28
Algoritmedesign.

Algoritmeskabelon for ‘Del og Hersk’.
Eksempler på ‘Del og Hersk’ algoritmer.
- Binær søgning i et ordnet array.
- Sortering ved fletning og Quicksort.
- Maksimal delsums problem.
- Tætteste par af punkter i et plan.
- Multiplikation af to heltal.
- Hanois Tårne.

Dynamisk programmering.
Algoritmeskabelon for Dynamisk Programmering.
Beregning af Fibonaccital ved Dynamisk programmering.
Knapsack problemet og veksleproblemet.

PS1 -- Algoritmedesign

Noter

Algoritmeskabelon for 'Del og Hersk'.

Løsning af problemet P:

- *Opdel* P problemet i et antal delproblemer P1, P2, ... Pn.
- *Løs* hver af delproblemerne Pi, og få del-løsningerne Li, i = 1 .. n.
- *Kombiner* del-løsningerne Li, i = 1..n, til en løsning L af det samlede problem P.

```
Løs-del-og-hersk(P: Problem_type): Løsning-type is
do
  If  P er et simpelt problem
  then  returner den umiddelbare løsning af P
  else  Opdel P i  delproblemerne  P1, P2,  ... Pn ;
        Let L1 be Løs-del-og-hersk(P1)
           ..
           Ln  be Løs-del-og-hersk(Pn)
        in
          Kombiner   L1  og ... og  Ln  til   L ;
          returner   L
end --Løs
```

PS1 -- Algoritmedesign

Noter

I forbindelse med sortering har vi tidligere på kurset set to gode eksempler på del og hersk algoritmer: Sortering ved fletning og quicksort. Disse adskiller sig dog ved

at quicksort bruger O(n) til på opdelingen af problemet i to delproblemer

og at sortering ved fletning bruger O(n) tid på kombinationen af de to delløsninger til en samlet løsning på "det store problem".

Det er "det mest almindelige og typiske" at der betales en eller anden ikke konstant pris for kombinationen af delløsninger, og at problemopdelingen er "billig" (ofte af konstant køretid).

Både quicksort og sortering ved fletning gav anledning til de samme rekursionsligninger for køretiden:

$T(1) = 1$

$T(n) = 2\ T(n/2) + c\ n$

hvilke som bekendt har løsningen $T(n) = n \log(n)$. En stor delmængde af del-og-hersk algoritmer kan beskrivives ved et generaliseret ligningssystem:

$T(1) = 1$

$T(n) = a\ T(n/b) + c\ n^k$

Denne ligning afspejler at der løses a delproblemer, som hver er 1/b-te del af n´s størrelse. Løsningen på ligningerne fremgår af sætning 10.6 i Weiss.

Det maksimale delsumsproblem (1).

Problemet:

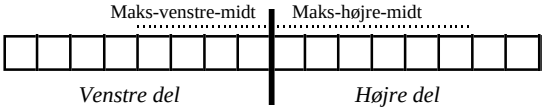
Givet et array med n heltal $a_1, ..., a_n$.
Find den maksimale delsum

$$\sum_{k=i}^{j} a_i$$

dog således, at resultatet er 0 hvis alle tal $a_1, ..., a_n$ er negative.

Simpel løsning: Systematisk gennemsøgning af alle mulige dellister: $O(n^2)$

Del og Hersk løsning:



En løsning på problemet er den største delsum i enten
- Den venstre del, eller
- Den højre del, eller
- En del, som krydser midten.

PS1 -- Algoritmedesign

Noter

Eksempel:

i:	1	2	3	4	5	6	7	8	9	10
a[i]:	5	-3	2	6	-2	-1	5	-1	2	0

Arrayet a opdeles i en venstre del a[1..5] og en højre del a[6..10]. I den venstre del er a[1..4] den maksimale delsum (sum 10). I den højre del er a[7..9] en maksimal delsum (sum 6). Over midten er a[1..9] den maksimale delsum (sum 13). Det er klart at det her er midtersummet som bliver det samlede resultat.

Det maksimale delsumsproblem (2).

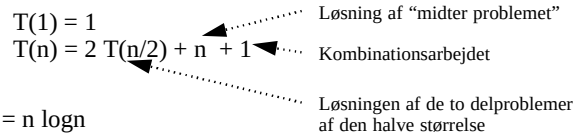
```
Maksimal-delsum(a: array[integer]; v, h: integer): integer:
 if v = h
 then returner max(a[v], 0)
 else center := (v + h) div 2;
      venstre-sum := Maksimal-delsum(a, v, center);
      højre-sum := Maksimal-delsum(a, center+1, h);

      let maks-venstre-midt be den største delsum i venstre del som indeholder a[center]
          maks-højre-midt be den største delsum i højre del, som indeholder a[center+1]

     returner max(venstre-sum, højre-sum, maks-venstre-midt + maks-højre-midt)
```

Køretidsfunktion:

n = antal elementer i array.



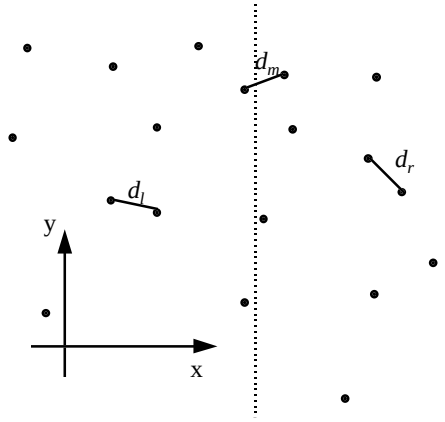
T(n) = n logn

Noter

Der henvises til afsnit 2.4.3 i Weiss, samt figur 2.7, for en mere præcis beskrivelse af ovenstående algoritme.

Den minimale afstand mellem to punkter i et plan.

Problemet: Find den minimale afstand mellem to forskellig punkter i planen.



PS1 -- Algoritmedesign

Noter

Dette problem beskrives i detaljer i Weiss afsnit 10.2.2. Ovenstående illustration vil tjene som udgangspunkt af en diskusion af en effektiv del og hersk løsning ved forelæsningen.

Multiplikation af to tal.

Problemet: To store n-cifrede tal x og y ønskes multipliceret.

Simpel løsning: "Skolebogs multiplikation".

Eksempel: Køretid: $O(n^2)$

$$\begin{array}{r} 12345 \cdot 678 \\ \hline 98760 \\ 864150 \\ 7407000 \\ \hline 8369910 \end{array}$$

Ad A $T(1) = 1$, $T(n) = 4\,T(n/2) + cn$ $\quad T(n) = \Theta(n^2)$

Ad B $T(1) = 1$, $T(n) = 3\,T(n/2) + cn$ $\quad T(n) = \Theta(n^{1.59})$

Del og Hersk løsning:

$x = x_{left} \cdot 10^i + x_{right}$
$y = y_{left} \cdot 10^i + y_{right}$

$$x\,y = x_{left}\,y_{left} \cdot 10^{2i} + (x_{left}\,y_{right} + x_{right}\,y_{left})\,10^i + x_{right}\,y_{right} = \quad (A)$$

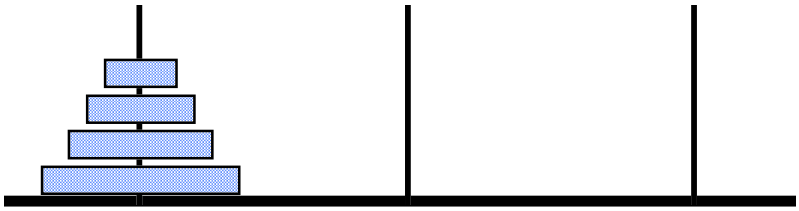
$$x_{left}\,y_{left} \cdot 10^{2i} + ((x_{left} - x_{right})(y_{right} - y_{left}) + x_{left} \cdot y_{left} + x_{right}\,y_{right})10^i + x_{right}\,y_{right} \quad (B)$$

PS1 -- Algoritmedesign

Noter

Weiss behandler dette problem i afsnit 10.2.4.

Towers of Hanoi (1).



Givet:

Et antal skiver på venstre stang, hvor
mindre skiver er placeret oven på større skiver.

Problemet:

Flytte stakken af skiver fra venstre til højre stang, således at

- skiverne flyttes én ad gangen.
- en større skive må aldrig placeres oven på en mindre skive undervejs i flytteprocessen.

Noter

I “Towers og Hanoi” opdeles det oprindelige problem i to delproblemer, som begge er én mindre end det oprindelige problem. Dette fremgår klart af detaljerne på næste slide. Derved kan “Towers of Hanoi” siges at være en del og hersk algoritme, omend den er væsensforskellig fra de andre eksempler, vi har studeret.

Det viser sig, at de to delproblemer er overlappende. Dette bliver årsagen til, at køretiden for “Towers of Hanoi” er eksponentiel. Dette er et iboende problem i “Towers of Hanoi”. I resten af dette kapitel vil vi imidlertid se på et antal tilsvarende “overlappende del og hersk” problemer, hvor vi kan høste en betydelig gevinst ved at tage hånd om overlappet i delproblemerne (og dermed også i løsningerne).

Towers of Hanoi (2).

```
Hanoi (n: integer; A, B, C: stang)
   -- Et antal skiver  >= n befinder sig på stang A.
   -- Flyt de n øverste af disse over på stang B,
   -- med brug af stang C som "mellemstation"

 If n = 1
 then Flyt_én_skive(A, B)
 else Hanoi (n-1, A, C, B);
      Flyt_én_skive(A, B);
      Hanoi (n-1, C, B, A);
```

Lad T(n) betegne antallet af "skive flytninger".

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

$T(n) = O(2^n)$

Hanoi(3, venstre, højre, midten):

1: venstre -> højre
2: venstre -> midten
1: højre -> midten
3: venstre -> højre
1: midten -> venstre
2: midten -> højre
1: venstre -> højre

Hanoi(4, venstre, højre, midten):

1: venstre -> midten
2: venstre -> højre
1: midten -> højre
3: venstre -> midten
1: højre -> venstre
2: højre -> midten
1: venstre -> midten
4: venstre -> højre
1: midten -> højre
2: midten -> venstre
1: højre -> venstre
3: midten -> højre
1: venstre -> midten
2: venstre -> højre
1: midten -> højre

Noter

Der er ovenfor vist eksempler på Towers of Hanoi for n=3 og n=4. Hvert kald af proceduren Flyt-én-skive give anledning til et primitiv bidrag til løsningen på formen:

i: x -> y

hvor x og y er enten venstre, midten eller højre. Symbolet i er værdien af parameteren n i den inkarnation af 'Hanoi', som forårsagede trækket.

Towers of Hanoi (3).

Hanoi(5, venstre, midter, højre):

Hanoi 5 venstre midter højre
Hanoi 4 venstre højre midter
Hanoi 3 venstre midter højre
Hanoi 2 venstre højre midter
Hanoi 1 venstre midter højre
Hanoi 1 midter højre venstre
Hanoi 2 højre midter venstre
Hanoi 1 højre venstre midter
Hanoi 1 venstre midter højre
Hanoi 3 midter højre venstre
Hanoi 2 midter venstre højre
Hanoi 1 midter højre venstre
Hanoi 1 højre venstre midter
Hanoi 2 venstre højre midter
Hanoi 1 venstre midter højre
Hanoi 1 midter højre venstre

Hanoi 4 højre midter venstre
Hanoi 3 højre venstre midter
Hanoi 2 højre midter venstre
Hanoi 1 højre venstre midter
Hanoi 1 venstre midter højre
Hanoi 2 midter venstre højre
Hanoi 1 midter højre venstre
Hanoi 1 højre venstre midter
Hanoi 3 venstre midter højre
Hanoi 2 venstre højre midter
Hanoi 1 venstre midter højre
Hanoi 1 midter højre venstre
Hanoi 2 højre midter venstre
Hanoi 1 højre venstre midter
Hanoi 1 venstre midter højre

Noter

Denne slide viser de rekursive kald, i stedet for de primitive flytninger i Towers of Hanoi. Det bemærkes, at store dele af processen gentager sig selv (de to afsnit med **fed skrift**, og de to afsnit med *kursiv skrift*)

Introduktion til 'Dynamisk Programmering': Fibonacci tal (1).

```
fib_husker: hukommelse[integer, integer];
...

fib(n: integer): integer is
 local fib_part1, fib_part2: integer
  do
    if n = 0 then result := 0
    elsif n = 1 then result := 1
    else  if fib_husker.er_beregnet(n - 1)
          then fib_part1 := fib_husker.værdi(n - 1)
          else fib_part1 := fib(n - 1);
               fib_husker.husk(n-1, fib_part1)
          end;
          if fib_husker.er_beregnet(n - 2)
          then fib_part2 := fib_husker.værdi(n - 2)
          else fib_part2 := fib(n - 2);
               fib_husker.husk(n -2, fib_part2)
          end;
          result := fib_part1 + fib_part2;
    end
 end;
```

```
class hukommelse[T,S]
-- Afbilder en værdi af type T til
-- den huskede værdi af typen S
feature
   husk(x:T, y: S)
   -- x huskes til at have værdi y

   er_beregnet(x:T): boolean
   -- returnerer hvorvidt der er husket en
   -- værdi for x.

   værdi(x:T):S
   -- returner den huskede værdi for x
   require er_beregnet(x)
end;
```

Køretidsfunktion T.
Problemstørrelse n.

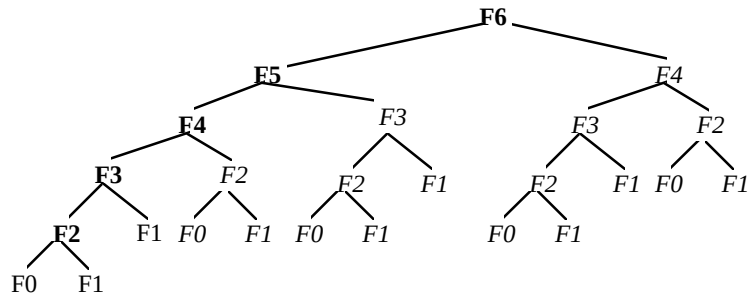
T(n) = O(n).

Noter

Denne slide viser en modificeret udgave af den rekursive udgave af Fibonaccital beregning, som vi har set på ved adskillige lejligheder på dette kursus. Der mindes om, at køretiden R(n) af den rekursive Finoaccitals funktion er •((3/2)^n), altså også af eksponentiel tidskompleksitet.

Ideen er at undgå genberegninger af fib(n), for givne n. Dertil har vi introduceret en instans af en klasse, som kaldes Hukommelse. Et objekt af denne klasse afbilder et problem til en løsning; I vort tilfælde f.eks. problemet "af bregne fib(6)" som har løsningen 8. Intern vil det være attraktive at bruge en hashtabel til realisering af afbildningen.

Introduktion til 'Dynamisk Programmering': Fibonacci tal (2).



PS1 -- Algoritmedesign

Noter

Denne slide illustrerer beregningsprocessen for fib(6), her kaldet F6. Figuren svarer til figur 10.42 i Weiss. De **fede** afsnit i figuren beregnes og huskes i hukommelsen. De *kursiverede* afsnit tages fra hukommelsen. Først lægges F2 ind, så F3, F4, F5 og tilsidst F6. Når F2 skal benyttes ifm. beregningen af F4 kan F2 trækkes F2 ud af tabellen i stedet for at skulle genberegnes. Endnu større gevinst er der med F3 og F4 ifm. beregning af hhv. F5 og F6.

Det er let ud fra figuren at se, at fib, med brug af denne teknik, har køretid der er lineær i n. Dette er en dramatisk forandring i forhold til den tidligere eksponentielle tidskompleksitet. Bemærk dog, at pladskompleksiteten er vokset med denne løsning (fra konstant til lineær pladsforbrug i n).

I litteraturn kaldes ovenstående, generelt anvendelige teknik ofte for 'dynamisk programmering'. Andre steder tales der om 'memoization'.

Algoritmeskabelon for Dynamisk Programmering.

```
Husker: hukommelse[Problem_type, Løsning-type]

Løs-dynamisk-prog(P: Problem_type): Løsning-type is
do
  If Husker.er-bereget(P)
  then returner  Husker.værdi(P)
  else  If P er et simpelt problem
  then returner den umiddelbare løsning af P
  else  Opdel P i  delproblemerne  P1, P2,  ... Pn ;
        Let L1 be Løs-dynamisk-prog(P1)
           ..
           Ln  be Løs-dynamisk-prog(Pn)
        in
          Kombiner   L1  og ... og  Ln  til   L ;
           Husker.husk(P, L);
          returner   L
end --Løs-dynamisk-prog
```

PS1 -- Algoritmedesign

Noter

Ovenstående forståes bedst som en videreudvikling af skabelonen for 'Del og Hersk' som er vist tidligere i dette kapitel. Vi har tilføjet administrationen af huskede løsninger på problemer via en instans af klassen Hukommelse.

Knapsack problemet (1).

Problemet:

Givet n typer af objekter, som har hhv. værdi v_j og vægt m_j, j = 1 .. n.
Fyld en rygsæk med objekter (0, 1 eller flere af type 1; 0, 1 eller flere af type 2, ...)
således at rygsækken ender med at have den størst mulige værdi
uden at overskride totalvægt T.

Simpel løsning:

```
Masse, værdi: array[1..n] of integer;
...

Knapsack(totalvægt: integer): integer:
var resultat_værdi: integer

  result-værdi := 0 ;
  for j := 1 to n do
    if masse[j] <= totalvægt
    then resultat-værdi := max(resultat_værdi,
                               værdi[j] + Knapsack(totalvægt - masse[j]) );
  return resultat-værdi
```

Køretidsfunktion T.
Problemstørrelse: totalvægten.
Hvis $m_j = 1$, j = 1 .. n

T(0) = n
T(m) = n * T(m -1)

$T(m) = n^{m+1}$

Noter

Resultatet af funktionen er den opnåelige maksimale værdi af rygsækkens indhold.

Det kan observeres, at Knapsack funktionen kan kaldes mange gange med den samme totalvægt. Ved hvert kald udføres den samme (dyre) beregning af den maksimale værdi relativ til totalvægten. Ligesom de andre problemer, vi har studeret, er nøglen til en forbedring at "memoizere" allerede løste delproblemer, og deres løsninger. Dette vises på næste slide.

Knapsackproblemet (2)

Køretidsfunktion T.
Problemstørrelse: totalvægten (m).
$T(m) = O(m \cdot n)$

```
Masse, værdi: array[1..n] of integer;

husker: hukommelse(integer, integer);
...

Knapsack(totalvægt: integer): integer:
var resultat_værdi: integer;

  if husker.er-beregnet(totalvægt)
  then return husker.værdi(totalvægt)
  else result-værdi := 0 ;
       for j := 1 to n do
       if masse[j] <= totalvægt
       then resultat-værdi := max(resultat_værdi,
                                  værdi[j] + Knapsack(totalvægt - masse[j]) ) ;
       husker.husk(total_vægt, resultat-værdi)
       return resultat-værdi
```


Noter

Vi postulerer ovenstående køretidsresultat.

Variation af Knapsack problemet: Veksleproblemet (1).

Problem:

På hvormange forskellige måder kan man veksle n kroner i gængse danske mønter (25 øre, 50 øre, 1 kroner, 2 kroner, 5 kroner, 10 kroner og 20 kroner)?

Algoritmisk ide

Antal af måder at veksle n kroner ved brug af ovenstående mønter =

\+ Antal af måder at veksle n kroner uden brug af 25 ører
Antal af måder at veksle (n - 0.25) kroner ved brug af alle ovenstående mønter.

```
Veksle(beløb: Ører; antal-mønt-typer: Integer): Integer
 if beløb = 0 then result := 1
 elseif beløb < 0  or
        antal-mønt-typer = 0 then result := 0
 else result := Veksle(beløb - pålydende(antal-mønt-typer), antal-mønt-typer) +
                Veksle(beløb, antal-mønt-typer -1)
```


Noter

Den algoritmiske ide stammer fra følgende betragtninger. Studer f.eks. alle mulige vekslinger af 20 kroner. Del disse vekslinger i to grupper:

1. De vekslinger, som ikke gør brug af 25 ører.
2. De vekslinger, som gør brug af 25 ører.

Det er klart at antallet af måder vi kan veksle 20 kroner er summen af antal vekslinge af 1 og 2 ovenfor.

Ad 2: Det ses let, at antallet af disse er det samme som antallet af måder vi kan veksle 19,75 kr ved brug af alle mønter.

For de nysgerrige kan det oplyses, at 20 kroner kan veksles på 3275 måder.

I ovenstående algoritmeskitse er det antaget, at mønterne ordnes, f.eks. efter pålydende. Når alle mønttyper skal tages i betragtning er 2. parameter til Veksle lig med 7. I det sidste rekursive kald bliver antal-mønt-typer lig med 6, hvilket afspejler at den 7. mønt (f.eks. 25 øren) ikke er med længere. Udtrykket pålydende(i) returnerer den pålydende værdi af den i'te mønttype.

Det ses, at veksleproblemet er et specialtilfælde af Knapsack-problemet. Vi kan sige, at massen og værdien er sammenfaldende (eller at vi kun interesserer os for værdien). Vi kræver imidlertid ved veksleproblemet, at den vægtede sum af værdier anddrager præcist det beløb, som skal veksles.

Variation af Knapsack problemet: Veksleproblemet(2).

(Veksle 200 4) = 10 Veksling af 2 kroner med de 4 mindste mønter.

- 200 200
- 200 100
- 200 50
- 200 25
- 200 0
- 175 25
- 175 0
- 150 25
- 150 0
- 125 25
- 125 0
- 100 25
- 100 0
- 75 25
- 75 0
- 50 25
- 50 0
- 25 25
- 25 0
- 0 25
- 150 50
+ 150 25
+ 150 0

+ 125 25
+ 125 0
+ 100 25
+ 100 0
+ 75 25
+ 75 0
+ 50 25
+ 50 0
+ 25 25
+ 25 0
+ 0 25
- 100 50
+ 100 25
+ 100 0
+ 75 25
+ 75 0
+ 50 25
+ 50 0
+ 25 25
+ 25 0
+ 0 25
- 50 50
+ 50 25

+ 50 0
+ 25 25
+ 25 0
+ 0 25
- 0 50
- 100 100
+ 100 50
+ 100 25
+ 100 0
+ 75 25
+ 75 0
+ 50 25
+ 50 0
+ 25 25
+ 25 0
+ 0 25
+ 50 50
+ 50 25
+ 50 0
+ 25 25
+ 25 0
+ 0 25

+ 0 50
- 0 100
- 0 200

PS1 -- Algoritmedesign

Noter

Denne slide viser de rekursive kald af veksleproceduren.

+/- beløb mønt

\+ betyder, at der er tale om en gen-beregning.

\- betyder, at beregningen ikke er foretaget tidligere.

Beløb er det beløb, som forsøges vekslet.

Mønt er den særlige mønt, der tages hensyn til i det aktuelle problem.

Bemærk, at i ovenstående tager vi kun hensyn til 25 ører, 50 ører, 1 kroner og 2 kroner.

Verfice selv at Veksle(100, 3) = 4 (veksling af en 1 krone ved brug af 25 ører, 50 ører, og 1 krone).