$T[\![\ll$p er i S$\gg]\!] \in O(k)$
$T[\![\ll$find løsning l direkte$\gg]\!] \in O(k)$
$T[\![\ll$del p i $p_1,\ldots,p_k \gg]\!] \in O(k)$
$T[\![\ll$kombiner $l_1 \ldots l_k$ til l$\gg]\!] \in O(k)$

ALGORITMISK PROBLEMLØSNINGSTEKNIK

Contents

0 Indledning

Dette hæfte introducerer de tre mest udbredte generelle teknikker til løsning af algoritmiske problemer. Hver teknik præsenteres v.hj.a. et antal eksempler og karakteriseres derudover i form af en såkaldt *algoritmeskabelon*, som er en slags abstrakt standardalgoritme, der omfatter alle konkrete algoritmer af den pågældende type.

Den første teknik, del-og-kombiner, er den vigtigste og er den, der har de bredeste anvendelsesområder. Den belyses bl.a. ved hjælp af tre centrale algoritmiske problemer, hvor den fører til effektive løsninger.

De to øvrige teknikker, dynamisk programmering og kombinatorisk søgning, kan opfattes som en slags 1. og 2. reserver, der evt. kan bringes i anvendelse, når del-og-kombiner teknikken svigter. Det er naturligvis ikke altid et algoritmisk problem lader sig løse på hensigtsmæssig vis v.hj.a. en af disse standardteknikker. I sådanne situationer må man starte fra bunden og konstruere algoritmer på systematisk vis v.hj.a. udsagn, invarianter, termineringsfunktioner osv. (jfr. [Programmeringsteori I + II]). Teknikkerne i dette hæfte skal altså opfattes som en slags supplement til den "algoritmiske værktøjskasse", der i forvejen indeholder programmering (i Trine), centrale begreber fra programmeringsteorien og datastrukturer (jfr. [Datastrukturer]).

1 Del-og-kombiner

Del-og-kombiner[1] teknikken bygger på den simple observation, at hvis et problem kan opdeles i et antal simplere problemer, hvis løsninger kan kombineres til en løsning af det oprindelige problem, så er en sådan opdeling et skridt på vejen mod en løsning af selve problemet. Sagt lidt mere præcist kan del-og-kombiner teknikken anvendes i situationer, hvor et problem p

[1] Det engelske navn for teknikken er *divide-and-conquer*, som vel nærmest burde oversættes ved del-og-hersk, men del-og-kombiner er en mere dækkende betegnelse.

kan opdeles i et antal delproblemer $p_1, \ldots, p_k$ på en sådan måde at

1. hvert delproblem p_i er simplere end p
2. løsningerne til $p_1, \ldots, p_k$ kan kombineres til en løsning af p.

Det skulle være klart, at man ved gentagen anvendelse af denne teknik får reduceret sit problem til et (muligvis stort) antal simple problemer, som man så antages at kunne løse direkte. I konkrete anvendelser af denne teknik vil man ofte realisere løsningerne af delproblemerne $p_1, \ldots, p_k$ i form af kald af én eller flere rutiner, som hver løser et delproblem.

Hvis disse delproblemer er af *samme slags* som det oprindelige problem, kan man introducere en procedure til at løse det oprindelige problem, som vil kunne skrives (ofte meget elegant og overskueligt) under anvendelse af *rekursion*.

Procedurerne `reverse1` og `reverse2` s. 35 i [Programmeringsteori I] er et udmærket eksempel herpå, idet ligningen

$$\tilde{L} = L(i..|L|) + + L(0..i)$$

på meget direkte måde repræsenterer følgende anvendelse af del-og-kombiner:

> Problemet *spejling af en følge* L løses ved at opdele L i to mindre problemer, *spejling af* $L(0..i)$ og *spejling af* $L(i..|L|)$, hvis løsninger, $L(0..i)$ og $L(i..|L|)$, kombineres til $\tilde{L}$ ved simpel omvendt sammensætning.

Det skulle være nemt at se, at proceduren `Kochline` s. 159 i [TRINE] også er skrevet under anvendelse af del-og-kombiner teknikken. Her er problemet at tegne en kurve af en vis længde i en vis retning. Dette deles op i det (simple) problem at tegne en streg, efterfulgt af fire delproblemer, der består i at tegne mindre kurver af passende længde i passende retninger.

1.1 Skabelonen

Da del-og-kombiner teknikken er anvendelig i en lang række forskellige situationer, præciserer vi den i form af følgende såkaldte *algoritme-skabelon*, som er en (abstrakt) algoritmebeskrivelse, i hvilken der udover stimulans, respons og metode også optræder et såkaldt *univers*, som er en beskrivelse af egenskaberne ved de elementer metode-delen manipulerer samt af de operatorer, der anvendes på dem.

Skabelon: Del-og-kombiner

Univers : En klasse af *problemer* P, som har *løsninger* i en klasse L. P består af *simple* problemer S og *komplekse* problemer K, dvs. $P = S \cup K$. Problemer $p \in S$ har en *direkte* løsning, medens problemer $p \in K$ kan *deles* i et antal mindre problemer $p_1, \ldots, p_k \in P$, hvis løsninger $l_1, \ldots, l_k \in L$ kan *kombineres* til $l \in L$, hvor l er en løsning til P.

Stimulans : p: P, problem der ønskes løst

Respons : l: L, løsning af p

Metode :

```
proc løs[p : P, l : L]
    if ≪ p er i S ≫→
        ≪find løsning l direkte≫
    & true →
        (+ Var p_1, ..., p_k : P
           Var l_1, ..., l_k : L
           ≪del p i p_1, ..., p_k ≫
           løs[p_1, l_1]
           ⋮
           løs[p_k, l_k]
           ≪kombiner l_1, ..., l_k til l ≫
        +)
    fi
end løs

løs[p, l]
```

Eksemplet med spejling af tegnfølger repræsenterer en konkretisering af skabelonen, hvor problemerne P består i at spejle tegnfølger af vilkårlig længde. De simple problemer er dem, hvor længden af følgerne er mindre end eller lig med 1, og løsningen består i at "gøre ingenting". Tegnfølger med længde større end 1 deles i to kortere tegnfølger, og kombinationen af løsninger består i at sætte løsningsfølgerne efter hinanden.

Vi overlader det til læseren at overbevise sig om, at programmet Kochline også repræsenterer en konkretisering af del-og-kombiner skabelonen. Bemærk

i den forbindelse, at *løsninger* ikke altid repræsenteres i form af eksplicitte parametre i del-og-kombiner procedurerne. Løsninger kan, afhængigt af situationen, f.eks. også repræsenteres af følger af udskrifter (i grafikpakken). Når vi i det følgende konkretiserer skabelonen, vil det altid fremgå af sammenhængen, hvordan løsninger repræsenteres.

1.2 Hanois Tårne

Hanois Tårne er også et eksempel på en anvendelse af del-og-kombiner, hvor skabelonen er konkretiseret på følgende måde.

P: flytning af n ($n > 0$) skiver fra én stang til én af de to tomme stænger under anvendelse af den anden tomme stang som mellemstation.

L: følger af enkeltflytninger, hvor en enkeltflytning består i at flytte den øverste skive fra én stang til en anden. En enkeltflytning angives ved

$$\alpha \rightarrow \beta$$

hvor α og β er navnet på en stang.

K: problemer med mindst 2 skiver.

Det er naturligvis trivielt at løse de simple problemer. Med hensyn til de komplekse problemer i K, løses de som bekendt ved, at flytning af n skiver fra α til δ via β kan opdeles i tre delproblemer på følgende måde

p_1: flyt $n-1$ skiver fr α til β via δ
p_2: flyt 1 skive direkte fra α til δ
p_3: flyt $n-1$ skiver fra β til δ via α

Hvert af disse problemer er mindre end det oprindelige, og deres løsninger udgør tilsammen en løsning af p. Det er imidlertid ikke umiddelbart klart, at p_1, p_2 og p_3 tilhører klassen P, fordi vi her har forudsat at de to af stængerne er *tomme*, og det er δ-stangen ikke, når vi løser p_3. Det er imidlertid oplagt, at da δ-stangen indeholder den *største* af skiverne, som

alle andre skiver kan ligge oven på, gør det ikke noget, at den ikke er tom. Vi kan klare dette formelle problem ved at *generalisere* problemklassen P til

> P: flytning af de øverste $n (n \geq 0)$ skiver fra én stang til én af to andre stænger, som begge kun indeholder skiver, der er større end de n skiver, der skal flyttes

og så er det nemt at se, at såvel p som p_1, p_2 og p_3 alle tilhører P.

Sortering er en anden problemstilling, hvor del-og-kombiner finder anvendelse og kan føre til hensigtsmæssige algoritmer. Vi skal se på to eksempler, som går under navnene hhv. Quicksort og Flettesortering.

1.3 Quicksort

Quicksort sorterer elementerne i en vektor i ikke-aftagende orden v.hj.a. omflytninger. Ideen i algoritmen kan illustreres på følgende måde. Hvis elementerne i en vektor

w:

opdeles på følgende måde

w: e

elementer$\leq$ e e$\leq$elementer

kan w sorteres ved at sortere de to "halvdele", som hver repræsenterer et mindre problem af samme slags. Til selve opdelingen kan vi bruge algoritmen Opdel s. 22 i [Programmeringsteori I], idet vi som deleelement vælger et tilfældigt element i vektoren.

Idet vi repræsenterer delvektorer v.hj.a. to "pegepinde" venstre og højre, foregår sorteringen mere præcist på følgende måde.

Algoritme: Quicksort
Stimulans : w: vector
Respons : w: sorteret i ikke-aftagende orden
Metode : **proc** opdel [w: Vector, r: Int](*venstre,højre,e*: Int)
≪Udfør algoritmen Opdel med stimulans
w(*venstre..højre*) og e, og med respons r ≫
end Opdel

proc quicksort [w: Vector](*venstre,højre*: Int)
if *venstre* < *højre*−1 →
(+ **var** i, r: int
≪vælg i så *venstre* ≤ i < *højre*≫
opdel [w, r](*venstre,højre,w.*(i))
{ w(*venstre..r*) ≤ e ≤ w(*r..højre*) og
$w.(i)$ er ikke blevet flyttet}
if $i < r \rightarrow r := r - 1$ **fi**
$w.(i) :=: w.(r)$
{$w.(r)$ står rigtigt }
quicksort[w](*venstre,r*)
quicksort[w](*r+1,højre*)
+)
fi
end quicksort

quicksort[w]$(0, |w|)$

Følgende komplette TRINE program realiserer algoritmen:

```
Process Quicksort
    (+ Proc Opdel[w: Vector, r: Int](venstre, højre, e: Int)
          (+ Var lav, høj: Int
             lav, høj := venstre, højre
             do lav<høj →
                   if w.(lav) ≤ e → lav := lav+1
                   | (w.(høj-1)<e) ∧ (e<w.(lav)) →
                         w.(lav) :=: w.(høj-1)
                         lav, høj := lav+1, høj-1
                   | e ≤ w.(høj-1) →  høj := høj-1
                   fi
             od
             r := lav
          +)
       end Opdel

       Proc Quicksort[w: Vector](venstre, højre: Int)
          if venstre<højre-1 →
                (+ Var r, i: Int
                   i := random(venstre, højre)
                   Opdel[w, r](venstre, højre, w. (i))
                   if i<r → r := r-1 fi
                   w.(i) :=: w.(r)
                   Quicksort[w](venstre, r-1)
                   Quicksort[w](r+1, højre)
                +)
          fi
       end Quicksort

       Var u: Vector

       write("Indlæs vektor:  ")
       read[u]
       Quicksort[u](0, | u |)
       write("Resultat:  ", u, eol)
    +)
end Quicksort
```

Læseren opfordres til at overbevise sig om, at quicksort er en korrekt konkretisering af del-og-kombiner skabelonen, herunder især at de i algoritmen anførte udsagn er gyldige.

Da algoritmen bærer navnet *quick* sort, ville man forvente, at den var effektiv. Det er den imidlertid ikke altid, hvilket kan indses på følgende måde.

Det er klart, at der for algoritmen Opdel gælder

$$T[\![\text{Opdel}]\!](n) \in \Theta(n)$$

hvor n nu er længden af vektoren $w(\textit{venstre..højre})$, dvs. $n = \textit{højre-venstre}$.

De eneste sætninger i kroppen af quicksort, der har ikke-konstant omkostning, er kaldet af opdel samt de to rekursive kald af quicksort. Vi har

derfor

$$\begin{aligned} T[\![\textbf{proc}\ \text{quicksort}]\!](n) \ \approx\ & T[\![\textbf{proc}\ \text{opdel}]\!](n) + \\ & T[\![\textbf{proc}\ \text{quicksort}]\!](m) + \\ & T[\![\textbf{proc}\ \text{quicksort}]\!](n-m-1) \end{aligned}$$

hvor $m = r\text{-}\mathit{venstre}$. Størrelsen af m afhænger altså af, hvor heldige vi er med valget af det element $w.(i)$, der bruges til at opdele efter. Hvis uheldet er ude, kan vi risikere hver gang at vælge det mindste element i w. Det betyder at $r = \mathit{venstre}$, dvs. at $m = 0$. Nu er det klart at

$$T[\![\textbf{proc}\ \text{quicksort}]\!](0) \in O(1)$$

hvorfor ligningen ovenfor bliver

$$T[\![\textbf{proc}\ \text{quicksort}]\!](n) \approx n + T[\![\textbf{proc}\ \text{quicksort}]\!](n-1)$$

Men så følger det at

$$T[\![\textbf{proc}\ \text{quicksort}]\!](n) \in \Theta(n^2)$$

Worst case tidskompleksiteten af quicksort er således ikke bedre end de simple sorteringsalgoritmer.

Betragter man derimod den *forventede* udførelsestid, ser sagen anderledes ud. Dette følger af, at det element der udvælges til at dele på, $w.(i)$, vælges *tilfældigt* blandt elementerne i $w(\mathit{venstre..højre})$. Idet vi for resten af denne analyse antager, at alle elementer i w er parvis forskellige, betyder dette, at værdien af r (dvs. placeringen af delepunktet) også er tilfældig. Men så kan vi finde den forventede tidskompleksitet af quicksort (som vi skal betegne med T_E) på følgende måde

$$\begin{aligned} & T_E[\![\textbf{proc}\ \text{quicksort}]\!](n) \approx \\ & \quad n + \sum_{m=0}^{n-1} \frac{1}{n}(T_E[\![\textbf{proc}\ \text{quicksort}]\!](m) + T_E[\![\textbf{proc}\ \text{quicksort}]\!](n-m-1)) \end{aligned}$$

idet der er samme sandsynlighed, $\frac{1}{n}$, for alle værdier af m.

Man kan, ved induktion efter n, bevise, at når $T_E[\![\textbf{proc}\ \text{quicksort}]\!]$ tilfredsstiller denne ligning, så findes der en positiv konstant k, således at

$$T_E[\![\textbf{proc}\ \text{quicksort}]\!](n) \leq kn\log(n)$$

hvoraf vi kan konkludere, at

$$T_E[\![\textbf{proc quicksort}]\!](n) \in O(n\log(n))$$

Dette indikerer, og praksis bekræfter da også, at Quicksort faktisk er en effektiv sorteringsmetode.

1.4 Flettesortering

Det næste eksempel på del-og-kombiner er som nævnt flettesortering. Ideen i denne algoritme er, at man kan sortere en vektor ved at dele den i to halvdele, som sorteres hver for sig, og derefter flette de to halvdele sammen til én sorteret følge.

Algoritme: Flettesortering
Stimulans : w: vector
Respons : w: sorteret
Metode : **proc** flet $[w_1, w_2, w$: Vector]
≪udfør algoritmen Fletning med
stimulans w_1 og w_2 og respons w ≫
end flet

proc sorter $[w$: Vector]
if $|w| > 1 \rightarrow$
(+ **var** $w1, w2$: vector
$w1, w2 := w(0..|w|/2), w(|w|/2..|w|)$
sorter$[w1]$
sorter $[w2]$
flet $[w1, w2, w]$
+)
fi
end sorter

sorter $[w]$

Det skulle være klart, at algoritmen er korrekt, og m.h.t. dens kompleksitet er det også klart, at der gælder

$$\begin{aligned} T[\![\textbf{proc}\ \text{sorter}]\!](n) \ \approx\ & n + \\ & 2 * T[\![\textbf{proc}\ \text{sorter}]\!](\frac{n}{2}) + \\ & T[\![\textbf{proc}\ \text{flet}]\!](n) \end{aligned}$$

hvor n angiver længden af w. Hvis flettealgoritmen kan implementeres så den får kompleksitet $O(n)$, giver dette anledning til ligningen

$$T[\![\textbf{proc}\ \text{sorter}]\!](n) \approx 2 * (n + T[\![\textbf{proc}\ \text{sorter}]\!](\frac{n}{2}))$$

Dette er den samme ligning som opstod under analysen af proceduren `reverse2`, og løsningen er

$$T[\![\textbf{proc}\ \text{sorter}]\!](n) \in \Theta(n \log(n))$$

Følgende TRINE program viser en implementation af flettesortering, hvor kompleksiteten af fletningen er $O(n)$. Læseren opfordres til at studere hvordan dette foregår.

```
Process Flettesortering
    (+ Proc flet[w1, w2, w: Vector]
        (+ Var i1, i2, i: Int
           w := Vector(0 | | w1 |+| w2 |)
           i1, i2, i := 0, 0, 0
           do ¬ (i1 = | w1 |) ∧ (i2 = | w2 |) →
                 w.(i) := w1.(i1)
                 i1, i := i1+1, i+1
            | (i1 = | w1 |) ∧ ¬ (i2 = | w2 |) →
                 w.(i) := w2.(i2)
                 i2, i := i2+1, i+1
            | ¬ (i1 = | w1 |) ∧ ¬ (i2 = | w2 |) →
                 if w1.(i1) ≤ w2.(i2) →
                       w.(i) := w1.(i1)
                       i1, i := i1+1, i+1
                  | w2.(i2) ≤ w1.(i1) →
                       w.(i) := w2.(i2)
                       i2, i := i2+1, i+1
                 fi
           od
        +)
    end flet

    Proc sorter[w: Vector]
        if | w |>1 →
            (+ Var w1, w2: Vector
               w1, w2 := w(0 .. | w |/2), w(| w |/2 .. | w |)
               sorter[w1]
               sorter[w2]
               flet[w1, w2, w]
            +)
```

```
        fi
    end sorter

    Var u: Vector

    write("Indlæs vektor:  ")
    read[u]
    sorter[u]
    write("Resultat:  ", u, eol)
+)
end Flettesortering
```

Det skal bemærkes, at Flettesortering, i modsætning til Quicksort, *ikke* er et eksempel på en omflytningsalgoritme, således som dette er defineret i [Programmeringsteori I]. Årsagen er, at fletningen *kopierer* elementerne fra listerne w1 og w2 til listen w, dvs. den indskrænker sig ikke til blot at bruge ombytninger. Dette er af betydning for procedurens *lagerforbrug*, hvilket kan ses på følgende måde.

Hvis vi, i analogi med betegnelsen for tidskompleksitet, betegner procedurens pladsforbrug *eksklusive* hvad parametrene fylder, med $P[\![\textbf{proc}\ \text{sorter}]\!]$ får vi

$$\begin{aligned} P[\![\textbf{proc}\ \text{sorter}]\!](n) &\approx n + \\ &\quad 2 * P[\![\textbf{proc}\ \text{sorter}]\!](\frac{n}{2}) + \\ &\quad P[\![\textbf{proc}\ \text{flet}]\!](n) \\ P[\![\textbf{proc}\ \text{flet}]\!](n) &\approx 1 \end{aligned}$$

fordi sorter indeholder to lokale variabler w_1, w_2, hvis samlede længde er n, medens proceduren flet kun indeholder simple lokale variabler. (Bemærk, at udsagnet i procedure flet angiver, at der for ethvert kald flet[w_1,w_2,w] gælder, at $|\mathrm{w}| = |\mathrm{w}_1| + |\mathrm{w}_2|$.)

Denne ligning for $P[\![\textbf{proc}\ \text{sorter}]\!]$ er igen velkendt og har løsning

$$P[\![\textbf{proc}\ \text{sorter}]\!](n) \in \Theta(n \log(n))$$

Der bruges med andre ord lige så meget ekstra lager som der bruges tid.

Dette kan være kritisk i praktiske anvendelser, men heldigvis kan man klare sig med mindre. Der kræves nemlig blot én hjælpevektor w af længde n, idet man kan flette skiftevis fra w til w_1 og fra w_1 til w. De to stumper, der flettes, er altid naboer og deres afgrænsning kan derfor angives på følgende måde

w:

Efter fletningen vil w_1 have følgende udseende

w_1:

hvor w_1(lav..høj) indeholder fletningen af w(lav..midt) og w(midt..høj).

Et program, der fungerer på denne måde, ser ud som følger. Variablen p af type Position bruges til at angive, om data befinder sig på w_1 eller w_2.

```
Process FlettesorteringEnTo
    (+ Proc FletEnTo[w1, w2: Vector] (lav, midt, høj: Int)
        (+ Var i1, i2, i: Int
           i1, i2, i := lav, midt, lav
           do ¬ (i1 = midt) ∧ (i2 = høj) →
                 w2.(i) := w1.(i1)
                 i1, i := i1+1, i+1
            | (i1 = midt) ∧ ¬ (i2 = høj) →
                 w2.(i) := w1.(i2)
                 i2, i := i2+1, i+1
            | ¬ (i1 = midt) ∧ ¬ (i2 = høj) →
                 if w1.(i1) ≤ w1.(i2) →
                       w2.(i) := w1.(i1)
                       i1, i := i1+1, i+1
                  | w1.(i2) ≤ w1.(i1) →
                       w2.(i) := w1.(i2)
                       i2, i := i2+1, i+1
                 fi
           od
        +)
    end FletEnTo

    Type Position = Sum(en, to: Unit)

    Proc SorterEnTo[w1, w2: Vector, p: Position] (lav, høj: Int)
        if høj-lav>1 →
            (+ Var midt: Int
               Var p1, p2: Position
               midt, p1, p2 := (lav+høj)/2, p, p
               SorterEnTo[w1, w2, p1] (lav, midt)
               SorterEnTo[w1, w2, p2] (midt, høj)
               if p1 ≠ p2 → ≪w1(midt..høj):=:w2(midt..høj≫ fi
               if is(p1, en) →
                     FletEnTo[w1, w2] (lav, midt, høj)
                     p := Position(to: #)
```

```
                    | is(p1, to) →
                        FletEnTo[w2, w1](lav, midt, høj)
                        p := Position(en: #)
                fi
            +)
        fi
    end SorterEnTo

    Var u, v: Vector
    Var p: Position

    write("Indlæs vektor:  ")
    read[u]
    v, p := Vector(0 | | u |), Position(en: #)
    SorterEnTo[u, v, p](0, | u |)
    write("Resultat:  ")
    if is(p, en) → write(u, eol)
    | is(p, to) → write(v, eol)
    fi
  +)
end FlettesorteringEnTo
where ≪w1(midt..høj):=:w2(midt..høj≫ is
        (+ Var i: Int
           i := midt
           do i<høj →
                 w1.(i) :=: w2.(i)
                 i := i+1
           od
        +)
```

1.5 Selektion

Et problem, der er beslægtet med sortering, er det såkaldte selektionsproblem, som går ud på at skrive en procedure

Proc Select (w: Vector, k: Int) → (Int)

som finder det k'te element (efter størrelse) i w. Mere præcist skal Select tilfredsstille specifikationen

Select(w,k) **sat** $1 \leq k \leq |w| \rightarrow$ SL(Select′,k,w)

hvor SL er prædikatet

SL(e,k,w): $|\{i \in 0..|w| \mid w.(i)<e\}| < k \leq |\{i \in 0..|w| \mid w.(i) \leq e\}|$

Det er klart, at man kan løse problemet ved at sortere w og dernæst returnere w.(k-1). Dette giver imidlertid en udførelsestid på $\Theta(n \log(n))$, og

vi er ude efter noget bedre.

Det viser sig, at man kan bruge samme grundlæggende idé som i Quicksort, dvs. udvælge et tilfældigt element e i w, opdele w's elementer efter e og så undersøge, hvor tæt delepunktet er på k. Nedenstående algoritme viser hvordan.

Proceduren Split gør lidt mere end opdel i Quicksort, idet den flytter rundt på elementerne i w, så de der er mindre end e står til venstre; de der er lig med e i midten; og de der er større end e står til højre. Den tilfredsstiller følgende specifikation

$$\begin{aligned}&\text{Split[w,m,s](e) } \textbf{sat}\\ &\quad (w'(0..m') < e = w'(m'..s') < w'(s'..|w|)) \wedge\\ &\quad (0 \le m' \le s' \le |w|) \wedge \text{perm}(w,w')\end{aligned}$$

hvor perm(w,w′) betyder, at w′ er en permutation af w.

Algoritme: Selektion
Stimulans: w,k: $1 \le k \le |w|$
Respons : e: SL(e,k,w)
Metode : **Proc** Select(w: Vector, k: Int) → (Int)
(+ **Var** e, m, s: Int
e := w.(random(0, |w|))
Split[w, m, s](e)
if $k \le m \rightarrow$ **return** Select(w(0 .. m), k)
| $m < k \le s \rightarrow$ **return** e
| $s < k \rightarrow$ **return** Select(w(s .. |w|), k-s)
fi
+)
end Select
e := Select(w,k)

Med hensyn til udførelsestid er det nemt at se, at hvis uheldet er ude, opfører algoritmen sig nogenlunde lige som Quicksort, dvs.

$$T[\![\textbf{Proc Select}]\!](n) \in \Theta(n^2)$$

Omvendt kan man (igen i lighed med Quicksort) indse, at hvis alle elementer i w er forskellige, dvs. $m = s - 1$, så er den forventede udførelsestid løsning til ligningen

$$T_E[\![\textbf{Proc Select}]\!]\ (\mathrm{n}) \approx$$
$$n + \frac{1}{n}(1 + \sum_{s=1}^{n} \max(T_E[\![\textbf{proc Select}]\!](s-1),\ T_E[\![\ \textbf{proc Select}]\!](n-s)))$$

Det kan, igen ved induktion efter n, vises at der findes en positiv konstant c så

$$T_E[\![\textbf{proc Select}]\!](n) \leq cn$$

dvs.

$$T_E[\![\textbf{proc Select}]\!](n) \in O(n)$$

1.6 Heltalsmultiplikation

Betragt som sidste eksempel følgende samling procedurer, der understøtter manipulation af lange (decimale) heltal.

De mest centrale er Plus, Minus; CMult; TDiv, TMult og TMod, som hhv. adderer og subtraherer to lange tal; multiplicerer et langt tal med et ciffer; dividerer, multiplicerer og tager modulus af et langt heltal med en 10'er potens. Procedurerne tilfredsstiller følgende specifikationer

$\mathrm{Plus}(t_1, t_2)$	**sat**	$\mathrm{Plus}' = t_1 + t_2$
$\mathrm{Minus}(t_1, t_2)$	**sat**	$t_1 \geq t_2 \rightarrow \mathrm{Minus}' = t_1 - t_2$
$\mathrm{CMult}(t, d)$	**sat**	$\mathrm{CMult}' = t * d$
$\mathrm{TDiv}(t, p)$	**sat**	$\mathrm{TDiv}' = t/10^p$
$\mathrm{TMult}(t, p)$	**sat**	$\mathrm{TMult}' = t * 10^p$
$\mathrm{TMod}(t, p)$	**sat**	$\mathrm{TMod}' = t$ **mod** 10^p

```
(+ @"sequence.tri"
```

Box N
Sequence(Int)

```
end N

Type Tal = N'Seq

Proc Konst(v: Vector) → (Tal)
    (+ Var t: Tal
       N'Con[t](v)
       return t
    +)
end Konst

Proc Plus(t1, t2: Tal) → (Tal)
    (+ Var t: Tal
       Var m, i, s1, s2: Int
       N'Init[t]
       s1, s2 := N'Len[t1], N'Len[t2]
       if s1>s2 →
             s1 :=: s2
             t1 :=: t2
       fi
       i := s1
       do i ≠ s2 →
             N'Rpush[t1](0)
             i := i+1
       od (* t1 og t2 er lige lange *)
       m, i := 0, 0
       do i ≠ s2 →
             m := m+N'Sel[t1](i)+N'Sel[t2](i)
             N'Rpush[t](m mod 10)
             m, i := m/10, i+1
       od
       if m>0 → N'Rpush[t](m) fi
       return t
    +)
end Plus

Proc Minus(t1, t2: Tal) → ( Tal) (* t2 forudsættes at være mindre end eller lig t1 *)
    (+ Var t: Tal
       Var i, s1, s2, s: Int
       s1, s2 := N'Len[t1], N'Len[t2]
       i := 0
       do i ≠ s2 →
             N'Sel[t2](i) := -N'Sel[t2](i)
             i := i+1
       od
       t := Plus(t1, t2)
       s := N'Len[t]
       i := 0
       do i ≠ s →
             if N'Sel[t](i)<0 →
                   N'Sel[t](i) := N'Sel[t](i)+10
                   N'Sel[t](i+1) := N'Sel[t](i+1)-1
             fi
             i := i+1
       od
       return t
    +)
end Minus

Proc Length(t: Tal) → (Int)
    return N'Len[t]
end Length

Proc CMult(t: Tal, d: Int) → (Tal)
    (+ Var i, s, m: Int
       s := N'Len[t]
```

```
        m, i:=0, 0
        do i≠s →
              m:=m+N'Sel[t](i)*d
              N'Sel[t](i), m:=m mod 10, m/10
              i:=i+1
        od
        if m>0 → N'Rpush[t](m) fi
        return t
    +)
end CMult

Proc TDiv(t: Tal, p: Int) → (Tal)
    (+ Var i: Int
        i:=0
        do i≠p →
              N'Lpop[t]
              i:=i+1
        od
        return t
    +)
end TDiv

Proc TMult(t: Tal, p: Int) → (Tal)
    (+ Var i: Int
        i:=0
        do i≠p →
              N'Rpush[t](0)
              i:=i+1
        od
        i:=N'Len[t]
        do i≠p →
              i:=i-1
              N'Sel[t](i):=N'Sel[t](i-p)
        od
        do i≠0 →
              i:=i-1
              N'Sel[t](i):=0
        od
        return t
    +)
end TMult

Proc TMod(t: Tal, p: Int) → (Tal)
    (+ Var i: Int
        Var r: Tal
        i:=0
        N'Init[r]
        do i≠p →
              N'Rpush[r](N'Sel[t](i))
              i:=i+1
        od
        return r
    +)
end TMod

Proc Skriv(t: Tal)
    (+ Var i: Int
        i:=N'Len[t]
        do i>0 →
              i:=i-1
              write(N'Sel[t](i))
        od
        if i=N'Len[t] → write("0") fi
    +)
end Skriv
```

```
Proc Læs[t: Tal] (* Forudsætter at der læses en ikke-tom følge af cifre *)
    (+ Var x: Text
       Var i: Int
       read[x]
       i := 0
       do (i<| x |) ∧ (x.(i) = '0') → i := i+1 od
       x := x(i .. | x |-1)
       N'Init[t]
       i := | x |
       do i ≠ 0 →
            i := i-1
            N'Rpush[t](ci(x.(i))-ci('0'))
       od
    +)
end Læs

Proc Mult1(x, y: Tal) → (Tal)
    (+ Var t: Tal
       Var i: Int
       i := 0
       t := Konst(Vector())
       do i ≠ Length(y) →
            t := Plus(t, TMult(CMult(x, N'Sel[y](i)), i))
            i := i+1
       od
       return t
    +)
end Mult1

Proc Mult2(x, y: Tal) → ( Tal)
    (+ Var t: Tal
       Var p, n: Int
       Var u, v, w, z: Tal
       if Length(x) ≤ Length(y) → n := Length(x)
       | Length(x) ≥ Length(y) → n := Length(y)
       fi
       if n ≤ 2 → return Mult1(x, y) fi
       p := n/2
       u, v := TDiv(x, p), TMod(x, p)
       w, z := TDiv(y, p), TMod(y, p)
       return Plus(TMult(Mult2(u, w), 2*p),
              Plus(TMult(Plus(Mult2(u, z), Mult2(w, v)), p), Mult2(v, z)))
    +)
end Mult2

Proc Mult3(x, y: Tal) → (Tal)
    (+ Var t: Tal
       Var p, n: Int
       Var u, v, w, z, r, uw, vz: Tal
       if Length(x) ≤ Length(y) → n := Length(x)
       | Length(x) ≥ Length(y) → n := Length(y)
       fi
       if n ≤ 2 → return Mult1(x, y) fi
       p := n/2
       u, v := TDiv(x, p), TMod(x, p)
       w, z := TDiv(y, p), TMod(y, p)
       r := Mult3(Plus(u, v), Plus(w, z))
       uw := Mult3(u, w)
       vz := Mult3(v, z)
       return Plus(Plus(TMult(uw, 2*p), TMult(Minus(Minus(r, uw), vz), p)), vz)
    +)
end Mult3

≪Program≫
```

```
+)
```

Det skulle være nemt at se, at alle procedurerne har udførelsestid $O(n)$, hvor n er længden af argumenterne.

Vi ønsker nu at tilføje en procedure, der *multiplicerer* to tal. Følgende procedure anvender den sædvanlige skolealgoritme.

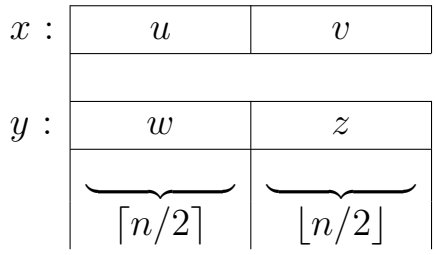
```
Proc Mult1(x, y: Tal) → (Tal)
    (+ Var t: Tal
       Var i: Int
       i := 0
       t := Konst(Vector())
       do i ≠ Length(y) →
             t := Plus(t, TMult(CMult(x, N'Sel[y](i)), i))
             i := i+1
       od
       return t
    +)
end Mult1
```

Det skulle være klart, at

$$T[\![\textbf{proc}\ \mathrm{Mult}_1]\!](n, m) \in \Theta(n * m)$$

hvor n og m er længden af x og y.

Man kan konstruere en multiplikationsalgoritme v.hj.a. del-og-kombiner ved at opdele tallene x og y på følgende måde.



Her er n maksimum af længderne af x og y, og $\lceil r \rceil$ ($\lfloor r \rfloor$) betegner det mindste (største) hele tal, der er større (mindre) end eller lig med r ($\lceil\ \rceil$ og $\lfloor\ \rfloor$ betegnes også hhv. som *loft* og *gulv*).

Hvis vi sætter $p = \lfloor n/2 \rfloor$, er det klart, at

$$\begin{aligned} x &= u * 10^p + v \\ y &= w * 10^p + z \end{aligned}$$

hvoraf

$$(*) \qquad x * y = u * w * 10^{2p} + (u * z + w * v) * 10^p + v * z$$

Dette giver umiddelbart anledning til følgende alternative procedure

```
Proc Mult2(x, y: Tal) → (Tal)
  (+ Var t: Tal
     Var p, n: Int
     Var u, v, w, z: Tal
     if Length(x) ≤ Length(y) → n := Length(x)
      | Length(x) ≥ Length(y) → n := Length(y)
     fi
     if n ≤ 2 → return Mult1(x, y) fi
     p := n/2
     u, v := TDiv(x, p), TMod(x, p)
     w, z := TDiv(y, p), TMod(y, p)
     return Plus(TMult(Mult2(u, w), 2*p),
                 Plus(TMult(Plus(Mult2(u, z), Mult2(w, v)), p),
                 Mult2(v, z)))
  +)
end Mult2
```

Hvis n igen betegner længden af det længste argument, får vi nu

$$T[\![\text{Mult}_2]\!](n) \approx n + 4 * T[\![\text{Mult}_2]\!](\frac{n}{2})$$

hvilket følger af, at (*) udtrykker multiplikation af to n-cifrede tal ved 4 multiplikationer af $n/2$-cifrede tal samt et antal lineære operationer (additioner og multiplikationer med 10'er potenser).

Løsning af denne ligning giver

$$T[\![\text{Mult}_2]\!](n) \in \Theta(n^2)$$

dvs. proceduren er ikke bedre end skolealgoritmen.

Nu kan vi imidlertid observere, at vi kan erstatte en multiplikation med to subtraktioner på følgende måde.

Hvis vi sætter

$$\begin{aligned} r &= (u + v) * (w + z) \\ &= u * w + (u * z + v * w) + v * z \end{aligned}$$

kan vi skrive ligningen (*) som

$$x * y = u * w * 10^{2p} + (r - u * w - v * z) * 10^p + v * z$$

og da vi kun behøver at udregne uw og vz én gang, er vi nede på tre multiplikationer af $n/2$-cifrede tal. Dette fører til følgende procedure

```
Proc Mult3(x, y: Tal) → (Tal)
  (+ Var t: Tal
     Var p, n: Int
     Var u, v, w, z, r, uw, vz: Tal
     if Length(x) ≤ Length(y) → n := Length(x)
      | Length(x) ≥ Length(y) → n := Length(x)
     fi
     if n ≤ 2 → return Mult1(x, y) fi
     p := n/2
     u, v := TDiv(x, p), TMod(x, p)
     w, z := TDiv(y, p), TMod(y, p)
     r := Mult3(Plus(u, v), Plus(w, z))
     uw := Mult3(u, w)
     vz := Mult3(v, z)
     return Plus(Plus(TMult(uw, 2*p),
                 TMult(Minus(Minus(r, uw), vz), p)), vz)
  +)
end Mult3
```

hvis analyse giver

$$T[\![\textbf{proc Mult}_3]\!](n) \approx n + 3 * T[\![\text{Mult}_3]\!](n/2)$$

Denne ligning har løsningen

$$\begin{aligned}
T[\![\textbf{proc Mult}_3]\!](n) &\approx n + 3(\frac{n}{2} + 3(\frac{n}{4} + \ldots + 3))\ldots) \\
&= n(1 + \frac{3}{2} + (\frac{3}{2})^2 + \ldots + (\frac{3}{2})^{\log(n)}) \\
&= n * \frac{1 - (\frac{3}{2})^{\log(n)+1}}{1 - \frac{3}{2}} \\
&= 2n((\frac{3}{2})^{\log(n)+1} - 1) \\
&\approx n * \frac{3^{\log(n)+1}}{2^{\log(n)+1}} \\
&\approx 3^{\log(n)+1} \\
&= n^{\log(3)} \\
&\approx n^{1.59}
\end{aligned}$$

Vi ser altså at del-og-kombiner, sammen med "tricket" med at spare en multiplikation fører til en asymptotisk forbedring i forhold til standard-metoden.

Nedenstående rette linier (tegnet i et dobbeltlogaritmisk hoordinatsystem) viser resultatet af et antal kørsler med de tre metoder. Da analysen kun gælder for store værdier af n, er de tre rette linier kun tilpasset punkter, hvis abcisse er større end eller lig med 16, svarende til multiplikation af 16-cifrede tal. De tre liniers hældningskoefficienter er som følger

L_1 : 1.95
L_2 : 2.01
L_3 : 1.64

Hvis dette sammenlignes med de af analysen forudsagte værdier

L_1 : 2.00
L_2 : 2.00
L_3 : 1.59

er der altså tale om ganske god overensstemmelse.

1.7 Sammenfatning

Del-og-kombiner er som nævnt i indledningen en generel algoritmisk problemløsningsteknik. Når man skal anvende skabelonen i en konkret situation, skal man fastlægge

- problemklassen P
- de simple problemer S
- hvordan man *deler* et problem i *mindre* delproblemer
- hvordan man *kombinerer* løsninger til delproblemer.

Her har man naturligvis en del frihed, men det er altid en god idé at søge at gøre delproblemerne disjunkte og nogenlunde lige store, samt at holde udgifterne til deling og kombination under kontrol, helst så de er $O(n)$. Dette skyldes, at ligningen for tidskompleksiteten af proceduren løs i skabelonen da får følgende udseende

$$\begin{aligned} T[\![\textbf{proc}\ \text{løs}]\!](1) &\approx 1 \\ T[\![\textbf{proc}\ \text{løs}]\!](n) &\approx n + \\ & \quad k * T[\![\textbf{proc}\ \text{løs}]\!](\frac{n}{k}) + \\ & \quad n \end{aligned}$$

hvor vi har antaget, at simple problemer har størrelse 1 og kan løses i konstant tid. Denne ligning kan løses på præcis samme måde som tidligere og resultatet bliver

$$T[\![\textbf{proc}\ \text{løs}]\!](n) \in \Theta(n \log(n))$$

Hvis man modsætningsvis kommer til at dele et problem i (f.eks. 2) problemer, hvoraf det ene er meget lille og det andet er meget stort, bliver resultatet en ligning af formen

$$\begin{aligned} T[\![\textbf{proc}\ \text{løs}]\!](n) &\approx n + \\ & \quad T[\![\textbf{proc}\ \text{løs}]\!](1) + \\ & \quad T[\![\textbf{proc}\ \text{løs}]\!](n-1) + \\ & \quad n \end{aligned}$$

hvilket som bekendt giver

$$T[\![\textbf{proc}\ \text{løs}]\!](n) \in \Theta(n^2)$$

Sammenfattende kan det altså siges, at anvendelser af del-og-kombiner, hvor

- simple problemer løses i tid $O(1)$
- deling og kombination har tidskompleksitet $O(n)$
- delproblemerne er disjunkte og nogenlunde lige store

fører til effektive løsninger med tidskompleksitet $\Theta(n \log(n))$.

Der findes faktisk flere eksempler på problemer, hvor del-og-kombiner fører til *optimale* algoritmer, dvs. algoritmer der beviseligt er lige så gode som alle andre algoritmer, der løser det pågældende problem.

Vi kan betragte Binær søgning som en anvendelse af del-og-kombiner hvor opdelingen består i at dele søgedomænet i to lige store dele, og hvor man i kombinationen ignorerer den halvdel, målelementet ikke kan befinde sig i. Betragtet på denne måde fås følgende ligning

$$T[\![\text{Binær søgning}]\!](n) \approx 1 + T[\![\text{Binær søgning}]\!](\frac{n}{2})$$

som har løsningen

$$T[\![\text{Binær søgning}]\!](n) \in \Theta(\log(n))$$

Det er klart, at Binær søgning udelukkende baserer sig på *sammenligninger*, og man kan vise, at den er optimal blandt sammenligningssøgealgoritmer.

Situationen er den samme m.h.t. flettesortering. Det er ikke svært at se, at argumentet for at enhver sammenligningssøgning må udføre $\Omega(\log(n))$ sammenligninger, kan udvides til at vise, at enhver sammenligningssorterings-algoritme må udføre $\Omega(n \log(n))$ sammenligninger. Da flettesortering kun baserer sig på sammenligninger, er den derfor optimal inden for denne klasse.

Det er ofte sådan i del-og-kombiner algoritmer, at delingen og kombinationen er "omvendt proportionale" i den forstand, at hvis der "sker noget" i den ene, er den anden "triviel". Dette ses tydeligt i de to sorteringseksempler. I quicksort er situationen således

deling: Opdel

kombination: **skip**

og i flettesortering

deling: ≪del listen i to halvdele≫
kombination: Fletning

Sagt på en anden måde, laver Quicksort arbejdet på vejen "ned" i rekursionen, medens flettesortering gør det på vejen "tilbage". Der er mange eksempler på del-og-kombiner algoritmer, der udviser en tilsvarende asymmetri imellem deling og kombination.

Som afslutning på dette afsnit opsamler vi de forskellige løsninger til del-og-kombiner rekursionsligninger i følgende nyttige sætning.

Sætning Rekursionsligningen

$$\begin{aligned} T(1) &\approx 1 \\ T(n) &\approx k * T(\frac{n}{d}) + n^p \text{ for } n > 1 \end{aligned}$$

hvor k, d og p er positive tal, har løsning

$$T(n) \in \begin{cases} \Theta(n^p) & \text{for} \quad k < d^p \\ \Theta(n^p \log(n)) & \text{for} \quad k = d^p \\ \Theta(n^{\log_d(k)}) & \text{for} \quad k > d^p \end{cases}$$

Bevis
Vi anvender den sædvanlige udfoldning og får

$$\begin{aligned} T(n) &\approx n^p + k * ((\frac{n}{d})^p + k * ((\frac{n}{d^2})^p + \ldots + k * 1)) \ldots) \\ &= n^p * (1 + \frac{k}{d^p} + (\frac{k}{d^p})^2 + \ldots + (\frac{k}{d^p})^{\log_d(n)}) \end{aligned}$$

Vi betragter nu de tre tilfælde.

$k < d^p$: Da $(\frac{k}{d^p}) < 1$ er den uendelige kvotientrække $\sum_{i=0}^{\infty}(\frac{k}{d^p})^i$ konvergent. Men så er den endelige delrække det også, dvs. $T(n) \approx n^p$.

$k = d^p$: Summen indeholder $\log_d(n)$ led, dvs. $T(n) \approx n^p * \log(n)$.

$k > d^p$: Vi bruger sumformelen for en kvotientrække (jfr. side 20) og får

$$\begin{aligned} T(n) &\approx n^p * \frac{1 - (\frac{k}{d^p})^{\log_d(n)+1}}{1 - (\frac{k}{d^p})} \\ &\approx n^p * (\frac{k}{d^p})^{\log_d(n)} \\ &= n^{\log_d(k)} \end{aligned}$$

□

Denne sætning giver nu direkte udførelsestiden for del-og -kombiner løsninger, hvor et (stort) problem deles op i k (små) problemer, som hvert er d gange så småt, og hvor opdelings- og kombinationstiderne er et polynomium i størrelsen af det store problem.

2 Dynamisk Programmering

Der er mange interessante problemer, der ikke umiddelbart lader sig løse v.hj.a. del-og-kombiner strategien på en sådan måde, at antallet af delproblemer er en passende (lille) konstant. Hvis man i sådanne tilfælde fristes til blot at generere så mange delproblemer som nødvendigt, ender man let med helt uacceptabelt ineffektive algoritmer. Dette illustreres af følgende vigtige *optimeringsproblem*, det såkaldte *Knapsackproblem*.

2.1 Knapsack

I dette problem er der givet et antal objekter $o_1, \ldots, o_n$, som hver har en størrelse $s_1, \ldots, s_n$ og en værdi $v_1, \ldots, v_n$. Problemet går ud på at pakke en rygsæk (engelsk: knapsack) af kapacitet C på en sådan måde, at indholdet har den størst mulige værdi (man kan vælge vilkårligt mange objekter af hver slags). En mere matematisk formulering af problemet er, at man skal maksimere summen

$$\sum_{i=1}^{n} x_i v_i$$

under randbetingelsen

$$\sum_{i=1}^{n} x_i s_i \leq C$$

hvor $s_1, \ldots, s_n$ er positive heltal og C, $v_1, \ldots, v_n$, $x_1, \ldots, x_n$ er ikke-negative heltal. Lad os betegne denne maksimale værdi med $MV(C)$.

Følgende rekursive algoritme løser problemet

Algoritme: Knapsack
Stimulans : $s_1, \ldots, s_n \geq 1; C, v_1, \ldots, v_n \geq 0$
Respons : $V_{opt} = MV(C)$
Metode : **proc** knap(c: Int) $\rightarrow$ (Int)
(+ **var** j, mv: Int
$mv := 0$
for $j := 1$ **to** n **do**
if $s_j \leq c \rightarrow mv := max\{mv, v_j +$ knap$(c - s_j)\}$ **fi**
return mv
+)
end knap

$V_{opt} :=$ knap(C)

Proceduren virker som følger. Hvis kapaciteten c i et kald er mindre end størrelsen af det mindste objekt, $\min\{s_j\}$, så er $MV(c)$ lig med 0. Ellers er $MV(c)$ lig med maximum af $(v_j + MV(c - s_j)$ over de s_j'er for hvilke $s_j \leq c$, fordi vi ved at medtage et eksemplar af objektet o_j forøger værdien med v_j mod til gengæld at bruge plads s_j. Det følger heraf, at algoritmen er korrekt, hvis den ellers standser, men det gør den, fordi argumenterne til rekursive kald aftager (alle s_j'erne er positive).

Til gengæld er algoritmen næppe særlig effektiv, fordi det må forventes, at der udføres en hel række af kald knap(c) for samme værdi af c, dvs. at én gang udregnede værdier beregnes igen og igen. Dette bekræftes af følgende analyse (for nemheds skyld ser vi på det udartede tilfælde, hvor alle s_j'erne er lig med 1). I dette tilfælde fås

$$\begin{aligned} T[\![\textbf{proc knap}]\!](0) &\approx O(n) \\ T[\![\ \text{knap}]\!](c) &\approx T[\![\textbf{proc knap}]\!](c) \\ &\approx n * T[\![\ \text{knap}(c-1)]\!] \\ &\approx n * T[\![\textbf{proc knap}]\!](c-1) \end{aligned}$$

som har løsningen

$$T[\![\textbf{proc knap}]\!](c) \in \Theta(n^{c+1})$$

Der er altså tale om eksponentiel udførelsestid. Det kan indses (men regnerierne er lidt mere kompliceret), at dette også gælder, når s_j'erne er mere "normale".

Algoritmens udførelsestid kan imidlertid effektiviseres dramatisk ved at observere, at der ikke forekommer mere end $C+1$ *forskellige* kald af proceduren knap, og at man derfor med fordel kan gå over til *genbrug*, dvs. gemme én gang udregnede værdier i en passende datastruktur, og så blot returnere disse næste gang, der er brug for dem. For Knapsack skal der bruges en vektor H(0..C), hvis elementer initialiseres til " at være ukendte", og hvor H.(c) sættes til $MV(c)$ første gang denne udregnes.

Disse overvejelser resulterer i følgende alternative algoritme – typen Huk (for hukommelse) er givet ved

Type Huk = **List**(Celle)
Type Celle = **Sum**(kendt: Int, ukendt: Unit)

Algoritme: Tabel knapsack
Stimulans : $s_1, \ldots, s_n \geq 1; C, v_1, \ldots, v_n \geq 0$
Respons : $V_{opt} = MV(C)$
Metode : **proc** Tknap[H: Huk](c: Int) → (Int)
if is (H.(c),kendt) → **skip**
| **is** (H.(c),ukendt) → (+ **Var** j, mv: Int
$mv := 0$
for $j := 1$ **to** n **do**
if $s_j \leq c \rightarrow mv :=$
$max\{mv, v_j +$ Tknap[H](c-s$_j$)$\}$
fi
H.(c):= Celle(kendt: mv)
+)
fi
return H.(c).kendt
end Tknap

$H :=$ Huk(Celle(ukendt: #)|$C+1$)
$V_{opt} :=$ Tknap[H](C)

Det skulle være klart, at Tabel knapsack er ækvivalent med Knapsack, og at den dermed er korrekt. Med hensyn til analysen af dens effektivitet kan vi bruge bogføringsteknikken fra s. 31 i [Programmeringsteori I] på følgende måde:

Det er klart, at hver gang der udføres et kald af formen Tknap[H](c), hvor H.(c) er kendt, koster kaldet kun 1 krone, fordi der blot er tale om et tabelopslag. Vi påstår nu, at algoritmen (næsten) kan finansieres i sin helhed, hvis vi fra begyndelsen deponerer $2n$ kroner i hvert element i H, dvs. ialt deponerer $2(C+1)n$ kroner.

Argumentet er et induktionsargument, som består i at vise, at det på et vilkårligt tidspunkt af udførelsen gælder, at hvis alle ukendte elementer i H besidder $2n$ kroner (de kendte elementer har ingen penge), så kan vi finansiere et kald af formen Tknap[H](c), hvor H.(c) er ukendt, (næsten) udelukkende v.hj.a. de midler, der befinder sig i H.

Til brug for beviset skal vi kalde tabellen H *velsitueret*, hvis der gælder følgende

is (H.(c),kendt) $\Rightarrow$ H.(c) = MV(c) og H.(c) har ingen penge
is (H.(c),ukendt)$\Rightarrow$ H.(c) besidder $2n$ kroner

Beviset føres ved induktion efter c med følgende induktionsantagelse

I(c): Hvis H er velsitueret, så er H stadig velsitueret efter kaldet Tknap[H](c), og udgiften til kaldet kan, pånær eventuelt 1 krone, betales af H selv.

Selve beviset går som følger

Basis

$c < min\{s_j\}$: Hvis H.(c) er kendt, bruger vi den ekstra krone til at betale for tabelopslaget. Hvis H.(c) er ukendt, udføres der ingen rekursive kald. Det fås derfor umiddelbart, at $T[\![\textbf{proc}\ \text{Tknap}]\!](c) \in O(n)$, og det kan rigeligt finansieres med de $2n$ kroner, der er deponeret i H.(c).

Induktionsskridt

Vi antager, at $I(c')$ er opfyldt for alle $c' < c$. Beviset for $I(c)$ går som følger.

Hvis H.(c) er kendt, bruger vi igen den ekstra krone til at betale for tabelopslaget. Ellers starter vi med at hæve de $2n$ kroner i H.(c). De n kroner dækker udgiften til udførelsen af procedurekroppen , og iflg. induktionsantagelsen kan de øvrige n kroner sammen med H selv finansiere de højst n rekursive kald. Dette *forudsætter*, at H er velsitueret ved indgangen til disse kald. Det er H imidlertid ikke, for vi har jo hævet pengene i H.(c), uden at ændre værdien af H.(c). Det gør imidlertid ikke noget, fordi der for samtlige rekursive kald Tknap[H](c′) gælder, at $c' < c$, og derfor kommer vi ikke under kaldet af Tknap[H](c) til at mangle de $2n$ kroner, der blev hævet.

Efter afslutningen af de rekursive kald sættes H.(c) lig den netop beregnede værdi MV(c), dvs. H bliver igen velsitueret.

Det følger af ovenstående, at

$$T[\![\text{Tabel knapsack}]\!](c) \in O(c * n)$$

hvilket er en meget konkret demonstration af forbedringen i forhold til $O(n^{c+1})$.

Bemærk, at vi her har set et eksempel på, at der på en og samme tid argumenteres for *korrekthed*, *terminering* og *kompleksitet* af en rekursiv procedure.

Det skal også bemærkes, at de her viste egenskaber for proceduren Tknap (naturligvis) også kunne have været vist v.hj.a. bevisprincippet for rekursive procedurer (jfr. [Programmeringsteori II]). Mere præcist kunne vi have vist, at følgende specifikation af Tknap er korrekt

Tknap[H](c) **sat**
(velsitueret(H) →
(velsitueret(H′)) ∧ **is** (H′.(c),kendt) ∧
(Tknap′ = MV(c)) ∧
(T[**proc** Tknap](c) ∈ O(Φ(H)−Φ(H′)+1)

hvor Φ er en potentialfunktion, der angiver "H's rigdom", dvs.

$$\Phi(\mathrm{H}) = \alpha n * |\{c \mid \textbf{is } (\mathrm{H}.(\mathrm{c}),\mathrm{ukendt})\}|$$

hvor α er en passende konstant (jfr. [Programmeringsteori II]).

2.2 Skabelonen

Vi kan også beskrive dynamisk programmeringsteknikken lidt mere abstrakt i form af en skabelon. Vi bruger følgende modifikation af del-og-kombiner skabelonen, hvor vi udover begreberne fra dennes univers skal bruge en hukommelse

$$H : P \to R$$

som er en funktion, der til hvert problem i problemklassen P knytter dets løsning fra L (eller er udefineret). Som R kan vi bruge typen

Sum (kendt: L, ukendt: Unit)

Skabelonen ser ud som følger

Skabelon: Dynamisk Programmering
Univers : Universet fra del-og-kombiner skabelonen udvidet med funktionen $H : P \to R$
Stimulans: $p : P$, problem der ønskes løst
Respons : $l : L$, løsning af p
Metode : **proc** Tløs[H: Huk, $p : P$, $l : L$]
if is ($H(p)$,kendt) $\to$ $l := H(p)$.kendt
& true $\to$ **if** $\ll p$ er i $S \gg \to$
$\ll$find løsning l direkte$\gg$
& true $\to$
(+ **Var** $p_1, \ldots, p_k : P$
Var $l_1, \ldots, l_k : L$
$\ll$del p i $p_1, \ldots, p_k \gg$
Tløs[p_1, l_1]
$\vdots$
Tløs[p_k, l_k]
$\ll$kombiner $l_1, \ldots, l_k$ til $l \gg$
+)
fi
$H(p) := R$(kendt: l)
fi
end Tløs

$\ll$sæt $H(q)$ til R(ukendt: #) for alle q der er mindre end $p \gg$
Tløs[H, p, l]

Vi kan nu finde udførelsestiden T⟦**Proc** Tløs⟧(k, s), hvor k er antallet af små problemer et stort problem opdeles i, og hvor s er lig med det samlede antal problemer, der er mindre end p. Vi skal antage, at der for de ubestemte stumper i skabelonen gælder

T⟦≪p er i S≫⟧ ∈ O(k)
T⟦≪find løsning l direkte≫⟧ ∈ O(k)
T⟦≪del p i p_1,...,p_k ≫⟧ ∈ O(k)
T⟦≪kombiner $l_1 \ldots l_k$ til l≫⟧ ∈ O(k)

Analysen er identisk med den, der blev gennemført i forrige afsnit og giver

følgende udførelsestid

$$\mathrm{T}[\![\textbf{Proc}\ \mathrm{Tløs}]\!](k, s) \in O(k * s)$$

At dette passer med analysen af Tabel knapsack følger af, at vi i det tilfælde havde $k = n$ og $s = c$.

2.3 Binomialkoefficienter

At princippet bag dynamisk programmering – introduktion af en tabel til at indeholde resultatet af én gang beregnede værdier – er velkendt fra andre sammenhænge, illustreres på udmærket vis af algoritmer til beregning af *binomialkoefficienter*.

Binomialkoefficienten $\binom{n}{m}$ angiver som bekendt antallet af måder, hvorpå man kan udtage m elementer fra en mængde med n elementer. Da såvel den tomme mængde som hele mængden kun kan udtages på én måde, er det klart, at der for alle $n \geq 0$ gælder

$$\binom{n}{0} = \binom{n}{n} = 1$$

Det er også let at se, at man for $0 < m < n$ kan udtrykke $\binom{n}{m}$ ved

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$$

Dette skyldes, at antallet af måder hvorpå man kan udtage m elementer (naturligvis) er lig med antallet af forskellige delmængder, der indeholder præcis m elementer (lad os kalde en sådan for en *m-delmængde*). Betragt nu et tilfældigt element e. Antallet af m-delmængder er lig med antallet af m-delmængder hvor *e er med*, plus antallet af m-delmængder, hvor *e ikke er med*. Det første af disse antal er

$$\binom{n-1}{m-1}$$

fordi der skal vælges $m-1$ ud af de resterende $n-1$ (e skal jo være med), og det andet er

$$\binom{n-1}{m}$$

fordi der nu skal vælges m ud af de resterende $n-1$ (e skal jo ikke være med).

Sammenfattende har vi altså for $n \geq 0$

$$\binom{n}{m} = \begin{cases} 1 & \text{for } m = 0 \\ \binom{n-1}{m-1} + \binom{n-1}{m} & \text{for } 0 < m < n \\ 1 & \text{for } m = n \end{cases}$$

Denne definition fører umiddelbart til følgende procedure

```
Proc Bin(n, m: Int) → (Int)
    if (m < 0) ∨ (n < m) → abort("Forkerte argumenter")
    &(m = 0) ∨ (n = m) → return 1
    & true → return Bin(n-1, m)+Bin(n-1, m-1)
    fi
end bin
```

Da værdierne i "bunden" af rekursionen alle er lig med 1, er det klart, at der ved et kald af formen $Bin(n,m)$ udføres $\binom{n}{m} - 1$ additioner, dvs. der gælder

$$T[\![Bin(n,m)]\!] \in \Omega(\binom{n}{m})$$

Nu vides det, at for $m = n/2$ er

$$\binom{n}{m} \geq \frac{2^n}{n+1}$$

hvorfor vi igen står med en håbløs procedure.

Imidlertid kan det igen observeres, at procedure Bin højst kaldes med $(n+1)(m+1)$ forskellige parametre. Vi kan derfor introducere en tabel på præcis samme måde som før, og får da følgende program

```
Process Binomial
    (+ Type Huk = List(Række)
       Type Række = List(Celle)
       Type Celle = Sum(kendt: Int, ukendt: Unit)

       Proc Tbin[H: Huk](n, m: Int) → (Int)
            if (m<0) ∨ (n<m) → abort("Forkerte argumenter")
            & true →
                 if is(H.(n, m), ukendt) →
                      (+ Var r: Int
                         if (m = 0) ∨ (n = m) → r := 1
                         & true → r := Tbin[H](n-1, m)+Tbin[H](n-1, m-1)
                         fi
                         H.(n, m) := Celle(kendt: r)
                      +)
                 fi
                 return H.(n, m).kendt
            fi
       end Tbin

       Var n, m: Int

       ≪Indlæs n og m≫
       do n ≥ 0 →
            (+ Var H: Huk
               H := Huk(Række(Celle(ukendt: #) | m+1) | n+1)
               write("Tbin(n,m)= ", Tbin[H](n, m), eol)
            +)
            ≪Indlæs n og m≫
       od
    +)
end Binomial
where ≪Indlæs n og m≫ is
          write("n= ")
          read[n]
          write("m= ")
          read[m]
```

Dette er en realisering af skabelonen, hvor $k = 2$ og $s = n*m$. Udførelsestiden bliver derfor

$$T[\![Tbin(n,m)]\!] \in \Theta(n * m)$$

Den rolle tabellen spiller her kan illustreres v.hj.a. den såkaldte *Pascals Trekant*, som er betegnelsen for følgende opstilling af binomialkoefficienterne

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5   10  10  5   1
    1   6   15  20  15  6   1
```

hvor hver række indeholder samtlige binomialkoefficienter for en bestemt værdi af n, mere præcist indeholder den n'te række koefficienterne $\binom{n}{m}$ for $0 \leq m \leq n$. Trekanten er konstrueret ud fra den rekursive definition af binomialkoefficienterne, idet hvert element fremkommer som summen af de to elementer, der står lige oven over.

Ved at studere programmet Binomial nøjere, kan man indse, at den del af tabellen H, der udfyldes af proceduren `Tbin`, er lig med følgende skraverede del af Pascals Trekant:

1

$$\binom{n}{0} \qquad \binom{n}{m} \qquad \binom{n}{n}$$

Det er klart, at en alternativ metode til beregning af $\binom{n}{m}$ er følgende iterative algoritme, som beregner hele Pascals Trekant.

Algoritme: Pascals Trekant

Stimulans : $n, m : n, m \geq 0$

Respons : $r = \binom{n}{m}$

Metode : $T, i :=$ List(Vector(1)), 1
do {T indeholder rækkerne $0..i$ i Pascals trekant}
$i \leq n \rightarrow T, j := T{+}{+}\,\text{List}(\text{Vector}(1|i+1)), 1$
do $j \neq i \rightarrow T.(i,j) := T.(i-1, j-1) + T.(i-1, j)$
$j := j + 1$
od
$i := i + 1$
od

$$r := T.(n, m)$$

Det er naturligvis klart, at vi igen har

$$T[\![\text{Pascals Trekant}]\!](n, m) \in \Theta(n * m)$$

Dette eksempel illustrerer en anden generel pointe vedrørende dynamisk programmering. Det er nemt at se, at man i stedet for at tilføje en tabel H til en rekursiv procedure, hvor H udfyldes i overensstemmelse med denne procedures opførsel, ligeså godt kunne skrive et *iterativt* program, der udfylder H systematisk fra bunden af i lighed med Pascals Trekant. Denne alternative formulering af dynamisk programmering er den, der oftest mødes i litteraturen. Vi skal imidlertid tænke på dynamisk programmering som "at smide en tabel efter en rekursiv procedure", fordi dette normalt er mere intuitivt og lettere at forstå.

3 Kombinatorisk søgning

Som det fremgik af forrige kapitel, kan man bruge dynamisk programmering til at løse et problem p, såfremt det *totale antal problemer der er mindre end p* ikke er for stort. Hermed menes, at det ikke må være større, end at man kan introducere en hukommelsesfunktion H, hvis domæne har denne størrelse. Dette betyder i praksis, at antallet af problemer der er mindre end p (som er en øvre grænse for antallet af forskellige kald i den rekursive løsning) skal være et polynimium af lav grad i p's størrelse.

Der findes imidlertid mange problemer, hvor dette ikke er tilfældet, og hvor man følgelig ikke kan klare sig med den ligefremme systematik, der præger dynamisk programmering (eller del-og-kombiner). I sådanne situationer er man henvist til at vælge de delproblemer, man løser – og rækkefølgen hvori man løser dem – med større omtanke. Kombinatorisk søgning dækker over en sådan problemløsningsteknik, hvor man udvælger delproblemer v.hj.a. en prioritetsmekanisme, som styres af en prioritetskø. Vi illustrerer teknikken v.hj.a. to eksempler, det såkaldte *0-1 Knapsack Problem* (som er en variant af Knapsack problemet fra sidste kapitel) og det

såkaldte *Dronninge Problem*, som er en klassiker inden for kombinatoriske søgeproblemer.

3.1 0-1 Knapsack

Vi betragter igen Knapsack problemet fra kapitel 2, men denne gang med den yderligere restriktion, at der højst må medtages ét objekt af hver slags. Sagt med andre ord, skal vi finde maksimum for

$$\sum_{i=1}^{n} x_i v_i$$

under randbetingelserne

$$x_i \in \{0, 1\}$$
$$\textstyle\sum_{i=1}^{n} x_i s_i \leq C$$

En oplagt løsning er følgende modifikation af algoritme Knapsack s. 30, hvor parameteren M er en delmængde af $\{1, 2, \ldots, n\}$.

Algoritme: 0-1 Knapsack
Stimulans : $s_1, \ldots, s_n \geq 1; C, v_1, \ldots, v_n \geq 0$
Respons : $V_{opt} = MV(C)$
Metode : **proc** Knap01(M: Mængde, c: Int) → (Int)
(+ **var** j, mv: Int
$mv := 0$
for $j : j \notin M$ **do**
if $s_j \leq c \rightarrow$
$mv := max\{mv, v_j +$ Knap01$(M \cup \{j\}, c - s_j)\}$ **fi**
return mv
+)
end Knap01

$V_{opt} :=$ Knap01$(\emptyset, C)$

Hvis vi med $\bar{m}$ betegner antallet af objekter, der *ikke* er med i M, dvs. $\bar{m} = n - |M|$ får vi

$$T[\![\textbf{proc Knap01}]\!](\bar{m}, c) \approx \bar{m} * T[\![\textbf{proc Knap01}]\!](\bar{m} - 1, c - 1)$$

hvoraf (idet vi gør den rimelige antagelse, at $n \leq C$)

$$T[\![\textbf{proc Knap01}]\!](n, C) \in O(n!)$$

hvilket igen er en uantagelig udførelsestid.

Hvis vi nu prøver at bruge skabelonen for dynamisk programmering, kan vi se, at problemklassen P består af *par* af formen (M, c), hvor M angiver de objekter, vi har udvalgt, og c er rest-kapaciteten. Nu udgøres mængden af problemer, der er mindre end (M, c), af de par (M', c') for hvilke der gælder

$$M' \supseteq M \text{ og } c' < c$$

Men da startproblemet er parret $(\emptyset, C)$ følger det, at det totale antal delproblemer er af størrelsesordenen $C * 2^n$, hvilket selv for moderate værdier af n (og C) umuliggør anvendelse af dynamisk programmering. Vi er derfor henvist til at "prøve os frem med omtanke", og til den ende er det nyttigt med følgende reformulering af 0-1 Knapsack problemet.

Betragt transitionssystemet

Transitionssystem: 0-1 Knapsack
Konfigurationer : $\{[M, c] \mid M \subseteq \{1, 2, \ldots, n\}, 0 \leq c \leq C\}$
Transitioner : $[M, c] \rhd [M \cup \{i\}, c - s_i]$ **hvis** $(i \notin M) \ \wedge \ (s_i \leq c)$

Hvis vi definerer værdien af mængden M som summen af værdierne af de objekter den "indeholder", dvs.

$$V(M) = \sum_{i \in M} v_i$$

er det klart, at 0-1 Knapsack problemet går ud på at finde den proces for transitionssystemet

$$[\emptyset, C] \rhd \ldots \rhd [M, c]$$

hvor værdien af M i slutkonfigurationen er maksimal.

Teknikken i kombinatorisk søgning består nu i at lede efter en sådan optimal proces for transitionssystemet ved på ethvert tidspunkt at udvælge den konfiguration, der er mest "lovende", dvs. som ser ud til at være den næste i den optimale proces. Det er klart, at vi kan repræsentere mængden af følger, der starter i startkonfigurationen $[\emptyset, C]$ v.hj.a. et *konfigurationstræ* på følgende måde

Her er roden startkonfigurationen $[\emptyset, C]$ og for enhver knude k gælder, at enten er k et blad, eller også har k en datter k' for hver konfiguration k', for hvilken $k \rhd k'$.

Det følger, at ikke alene er hver vej fra roden til et blad en følge for transitionssystemet, men der er også en af disse der kan forlænges til den optimale proces vi leder efter. Ideen er nu, at vi på ethvert tidspunkt holder styr på bladene i konfigurationstræet og i hvert skridt vælger at *ekspandere* det blad, der i en eller anden forstand er mest lovende.

Vi har altså brug for en datastruktur, hvis elementer udgøres af træets blade og som på et vilkårligt tidspunkt kan levere det "bedste" blad. En datastruktur med disse egenskaber er en prioritetskø, hvor prioriteten af et element (et blad) er udtryk for elementets (bladets) "godhed".

Vi kan nu skrive følgende abstrakte skitse til en løsning. Q er en prioritetskø, hvis mere detaljerede indretning vi skal se på i det følgende.

Algoritme: Skitse til kombinatorisk 0-1 Knapsack

```
Stimulans : s_1, ..., s_n ≥ 1; C, v_1, ..., v_n ≥ 0
Respons   : V_opt = MV(C)
Metode    : init[Q]
            insert[Q]([∅, C])
            V_opt := 0
            do ¬ empty[Q] →  delbest[Q, k]
                             V_opt := max{V_opt, Værdi(k)}
                             for  k' : k ▷ k' do
                                  insert[Q](k')
                             od
            od
```

Det skulle være nemt at indse, at denne algoritme er korrekt (den fungerer faktisk ligesom den rekursive løsning). Den lider imidlertid af den skavank, at én og samme konfiguration kan blive indsat i prioritetskøen flere gange. Dette skyldes, at transitionssystemet har den egenskab, at de fleste konfigurationer kan nås på mange måder fra startkonfigurationen. Problemet kan klares på flere måder. Én er at indføre en mekanisme, der husker hvilke konfigurationer der allerede er set, men så risikerer man igen at pladskravet bliver for stort.

En bedre metode er at generere konfigurationerne mere systematisk. Det

kan gøres ved at finde et andet transitionssystem, hvor en konfiguration kun optræder én gang i konfigurationstræet.

Følgende transitionssystem har denne egenskab.

Transitionssystem: 0-1 Knapsack

$$\begin{array}{ll} \text{Konfigurationer} : & \{[M,i,c] \mid M \subseteq \{1,2,\ldots,i\}, 0 \leq i \leq n, 0 \leq c \leq C\} \\ \text{Transitioner} : & [M,i,c] \rhd [M,i+1,c] \textbf{ hvis } i < n \\ & [M,i,c] \rhd [M \cup \{i+1\}, i+1, c - s_{i+1}] \\ & \qquad \textbf{hvis } (i < n) \ \wedge \ (s_{i+1} \leq c) \end{array}$$

Konfigurationerne $[M,i,c]$ indeholder nu parameteren i, som angiver nummeret på det objekt, der sidst er betragtet – og de to transitioner angiver muligheden for hhv. at udelukke og medtage det $(i+1)$'te objekt i M.

Vi er nu interesseret i optimale processer af formen

$$[\emptyset, 0, C] \rhd \ldots \rhd [M, n, c]$$

og de kan findes på samme måde som i skitsen ovenfor. Denne skitse er imidlertid for "liberal" med hensyn til hvilke konfigurationer den indsætter i prioritetskøen. Man bør kun indsætte konfigurationer, der har mulighed for at føre til den søgte optimale proces, dvs. blindgyder skal elimineres så tidligt som muligt. Afskæring af blindgyder sker v.hj.a. en såkaldt *afskæringsmekanisme*, og det er den anden karakteristiske egenskab ved kombinatorisk søgning. Principielt udmønter den sig i at sætningen

for $k' : k \rhd k'$ **do**
  insert$[Q](k')$
od

erstattes med

for $k' : k \rhd k'$ **do**
  if $\ll k'$ er lovende$\gg \rightarrow$
    insert$[Q](k')$
  fi
od

hvor afskæringsmekanismen, her repræsenteret ved

$$\ll k' \text{ er lovende} \gg$$

udtrykker, at det kun er de "interessante" efterfølgere vi genererer. Det er konkretiseringen af $\ll k'$ er lovende $\gg$ som, sammen med prioritetsmekanismen, bestemmer effektiviteten af en kombinatorisk søgealgoritme.

Vi skal nu fastlægge disse to parametre for vores konkrete algoritme (da vi taler om et *maksimerings*problem, vil vores prioritetskø levere det element der har den *største* prioritet).

Vi definerer prioriteten af en konfiguration $[M, i, c]$ som den største værdi M potentielt kan udvides til uden hensyntagen til kapacitetsbegrænsninger, dvs. vi definerer

$$prt([M, i, c]) = V(M) + \sum_{i<j\leq n} v_j$$

Vi kalder endvidere en konfiguration lovende hvis dens potentiel (som altså i dette tilfælde er lig med dens prioritet) er større end det hidtil fundne maksimum. Dette giver anledning til følgende algoritme, hvor vi bruger en prioritetskø af "type" PolyPri(Int), dvs. en prioritetskø der indeholder proceduren

proc DeleteBest$[Q : \text{Pkø}, x : E, p : \text{Int}]$

som fjerner og returnerer det "bedste" element x i Q og som samtidig angiver dettes prioritet p.

Algoritme: Kombinatorisk 0-1 Knapsack

```
Stimulans : s_1, ..., s_n ≥ 1; C, v_1, ..., v_n ≥ 0
Respons   : V_opt = MV(C)
Metode    : Init[Q, false]
            Insert[Q]([∅, 0, C], Σ_{i=1}^{n} v_i)
            V_opt := 0
            do ¬ empty[Q] →  DeleteBest[Q, k, p]
                             if k.i = n → V_opt := max{V_opt, p}
                             & true →  for k' : k ▷ k' do
                                           if prt(k') > V_opt →
                                              Insert[Q](k', prt(k'))
                                           fi
                                       od
                             fi
            od
```

Med hensyn til algoritmens effektivitet, kan man godt risikere at komme til at generere en konfiguration for hver eneste delmængde af $\{1, 2, \ldots, n\}$, dvs. udførelsestiden bliver i værste fald eksponentiel i n. Der er imidlertid gode chancer for at den såkaldte *heuristik*, som afskæringsmekanismen og prioritetsmekanismen repræsenterer, kan begrænse omkostningerne i praktiske tilfælde.

Følgende program, der nøje følger algoritmen, bruger en instans af den polymorfe prioritetskø PolyPri side 15 i [Datastrukturer]. Som det fremgår af initialiseringen $PQ'\text{Init}[Q](n^3, \mathit{false})$ er det *bedste* element det *største* element. Bemærk i øvrigt, at da hver konfiguration i transitionssystemet højst har to efterfølgere, kan **for**-sætningen

```
for k' : k ▷ k' do
    ≪Håndter k' ≫
od
```

realiseres som

```
≪Håndter k'_1 ≫
≪Håndter k'_2 ≫
```

hvor k'_1 og k'_2 er de to efterfølgere for k.

```
Process Knapsack
    (+ @"PolyPri.tri"

        Type Sel = List(Bool)
        Type Config = Prod(M: Sel, i, c: Int)

        Box PQ
            PolyPri(Config)
        end PQ

        Var Q: PQ'Queue
        Var C, Vopt, n, Vsum: Int
        Var Kopt: Config
        Var S, V: Vector

        ≪Indlæs n, C, S og V≫
        ≪Beregn Vsum≫
        PQ'Init[Q](n*n*n, false)
        PQ'Insert[Q](Config(Sel(false | n), 0, C), Vsum)
        Vopt := 0
        do ¬ PQ'Empty[Q] →
            (+ Var k: Config
               Var p: Int
               PQ'DeleteBest[Q, k, p]
               if k.i = n →
                    if p>Vopt → Vopt, Kopt := p, k fi
               & true →
                    (+ Var ki: Int
                       Var kny: Config
                       ki := k.i
                       if (S.(ki) ≤ k.c) ∧ (p>Vopt) →
                            kny := k
                            kny.M.(ki), kny.i, kny.c := true, ki+1, k.c-S.(ki)
                            PQ'Insert[Q](kny, p)
                       fi
                       if p-V.(ki)>Vopt →
                            kny := k
                            kny.i := ki+1
                            PQ'Insert[Q](kny, p-V.(ki))
                       fi
                    +)
               fi
            +)
        od
        write("Løsning:  ", Kopt, eol)
        write("Værdi:  ", Vopt, eol)
    +)
end Knapsack
where ≪Indlæs n, C, S og V≫ is
            write("Indlæs antal objekter:  ")
            read[n]
            write("Indlæs kapacitet:  ")
            read[C]
            write("Indlæs størrelser:  ")
            read[S]
            write("Indlæs værdier:  ")
            read[V]
where ≪Beregn Vsum≫ is
            (+ Var i: Int
```

```
    i, Vsum := 0, 0
    do i ≠ n → i, Vsum := i+1, Vsum+V.(i) od
+)
```

3.2 Skabelonen

Vi kan også angive en skabelon for kombinatorisk søgning.

Kravene til afskærings- og sammenligningsmekanismerne er angivet indirekte ved, at de skal sørge for gyldigheden af en invariant, der udtrykker at

- alle konfigurationer i Q kan nås fra startkonfigurationen k_0
- enhver konfiguration, der kan nås fra k_0, og som er bedre end det aktuelle bud på den optimale konfiguration, kan også nås fra en konfiguration i Q.

Dette er et præcist udtryk for, at prioritetskøen indeholder alle de "interessante" konfigurationer.

Skabelon: Kombinatorisk Søgning

Univers : $S = (K, T)$ er et gentagelsesfrit transitionssystem, hvori alle processer er endelige. $prt : K \to \mathbb{R}$ og $val : K \to \mathbb{R}$ er konfigurationernes *prioritet* og *værdi*. Q er en prioritetskø med prt som prioritetsmekanisme. $\ll k$ er lovende$\gg$ og $\ll k_1$ er bedre end $k_2 \gg$ er en afskærings- og sammenligningsmekanisme, der gør algoritmen gyldig. Invarianten er

$$I(Q, k_0): (\forall \bar{k} \in Q : k_0 \overset{*}{\triangleright} \bar{k}) \wedge$$
$$\forall k \in K : (k_0 \overset{*}{\triangleright} k \wedge \ll k \text{ er bedre end } k_{opt} \gg) \Rightarrow$$
$$\exists \bar{k} \in Q : \bar{k} \overset{*}{\triangleright} k$$

Stimulans: k_0: startkonfiguration

Respons : $k_{opt}, V_{opt} : (V_{opt} = val(k_{opt})) \wedge$
$$\neg(\exists k \in K : k_0 \overset{*}{\triangleright} k \wedge \ll k \text{ er bedre end } k_{opt} \gg)$$

Metode : Init$[Q]$

```
Insert[Q](k_0, prt(k_0))
k_opt, V_opt := k_0, val(k_0)
do {I(Q, k_0)}
  ¬ Empty[Q] →
        DeleteBest[Q, k, p]
        if ≪ k er død≫ ∧ ≪ k er bedre end k_opt ≫→
              k_opt, V_opt := k, val(k)
        & true → for k' : k ▷ k' do
                    if ≪ k' er lovende≫→
                          Insert[Q](k', prt(k'))
                    fi
                 od
        fi
od
```

Når man skal anvende denne skabelon i en konkret situation, skal man naturligvis først fastlægge transitionssystemet. Dernæst vælges de fire "parametre"

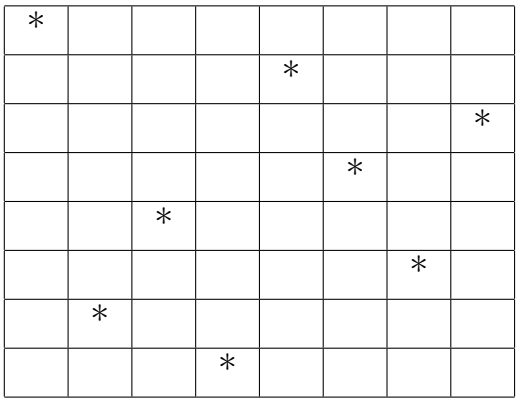
val
prt
$\ll k$ er lovende$\gg$
$\ll k_1$ er bedre end $k_2 \gg$

så hensigtsmæssigt som muligt for det givne problem. Valget af $\ll k_1$ er bedre end $k_2 \gg$ giver normalt sig selv, afhængigt af om der er tale om et maksimerings- eller minimeringsproblem. Naturen af val er som regel givet ved problemformuleringen, så den egentlige udfordring ligger i valget af, og samspillet mellem, prioritets- og afskæringsmekanismerne. (I nogle tilfælde, f.eks. det skal vi se i næste afsnit, er man kun interesseret i at finde en død konfiguration, og så kan man helt ignorere værdifunktionen.)

Valget af prioritets- og afskæringsmekanisme kan der ikke siges ret meget om i almindelighed. Det er et spørgsmål om erfaring og intuition og skal overvejes i hver enkelt konkret situation.

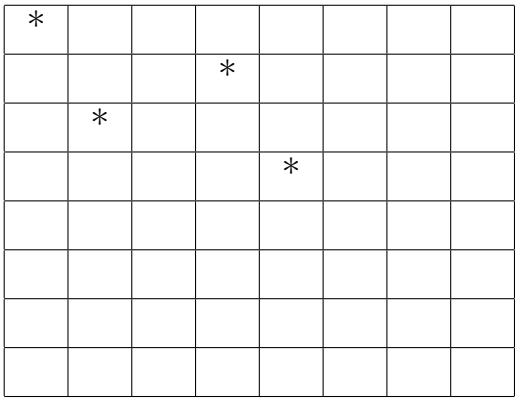
3.3 Dronningeproblemet

I dette afsnit betragtes et klassisk kombinatorisk problem, som kun kan løses v.hj.a. kombinatorisk søgning. Problemet hører til i "legetøjskategorien", men det er velegnet til at illustrere en konkretisering af skabelonerne, hvor værdier ikke spiller nogen rolle. Problemet går ud på at finde en måde at placere 8 dronninger på et skakbræt så ingen af dronningerne er i *konflikt* med (dvs. kan slå) nogen af de andre. To dronninger må altså ikke stå i samme søjle, række eller diagonal. Følgende er et eksempel på en løsning.



I det følgende udvikler vi et program, der kan finde en (alle) løsning(er) til det generaliserede dronningeproblem, der består i at betragte skakbrætter af vilkårlig størrelse. Fremgangsmåden er den samme som i sidste afsnit, dvs. vi definerer først et passende transitionssystem og konstruerer derefter programmet v.hj.a. skabelonen.

Konfigurationerne i transitionssystemet beskriver dronningernes positioner på skakbrættet. Vi skal med det samme gøre transitionssystemet gentagelsesfrit, hvilket opnås ved at placere dronningerne række for række startende fra toppen. En konfiguration beskriver således en situation, hvor et antal af de øverste rækker alle indeholder en dronning, mens de resterende rækker er tomme. Vi kan derfor lade konfigurationerne være følger af tal, der angiver positionerne for dronningerne i de øverste rækker. Følgende situation på et 8×8 bræt



beskrives således af talfølgen

$$(0, 3, 1, 4)$$

hvor det i'te element indeholder positionen for dronningen i den i'te række (bemærk at nummereringen starter med 0).

Givet en konfiguration k kan denne nu udvides til konfigurationen $k \cdot p$, hvor $0 \leq p < n$, forudsat at position p i den første tomme række ikke er i konflikt med nogen af dronningerne i k. Vi definerer prædikatet $\text{fred}(k, p)$ ved

$$\begin{array}{rl} \text{fred}(k,p): & \text{en dronning i position } p \text{ i række } |k| \text{ er ikke i} \\ & \text{konflikt med nogen dronning i } k \end{array}$$

Vi bemærker, at betydningen af dette prædikat jo defineres præcist v.hj.a. reglerne for skak.

Følgende transitionssystem er nu relevant.

$$\begin{array}{l} \underline{\text{Transitionssystem: Dronninger på } n \times n \text{ bræt}} \\ \text{Konfigurationer} : \{[k] \mid k \in N_0^* \wedge |k| \leq n\} \\ \text{Transitioner} \quad\; : [k] \rhd [k \cdot p] \textbf{ hvis } (|k| < n) \wedge \text{fred}(k, p) \end{array}$$

Det skulle være klart, at hvis

$$[\lambda] \rhd \ldots \rhd [k]$$

er en proces, så er $|k| = n$, dvs. k repræsenterer en løsning til problemet.

Konstruktionen af programmet foregår nu v.hj.a. skabelonen. Som prioritetsmekanisme bruger vi konfigurationens *længde*, dvs. vi videreudvikler

først de konfigurationer, der allerede indeholder mange dronninger. Da vi kun er interesseret i at finde døde konfigurationer, har vi ingen brug for skabelonens værdifunktion, dvs. den kan ignoreres.

Følgende program finder en enkelt løsning til $n \times n$ problemet (et program der finder alle løsninger opnås ved blot at fjerne **stop** sætningen). For simpelheds skyld udskrives løsningen(erne) straks de findes.

```
Process Dronninger
    (+ @"PolyPri.tri"

        Type Config = Vector

        Box PQ
            PolyPri(Config)
        end PQ

        Var Q: PQ'Queue
        Var n: Int
        Var løsning: Bool

        write("Indlæs antal dronninger:  ")
        read[n]
        løsning := false
        PQ'Init[Q](n*n*n, false)
        PQ'Insert[Q](Config(), 0)
        do ¬ PQ'Empty[Q] →
              (+ Var k: Config
                 Var p: Int
                 PQ'DeleteBest[Q, k, p]
                 if | k | = n →
                       write("Løsning:  ", k, eol)
                       løsning := true
                       stop
                 & true →
                       (+ Proc fred[k: Config](p: Int) → (Bool)
                             (+ Var i: Int
                                i := 1
                                do i ≤ | k | →
                                      if (k.(| k | -i) = p) ∨ (k.(| k | -i) = p-i) ∨
                                         (k.(| k |-i) = p+i) → return false fi
                                      i := i+1
                                od
                                return true
                             +)
                          end fred

                          Var p: Int

                          p := 0
                          do p ≠ n →
                                if fred[k](p)  → PQ'Insert[Q](k++Config( p), | k |+1) fi
                                p := p+1
                          od
                       +)
                 fi
              +)
        od
        if ¬ løsning → write("Ingen løsninger!", eol) fi
```

```
    +)
end Dronninger
```

3.4 Rekursive løsninger

Den normale måde at præsentere løsninger til kombinatoriske søgeproblemer er i form af rekursive procedurer, der gennemløber konfigurationstræet på mere systematisk vis end den "springen fra blad til blad", der repræsenteres af prioritetskø algoritmerne. Baggrunden herfor er den observation, at hvis man er villig til at opgive den eksplicitte kontrol (v.hj.a. prioriteterne) over den rækkefølge hvori konfigurationstræets blade ekspanderes, så behøver man heller ikke en eksplicit prioritetskø, idet denne da kan repræsenteres implicit i form af forskellige inkarnationer af en passende rekursiv procedure.

Vi illustrerer teknikken ved hjælp af følgende løsning til dronningeproblemet.

```
Process RecDronninger
    (+ Type Config = Vector

        Proc fred[k: Config](p: Int) → (Bool)
            (+ Var j: Int
               j := 1
               do j ≤ |k| →
                     if (k.(|k|-j) = p) ∨
                        (k.(|k|-j) = p-j) ∨ (k.(|k|-j) = p+j) → return false fi
                     j := j+1
               od
               return true
            +)
        end fred

        Proc Dronning[n: Int](k: Config)
            if |k| = n →
                  write("Løsning:  ", k, eol)
                  stop
            & true →
                  (+ Var p: Int
                     p := 0
                     do p ≠ n →
                           if fred[k](p) → Dronning[n](k++Config(p)) fi
                           p := p+1
                     od
                  +)
            fi
        end Dronning

        Var n: Int

        write("Indlæs antal dronninger:  ")
        read[n]
```

```
        Dronning[n](Config())
        write("Ingen løsninger!", eol)
    +)
end RecDronninger
```

Hvis vi betragter det komplette konfigurationstræ med rod $[\lambda]$ for 4×4 problemet

er det nemt at se, at proceduren Dronning gennemsøger træet ved (rekursivt) at undersøge undertræerne fra venstre mod højre (et såkaldt *dybde-først venstre-til-højre* gennemløb). Dette fortsætter indtil den finder det første blad i dybde 4, idet dette repræsenterer den første løsning.

Vi kan nu også se sammenhængen med prioritetskø-løsningen. Hvis vi nummererer træets knuder som følger

og tildeler konfigurationerne disse tal som prioriteter i en udførelse af prioritetskø-algoritmen, så vil denne behandle konfigurationerne i *præcis samme rækkefølge* som den rekursive procedure. Dette er en meget håndgribelig illustration af, hvordan de forskellige inkarnationer af den rekursive procedure definerer en implicit prioritetskø.

Vi slutter dette kapitel med (for fuldstændighedens skyld) også at angive en rekursiv løsning til 0-1 Knapsack problemet.

```
Process Rec01Knapsack
    (+ Type Sel = List(Bool)
       Type Config = Prod(M: Sel, i, c: Int)

       Proc Rec01Knap[S, V: Vector, n: Int, Kopt: Config, Vopt: Int](k: Config, Vpot: Int)
           if k.i = n →
                 if Vpot>Vopt → Vopt, Kopt := Vpot, k fi
           & true →
                 (+ Var ki: Int
                    Var nyk: Config
                    ki := k.i
                    if (S.(ki) ≤ k.c) ∧ (Vpot>Vopt) →
                          nyk := k
                          nyk.M.(ki), nyk.i, nyk.c := true, ki+1, k.c-S.(ki)
                          Rec01Knap[S, V, n, Kopt, Vopt](nyk, Vpot)
                    fi
                    if Vpot-V.(ki)>Vopt →
```

```
                        nyk:=k
                        nyk.i:=ki+1
                        Rec01Knap[S, V, n, Kopt, Vopt](nyk, Vpot-V.(ki))
                    fi
                +)
            fi
        end Rec01Knap

        Var C, Vopt, n, Vsum: Int
        Var Kopt: Config
        Var S, V: Vector

        ≪Indlæs n, C, S og V≫
        ≪Beregn Vsum≫
        Kopt, Vopt:=Config(Sel(false | n), 0, C), 0
        Rec01Knap[S, V, n, Kopt, Vopt](Config(Sel(false | n), 0, C), Vsum)
        write("Løsning:  ", Kopt, eol)
        write("Værdi:  ", Vopt, eol)
    +)
end Rec01Knapsack
where ≪Indlæs n, C, S og V≫ is
            write("Indlæs antal objekter:  ")
            read[n]
            write("Indlæs kapacitet:  ")
            read[C]
            write("Indlæs størrelser:  ")
            read[S]
            write("Indlæs værdier:  ")
            read[V]

where ≪Beregn Vsum≫ is
            (+ Var i: Int
               i, Vsum:=0, 0
               do i≠n → i, Vsum:=i+1, Vsum+V.(i) od
            +)
```

4 Sammenfatning

De tre teknikker, der er præsenteret i denne note, kan opfattes som beregnet til at løse kombinatoriske problemer af varierende vanskelighed.

Den første, del-og-kombiner, er god, når hvert problem kan opdeles i et *lille* antal delproblemer, der ikke lapper for meget over hinanden.

Den anden, dynamisk programmering, kan anvendes selv om opdelingen af nogle problemer fører til et stort antal evt. overlappende delproblemer, blot der i den totale samling delproblemer ikke er for mange, der er forskellige.

Den tredie, kombinatorisk søgning, kan bruges i tilfælde, hvor det samlede antal delproblemer er for stort til at deres løsninger eksplicit kan lagres samtidigt. Der skal imidlertid findes en mekanisme, der styrer den rækkefølge hvori delproblemer genereres og løses.

Den varierende kompleksitet af de tre teknikker ses også af deres udførelsestider. Del-og-kombiner løsninger har ofte udførelsestider i $O(n \log n)$, dynamisk programmering i $O(n^\alpha)$, hvor α er et lille tal, medens kombinatorisk søgning normalt tilhører $O(2^{n^\alpha})$ hvor $0 < \alpha \leq 1$.

Der er ikke unormalt, i bøger om algoritmisk problemløsningsteknik, at omtale en fjerde teknik, den såkaldte *grådige algoritme*. Denne dækker over de situationer, hvor man kan beskrive løsningen til sit problem som en proces for et transitionssystem, hvor transitionssystemet har den egenskab, at det altid er "let" at beregne næste konfiguration i den optimale proces. En grådig algoritme er nu en simpel iteration, der ud fra startkonfigurationen successivt beregner konfigurationerne i den optimale proces og standser ved den første døde konfiguration.

Da mange af de algoritmer, vi tidligere har set, kan karakteriseres som grådige, og da begrebet grådighed i øvrigt ikke er særlig godt karakteriseret, skal vi ikke komme nærmere ind på det i denne sammenhæng.

Referencer

[**Programmeringsteori I**] Erik Meineche Schmidt: *Programmeringsteori*. DAIMI FN-51, Januar 1994.

[**Programmeringsteori II**] Erik Meineche Schmidt: *Programmeringsteori*. DAIMI FN-52, Marts 1994.

[**Trine**] Erik Meineche Schmidt & Michael I. Schwartzbach: *Programmering og programmeringssproget Trine*. DAIMI FN-36, August 1993.

[**Datastrukturer**] Michael I. Schwartzbach: *Datastrukturer*. DAIMI FN-38, Februar 1994.