Projects & Training

a.) J. Kepler found a formula to approximate the i-element of the Fibonacci Sequence. Why is this formula not suitable for the computation?

Kepler hat herausgefunden, dass die Fibonaccizahlen anhand der Zahl Phi bestimmt werden kann :

$$Fib(n) = \frac{Phi^n - (-Phi)^{-n}}{\sqrt{5}} = \frac{Phi^n - (-phi)^n}{\sqrt{5}}$$

https://www.google.com/url?sa=i&url=https%3A%2F%2Fblog.rinatussenov.com%2Fsequencing-fibonacci-numbers-with-javascript-c510561c6feb&psig=AOvVaw1vvATPAZwXO3byUjicJPKC&ust=1634675056632000&source=images&cd=vfe&ved=0CAsQjRxqFwoTCKi6nKnl1PMCFQAAAAAdAAAAABAD

Diese Formel eignet sich nicht zum berechnen von grösseren Fibonacci stellen. In Java hat Double einen MAX_VALUE von 1.7 E308. Das heisst ab Fibonacci Stelle 1474 kann diese Methode in Java nicht mehr verwendet werden ausser man verwendet eine erweiterte mir unbekannte Wurzelfunktion. Desweiteren wird der Rechenaufwand grösser je genauer die irrationale Zahl sqrt(5) berechnet werden muss.

b.) Describe a method to compute efficiently a power of a value x with logarithmic complexity

Die normale Methode um die Potenz einer Zahl zu berechnen hat eine Komplexität von BigOh(n). Heisst für 2^n wird die Zahl 2 n-Mal mit sich selbst multipliziert was bedeutet, dass eine Schlaufe n-Mal durchlaufen wird.

Wenn die Methode eine Komplexität von BigOh(log(n)) haben soll muss n bei möglichst jedem Durchlauf der Schlaufe halbiert werden. Eine Möglichkeit sieht so aus:

```
public static long power_calc(long base, long power) {
    if(power == 0) {
        return 1;
    }
    if (power < 2) {
        return base;
    }
    if(power % 2 == 0) {
        return power_calc(base, power/2) * power_calc(base, power/2);
    } else {
        return base * power_calc(base, power-1);
    }
}
```

Bei jedem rekursiven Aufruf der Methode wird überprüft ob es sich um eine gerade oder ungerade Hochzahl (`power`) handelt.

Falls `power` ungerade ist soll die Basis mit dem nächst kleineren rekursiven Aufruf multipliziert werden.

Falls `power` gerade ist soll der rekursive Aufruf von der hälfte der Hochzahl mit sich selbst multipliziert werden. Sobald ein rekursiver Aufruf mit `power < 2` stattfindet wird die Basis zurück gegeben.

Die Rekursive Methode ist verständlicher aber ziemlich langsam da für mindestens jeden zweiten Aufruf der Methode ein zusätzlicher Aufruf dazu kommt.

```
return power_calc(base, power/2) * power_calc(base, power/2);
```

So sieht der gleiche Vorgang als Iterative Methode aus:

```
public static void power_calc(long base, long power) {
    while(true) {
        if(power % 2 != 0){
            res *= base;
        }
        power >>= 1;
        if(power <= 0) {
            break;
        }
        base *= base;
    }
}
```

Auch hier wird bei jedem Durchgang überprüft ob es sich um eine gerade Hochzahl handelt. Wenn ja, wird das aktuelle Resultat mit der aktuellen Basis multipliziert. Anschliessend kann das Bit ganz rechts mit einem Shift Right entfernt werden (Hier kann auch einfach durch zwei geteilt werden solang der Rest abgerundet wird). Falls die Hochzahl noch nicht 0 beträgt wird anschliessend die aktuelle Basis mit sich selbst multipliziert.

c.) Give a formula for efficient computation of f(i) based on the matrix product.

Eine gewünschte Fibonaccistelle kann anhand des Matrixprodukts berechnet werden. Dazu muss die Matrix $\begin{bmatrix}1 & 1\\1 & 0\end{bmatrix}$ mit der gewünschte Fibonaccistelle hochgerechnet werden. Heisst, um Fibonaccistelle 100 zu berechnen muss $\begin{bmatrix}1 & 1\\1 & 0\end{bmatrix}^{100}$ berechnet werden.

Da wir jetzt eine Methode haben um Potenzen schneller zu berechnen können diese beiden Ideen kombiniert werden.

d.) Implement an efficient algorithm to compute f(i).

Der Code, inklusive Timer und whileCounter, ist auf meine GitHub zu finden.

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

        // Declare Variables
        BigInteger base[][] = {{new BigInteger("0"),new BigInteger("1")},{new BigInteger("1"),new BigInteger("1")}};
        BigInteger result[][] = {{new BigInteger("0"),new BigInteger("1")},{new BigInteger("1"),new BigInteger("1")}};
        fib = 1000000;

        // Calculate Fibonacci
        while(fib > 0) {
            // Check if even and subtract fib
            if((fib & 1) != 0){
                result = matrix_calc(result, base);
                fib -= 1;
            }
            // Divide by 2
            fib >>= 1;
            if(fib >= 0) {
                base = matrix_calc(base, base);
            }
        }

        // Print result
        System.out.println("Fibonaccinummer " + fibPrint + " = " + result[0][0].toString());

    }
```

```
public static BigInteger[][] matrix_calc(BigInteger[][] a, BigInteger[][] b) {
        BigInteger res[][] = {{new BigInteger("0"),new BigInteger("0")},{new BigInteger("0"),new BigInteger("0")}};
        for(int i=0;i<2;i++) {
            for(int j=0;j<2;j++) {
                for(int k=0;k<2;k++) {
                    res[i][j] = res[i][j].add(a[i][k].multiply(b[k][j]));
                }
            }
        }
        return res;
    }
}
```