Øving 2 - Algoritmer og Datastrukturer

Innhold

- Oppgavebeskrivelse
- Algoritmen
- Tidskompleksitet
- Tidsmålinger
- Konklusjon

Oppgavebeskrivelse

Oppgaven handler om rekursjon. I main.cpp er det to forskjellige rekursive metoder for å regne ut eksponenter, i tillegg en tredje funksjon som bruker standardbiblioteket sin eskponentfunksjon. Den sammenligner så hastigheten til de tre metodene ved bruk av std::chrono.

Algoritmen

`main.cpp`:

```cpp
#include <chrono>
#include <iostream>

using namespace std;

double eksponentMetode1(double x, int n);
double eksponentMetode2(double x, int n);
double eksponentMetode3(double x, int n);

typedef double (*eksponent_method)(double, int);

void tidtaking(eksponent_method method, double x, int n);

int main() {
    cout << "Oving 2" << endl;
    int n_arr[] = {100, 1000, 10000, 100000};
    double x = 1.002;

    cout << "x = " << x << endl
         << endl;
```

```
    for (int n : n_arr) {
        cout << "n = " << n << endl;

        cout << "Metode 1: " << endl;
        tidtaking(eksponentMetode1, x, n);
        cout << endl;

        cout << "Metode 2: " << endl;
        tidtaking(eksponentMetode2, x, n);
        cout << endl;

        cout << "Metode 3: " << endl;
        tidtaking(eksponentMetode3, x, n);
        cout << endl;

        cout << "Kontroll: " << pow(x, n) << endl
             << endl;

        cout << "----------------------------------------" << endl
             << endl;
    }

    return 0;
}

double eksponentMetode1(double x, int n) {
    if (n == 1)
        return x;

    return x * eksponentMetode1(x, n - 1);
}

double eksponentMetode2(double x, int n) {
    if (n == 1)
        return x;

    if (n & 1)
        return x * eksponentMetode2(x * x, (n - 1) / 2);

    return eksponentMetode2(x * x, n / 2);
}

double eksponentMetode3(double x, int n) {
    return pow(x, n);
}

void tidtaking(eksponent_method method, double x, int n) {
    int runder = 0;
    double verdi;

    auto start = chrono::high_resolution_clock::now();
    auto finish = chrono::high_resolution_clock::now();

    do {
        verdi = method(x, n);
        finish = chrono::high_resolution_clock::now();
        runder++;
```

```
    } while (chrono::duration_cast<chrono::duration<double>>(finish - start).count()
< 1.0);

    cout << "Verdi: " << verdi << endl;
    auto elapsed = chrono::duration_cast<chrono::duration<double, std::micro>>
(finish - start);
    cout << "Tid: " << round(elapsed.count() / runder * 100) / 100 << "
mikrosekunder per runde" << endl;
}
```

Tidskompleksitet

Metode 1 har asymptotisk tidskompleksitet på $\Theta(n)$, mens metode 2 har $\Theta(log(n))$. Dette er fordi metode 1 kaller på seg selv n ganger, mens metode 2 halverer n for hver gang den kaller på seg selv slik at etter log(n) kjøringer er n=1 og returneres.

Tidsmålinger

Målingene er gjort med std::chrono over nok runder slik at den totale måletiden tilsvarer 1 sekund og vi får et mer nøyaktig resultat. Kjøretiden er i mikrosekunder.

(N)	METODE 1	METODE 2	METODE 3
100	0.8 ms	0.05 ms	0.05 ms
1000	7.42 ms	0.08 ms	0.05 ms
10000	78.59 ms	0.09 ms	0.05 ms
100000	817.59 ms	0.11 ms	0.05 ms

Konklusjon

Etter å ha utført en praktisk tidsmåling med chrono ser vi at målingene for metode 1 stemmer overens med den teoretiske tidskompleksiteten altså $\Theta(n)$. Målingene for metode 2 ser ut til å øke lineært for hver gang vi ganger `n` med 10. Dette stemmer overens med den teoretiske tidskompleksiteten $\Theta(\log(n))$. Metode 3 ser ut til å kjøre i konstant tid.
