Arbeidskrav 1 - Kompleksitet og rekursjon

Gruppemedlemmer: Karl Labrador, Eivind Berger-Nilsen, Mats Sollid Eide, Lars-Håvard Bråten

Formålet med dette arbeidskravet var grunnleggende tidsanalyse for to måter å beregne potenser på. Disse ble beskrevet i oppgaveteksten, og så testet vha. Java-kode. Hensikten med oppgaven var å tilegne forståelse for tidskompleksitet til forskjellige algoritmer, og betydningen dette har når man prosesserer små til store datamengder.

Den første oppgaven beskrev en algoritme som beregner potenser gitt grunntallet og eksponenten. Denne metoden tar høyde for et spesialtilfelle, nemlig hvis eksponenten er null. Dette er "best case" m.t.p. tidskompleksitet, da dette løses i konstant tid, eller Ω(1), men ellers vil eksponenten være proporsjonal med antall metodekall, noe som tilsier en O(n) under normale forutsetninger.

Den andre algoritmen beregnet også potenser vha. samme parametre, men behandlet par- og oddetall på forskjellig måte. Spesialtilfellet hvor eksponenten er 0 vedvarte, som gir Ω(1) for følgende algoritme, men normalt sett vil kompleksiteten være O(logN) som vi avdekket ved hjelp av master-metoden.

Metode

Utbytte av disse testene er ikke vitenskapelige målinger, da vi også må ta høyde for avvik i systemklokka, og annen "overhead" som kan forstyrre måleresultatet (JVM, Windows Kernel, osv.), eller at hovedtråden (main-thread) blir satt på pause under runtime fordi windows rett og slett prioriterer andre prosesser i stedet. Vi har ikke tatt høyde for målefeil, og har kun tilstrebet med minst mulig "overhead" under våre egen kildekode. Eksempel på tidsmåling:

```
Int startTid = System.currentTimeMillis();
for(int i = 0; i < syklusAntall; i++){
------ Metodekall til algoritmen her, ingenting annet -------
}
System.out.println(System.currentTimeMillis() - startTid)
```

Systemkallet til klokken oppgir utgått tid f.om. epoch tiden (1. Januar 1970), men oppløsningen avhenger av operativsystemet samt, maskinvaren selv. Vi kan ikke forutsi hvor stort tidsavviket blir under målinger, men ved å innføre mange nok målinger under et målesett kan vi gjøre tidsavviket neglisjerbart. Avhengig av hvor prosessor-intens operasjonen, vil det kanskje være nærliggende å kalle testmetoden mange nok ganger, slik at systemklokken rekker å oppdatere seg.

Rekursiv metode I

Den første rekursive metoden er basert på beskrivelsen i oppgave 2.1-1 på side 28 i læreboka. I denne oppgaven er x^n definert slik: $x^n = \{ 1 \text{ når } n = 0,\ x * x^{n-1} \text{når } n > 0 \}.$ Metoden kjører kalkulasjonen og kaller seg selv frem til et endelig resultat.

Rekursiv metode II

Den andre rekursive metoden er basert på beskrivelsen i oppgave 2.2-3 på side 38 i læreboka. I denne oppgaven er x^2 definert slik:
$x^n = \{ 1 \text{ når } n = 0,\ x * (x^2)^{(n-1)/2} \text{ når } n \text{ er et oddetall},\ (x^2)^{n/2} \text{ når } n \text{ er et partall} \}.$ På samme måte som den første rekursive metoden, kjører metoden kalkulasjonen og kaller seg selv frem til et endelig resultat.

Kompleksitet for rekursive algoritmer - mastermetoden

Dersom tidsforbruket, T(n), til en rekursiv algoritme kan beskrives på formen

$$T(n) = a * T(n/b) + cn^k,$$

har vi:

$$T(n) \in \Theta(n^{log_b a}),\ hvis\ b^k < a$$
$$T(n) \in \Theta(n^k * log\ n),\ hvis\ b^k = a$$
$$T(n) \in \Theta(n^k),\ hvis\ b^k > a.$$

Kompleksitet til metode I

Tidsforbruket til metode I kan telles opp på denne måten:

```
public static double recursiveTest1(double x, int n) {
    // x*x^n-1 when n > 0
  1 if (n > 0) {
        return x * recursiveTest1(x, n: n - 1);
    }                 ↑          ↑
                      1        T(n-1)

    // 1 when n = 0
    return 1;
}
```

Ut ifra dette får vi likningen T(n) = T(n-1) + 2, som ikke er på riktig form til å kunne anvende mastermetoden, men vi kommer tilbake til et uttrykk for tidskompleksiteten til denne metoden senere.

Kompleksitet til metode II

Vi kan telle tidsforbruket linje for linje slik:

```
public static double recursiveTest2(double x, int n) {
    if (n > 0) {                                              1
        // Even: (x^2)^(n/2)
        if (n % 2 == 0) {                                      1
            return recursiveTest2( x: x*x, n: n/2);
        }                                          ~T(1/2)

        // Odd: x * (x^2)^((n-1)/2)
        else {
            return x * recursiveTest2( x: x*x, n: (n - 1)/2);
        }
    }

    // 1 when n = 0
    return 1;
}
```

Dermed har vi T(n) = T(n/2) + 2. Da er a = 1, b = 2, c = 2, k = 0 i formelen for kompleksitet (mastermetoden).. 2^0=1, dermed er $T(n) \in \Theta(n^0 * log\ n) = \Theta(log(n))$

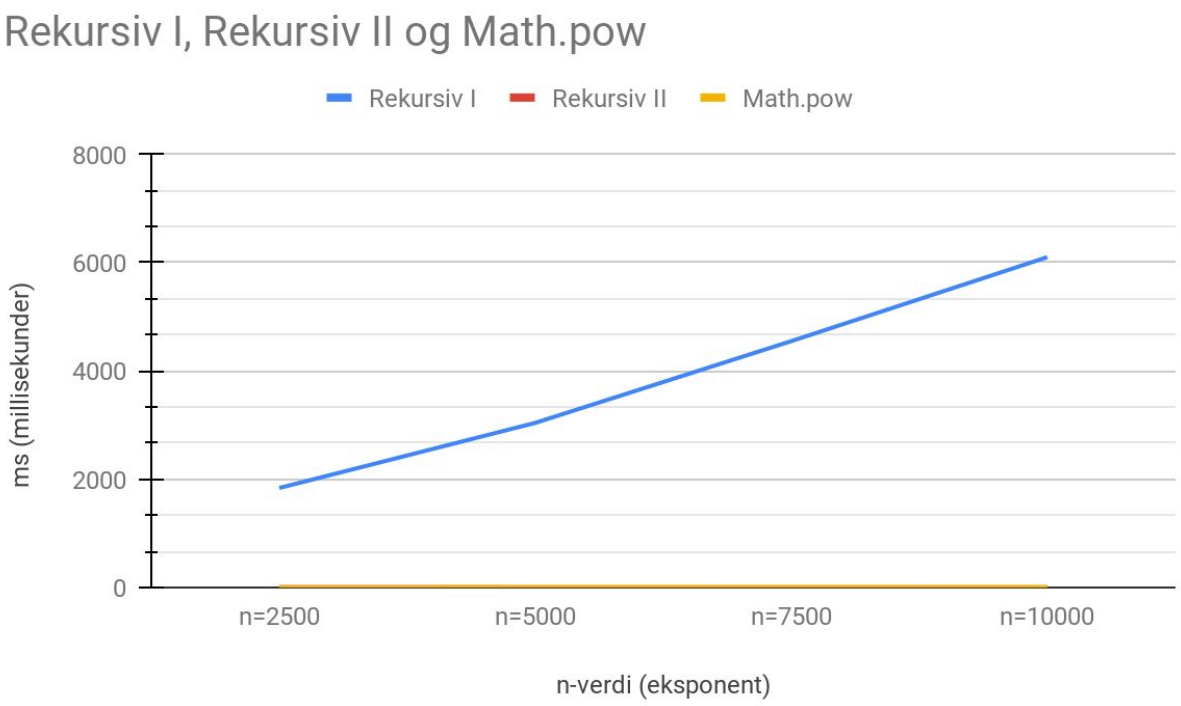
Denne metoden har altså logaritmisk tidskompleksitet.

Tidsmålinger

Vi ønsker å undersøke tidsbruken til de ulike rekursive metodene. For å gjøre dette er det skrevet benchmark-tester der vi måler tidsbruken for hver metode, inkludert Math.pow. Testen kjører hver rekursive metode 200.000 ganger, der vi måler tiden det tar i millisekunder å gjennomføre dette antallet kalkulasjoner.

N-verdi / Metode	Rekursiv I	Rekursiv II	Math.pow
2500	1838 ms	7 ms	1 ms
5000	3041 ms	5 ms	4 ms
7500	4551 ms	5 ms	4 ms
10000	6102 ms	5 ms	3 ms

Figur 1: Tabell av målingsresultater. Tabellen viser tidsmålinger i millisekunder.



Figur 2: Grafisk fremvisning av måleresultatene.

Vi gjennomførte i tillegg målinger med større variasjon i n (fra 10 til 10 000) med 10 000 000 repetisjoner, for å bedre illustrere hvordan tidsforbruket til metode II vokser.

N-verdi / Metode	Rekursiv I	Rekursiv II	Math.pow
10	114 ms	70 ms	196 ms
100	1774 ms	112 ms	198 ms
1000	26 981 ms	158 ms	193 ms

10000	289 980 ms	222 ms	122 ms

Figur 3: Flere måleresultater

Analyse av tidsmålingene

På tidsmålingene ser vi at tidsforbruket til metode 1 stiger med en størrelsesorden rundt 10 for hver gang n blir ganget med 10. Dette er som forventet ettersom den i teorien skal være proporsjonal med n.

Tidsforbruket til metode 2 stiger med et noenlunde konstant ledd (her 40-50 ms) hver gang n stiger med en konstant faktor (her 10), og bekrefter dermed at tidsforbruket utvikler seg logaritmisk med n. Uttrykket vi fikk med mastermetoden og annen regning stemmer altså overens med tidsmålingene.

Hvorfor er metode 2 raskere enn metode 1?

Metode 1 kaller på seg selv med en n som blir én mindre for hvert kall, helt til den kommer ned til null. Den må derfor utføre *n* kall dersom eksponenten er lik n.

Metode 2 kaller på seg selv med en n som mer eller mindre halverer seg for hver rekursjon. Vi lar *x* antyde antall rekursjoner og får denne ligningen som viser når argumentet i rekursjonen kommer ned til én:

$$n * (\tfrac{1}{2})^x = 1$$

Løser vi denne ligningen for x, får vi

$$log_2(n * (\tfrac{1}{2})^x) = log_2 1$$
$$log_2(n) + log_2(\tfrac{1}{2})^x = 0$$
$$x * log_2(\tfrac{1}{2}) = -log_2(n)$$
$$x * (-1) = -log_2(n)$$
$$x = log_2(n)$$

Antall rekursive kall blir altså toerlogaritmen til eksponenten, som er en funksjon som stiger mye saktere enn bare *n*, som var tilfellet med metode 1. Når n blir stor, og også lenge før den blir det, vil forskjellen på de to algoritmene bli svært tydelig.

Konklusjon

Vi har implementert to forskjellige algoritmer som kalkulerer potenser gjennom rekursjon. Ved å analysere tidskompleksiteten for de to algoritmene kan man finne ut at tidsforbruket til den ene algoritmen vil vokse lineært med eksponenten n, mens den andre vokser logaritmisk. Tidsmålingene vi gjennomførte bekreftet at tidsforbruket til algoritmene mer eller mindre fulgte de forventede funksjonsfamiliene.