Algoritmer og datastruktur - Arbeidskrav 02 - Analyse

Gruppe 12: John I. Eriksen og Emil Slettbakk

Felles for begge metodene og målingene ser vi at tidsmålingen varierte ganske mye for hver kjøring. Her har det blitt tatt med data fra de høyeste tallene som kom frem i kjøringene, som ble repetert til det kom et tall større enn det forrige for å illustrere en økning. Altså er det et kuratert utvalg av tall for illustrasjon.

Det er lite å skille på tidsforbruket mellom oddetall og partall, og det som ev. måtte være forskjell kan uten problemer tilskrives den variasjonen som blir observert mellom hver kjøring av *samme* tall. Dette gjelder for både Metode 1 og Metode 2.

Metode 1

Målinger for kjøretid med partall n:

n (int)	x (double)	Tid, millisekund	Antall kjøringer
10	9.5	5.5010	100 000
100	9.5	21.1098	100 000
1 000	9.5	327.8025	100 000
2 000	9.5	744.4444	100 000
4 000	9.5	1484.8549	100 000
8 000	9.5	2965.9121	100 000
13 500	9.5	4906.7994	100 000
14 000	9.5	Stack overflow error	100 000

Målinger for kjøretid med oddetall n:

n (int)	x (double)	Tid, millisekund	Antall kjøringer
1**1**	9.5	5.3196	100 000
10**1**	9.5	23.8438	100 000
1 00**1**	9.5	346.0506	100 000
2 00**1**	9.5	696.7288	100 000
4 00**1**	9.5	1496.9293	100 000
8 00**1**	9.5	2885.9111	100 000
13 50**1**	9.5	4975.1604	100 000
14 00**1**	9.5	Stack overflow error	100 000

Det er en tydelig økning i tiden det tar å kjøre gjennom rekursjonen med Metode 1.

Fra og med $n = 10\,000$ begynner kjøringene å tidvis resultere i Stack Overflow error. Feilmeldingene blir deretter hyppigere når n økes, og ved $n = 14\,000$ begynner kjøringer å resultere i stack overflow i de aller fleste tilfeller (10 manuelle kjøringer av metoden, alle med same utfall). For hver dobling av tallet n, dobles også tidsforbruket, mer eller mindre. Metode 1 er altså lineær.

Samlet grense for metode 1

$$
\begin{aligned}
T(n) &= T(n-1) + 1 \\
T(n) &= \theta(n) \\
\theta(n)
\end{aligned}
$$

Metode 2

Målinger for kjøretid med partall n. Kjøringer med både Modulo (`%`) og Bitwise(`&`) for å illustrere forskjellen i kjøretid.

n (int)	x (double)	Tid (ms), modulo	Tid (ms), bitwise	Antall kjøringer
10	9.5	14.8778	4.1023	100 000
100	9.5	17.2206	7.2790	100 000
1 000	9.5	27.5822	8.8953	100 000
10 000	9.5	37.8629	9.3323	100 000
100 000	9.5	60.0917	11.2889	100 000
1 000 000	9.5	62.6315	12.4907	100 000
10 000 000	9.5	95.3966	13.5969	100 000

Målinger for kjøretid med oddetall n

n (int)	x (double)	Tid (ms), Bitwise	Antall kjøringer
1**1**	9.5	5.0436	100 000
10**1**	9.5	6.5676	100 000
1 00**1**	9.5	7.3326	100 000
10 00**1**	9.5	8.0816	100 000
100 00**1**	9.5	9.0713	100 000
1 000 00**1**	9.5	11.0723	100 000
10 000 00**1**	9.5	13.9468	100 000

For hver 10-dobling av tallet n, øker tidsbruket, men med en minkende hastighet i forhold til datamengden som prosesseres, hvilket er kjennetegnet for logaritmiske funksjoner.

Målingene viser også en markant forskjell i effektivitet ved å bruke Bitwise (`&`) i stedet for Modulo (`%`) for å beregne om tallet n er et partall eller oddetall.

Samlet grense for metode 2

$$a = 1,\ b = 2,\ k = 0$$

$$T(n) = T(\frac{n}{2}) + 2$$
$$b^k = 2^0 = 1$$
$$b^k = a \implies 1 = 1$$

$$T(n) \in \theta(n^k \cdot \log(n))$$

Årsaken til forskjell i kjøretid

Metode 2 er mer effektiv enn Metode 1.

I Metode 1 trekkes det fra 1 for hvert kall, og gjør operasjonen $n - 1$ helt til $n = 1$, og må dermed gjøre n rekrusive kall for å komme dit. Antallet kall er da proporsjonalt med n, det vil si lineær.

```
1     public static double multiplication(int n, double x)
{
2         if (n > 1) {
3             return (x + multiplication(n - 1, x));
4         } else {    // Function time is proprtional to n
5             return x;
6         }
7     }
```

I Metode 2 deles n på 2 for hvert kall, og metoden halverer dermed datamengden. At n halveres for hver gang, resulterer i den avtagende (men fortsatt økende) tidsbruken, og vi får en logaritmisk funksjon.

```java
    public static double multiplicationBitwise(int n, double x) {
        if (n == 1) return x;
        if ((n & 1) == 0) return multiplicationBitwise(n/2, x+x);
        return x + multiplicationBitwise((n-1)/2, x+x);
                            // Dividing n by two results in logarithmic function
    }
```