

---

Kjøretidsanalyse av rekursive algoritmer.  
TDT4120 Algoritmer og Datastrukturer

*Simon Jonassen*

---

NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITETET  
INSTITUTT FOR DATATEKNIKK OG INFORMASJONSVITENSKAP



---

## Innhold

<b>1</b>	<b>Introduksjon</b>	<b>1</b>
<b>2</b>	<b>Kjøretidsanalyse av rekursive algoritmer</b>	<b>1</b>
<b>3</b>	<b>Rekurrensregning</b>	<b>1</b>
3.1	Iterasjonsmetoder . . . . .	1
3.1.1	Foroversubstitusjon . . . . .	1
3.1.2	Tilbakesubstitusjon . . . . .	3
3.2	Substitusjonsmetoden . . . . .	6
3.3	Tremetoden . . . . .	7
3.4	Masterteoremet . . . . .	8
<b>4</b>	<b>Videregående metoder</b>	<b>9</b>



## 1 Introduksjon

Formålet med dette notatet er å introdusere noen metoder som kan brukes til kjøretidsanalysen av rekursive algoritmer i faget TDT4120, Algoritmer og Datastrukturer.

Forutsatte ferdigheter for dette notatet er:

1. logaritmer, matematiske funksjoner, rekkeutviklinger (3MX [ea02] eller tilsvarende)
2. asymptotisk notasjon og kjøretidsanalyse for sekvensielle algoritmer (algsat [CLR90] eller tilsvarende [Lev02])

## 2 Kjøretidsanalyse av rekursive algoritmer

En algoritme er rekursiv hvis den kaller seg selv en eller flere ganger, eventuelt med en annen probleminstans.

Kjøretidsanalyse av rekursive algoritmer består av fire steg [CLR90]:

1. velge en metrikk som indikerer inputstørrelse
2. bestemme algoritmens grunnleggende operasjonsfunksjon  $f(n)$  som en funksjon av inputstørrelsen.
3. sette opp en rekurrensligning  $T(n) = \dots T(\dots) + \dots + f(n), T(1) = \dots$
4. løse rekurrensen på en eller en annen måte

## 3 Rekurrensregning

Det finnes mange ulike måter å løse rekurrenser på. Vi skal se på fire metoder som kan brukes i dette faget, disse er: iterasjonsmetoden, substitusjonsmetoden, tremetoden og bruk av masterteoremet. Noen av disse metodene gir en raskere løsning i enkelte tilfeller, en del trening vil gi mye innsikt i valg av den riktige metoden.

### 3.1 Iterasjonsmetoder

Iterasjonsmetoder karakteriseres med substitusjon av et uttrykk eller en løsning til en instans av en rekurrens inn i en større instans av den samme rekurrensen.

#### 3.1.1 Foroversubstitusjon

Denne metoden er ganske rett-fram, vi regner  $T(1), T(2), T(3), \dots$  inntil vi ser et generelt mønster. Løsningen kommer ved å finne hva det  $n$ -te resultatet blir.



Figur 1: Tower of Hanoi

**Tower of Hanoi** Et kjent eksempel for en rekursiv algoritme er løsningen til 'Tower of Hanoi'-problemet [Wik07]:

*Gitt tre stav (A,B,C) og N klosser. Klossene skal flyttes fra stav A til stav C via stav B. Det er forbudt å sette en mindre kloss opp på en som er større. Problemet er da å finne en ordning for å utføre flyttingen.*

Løsningen er som følger: N klosser kan flyttes fra A til C ved å flytte (N-1) klosser fra A til B, så flytte den ene klossen fra A til C, og flytte de (N-1) mindre klosser fra B til C. Løsningen er triviell for N=1.

```
def Hanoi(N, A, B, C):
    if N > 1 :
        Hanoi(N-1, A, C, B)
        print "Move one disk from " + A + " to " + C
        Hanoi(N-1, B, A, C)
    else :
        print "Move one disk from " + A + " to " + C
```

Brukseksempel:

```
>>> Hanoi(3, "A", "B", "C")
Move one disk from A to C
Move one disk from A to B
Move one disk from C to B
Move one disk from A to C
Move one disk from B to A
Move one disk from B to C
Move one disk from A to C
>>>
```

En kan fort se at  $N=1$  gir 1 flytting,  $N=2$  gir 3 flyttinger,  $N=3$  gir 7 flyttinger, osv. Kan vi si noe generelt om hvor mange flyttinger blir det utført for en gitt  $N$ ? For algoritmen over blir antall flyttinger proporsjonal til kjøretiden, dermed kan vi se en sammenheng mellom kjøretidsanalysen og det med å telle antall ganger en gitt operasjon vil utføres.

Når det gjelder kjøretidsanalysen: først velger vi inntputstørrelsen - det er hvor mange diskene som skal flyttes,  $N$ . Videre bestemmer vi den grunnleggende operasjonen - det med å flytte en disk/ å skrive ut en tekstinne,  $f(n) = 1$ . I tillegg til det, for hver  $\text{Hanoi}(N, \dots)$  kalles  $\text{Hanoi}(N-1, \dots)$  eksakt to ganger når  $N$  er større enn 1. Da kan vi beskrive algoritmen over med følgende rekurrenslikning:

$$T(n) = 2T(n-1) + 1, T(1) = 1 \quad (1)$$

**-Så! Hvordan skal en løse dette?!** La oss først finne  $T(2)$  ved hjelp av  $T(1)$ ,  $T(3)$  ved hjelp av  $T(2)$ ,  $T(4)$  ved hjelp av  $T(3)$ , osv. :

$$T(1) = 1 \quad (2)$$

$$T(2) = 2 * T(1) + 1 = 2 + 1 = 3 \quad (3)$$

$$T(3) = 2 * T(2) + 1 = 6 + 1 = 7 \quad (4)$$

$$T(4) = 2 * T(3) + 1 = 14 + 1 = 15 \quad (5)$$

$$(6)$$

Hvis vi bare fortsetter lenge nok, får vi følgende utvikling:

$$1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, 2047, \dots \quad (7)$$

Dette ligner veldig mye på en eksponentiell rekke, nemlig:

$$2^1 - 1, 2^2 - 1, 2^3 - 1, \dots \quad (8)$$

Tilsvarende blir det  $n$ -te tallet i denne utviklingen  $2^n - 1$ . Følgelig:

$$T(n) = 2^n - 1 = \Theta(2^n) \quad (9)$$

Som en ser, metoden er veldig grei og krever minst mulig opplæring. Derimot blir det fort umulig å gjenkjenne vanskelige rekker, derfor kan denne metoden brukes bare i noen enkelte tilfeller. For eksempel, hva er det neste tallet til følgende rekke: 2, 8, 24, 64, 160? Hvis du tipper at det er 384 - søk Mensa-medlemskap nå!

### 3.1.2 Tilbakesubstitusjon

Tilbakesubstitusjon er generelt en veldig god metode som kan brukes overalt. I motsetning til foroversubstitusjon, ved tilbakesubstitusjon går vi fra  $T(n)$  mot  $T(1)$  ved å uttrykke en større instans med en mindre og akkumulere alle resterende ledd til en matematisk rekke, så løse den istedet. Samtidig finnes det en rekke lure triks som kan lette og effektivisere prosessen.

Tilbake til 'Tower of Hanoi'. Vi vet at  $T(n) = 2T(n-1) + 1$ ,  $T(n-1) = 2T(n-2) + 1$ ,  $T(n-2) = 2T(n-3) + 1, \dots$ . Da kan vi uttrykke  $T(n)$  først ved hjelp av  $T(n-1)$ , så ved hjelp av  $T(n-2)$ , osv. Underveis observerer vi hva skjer med konstantene:

$$T(n) = 2T(n-1) + 1 \quad (10)$$

$$T(n) = 2(2T(n-2) + 1) + 1 \quad (11)$$

$$T(n) = 2(2(2T(n-3) + 1) + 1) + 1 \quad (12)$$

$$\dots \quad (13)$$

Hvis vi ganger dette ut:

$$T(n) = 2T(n-1) + 1 \quad (14)$$

$$T(n) = 4T(n-2) + 2 + 1 \quad (15)$$

$$T(n) = 8T(n-3) + 4 + 2 + 1 \quad (16)$$

$$\dots \quad (17)$$

Videre gjetter vi følgende løsning for  $T(n)$  som et uttrykk av  $T(n-j)$  for en  $0 < j < n$ :

$$T(n) = 2^j T(n-j) + 2^{j-1} + 2^{j-2} + \dots + 2^2 + 2^1 + 2^0 \quad (18)$$

Vi vet også at  $T(1) = 1$ . La  $T(n-j) = T(1)$ , altså  $j = n-1$ :

$$T(n) = 2^{n-1} T(1) + 2^{n-2} + 2^{n-3} + \dots + 2^2 + 2^1 + 2^0 \quad (19)$$

Siden  $T(1) = 1$ :

$$T(n) = 2^{n-1} + 2^{n-2} + 2^{n-3} + \dots + 2^2 + 2^1 + 2^0 T(n) = \sum_{j=0}^{n-1} 2^j \quad (20)$$

Nå kan vi bruke enten **Appendix A** i Cormen ([CLR90], s. 1060) eller Rottman sin formelsamling [Rot],  $\sum_{k=0}^n x^k = \frac{x^{n+1}-1}{x-1}$ :

$$T(n) = \frac{2^n - 1}{1} = 2^n - 1 = \Theta(2^n) \quad (21)$$

Ferdig!

**Variabelskifte** Hvis vi ser på algoritmen under:

```
def FindMax(list):
    n=len(list)
    if n > 1:
        a = FindMax(list[:n/2])
        b = FindMax(list[n/2:])
        if a > b:
```

```

    return a
  else if ;
    return b
else :
  return list [0]

```

Er denne algoritmen asymptotisk sett raskere enn en sekvensiell løsning? Er de like gode? Vi vet at den sekvensielle algoritmen, som vil bare gå gjennom listen en gang og sammenligne alle tall mot den beste verdien så langt, er  $O(n)$ .

La oss uttrykke løsningen over som en rekurrensligning. Inputstørrelsen er lengden til `list`, den grunnleggende operasjon for en instans er  $f(n) = O(1)$ , og for hver  $T(n)$  kalles  $T(n/2)$  eksakt to ganger:

$$T(n) = 2T(n/2) + O(1), T(1) = 1 \quad (22)$$

For enkelhets skyld (Obs! Egentlig skal en aldri gjøre noe sånt, spesielt på eksamen, fy!):

$$T(n) = 2T(n/2) + 1, T(1) = 1 \quad (23)$$

Som det ble nevnt tidligere, finnes det et ninja-triks for substitusjonsmetoden. Ideen bak er samme som ideen bak variabelskifte for integralregningen, vi erstatter en variabel med en annen som gir litt bedre karma i form av en enklere rekurrensligning.

La  $n = 2^m$  for en viss  $m$ , slik at  $T(n) = T(2^m)$  :

$$T(2^m) = 2T(2^{m-1}) + 1, T(2^0) = 1 \quad (24)$$

Dette hjelper ikke så mye. Men la oss i tillegg definere en  $S(m) = T(2^m)$ . Hvis vi finner en løsning til  $S(m)$ , har vi funnet løsningen til  $T(2^m)$ . Samtidig er det ganske trivielt at  $S(m) = T(2^m)$ ,  $S(m-1) = T(2^{m-1})$ , .... Følgelig :

$$S(m) = 2S(m-1) + 1, S(0) = 1 \quad (25)$$

Denne rekurrensligningen ligner veldig mye på en vi hadde for 'Tower of Hanoi'. Da vet vi også at  $S(m) = \Theta(2^m)$ . Følgelig  $T(n) = T(2^m) = \Theta(n)$ . Nå ser vi også at `FindMax` var ikke noe bedre enn den sekvensielle metoden.

**Variabelskifte, VK** Samme trikset kan brukes også på rekurrenser som inneholder kvadratrot og lignende. La oss studere følgende rekurrensligning:

$$T(n) = 2T(\sqrt{n}) + 1, T(1) = 1 \quad (26)$$

Nett som før, la oss definere  $n = 2^m$  :

$$T(2^m) = 2T(\sqrt{2^m}) + 1, T(1) = 1 \quad (27)$$

Det er det samme som :

$$T(2^m) = 2T(2^{m/2}) + 1, T(1) = 1 \quad (28)$$

Igjen, la oss definere  $S(m) = T(2^m)$  :

$$S(m) = 2S(m/2) + 1, S(0) = 1 \quad (29)$$

Fra forrige eksempel vet vi at  $S(m) = \Theta(m)$ . Følgelig  $T(n) = T(2^m) = \Theta(m) = \Theta(\log(n))$ .

Som en ser, blir det fort mye regning. Men tenk ikke at det er *noen avanserte mattegreier*. Helt omvendt, det er en veldig kjapp og grei regnemetode der presisjon er veldig lite relevant. Konstanter i  $f(n)$  spiller ingen rolle for resultatet, det spiller heller ingen rolle om det står  $T(1) = 1$  eller  $T(0) = 1$ , eller faktisk  $T(54) = 42$ . Samme gjelder også om man bruker 1 eller  $O(1)$ , men prøv å bruke  $O(1)$  når det er  $O(1)$ , ihvertfall på eksamen.

### 3.2 Substitusjonsmetoden

Problematikken med iterasjonsmetodene er at det kan bli veldig vanskelig å sjekke at løsningen vi har funnet er riktig. Vi kan selvfølgelig sjekke at løsningen stemmer for  $n = 1$ , og for  $n = 2$ , og for  $n = 3$ , osv. Men vi får ikke bevise at løsningen er riktig for alle  $n$  på denne måten.

Induksjon er en generell metode for å bevise at en regel gjelder alle instanser og består av to steg:

- Basis: Vis at regelen gjelder for en viss instans  $M(k)$
- Induksjonssteg: Vis at dersom regelen gjelder for en instans  $M(j)$ , gjelder den også for en instans  $M(j + 1)$ . For en  $j \geq k$ .

Strategien bak substitusjonsmetoden er da å gjette en generell løsning som stemmer for en enkel instans av  $T$  og deretter bevise at dersom en løsning er gyldig for en instans, er den også gyldig for en større instans.

For eksempel, hvis vi bare gjetter at løsningen til  $T(n) = T(n/2) + 1, T(1) = 0$  er  $T(n) \leq \log_2(n)$ :

- $T(1) = 0 \leq \log_2(1) = 0$
- Antar at  $T(n) \leq \log_2(n)$  for  $n/2$ . I så fall:  

$$T(n) = T(n/2) + 1 \leq \log_2(n/2) + 1 = \log_2(n) - \log_2(2) + 1 = \log_2(n)$$

Ofte kan en bruke noe mer komplisert, for eksempel lange polynomer med flere konstante ledd. Husk også at vi kan se bort fra konstante ledd i selve  $T()$ , det spiller ingen rolle om det er  $T(n/3)$  eller  $T(n/3 + 42)$ .

I motsetning til iterasjonsmetodene, ved substitusjonsmetoden må en være ganske forsiktig og presis. For eksempel, rekurrensen er  $T(n) = 4T(n/2) + n$  og vi antar at løsningen er  $T(n) \leq cn^2$ :

$$T(n) = 4T(n/2) + n \leq 4c(n/2)^2 + n = cn^2 + n \quad (30)$$

Vi mislykkes å bevise at  $T(n) \leq cn^2$  siden vi får en ekstra  $n$  i tillegg til  $cn^2$ . Å bruke  $O$ -notasjon blir en feil. Så, hva kan vi gjøre? En god løsning er å utnytte feilen ved å trekke det overflødig fra antagelsen, multiplisert med en konstant i tillegg. La oss gjette nå at  $T(n) \leq n^2 - kn$ .

Anta at  $T(n) \leq n^2 - kn$  holder for  $n/2$  :

$$T(n) = 4T(n/2) + n \leq 4c((n/2)^2 - k(n/2)) + n = cn^2 - kn - (kn - n) \quad (31)$$

Siden  $kn - (kn - n) \leq kn$  for  $k \geq 1$

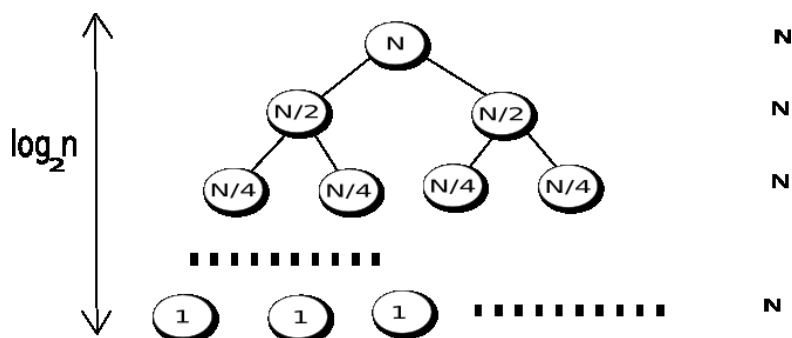
$$T(n) \leq n^2 - kn, k \geq 1 \quad (32)$$

Generelt kan en si at substitusjonsmetoden passer veldig godt som en gjette/bevismetode. Hovedproblemet med denne metoden er at den krever litt mer forståelse og innsikt enn iterasjonsmetodene.

### 3.3 Treemetoden

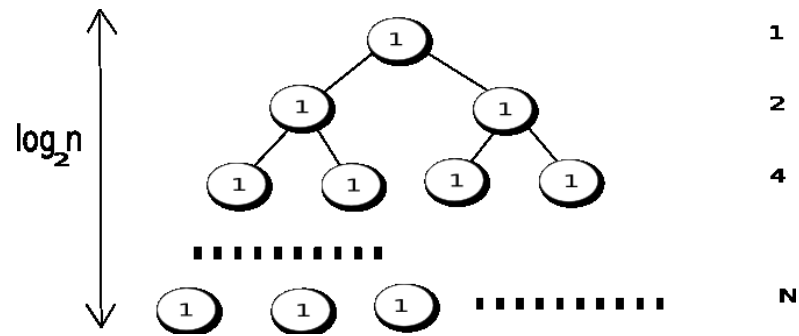
Treemetoden er en annen regnemetode. Hovedideen her er å tenge en rekursjonstre for  $T(n)$  og nedover. Hver node i treet får da verdien av  $f(n)$  i en tilsvarende  $T(n)$ . Deretter summeres alle  $f(n)$  verdier for hele treet. Som regel er det to tilfeller:

- Summen av alle  $f(n)$  for et nivå i treet er konstant. Da kan vi bare gange denne verdien med høyden til treet. Et godt eksempel er  $T(n) = 2T(n/2) + n, T(1) = 1$



Figur 2:  $T(n) = 2T(n/2) + n$

(Fig. 2). Høyden til treet er  $\log_2(n)$  og summen på hvert nivå er  $n$ . Den totale summen er derfor  $n \log_2(n)$

Figur 3:  $T(n) = 2T(n/2) + 1$ 

- Summen av alle  $f(n)$  for et nivå i treet er en funksjon av dybden til noden. Da kan vi sette opp en summasjon og løse den. Et godt eksempel er  $T(n) = 2T(n/2) + 1, T(1) = 1$  (Fig. 3). Summen for et nivå med dybde  $i$  er  $2^i$ . Den totale summen er i så fall  $\sum_1^{\log_2(n)} 2^i = \Theta(n)$ .

### 3.4 Masterteoremet

Masterteoremet fra Cormen [CLR90] sier følgende:

La  $a \geq 0$  og  $b > 0$  være konstanter, la  $f(n)$  være en funksjon, og la  $T(n)$  være definert for ikke negative heltall med rekurrensen  $T(n) = aT(n/b) + f(n)$ , hvor  $n/b$  betyr enten  $\lfloor n/b \rfloor$  eller  $\lceil n/b \rceil$ .

- Hvis  $f(n) = O(n^{\log_b(a)-\epsilon})$  for en  $\epsilon > 0$ ,  $T(n) = \Theta(n^{\log_b(a)})$
- Hvis  $f(n) = \Theta(n^{\log_b(a)})$ ,  $T(n) = \Theta(n^{\log_b(a)} \log(n))$
- Hvis  $f(n) = \Omega(n^{\log_b(a)+\epsilon})$  for en  $\epsilon > 0$  hvor  $af(n/b) \leq cf(n)$  for en konstant  $c < 1$  for alle store  $n$ ,  $T(n) = \Theta(f(n))$

Metoden virker like enkelt som *Smith and Wesson*, en skriver opp variablene, finner riktig case, pof! Samtidig finnes det en del ting en må være forsiktig med for å ikke skyte seg selv i foten. De viktigste er logaritmer og regularitetskondisjon. Regularitetskondisjonen for case 3 ( $af(n/b) \leq cf(n)$ ) gjør at masterteoremet blir ubrukelig på rekurrenser som for eksempel  $T(n) = T(n/2) + n(\sin(n/2) + 2)$ . Hvis  $f(n) = \Theta(n^{\log_b(a)} \log^k(n))$ , virker ingen av casene (siden  $n^\epsilon = \Omega(\log^k(n))$ ) og masterteoremet vil feile<sup>1</sup>.

Noen eksempler på bruk av masterteoremet:

- $T(n) = 2T(n/2) + 1$  gir  $\Theta(n)$  etter case 1.

<sup>1</sup>Det finnes en hemmelig versjon av case 2 i Cormen:

- Hvis  $f(n) = \Theta(n^{\log_b(a)} \log^k(n))$ ,  $T(n) = \Theta(n^{\log_b(a)} \log^{k+1}(n))$

- $T(n) = 2T(n/2) + n$  gir  $\Theta(n \log(n))$  etter case 2
- $T(n) = 2T(n/2) + n \log(n)$  har ingen løsning med masterteoremet, men  $T(n) = 3T(n/2) + n \log(n)$  faller inn for case 1. Vær forsiktig!
- $T(n) = 2T(n/2) + n^2$  gir  $\Theta(n^2)$  etter case 3.

## 4 Videregående metoder

Det finnes mange andre interessante metoder som kan brukes til diverse aspekter av kjøretidsanalysen. Disse ligger dessverre langt utenfor pensum i dette faget, men kan likevel leses for å skaffe en bedre forståelse av emnet:

- Regning av lineære homogene og inhomogene rekurrensrelasjoner med konstante koeffisienter - *'Discrete Mathematics and It's Applications' by Rosen, Chapter 5, 'Recurrence Relations' and 'Solving Recurrence Relations'* [Ros98]
- Amortisert kjøretidsanalyse - *'Introduction to Algorithms' by Cormen et. al., Chapter 17, 'Amortized Analysis'* [CLR90]
- Empirisk kjøretidsanalyse - *'Introduction to the design and analysis of algorithms' by Levitin, Chapter 2, 'Empirical Analysis of Algorithms'* [Lev02]



---

## Referanser

- [CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.
- [ea02] Gunnar Erstad et al. *Matematikk 3MX*. Ascheloug, 2002.
- [Lev02] Anany V. Levitin. *Introduction to the Design and Analysis of Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [Ros98] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Higher Education, 1998.
- [Rot] Karl Rottman. *Matematisk formelsamling*.
- [Wik07] Wikipedia. Tower of hanoi. [http://en.wikipedia.org/wiki/Tower\\_of\\_Hanoi](http://en.wikipedia.org/wiki/Tower_of_Hanoi), 2007.