too long
8	0 ms	0 ms	0 ms	0 ms	0 ms

c) Compare your analysis with the actual number of steps (i.e. the value of sum after the loop) for your report.

We wrote another method called *countSteps()*. In this method we set our long *sum to be 0* and call our fragment method similar to the previous exercise. At the end of this method there is another *System.out.println* that returns the (final) value of *sum*.

```java
/**
 * Exercise 1c
 * Method to count the number of steps
 * @param n - Input number
 */
private void countSteps(int n) {
    sum = 0;
    frag1(n);

    System.out.println("-----------------------------------------------\n");
    System.out.println("Results for " + n);
    System.out.println("Number of steps: " + sum + "\n");
}
```

This methods also gets called in the main method similar to the previous exercise.

```java
// Exercise 1c: Number of Steps
et.countSteps( n: 10);
et.countSteps( n: 100);
et.countSteps( n: 1000);
et.countSteps( n: 10000);
et.countSteps( n: 100000);
```

```
Results for 10
Number of steps: 10

-----------------------------------------------

Results for 100
Number of steps: 100

-----------------------------------------------

Results for 1000
Number of steps: 1000

-----------------------------------------------

Results for 10000
Number of steps: 10000

-----------------------------------------------

Results for 100000
Number of steps: 100000
```

Our results looked like this: (the red ones took too long, so we left them out)

Fragment	N = 10	N = 100	N = 1,000	N = 10,000	N = 100,000
1	10	100	1,000	10,000	100,000
2	100	10,000	1,000,000	100,000,000	10,000,000,000
3	55	5,050	500,500	50,005,000	5,000,050,000
4	20	200	2,000	20,000	200,000
5	1,000	1,000,000	100,000,000	1,000,000,000,000	
6	45	4,950	499,500	49,995,000	4,999,950,000
7	14,002	258,845,742	3,742,257,028,683		
8	3	6	9	13	16

Results:

Fragment 1:

We guessed that it was linear and the steps confirmed that we were right.

Fragment 2:

The sum is increasing quadratic, therefore our guess was right again.

Fragment 3:

According to the steps, the slope must be linear. We were wrong.

Fragment 4:

The increment here is linear again, even though we thought it was quadratic because of the nested loop.

Fragment 5:

10^3 is 1000 and 1000^3 is 1.000.000.000. It is $0(n^3)$.

Fragment 6:

45 x 11 x 10, 45 x 111 x 100, 45 x 1111 x 1000, 45 x 11111 x 10000

This is the pattern we found out by looking at each iteration. The first factor keeps increasing by tenfold + 1, while the 2nd factor increases tenfold every time. The slope is linear, so that would be O(n).

Fragment 7:

The complexity is $O(2^N \log N)$.

Fragment 8:

The complexity is $O(\log N)$.

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.

We took our method from an older exercise we did in the first semester. We added another *if statement* in case the entered number (*int n*) is negative. If so a corresponding message gets printed to the console via *System.out.println*.

```
/**
 * Exercise 2
 * Method to determine whether an Integer is a prime number or not
 * @param n - Integer you want to check
 * @return true if prime, false if not
 */
private static boolean isPrime(int n) {
    if (n < 0) {
        System.out.println("This number is not valid. Please use a positive number.");
        return false;
    }
    if (n <= 1) {
        return false;
    }
    for (int i = 2; i <= Math.sqrt(n); i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}
```

To ensure that our method is working as expected we tested it within a *for loop* from *0* to *100* and compared the output with a table of prime numbers. The results were correct.

```
// isPrime() method test
for (int i = 0; i < 100; i++) {
    if (pn.isPrime(i))
        System.out.println(i);
}
```

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
```

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

The running time of our algorithm is *O(N)* in every case. The worst-case is when *N* is *prime*, but that doesn‘t affect the complexity. Checking a range of numbers (as in our test from *1* to *100*) however would increase the complexity to $O(N^2)$.

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

We started by creating two methods to both convert our Integer in a Binary String and to count the Number of Bits (*length* of the *String*). Therefore we used the Method *.toBinaryString()* which we have found in the API of *Integer*. To count the number of bits we simply used the *.length()* method of *String*.

```
/**
 * Method to convert int to a Binary String
 * @param n - Integer you want to convert
 * @return str - binary String
 */
private String intToBinaryString(int n) {
    String str = Integer.toBinaryString(n);
    return str;
}

/**
 * Method to count the Number of Bits
 * @param n - Integer you want to convert
 * @return length of String / Number of Bits in binary representation
 */
private int NumberOfBits(int n) {
    String str = intToBinaryString(n);
    int length = str.length();
    return length;
}
```

Afterwards we created another method to print the results of different values (*10* to *1,000,000*) to our console to analyze them.

```
private void execute(int n) {
    System.out.println("-----------------------------------------------------------\n");
    System.out.println("Results for  " + n);
    System.out.println("In Binary:   " + intToBinaryString(n));
    System.out.println("No. of Bits: " + NumberOfBits(n));
    System.out.println();
}
```

```
// Check different (Big)Integers for their binary value + number of bits
pn.execute( n: 10);
pn.execute( n: 100);
pn.execute( n: 1000);
pn.execute( n: 10000);
pn.execute( n: 100000);
pn.execute( n: 1000000);
```

```
Results for  10
In Binary:   1010
No. of Bits: 4

-----------------------------------------------------------

Results for  100
In Binary:   1100100
No. of Bits: 7

-----------------------------------------------------------

Results for  1000
In Binary:   1111101000
No. of Bits: 10

-----------------------------------------------------------

Results for  10000
In Binary:   10011100010000
No. of Bits: 14

-----------------------------------------------------------

Results for  100000
In Binary:   11000011010100000
No. of Bits: 17

-----------------------------------------------------------

Results for  1000000
In Binary:   11110100001001000000
No. of Bits: 20
```

The number of bits in the binary representation (B) corresponds to a *power of 2*. B is the logarithm of N + 1 ($B = \log_2(N) + 1$) and $N < 2^B$.

Example: We have tested the *int 10*. The number of bits in its binary representation is *4*. The fourth power of two is *16*. This means a number smaller than 16 is represented in *4 bits*. ($\log_2 10 + 1 = 4{,}32$)

```
Results for  15
In Binary:   1111
No. of Bits: 4

------------------------------------------------------

Results for  16
In Binary:   10000
No. of Bits: 5

------------------------------------------------------

Results for  17
In Binary:   10001
No. of Bits: 5
```

◀ **Another test** with the integers 15, 16 and 17 and its results

The powers of 2 (wikipedia.org) ▶
https://en.wikipedia.org/wiki/Power_of_two

2^0	=	1
2^1	=	2
2^2	=	4
2^3	=	8
2^4	=	16
2^5	=	32
2^6	=	64
2^7	=	128
2^8	=	256
2^9	=	512
2^{10}	=	1,024
2^{11}	=	2,048
2^{12}	=	4,096
2^{13}	=	8,192
2^{14}	=	16,384
2^{15}	=	32,768

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

$O(N) \le (2^B - 1) < 2^B$

So the worst-case running time is $O(2^B)$.

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.

We started this exercise by rewriting all of our methods to take a *BigInteger b* as *parameter* instead of an *int*. The *isPrime()* method was a bit difficult in the beginning since *BigInteger*s are not working in the same way as *Integer*s. We figured out how to solve it by taking a look in the API of *BigInteger* and found a method *.signum()*. This method returns *−1* if the *b* is negative. We also found a method *.isProbablePrime()* which takes a certainty as parameter and returns either *true* or *false* if *b* is a prime number or not.

```java
/**
 * Exercise 2
 * Method to determine whether a BigInteger is a prime number or not
 * @param b - BigInteger you want to check
 * @return true if prime, false if not
 */
private boolean isPrime(BigInteger b) {
    if (b.signum() == -1) {
        System.out.println("This number is negative and not valid.");
        return false;
    }
    if (b.isProbablePrime( certainty: 100))
        return true;
    else
        return false;
}
```

After that we created a new method called *test()* which takes the number of Bits (*int numBits*) as *parameter*. We also found an appropriate constructor for our BigInteger in the API:

BigInteger

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

Constructs a randomly generated BigInteger, uniformly distributed over the range 0 to ($2^{numBits}$ - 1), inclusive. The uniformity of the distribution assumes that a fair source of random bits is provid

Parameters:

`numBits` - maximum bitLength of the new BigInteger.

`rnd` - source of randomness to be used in computing the new BigInteger.

Source: https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html

We wrote a *for loop* from *1* to *100* which creates a *new BigInteger* on each pass with *int numBits* (number of bits) and a *random* as its parameter. For the latter we set up a new field *private static Random random;* in our *PrimeNumbers* class. We then created a new instance of it in the first line of our new method: *random = new Random();*. By using our new method *isPrime(BigInteger b)* we can check for all BigIntegers if they are prime numbers and print the results to the terminal.

After learning that *System.currentTimeMillis()* is not accurate enough we searched the internet for a better implementation. We found a solution with *java.time.Instant* which was introduced to Java in version 8. (Source: http://tutorials.jenkov.com/java-date-time/duration.html). We created two *new Instant*s *start* and *end* and wrapped them around our method call. With *Duration.between(start, end)* we can finally measure the time correctly. We also added a *.toMillis()* to make sure our results were converted to milliseconds. Last but not least we printed our results to the console.

```java
/**
 * Exercise 2e
 * This method creates 100 random BigIntegers and calls the isPrime()-method on them
 * @param numBits - Number of Bits
 */
private void test(int numBits) {

    random = new Random();

    Instant start = Instant.now();
    for (int i = 0; i < 100; i++) {
        BigInteger b = new BigInteger(numBits, random);
        System.out.println(b + " : " + isPrime(b));
    }
    Instant end = Instant.now();
    long timeElapsed = Duration.between(start, end).toMillis();
    System.out.println("\n" + "Time: " + timeElapsed + "ms. \n");

}
```

```java
// Exercise 2e
pn.test( numBits: 20);
pn.test( numBits: 40);
```

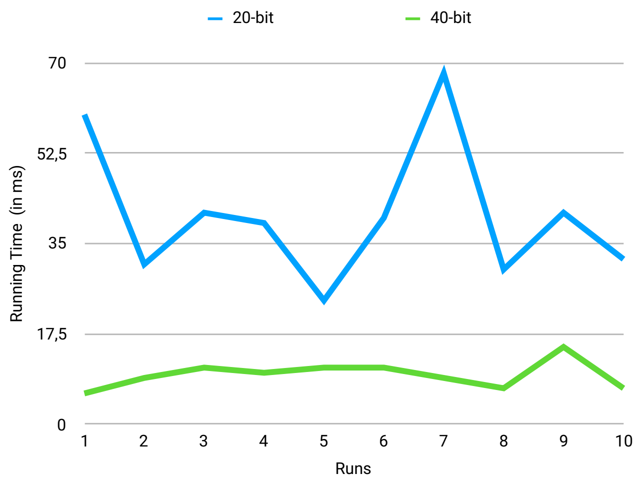
```
src — -bash — 42×35
737092 : false
233474 : false
26689 : false
230218 : false
910451 : true
413314 : false
408233 : false
796994 : false
941886 : false
827242 : false
29574 : false
387032 : false
146855 : false
779141 : false
133232 : false
928565 : false
659020 : false
747011 : false
64037 : true
590569 : false
636363 : false
839881 : false
14027 : false
584782 : false
527061 : false
886841 : false
89967 : false
368815 : false
443907 : false
693754 : false
832951 : false
127265 : false

Time: 32ms.
```

```
src — -bash — 42×35
1040349149294 : false
813939372135 : false
159006988116 : false
618464713249 : false
538022049515 : false
630527660088 : false
64737797409 : false
755621060850 : false
286429640916 : false
524684653937 : false
854495602363 : false
253687608121 : false
849734892935 : false
314446021103 : true
385328357701 : false
679616023139 : false
265405863173 : false
723746676944 : false
850444388544 : false
401851201122 : false
82834805878 : false
625151903991 : false
565846064791 : false
598521690409 : false
444977691299 : true
915325742732 : false
1067636904724 : false
353716851507 : false
597811784059 : false
648146937977 : false
9457485288 : false

Time: 7ms.

Timos-iMac:src timoschmidt$
```

Due to the ever-changing results we created a table and wrote down the 10 first runs of our method. We compared them and created a chart of it.

Running times in ms

Run	20-bit	40-bit
1	60	6
2	31	9
3	41	11
4	39	10
5	24	11
6	40	11
7	68	9
8	30	7
9	41	15
10	32	7



Surprisingly the *BigInteger*s with *40 bits* took less time than the *20 bit* ones in our implementation. We asked another group (Katja Hedemann, Kenneth Englisch) for their results in order to compare them. In their implementation the 40-bit Integers took so long they had to manually cancel the process.

We have made the assumption that thanks to the method *.isProbablePrime* of *BigInteger* our method might be much more efficient so it performs faster.

Time Management

In-Lab/Assignment	90 min
Task 1 a, b, c	*90 min (+30 min at home)*
Task 2 a, b	25 min
Task 2 c, d	30 min
Task 2 e	80 min

What have we learned?

Timo Schmidt

This week I‘ve learned how to measure the time that one method needs to execute. This will come in handy when we want to compare our algorithms with other people‘s ones at a later time. I also learned how to properly deal with BigIntegers since they are working in a different way than Integers.

Anh Pham Viet

The tasks helped me to further understand all about algorithm and their complexity. I guess in the real world people work with similiar big numbers so having efficient alogrithms seem very important. I for myself hope that we do learn the most important algorithms which could be applied to almost every problem because inventing new algorithm seems to be really scary.

ExecutionTimes

```
public class ExecutionTimes {

    private static int n;
    private static long sum;

    public static void main(String[] args) {

        ExecutionTimes et = new ExecutionTimes();

        // Initialize all fields with 0
        n = 0;
        sum = 0;

        // Exercise 1b: Time in milliseconds
        et.countTime(10);
        et.countTime(100);
        et.countTime(1000);
        et.countTime(10000);
        et.countTime(100000);

        // Exercise 1c: Number of Steps
        et.countSteps(10);
        et.countSteps(100);
        et.countSteps(1000);
        et.countSteps(10000);
        et.countSteps(100000);
    }

    /**
     * Exercise 1b
     * Method to measure how long the method takes to execute (in ms)
     * @param n - Input number
     */
    private void countTime(int n) {
        long start = System.currentTimeMillis();
        frag1(n);
        long end = System.currentTimeMillis();

        System.out.println(„---------------------------------------------\n“);
        System.out.println(„Results for „ + n);
        System.out.println(„Duration: „ + (end-start) + „ ms“ + „\n“);
    }
```

```
/**
     * Exercise 1c
     * Method to count the number of steps
     * @param n - Input number
     */
    private void countSteps(int n) {
        sum = 0;
        frag1(n);

        System.out.println(„---------------------------------------------------\n");
        System.out.println(„Results for „ + n);
        System.out.println(„Number of steps: „ + sum + „\n");
    }

    // Fragment #1
    private long frag1(int n) {
        for (int i = 0; i < n; i++)
            sum++;
        return sum;
    }

    // Fragment #2
    private long frag2(int n) {
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                sum++;
        return sum;
    }

    // Fragment #3
    private long frag3(int n) {
        for ( int i = 0; i < n; i ++)
            for ( int j = i; j < n; j ++)
                sum++;
        return sum;
    }

    // Fragment #4
    private long frag4(int n) {
        for ( int i = 0; i < n; i ++)
            sum++;
        for ( int j = 0; j < n; j ++)
            sum++;
        return sum;
    }

    // Fragment #5
    private long frag5(int n) {
        for ( int i = 0; i < n; i ++)
            for ( int j = 0; j < n*n; j ++)
                sum++;
                //counter = counter.add(BigInteger.ONE);
        return sum;
    }
```

```
// Fragment #6
    private long frag6(int n) {
        for ( int i = 0; i < n; i ++)
            for ( int j = 0; j < i; j ++)
                sum++;
        return sum;
    }

    // Fragment #7
    private long frag7(int n) {
        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;
    }


    // Fragment #8
    private long frag8(int n) {
        int i = n;
            while (i > 1) {
            i = i / 2;
            sum++;
            }
        return sum;
    }


}
```

PrimeNumbers

```
import java.math.BigInteger;
import java.time.Duration;
import java.time.Instant;
import java.util.Random;

public class PrimeNumbers {

    private static Random random;

    public static void main(String[] args) throws InterruptedException {
        PrimeNumbers pn = new PrimeNumbers();

        // Exercise 2a - isPrime() method test
        for (int i = 0; i < 100 ; i++) {
            if (pn.isPrime(i))
                System.out.println(i);
        }

        // Check different (Big)Integers for their binary value + number of bits
        pn.execute(10);
        pn.execute(100);
        pn.execute(1000);
        pn.execute(10000);
        pn.execute(100000);
        pn.execute(1000000);

        // Exercise 2e
        pn.test(20);
        pn.test(40);
    }

    /**
     * Exercise 2a
     * Method to determine whether an Integer is a prime number or not
     * @param n - Integer you want to check
     * @return true if prime, false if not
     */
    private static boolean isPrime(int n) {
        if (n < 0) {
           System.out.println(„This number is not valid. Please use a positive number.“);
            return false;
        }
        if (n <= 1) {
            return false;
        }
        for (int i = 2; i <= Math.sqrt(n); i++) {
            if (n % i == 0) {
                return false;
            }
        }
        return true;
    }
```

```
/**
 * Exercise 2e
 * Method to determine whether a BigInteger is a prime number or not
 * @param b - BigInteger you want to check
 * @return true if prime, false if not
 */
private boolean isPrime(BigInteger b) {
    if (b.signum() == -1) {
        System.out.println(„This number is negative and not valid.");
        return false;
    }
    if (b.isProbablePrime(100))
        return true;
    else
        return false;
}

/**
 * Exercise 2e
 * This method creates 100 random BigIntegers and calls the isPrime()-method on them
 * @param numBits - Number of Bits
 */
private void test(int numBits) {

    random = new Random();

    Instant start = Instant.now();
    for (int i = 0; i < 100; i++) {
        BigInteger b = new BigInteger(numBits, random);
        System.out.println(b + „ : „ + isPrime(b));
    }
    Instant end = Instant.now();
    long timeElapsed = Duration.between(start, end).toMillis();
    System.out.println(„\n" + „Time: „ + timeElapsed + „ms. \n");

}

/**
 * Exercise 2c
 * Method to print out the results of the intToBinaryString + NumberOfBits-methods
 * @param n - Integer
 */
private void execute(int n) {
    System.out.println(„------------------------------------------------------\n");
    System.out.println(„Results for  „ + n);
    System.out.println(„In Binary:   „ + intToBinaryString(n));
    System.out.println(„No. of Bits: „ + NumberOfBits(n));
    System.out.println();
}
```

```
    /**
     * Exercise 2c
     * Method to convert int to a Binary String
     * @param n - Integer you want to convert
     * @return str - binary String
     */
    private String intToBinaryString(int n) {
        String str = Integer.toBinaryString(n);
        return str;
    }

    /**
     * Exercise 2c
     * Method to count the Number of Bits
     * @param n - Integer you want to convert
     * @return length of String / Number of Bits in binary representation
     */
    private int NumberOfBits(int n) {
        String str = intToBinaryString(n);
        int length = str.length();
        return length;
    }

    /*
    private void execute(BigInteger b) {
        System.out.println(„-----------------------------------------------------\n“);
        System.out.println(„Results for  „ + b);
        System.out.println(„In Binary:   „ + intToBinaryString(b));
        System.out.println(„No. of Bits: „ + NumberOfBits(b));
        System.out.println();
    }

    private String intToBinaryString(BigInteger b) {
        String str = b.toString(2);
        return str;
    }

    private int NumberOfBits(BigInteger b) {
        String str = intToBinaryString(b);
        int length = str.length();
        return length;
    }
     */

}
```

Pre-Lab – Timo Schmidt

Exercise 3: Execution Times Timo Schmidt 29.10.2019

P1

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)?*

   10 100 > 10 000
   <u>*A:* 150N log N = 150 • 10 100 lg(11 000) = 20 152 632,52</u>
   B: N^2 = 10 100^2 = 102 010 000

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

   10 < 100
   A: 150N log N = 150 • 10 lg(10) = 4982,89
   <u>B: N^2 = 10^2 = 100</u>

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

   A: 150N log N = 150 • 1000 lg(1000) = 1 494 867,64
   <u>B: N^2 = 1000^2 = 1 000 000</u>

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

   No, because of the large values of N. B only performs better on small inputs.

P2

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	500/100 = N/0,5
2. O (N log N)	3,3737 ms	500log500/100log100 = N/0,5
3. quadratic	12,5 ms	500^2/100^2 = N/0,5
4. cubic	62,5 ms	500^3/100^3 = N/0,5

P3

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	x/100 = 60000/0,5
2. O (N log N)	3 656 807	xlogx/100log100 = 60000/0,5
3. quadratic	34 641	x^2/100^2 = 60000/0,5
4. cubic	4 932	x^3/100^3 = 60000/0,5

Pre-Lab – Timo Schmidt

Exercise 3: Execution Times Timo Schmidt 29.10.2019

P4

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
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

Pre-Lab – Anh Pham Viet

1. Programs A and B are analyzed and are found to have worst-case running times no greater than 150 N log N and N2 , 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)?

O(N log N)

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

still

O(N log N)

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

3.0 x 10 -3 s

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

accordingt to efficiency table, 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

Still ,05

2. O (N log N)

3. quadratic

4. cubic

Pre-Lab – Anh Pham Viet

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

maybe almost unlimited?

2. O (N log N)

n1 seems to aplly here too

3. quadratic

about 10000, little bit less

4. cubic

Must be around 200

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

2/N,

N

N1.5,

N2 ,

N3,

2N,

2N/2,

N2 log N

N log2 N,

N log N,

N log log N,

N log (N2),