AlgDat øving 2

Metode 1 –

Metoden recOne() er rekursiv og beskrives matematisk ved bildet til høyre. Metoden har ingen løkker og kjører gjennom dataen n-antall ganger. Metoden har derfor en tidskompleksitet på: $T(n) \in \theta(n)$. Som da blir O(n) (lineær).

$$x^n = \begin{cases} x & \text{hvis } n = 1 \\ x \cdot x^{n-1} & \text{hvis } n > 1 \end{cases}$$

```java
private static double recOne(double base, int n) {
    double sum = 0;
    if (n == 1) {
        return base;
    } else {
        sum = base * recOne(base, (n - 1));
        return sum;
    }
}
```

Metode 2 –

Metoden recTwo() har to forskjellige metoder den bytter mellom men til hver iterasjon brukes kun en av disse. Tidskompleksiteten kan gis ved:

$$x^n = \begin{cases} x & \text{hvis } n = 1 \\ (x \cdot x)^{n/2} & \text{hvis } n \text{ er partall} \\ x \cdot (x \cdot x)^{\frac{n-1}{2}} & \text{hvis } n \text{ er oddetall} \end{cases}$$

$$T(n) = a * T\left(\frac{n}{b}\right) + c * n^k$$

Der a er antall rekursive kall i metoden (1), b er brøkdelen av datasett vi behandler i et rekursivt kall (2) og k beksriver hvordan arbeidsmengden vokser iforhold til inputen/datasett (for-løkker) denne blir 0 så da gir formelen tidskompleksitet:

$$T(n) \in \theta(n^0 * \log(n))$$

$$T(n) \in \theta(\log(n))$$

```java
private static double recTwo(double base, int n) {
    double sum = 0;
    if (n == 1) {
        return base;
    }
    if ((n & 1) == 1) {
        sum = base * recTwo(base * base, (n - 1) / 2);
    } else {
        sum = recTwo(base * base, (n / 2));
    }
    return sum;
}
```

Metode 3 og Tidtaking –

Metoden runTimeTest() tar for seg tidtakingen for en av rekursjonsmetodene, med gitte n-verdier og en base. Det er denne metoden som tar og printer tid. Metode 3 brukes i switch-setningen ved default-caser.

```java
private static void runTimeTest(int methodIndex, int[] nValues, double base) {

    for (int i = 0; i <= nValues.length - 1; i += 1) {
        int expo = nValues[i];
        double time = 0;
        int rounds = 0;
        Date startTime = new Date();
        Date endTime;
        double result = 0;
        String formattedResult = "";
        do {
            switch (methodIndex) {
                case 0:
                    result = recOne(base, nValues[i]);
                    break;
                case 1:
                    result = recTwo(base, nValues[i]);
                    break;
                default:
                    result = Math.pow(base, nValues[i]);
            }
            endTime = new Date();
            rounds++;
        } while (endTime.getTime() - startTime.getTime() < 1000);
        formattedResult = String.format(format:"%.5f", result);
        time = (double) (endTime.getTime() - startTime.getTime()) / rounds;
        System.out.println("\n" + base + "^" + nValues[i] + " = " + formattedResult + " Time: " + time + "ms");
    }
}
```

Analyse av tidsmålinger -

Utskriften for main-metoden vises nedenfor og starter med å vise at de rekursive metodene og Math.pow-metoden får det samme riktige resultat. Math.pow og den lineære rekursive metoden kjører gjennom nValues, mens nValues2 kjører gjennom flere og høyere *n* for å få bedre forståelse for utvikling av tid.

Man ser at alle metodene gir samme svar for 5^11. Videre er det utskrift for de tre metodene med grunntall 1.002 og stigende eksponenter.

Man ser også at for hver dobling av n hos den lineære metoden så dobles tiden omtrentlig.

Ved den logaritmiske metoden ser man at den stiger relativt raskt (*n* = 2001 er relativt lavt, og det kan oppstå ujevnheter) før den flater ut mer og mer ved høyere *n* (utskrift viser det som $2.2 * 10^{-5}\ ms$). Dette er gjenkjennbart med logaritmiske funksjoner. Math.pow() ligger derimot relativt likt på alle verdier av *n*, på rundt $5.0 * 10^{-5}\ ms$.

```
Sjekk for grunntall 5, eksponent 11:
Method 1, O(n)4.8828125E7

Method 2, O(log(n))4.8828125E7

Method 3, Math.pow()4.8828125E7

Method 1, O(n)
----------------

1.002^1000 = 7.37431 Time: 0.002906131938390003ms

1.002^2001 = 54.48924 Time: 0.005600515247402761ms

1.002^4000 = 2957.23696 Time: 0.01206476365128007ms

1.002^8001 = 8762740.91705 Time: 0.02527869764149751ms

1.002^16000 = 76479404842391.83000 Time: 0.052764879696074296ms


Method 2, O(log(n))
----------------

1.002^1000 = 7.37431 Time: 2.0700815767397354E-5ms

1.002^2001 = 54.48924 Time: 2.1596050963397674E-5ms

1.002^4000 = 2957.23696 Time: 2.0617988749835675E-5ms

1.002^8001 = 8762740.91705 Time: 2.1311600729035763E-5ms

1.002^16000 = 76479404842364.40000 Time: 2.1340472318857215E-5ms

1.002^32001 = 5860797563772357000000000000.00000 Time: 2.1797958860930222E-5ms

1.002^64000 = 3421196338213791500000000000000000000000000000000000000000.00000 Time: 2.223592004310033E-5ms


Method 3, Math.pow()
-----------------

1.002^1000 = 7.37431 Time: 5.072757546944313E-5ms

1.002^2001 = 54.48924 Time: 5.066771690710271 4E-5ms

1.002^4000 = 2957.23696 Time: 5.085403245016152E-5ms

1.002^8001 = 8762740.91705 Time: 5.0836052257224934E-5ms

1.002^16000 = 76479404842391.92000 Time: 5.0838127538220616E-5ms
```