Øving 1 – Rekursjon

Ingebrigt Hovind & Erling Sung Sletta

Algoritmer og datastrukturer

Disse tallene ble funnet ved å telle antall utregninger som programmet rakk på 1000 ms. Disse tusen millisekundene ble så delt på antallet gjennomførte utregninger for å finne gjennomsnittlig antall millisekunder per utregning.

(Grunntall, Potens)	Oppgave 2-1-1	Oppgave 2-2-3	Pow(x,n)
(1.001, 4000)	0.0211318 ms	0.00013417 ms	0.00010685 ms
(1.001, 6000)	0.0325744 ms	0.000141511 ms	0.000107165 ms
(1.001, 8000)	0.0438347 ms	0.000141504 ms	0.000107706 ms

Kan lese ut ifra tabellen at programmet fra 2-1-1 stiger mye raskere i tillegg til å være høyere fra starten av enn programmet fra oppgave 2-2-3. Dette er på grunn av metodenes store O, noe som vil forklares i analysen.

Asymptotiske analyser

Dette er kun analyse av selve utregningen, andre deler av programmet har også innflytelse på kjøretiden, men disse delene skal ikke være avhengige av N, så de er ikke særlig interessante i denne analysen.

Oppgave 2-1-1

```
if(n == 0){ Ω(1)
    return 1;
}
if(n>0){
    return x * power211(x, n – 1); O(n-1) ≈ O(n)
}
else{
    return 1 / power211(x, (n * -1)); O(n)
}
```

Oppgave 2-2-3

```
if(n == 0){ Ω(1)
    return 1;
}
if(n<0){ //1
    return 1/power223(x,(n*-1));
} else if(n % 2 == 0){
    //partall
    return power223(x*x,n/2); O(log2(n))
} else{
    //oddetall
    return x*power223(x*x,(n-1)/2); O(log2(n))
}
```

Hvordan kan vi se kompleksiteten?

2-2-3

For å vise at $O(\log_2(n))$ blir riktig så kan man løse ligningen:

$$1=\frac{N}{2^x}$$

Man kan se at denne viser hvor mange ganger (x) man må halvere et heltall (N) helt til man kommer til N=1 som har den trivielle løsningen. Dette vil i realiteten regne ut én iterasjon mindre enn programmene vil bruke, da disse stopper ved N = 0, men dette er uviktig med tanke på utregning av store O.

Man ser under utledningen som viser at antall iterasjoner øker i takt med logaritmen til N:

$$2^x=N \rightarrow \log_2(2^x)=\log_2(N) \rightarrow x*\log_2(2)=\log_2(N) \rightarrow x=\log_2(N)$$

2-1-1

For oppgave 2-1-1, med O(N) så kan man løse en ligning på samme måte som over om ønskelig, og man ender da opp med;

$$1=N-x$$

Her så ser man at oppgave 2-1-1 må gjennomføre et likt antall iterasjoner som potensen for å nå den trivielle løsningen.

Hvordan påvirker store O den målte tiden

Her så ser man at funksjonen fra opg 2-1-1 må kjøre 8000 iterasjoner, mens funksjonen fra 2-2-3 kun må halvere ~13 ganger for å oppnå samme resultat, dette er kilden til de store tidsforskjellene:

N	O(N)	$O(\log_2(N))$
4000	4000	11,96
6000	6000	12,55
8000	8000	12,96

Funksjonen fra oppgave 2-1-1 ser man at er lineær fra målingene, da tiden utregningen øker lineært i takt med den lineære økningen av N. Med grunntall 1.001 så er økningen fra 4000 til 6000 = 11442,3 nanosekunder og økningen fra 6000 til 8000 = 11260,3 nanosekunder, noe man kan se at er lineært.

Funksjonen fra oppgave 2-2-3 brukte gjennomsnittlig: 0,00013417 ms = 134,17 nanosekunder per halvvering med grunntall 1.001 og eksponent på 4000. Dette gir 11,22 gjennomsnittlig nanosekunder per iterasjon. Differansen mellom 4000 og 8000 i eksponenten er 7,334 nanosekunder, dette er tilnærmet lik 11.96 nanosekunder slik at vi kan konkludere med at tiden virkelig øker i takt med $\log_2$ av eksponenten, da vi observerer at en dobling fører til en økning i tid som samsvarer med én ekstra iterasjon gjennom funksjonen. Denne økningen er mye tregere enn den lineære vi observerte i oppgave 2-1-1, noe som fører til at programmet blir raskere, spesielt med store eksponenter slik som i målingene.

Differansen på ca. 4 nanosekunder mellom målt ut utregnet tid per iterasjon til funksjon 2-2-3 kommer sannsynligvis fra unøyaktighet i maskinklokken eller små differanser i ledig prosessorkraft på maskinen mellom testene. Store O er naturligvis også bare et estimat på tidsforbruk, så man kan ikke forvente at den skal passe perfekt, men som man kan se, så passer estimatet relativt bra med det reelle tidsforbruket i testene.