14
Algoritmeanalyse.

Algoritmebegrebet.

Hvad er algoritmeanalyse?

Problemstørrelse og køretid.

O og Ω.

Køretid for forskellige kontrolstrukturer.

Eksempler på algoritmeanalyse.

Eksponentiel og polynomiel tidskompleksitet.

PS1 -- Algoritmeanalyse

Noter

Hvad er en algoritme?

En *algoritme* er en endelig mængde af instruktioner, som løser et bestemt problem.

En algoritme skal endvidere opfylde følgende kriterier:

- *Input-parametre*: Algoritmen er baseret på nul eller flere eksternt angivne input-parametre.
- *Resultat.* Algoritmen producerer mindst et resultat, som også kan betegnes som output.
- *Klarhed*: Hver instruktion skal være klar og utvetydig.
- *Endelighed*: Algoritmen vil terminere efter et endeligt antal trin.
- *Enkelthed*: Hver instruktion skal være så enkel, at instruktionen kan udføres af en person, som vil følge algoritmen.

Noter

I resten af dette kapitel vil vi bruge ordet “program” synonymt med ordet “algoritme”. Den eneste egentlige forskel er, at nogle programmer ikke opfylder endelighedskriteriet; programmer som er uendelige er eksempelvis operativsystem-drivere. Sådanne programmer er ikke relevante at diskutere inden for rammerne af dette kursus (og dette er altså årsagen til, at vi kan tillade os at veksle mellem de to betegnelser).

Ovenstående definition er hentet (og oversat) fra Horowitz og Shani’s bog “Fundamentals of Data Structures”. I denne bog kaldes klarhed for *definiteness*, endelighed for *finiteness*, og enkelthed for *effectiveness*.

Hvad er algoritmeanalyse?

Algoritmeanalyse handler om at måle, beregne og estimere en algoritmes ressourceforbrug.

Hvilke ressourcer?

- **algoritmens køretid.**
- algoritmens lagerforbrug.

Hvordan?

- Empiriske målinger af et algoritmens (programmets) faktiske køretid.
- Teoretisk beregning af algortments køretid.
- Estimat af størrelseorden af algoritmen køretid.

Hvorfor?

- For at kunne kontrollere og sikre, at programmets samlede køretid (og svartider i en dialog) ikke overstiger en fastsat grænse, og for at sikre at programmet kan køre i det tilgængelige lager.

PS1 -- Algoritmeanalyse

Noter

Problemstørrelse og køretid.

Problemets størrelse:

- Et kvantitativt mål for omfanget af de data, algoritmen bearbejder.
- Udtrykkes ved en eller flere parametre.

Algoritmens køretid:

- Det antal beregningstrin, der udføres af algoritmen, udtrykt som funktion af problemets størrelse.
- Relativ til en valgt beregningsmodel.
- Middelkøretid eller "worst case" køretid?
  - Det er ikke altid entydigt hvad "middel" (gennemsnit, "average") betyder.
  - Middelkøretider er ofte vanskelige at beregne.
  - "Worst case" køretid udtrykker nyttig *øvre grænse.*

En simpel og grov model:

- Alle primitive kommandoer og operationer tager én tidsenhed.
- Uendelig meget arbejdslager.

Noter

Antagelsen om uendelig meget arbejdslager skyldes, at vi ønsker at eliminere forhold såsom "swoping" (paging) til og fra pladelager fra vores model.

Hvad er primitive kommandoer og primitive operationer? Der er ingen faste definitioner af dette. Det er et valg som fastlægges af vores beregningsmodel. Til de primitive kommandoer hører dog klart assignments og visse kald af "primitive procedurer", som vi ikke kan eller ønsker at opdele yderligere i vores analyse. Artitmetiske operationer, sammenligningsoperationer, og boolske operationer tilhører de primitive operationer.

O og Ω.

- Vi ønsker på en simpel måde at kunne estimere øvre og nedre grænse af en funktion.
- Vi ønsker indbyrdes at kunne sammenligne funktioners vækst.

Øvre grænse.

T(n) = O(f(n)) hvis og kun hvis
Der findes to konstanter c og m så T(n) <= c f(n) for n >= m.

f(n) er den øvre grænse af T(n).

T(n) = O(f(n)): "T(n) er af orden f(n)" eller "T(n) er store O af f(n)".

Nedre grænse.

T(n) = Ω (g(n)) hvis og kun hvis
Der findes to konstanter c og m så T(n) >= c g(n) for n >= m.

PS1 -- Algoritmeanalyse

Noter

Definitionerne på denne slide kan læses uafhængig af den sammenhæng, vi befinder os i, nemlig algoritmeanalyse og programmers estimerede køretid. Når vi taler om funktioner, som vi ønsker at estimere og sammenligne, tænker vi dog primært på programmers køretid. Som omtalt på en tidligere slide, udtrykker vi netop programmers køretid som en funktion af en eller flere parametre, der tilsammen måler problemets størrelse.

Store O notationen er langt den mest anvendte, idet den udtrykker et "loft" over en funktion (køretiden). Undertiden er det dog også interessant at udtrykke en nedre grænse for (et "gulv" under) en funktion.

Når vi udtrykker den øvre grænse af en funktion med O-notation ønsker vi næsten altid så lav en grænse som muligt.

Eksempel:

> Lad f(n) = n * n + 2 n. Da gælder både at f(n) = O(n*n) og f(n) = O(n*n*n), men den første af disse er klart den bedste.

Hvis et programs køretid (relativ til en eller anden beregningsmodel) er k(n), hvor n udtrykker problemstørrelsen, og hvis k(n) = O(f(n)), da siger vi ofte at programmets (asymptotiske) *tidskompleksitet* er f(n).

Køretid for forskellige konstrolstrukturer.

Køretid(*command*) = k
Køretid(*command-1*) = k1
Køretid(*command-2*) = k2

Kontrolstruktur	***Køretid***
command-1 ; *command-2*	k1 + k2
for i := 1 **to** n **do** *command*.	n * (k + 1)
while *boolean-expr* **do** *command*	N * (k + e) + e *Hvor* : N betegner antallet af iterationer og e betegner køretiden for det boolske udtryk.
from *initialization-commands* **until** *boolean-exp* **loop** *command* **end**	i + N * (k + e) + e *Hvor* : i betegner køretiden for initialiseringer N betegner antallet af iterationer og e betegner køretiden for det boolske udtryk.
if *boolean-expr* **then** *command-1* **else** *command-2* **end**	e + max(k1, k2) *Hvor* : e betegner køretiden for det boolske udtryk.

PS1 -- Algoritmeanalyse

Noter

For-løkke: et-tallet i køretiden stammer fra (re-)initialisering af løkke-variablen.

I while-løkken, og i Eiffels loop konstruktion, kan man ikke udtale sig om antallet af iterationer på sammen måde som i for-løkken. Derfor betegnes antallet af faktisk udførte iterationer med variablen N.

I både while og loop strukturen testes den boolske betingelse én gang mere end løkken udføres. Dette afspejles med et ekstra e-bidrag i formlerne for køretiderne.

Eksempel på algoritmeanalyse (1)

Matrix addition.

```
class MATRIX
feature

 add(m: MATRIX): MATRIX is
 require number_of_rows = m.number_of_rows;
         number_of_columns = m.number_of_columns;
 local row, column, sum: integer
 do
   result.create(number_of_rows, number_of_columns);
   from row := 1
   until row > number_of_rows
   loop
     from column := 1
     until column > number_of_columns
     loop
       sum := item(row,column) + m.item(row,column);
       result.put(sum,row,column);
       column := column + 1
     end;
     row := row + 1
   end
 end -- add

end --class MATRIX
```

Analyse af matrix-addition (add):

Lad a = m.number_of_rows
b = m.number_of_columns

Problemets størrelse: (a,b).

Køretiden af add = O(a*b)

PS1 -- Algoritmeanalyse

Noter

Eksempel på algoritmeanalyse (2).

Beregning af Fibonacci tal.

```
fib(n: integer): integer is
do
  if n = 0
  then result := 0
  elsif n = 1
  then result := 1
  else result := fib(n-2) + fib(n-1)
  end
end
```

Analyse:

Problemets størrelse: n
Lad køretiden af kaldet fib(n) være K(n).

K(0) = K(1) = O(1).
K(n) = K(n-2) + K(n-1) + 3, for n >= 2.

Det ses umiddelbart, at K(n) > fib(n).

Det kan vises, at fib(n) >= $(3/2)^n$

Derfor er køretiden af fib eksponentiel.

PS1 -- Algoritmeanalyse

Noter

I kapitlet "Specifikation med logiske udtryk" har vi set en iterativ udgave til beregning af Fibonacci tal. Den så ud som følger:

```
myfib(n: integer): integer is
  local a: integer;
        b: integer;
        count: integer;
        br: integer
  do
   from a := 1; b := 0; count := n;
   until count = 0
   loop
     br := b; b:= a;
     a := a + br; count := count - 1
   end;
   result := b
  end -- myfib
```

Det ses umiddelbart, at kørentiden af myfib er O(n). Der er altså enorm stor forskel mellem tidskompleksiteten af myfib, og den rekursive fib, som formuleret ovenfor.

Senere i dette kapitel vil vi se på, hvor slem eksponentiel køretid egentlig er.

Eiffel funktionerne til beregning af Fibonacci tal fra denne slide er indeholdt i filen
/user/normark/public/p1/gcd_og_fib.e.

Eksempel på algoritmeanalyse (3).

Euclid's algoritme.

Rekursiv formulering

```
gcd(a, b: integer): integer is
require a >= b, b >= 0
do
  if b = 0
  then result := a
  else result := gcd(b, a mod b)
  end
ensure a mod result = 0 and
         b mod result = 0;
         -- result er stoerste divisor i a og b
end;
```

Iterativ formulering

```
gcd(a, b: integer): integer is
require a >= b; b >= 0
local rem, x, y: integer;
do
  from x := a; y := b
  until y = 0
  loop
    rem := x mod y;
    x := y;
    y := rem;
  end;
  result := x
ensure a mod result = 0 and
         b mod result = 0;
         -- result er stoerste divisor i a og b
end;
```

Analyse:

Problemets størrelse: a.
Lad N betegne antallet af iterationer i gcd.
Køretiden af gcd = O(N).
Det kan vises at N <=2 log a.
Køretiden af gcd er således O(log a).

PS1 -- Algoritmeanalyse

Noter

Funktionen gcd beregner den største divisor af to tal.

Lad os først indse, hvorfor gcd i de to ovenstående formuleringer rent faktisk beregner den største divisor af de to parametre. Det er ikke vanskeligt at vise, at

(a • b and b > 0) => gcd(a,b) = gcd(b, a mod b).

Bemærk at b > a mod b; derved kan vi umiddelbart anvende formlen igen:

gcd(b, a mod b) = gcd(a mod b, b mod (a mod b)).

Vi kan altså danne følgende række af tal

a, b, a mod b, b mod (a mod b), ..., r, 0.

Hvis rækken således er ... x, y, er det næste led lig med x mod y. Da

gcd(r, 0) = r

har vi, at det næstsidste led i rækken (r ovenfor) er lig med gcd(a,b).

Den rekursive formulering af Euclid's algoritme er "pænere" end den iterative formulering. Men i mange sprog vil den rekursive formulering afstedkomme et stort lagerforbrug, idet hvert rekursivt kald repræsenteres på kaldstakken. Ved nærmere eftertanke kan man dog let indse, at dette store lagerforbrug ikke er nødvendigt i dette tilfælde. Årsagen er, at gcd er **halerekursiv:** Det rekursive resultat benyttes ikke i en yderligere beregning inden funktionen returnerer. Så hvis vores programmeringssprog håndterer halerekursion optimalt, er den rekursive formulering lige så effektiv som den iterative.

Eiffel funktionerne til beregning af største fælles divisor fra denne slide er indeholdt i filen /user/normark/public/p1/gcd_og_fib.e.

Opgave: Imod den iterative løsning vist ovenfor kan man invende, at den ikke er "original og basal nok". Man kan formulere en version af Euclids algoritme som anvender gentagen subtraktion af b fra a i stedet for 'a mod b' (eller x mod y, som det forekommer i programmet). Skriv denne version af Euclids algoritme, og prøvekør den.

Eksponentiel tidskompleksitet.

Algoritmer med eksponentiel tidskompleksitet kan i praksis antages at have uendelig køretid for store input-størrelser.

n	2^n	$10^{-9} 2^n$
5	32	$3 * 10^{-8}$
25	$3*10^7$	0.03
50	10^{15}	12 dage
65	$4*10^{19}$	1330 år
80	10^{24}	$3 * 10^7$ år
100	10^{30}	$3 * 10^{13}$ år

PS1 -- Algoritmeanalyse

Noter

Den højre kolonne viser køretiden, hvis vi antager at hvert primitiv har en udførelsestid på 1 nanosekund.

Til sammenligning kan vi betragte en tilsvarende tabel for n * n *n, hvor vi dog i stedet for tilsidst at betragte n = 100 går op på n = 1000.

Tabellen er vist på næste slide.

Polynomiel tidskompleksitet.

n	n^3	$10^{-9}\ n^3$
5	125	$1.25 * 10^{-7}$
25	15625	$1.56 * 10^{-5}$
50	125000	$1.25 * 10^{-4}$
100	10^6	10^{-3}
1000	10^9	1 sekund

Noter

Sædvanligvis betragtes algoritmer med tidskompleksitet n*n*n ikke som særligt attraktive. Men sammenlignet med eksponentiel tidskompleksitet ser vi, at der er en gigantisk forskel.