Karo F. Mahmoud
Jonatan Andre Vevang
Stian Wilhelmsen

Øving 2

Oppgave 2.1-1 (1)

```java
public static double recursionAlgorithmOne(int x,double n){
    double answer=0;
    if(n==0){
        answer=1;
    }else if (n>0){
        answer=x*recursionAlgorithmOne(x, n: n-1);
    }
    return answer;
}

public static double recursionAlgorithmTwo(int x,double n){
    double answer=0;
    if (n==0){
        answer=1;
    }else if(n%2==0){
        answer=recursionAlgorithmTwo( x: x*x,(n/2));
    }else{
        answer=x*recursionAlgorithmTwo( x: x*x, n: (n-1)/2);
    }
    return answer;
}
public static double pow(double x, double n){
    return Math.pow(x,n);
}
```

Bilde 1: Bilde av de to algoritmene som ble utviklet og mattefunksjonen pow

Etter at algoritmene ble utviklet ved å ta i bruk rekursjon for å regne uttrykket x^n, testet vi med tilfeldig tall x=2 og n=12. Disse verdiene ga et felles svar (4096.0) på alle algoritmene og vi kan dermed konkludere med at algoritmene fungerer slik de skal da vi visste på forhånd at svaret skulle bli 4096.

Tabell med tidsmålinger for alle 3 metodene med samme x, men ulik n:

Metode	x, n	Kjøretid i ms (per runde)
pow	1.001, 5000	$1.84*10^{-5}$
recursionAlgorithmOne	1.001, 5000	0.0085
recursionAlgorithmTwo	1.001, 5000	$1.30*10^{-4}$
pow	1.001, 3500	$1.69*10^{-5}$
recursionAlgorithmOne	1.001, 3500	0.0060
recursionAlgorithmTwo	1.001, 3500	$1.22*10^{-4}$
pow	1.001, 2000	$1.69*10^{-5}$
recursionAlgorithmOne	1.001, 2000	0.0034

recursionAlgorithmTwo	1.001, 2000	$1.03*10^{-4}$
pow	1.001, 1500	$1.70*10^{-5}$
recursionAlgorithmOne	1.001, 1500	0.0025
recursionAlgorithmTwo	1.001, 1500	$1.07*10^{-4}$
pow	1.001, 1000	$1.71*10^{-5}$
recursionAlgorithmOne	1.001, 1000	0.0017
recursionAlgorithmTwo	1.001, 1000	$9.42*10^{-5}$
pow	1.001, 500	$1.72*10^{-5}$
recursionAlgorithmOne	1.001, 500	$8.42*10^{-4}$
recursionAlgorithmTwo	1.001, 500	$8.28*10^{-5}$
pow	1.001, 100	$1.33*10^{-5}$
recursionAlgorithmOne	1.001, 100	$1.58*10^{-5}$
recursionAlgorithmTwo	1.001, 100	$5.84*10^{-5}$

Ved å først se på der vi kjører med n = 100 så er kjøretiden nesten like lang for begge metodene. Men så snart vi setter n til å bli høyere enn 1000 så ser vi at tidsforskjellen øker mer og mer for høyere n. Dermed kan man si at recursionAlgorithmTwo er raskest av de to metodene når n blir stor. Grunnen til dette er for recursionAlgorithmOne regner med n-1 imens recursionAlgorithmTwo halverer datamengden med n/2 og (n-1)/2. Dette kan man regne på ved hjelp av å bruke mastermetoden.

Oppgave 2.2-3 (2) og oppgave 3

For å skjønne hvorfor den ene algoritmen er raskere enn den andre kan den lønne seg å regne på dette ved bruk av mastermetoden.

<u>Asymptotisk analyse for recursionAlgorithmOne:</u>

Det går ikke ant å gjennomføre analyse på denne algoritmen ved hjelp av mastermetoden da den ikke er satt opp på rett form. I denne algoritmen så har vi at n-1 noe som fører til at den ikke er satt på rett form på lik linje med tårnet i Hanoi (forklart i presentasjon om rekursjon). Men ved å se på likningen klarer man å se at den vil gi T(n) = Theta(n). Med andre ord så er den lineær. Dette er fordi man kan se bort ifra leddet -1 i likningen. Man kan dessuten se

den lineære stigningen av kjøretid når man øker datamengden (n). Et eksempel på dette er økningen fra $1.58*10^{-5}$ til $8.28*10^{-5}$, som er omtrent en femdobling (når datamengden femdobles fra 100 til 500).

Asymptotisk analyse for recursionAlgorithmTwo:

Generell formel for mastermetoden er som følger:

a*T(n/b) + c*n^k

Der a er antall rekursive kall i metoden, b er brøkdelen av datasettet vi behandler i et rekursivt kall og c*n^k kompleksitet for metoden. Ved å ta se på algoritmen som er definert på bilde 1 ser vi at det bare er ett rekursivt kall og dermed blir a satt til 1 i dette tilfellet. Videre som nevnt tidligere i rapporten har vi halvering av datamengden og dermed blir b satt til 2. Sist så har vi har k er 0 da vi ikke har noen form for løkker i denne algoritmen.

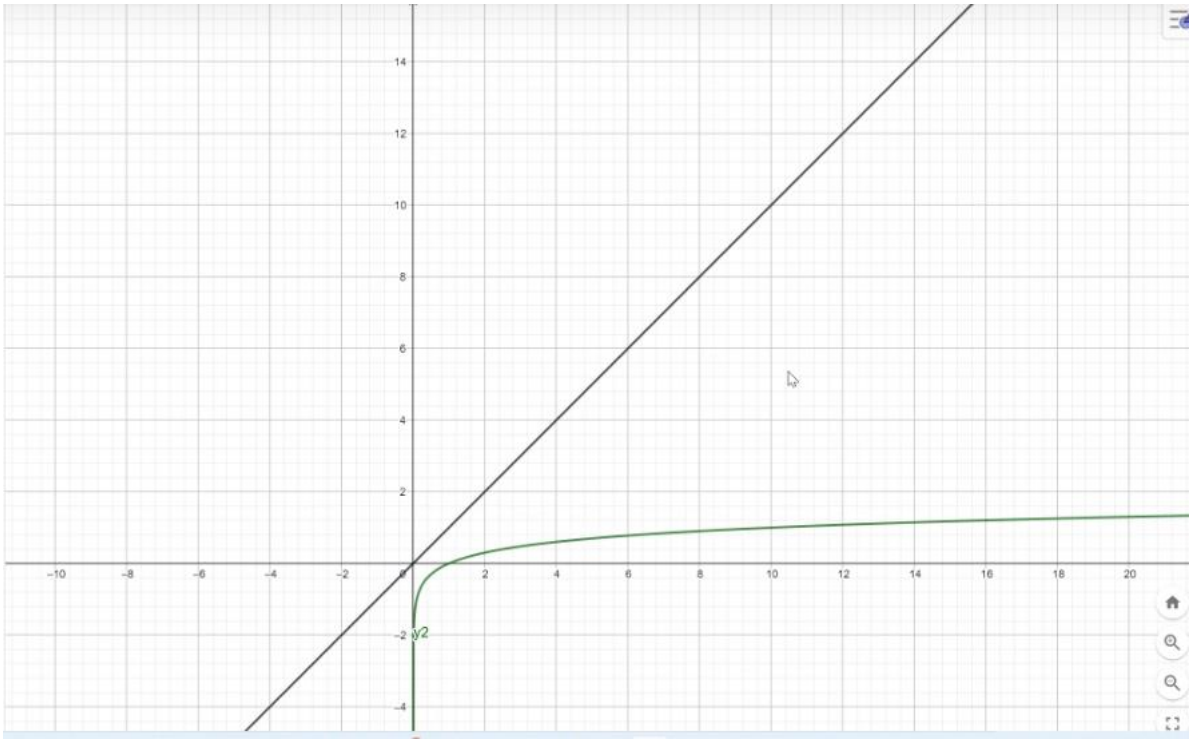
Når man vet dette kan man videre se på forholdet mellom b^k og a som er gitt på følgende måte:

Hvis $b^k < a$, har vi $T(n) \in \Theta(n^{\log_b a})$
Hvis $b^k = a$, har vi $T(n) \in \Theta(n^k \cdot \log n)$
Hvis $b^k > a$, har vi $T(n) \in \Theta(n^k)$

Siden vi har at a=1, b=2, k=0 og dermed får vi at b^k=a -> 2^0=a og dermed har vi at T(n) er et element av Theta(n^k*log(n)). Men siden k = 0 så blir dette utrykket forenklet til log(n). Dette fører til at man får T(n) = Theta(log(n)). Ved å se på de to første tidsmålingene der n er 100 og 500, kan man legge merke til en svært liten forskjell i kjøretid (fra $5.84*10^{-5}$ til $8.28*10^{-5}$). Dette er noe som stemmer overens med den logaritmiske funksjonen.

Konklusjon:

Ved å ta utgangspunkt i T(n) som vi har kommet fram til i begge algoritmene kan man forklare hvorfor recursionAlgorithmTwo er raskere enn nummer recursionAlgorithmTwo. Ved å se på bildet nedenfor, kan man legge merke til at en logaritmisk funksjon i dette tilfellet har en langsom endring av tid når man øker datamengden, sammenlignet med den lineære funksjonen. Man kan derfor konkludere med at recursionAlgorithmTwo er raskere enn recursionAlgorithmOne.



Bilde over lineær funksjon (svart) og funksjoner