Matematisk Institut
Aarhus Universitet
Datalogisk Afdeling Marts 1994

PROGRAMMERINGSTEORI II

Erik Meineche Schmidt

Contents

0 Introduktion **1**

1 Amortiseret analyse **2**

1.1 Binær kilometertæller som datastruktur 2

1.2 Potentialfunktion 3

1.3 En monoton Sequence 6

1.4 Datatypen Sequence 10

2 Korrekthed af procedurer **12**

2.1 Ikke-rekursive procedurer 12

2.2 Rekursive procedurer 17

3 Eksempel: Erathostenes Si **20**

3.1 Den abstrakte algoritme 20

3.2 Den konkrete algoritme 22

3.3 Kompleksiteten 24

3.4 TRINE programmet 25

3.5 Kompleksiteten af TRINE programmet 28

4 Referencer **30**

0 Introduktion

Dette hæfte er en direkte fortsættelse af Programmeringsteori I. Vi introducerer først begrebet *amortiseret* analyse og viser dernæst, hvordan man v.hj.a. specifikationer ræsonnerer omkring korrektheden af procedurer. Endelig binder dette hæftes sidste kapitel tingene fra begge hæfter sammen i form af en gennemgang af en (idealiseret) "tilblivelseshistorie" for et TRINE program, der implementerer Erathostenes Si til at finde primtal.

1 Amortiseret analyse

I [Datastrukturer] har vi set en lang række eksempler på analyser af en datastrukturs operationer. Fælles for disse har været, at vi som sædvanligt har interesseret os for *worst-case* udførelsestiderne. I dette kapitel skal vi betragte en anden type udførelsestid, nemlig den såkaldte *amortiserede* udførelsestid. Dette er en formel måde at udtrykke, at en operation med høj worst-case udførelsestid godt kan være effektiv, hvis "worst-case optræder tilstrækkeligt sjældent".

1.1 Binær kilometertæller som datastruktur

Betragt som eksempel følgende datastruktur, der indeholder en binær tæller, som kan nulstilles, aflæses og tælles op med én. Dette er en datastruktur variant af algoritmen Optælling side 30 i [Programmeringsteori I].

```
Box Counter
    Type Bcounter = Vector

    Proc Init[R: Bcounter]
        R := Vector()
    end Init

    Proc ReadOut[R: Bcounter] → (Int)
        (+ Var i, x, r: Int
           i, x, r := 0, 1, 0
           do (i<| R |) →
                r := r+R.(i)*x
                x, i := 2*x, i+1
           od
           return r
        +)
    end ReadOut

    Proc Inc[R: Bcounter]
        (+ Var i: Int
           i := 0
```

```
            do(i<| R |) ∧ (R.(i) =1) → R.(i), i:=0, i+1 od
            if i=| R | → R:=R++Vector(0) fi
            R.(i):=1
        +)
    end Inc
end Counter
```

Det er klart, at de tre procedurer har følgende worst-case udførelsestider

$$\begin{array}{lcl} T[\![\text{Init}[R]]\!] & \in & O(1) \\ T[\![\text{ReadOut}[R]]\!] & \in & O(|R|) \\ T[\![\text{Inc}[R]]\!] & \in & O(|R|) \end{array}$$

Vi kan imidlertid i lighed med algoritmen Optælling argumentere for, at det er sjældent, at vi i proceduren Inc skal overføre menten hele vejen igennem R. Faktisk kunne vi gentage argumentet med bogføringsinvarianten og vise, at hvis vi betaler (noget der er proportionalt med) 2 kr. for hvert kald af Inc, så kan *samtlige* kald af Inc finansieres ad denne vej. Vi siger, at den *amortiserede* udførelsestid for Inc er konstant, hvilket vi skriver som

$$T_A[\![\text{Inc}[R]]\!] \in O(1)$$

1.2 Potentialfunktion

På basis af ovenstående datastruktur kan vi nu på én og samme tid præcisere begrebet amortiseret kompleksitet og generalisere idéen med bogføringsinvarianter. Midlet er begrebet *potentialfunktion*, som er en funktion, der er knyttet til en datastruktur (som Counter ovenfor), og som udtrykker, hvor megen "potentiel energi", der er opsparet fra tidlige operationer til brug for senere operationer. En potentialfunktion, Φ, afbilder (i lighed med en termineringsfunkion) datastrukturen ind i de hele (eller måske reelle) tal. Den anvendes i følgende generelle sammenhæng.

Datastrukturen D har startværdi D_0 og "udsættes" for en følge af n operationer (som kald af procedurerne Init, ReadOut og Inc), som resulterer i, at den gennemløber værdierne

$$D_0, D_1, D_2, \ldots, D_i, \ldots, D_n$$

hvor den i'te operation transformerer D_{i-1} til D_i.

Hvis c_i er den *reelle* omkostning ved den i'te operation, definerer vi den *amortiserede omkostning* ved den i'te operation som

$$\bar{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

dvs. som summen af den reelle omkostning og stigningen i potentialet.

Hvis man nu sørger for at vælge Φ så den opfylder

(*) $\quad \Phi(D_0) \leq \Phi(D_n)$

gælder der

$$\begin{aligned}
\sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} (\bar{c}_i + \Phi(D_{i-1}) - \Phi(D_i)) \\
&= (\sum_{i=1}^{n} \bar{c}_i) + \Phi(D_0) - \Phi(D_n) \\
&\leq \sum_{i=1}^{n} \bar{c}_i
\end{aligned}$$

dvs. at summen af de amortiserede omkostninger er en øvre grænse for summen af de reelle omkostninger. (I praksis opfyldes (*) ofte ved at sørge for, at $\Phi(D_0) \leq \Phi(D_i)$ for alle $i \leq n$.)

Anvendelsen af potentialfunktionen på datastrukturen Counter er som følger. Selve datastrukturen er vektoren R, på hvilken vi definerer potentialfunktionen ved

$$\Phi(R) = \alpha * (\text{antallet af 1'ere i } R)$$

hvor α er en proportionalitetsfaktor, vi bestemmer senere. Idet vi går ud fra, at R_0 er tom, er det klart, at $\Phi(R_0) \leq \Phi(R_i)$ for alle $i \geq 0$.

Den amortiserede omkostning for en operation P er nu

$$T_A[\![P[R]]\!] = T[\![P[R]]\!] + \Phi(R') - \Phi(R)$$

hvor R' i lighed med specifikationen angiver værdien af R efter kaldet af P.

De amortiserede omkostninger ved operationerne Init, ReadOut og Inc er som følger.

Init

$$\begin{aligned} T_A[\![\text{Init}[R]]\!] &= T[\![\text{Init}[R]]\!] - \Phi(R) \\ &\leq T[\![\text{Init}[R]]\!] \\ &\in O(1) \end{aligned}$$

idet $R' = 0$.

ReadOut
ReadOut ændrer ikke på R, dvs.

$$T_A[\![\text{ReadOut}[R]]\!] = T[\![\text{ReadOut}[R]]\!] \in O(|R|))$$

Inc
Antag, at R har udseende

$$R = (\underbrace{1, 1, 1, \ldots, 1}_{x}, 0, ?, \ldots, ?)$$

Så er

$$R' = (\underbrace{0, 0, 0, \ldots, 0}_{x}, 1, ?, \ldots, ?)$$

hvorfor

$$\Phi(R') - \Phi(R) = \alpha - \alpha x = \alpha(1 - x)$$

Men nu følger det af kroppen af proceduren Inc, at udførelsestiden for Inc$[R]$ er proportional med $x + 1$, dvs. at der findes en konstant β, så

$$T[\![\text{Inc}[R]]\!] \leq \beta(x + 1)$$

Hvis vi nu vælger $\alpha = \beta$ får vi

$$\begin{aligned} T_A[\![\text{Inc}[R]]\!] &= T[\![\text{Inc}[R]]\!] + \Phi(R') - \Phi(R) \\ &\leq \alpha(x + 1) + \alpha(1 - x) = 2\alpha \\ &\in O(1) \end{aligned}$$

1.3 En monoton Sequence

Lad os som et mere interessant eksempel betragte box'en Sequence(T) fra afsnit 10.5 i [TRINE]. Vi ser først på følgende "monotone delbox", hvor vi har fjernet Lpop og Rpop.

```
Box SS(T)
    Type Tlist = List(T)
    Type Seq = Prod(low, high: Int, l: Tlist)

    Proc Init[s: Seq]
        s := Seq(0, 0, Tlist())
    end Init

    Proc Len[s: Seq] → (Int)
        return s.high-s.low
    end Len

    Proc Lpush[s: Seq](t: T)
        if |s.l| = 0 → s := Seq(0, 1, Tlist(t))
        & s.low>0 →
              s.low := s.low-1
              s.l.(s.low) :=: t
        & true → s.l, s.low, s.high := Tlist(t||s.l|)++s.l, |s.l|-1, s.high+|s.l|
        fi
    end Lpush

    Proc Ltop[s: Seq] → [T]
        if Len[s] = 0 → abort("Ltop on empty sequence") fi
        return s.l.(s.low)
    end Ltop

    Proc Rpush[s: Seq](t: T)
        if |s.l| = 0 → s := Seq(0, 1, Tlist(t))
        & s.high<|s.l| →
              s.l.(s.high) :=: t
              s.high := s.high+1
        & true → s.l, s.high: tt = s.l++Tlist(t||s.l|), s.high+1
```

```
        fi
    end Rpush

    Proc Rtop[s: Seq] → [T]
        if Len[s] = 0 → abort("Rtop on empty sequence") fi
        return s.l.(s.high-1)
    end Rtop

    Proc Sel[s: Seq](i: Int) → [T]
        if 0 ≤ i<Len[s] → return s.l.(s.low+i)
        & true → abort("Seq index out of range")
        fi
    end Sel

    Proc Print[s: Seq]
        write(s.l(s.low .. s.high) )
    end Print
end SS
```

Det er klart, at der for procedurerne Init, Len, Ltop, Rtop og Sel gælder, at de har konstante udførelsestider. Det er også klart, at Print har lineær udførelsestid.

Med hensyn til Rpush (og Lpush), kan vi udtrykke udførelsestiden på følgende måde. Typen Sequence er implementeret som en liste

hvor det skraverede er "spild". Rpush foretager sig kun noget "dyrt", når high peger på enden af listen, og i så fald er omkostningen proportional med længden af listen. Det er derfor relevant at udtrykke kompleksiteten af Rpush v.hj.a. følgende tre karakteristiske parametre n, m og k.

Hermed bliver tidskompleksiteten af Rpush

$$T[\![\textbf{proc Rpush}]\!](n,m,k) \in \begin{cases} O(1) & \text{hvis } k > 0 \\ O(n+m+1)^{\dagger} & \text{hvis } k = 0 \end{cases}$$

dvs. der findes en konstant α så

$$T[\![\textbf{proc Rpush}]\!](n,m,k) \leq \begin{cases} \alpha & \text{hvis } k > 0 \\ \alpha(n+m+1) & \text{hvis } k = 0 \end{cases}$$

Som potentialfunktion vælger vi nu

(*) $\quad \Phi(n,m,k) = \alpha(3m - (n+k))$

Vi får da følgende, idet vi betegner ændring i potentialet $\Phi(n',m',k') - \Phi(n,m,k)$ med $\Delta\Phi$

$$T_A[\![\textbf{proc Rpush}]\!] = T[\![\textbf{proc Rpush}]\!] + \Delta\Phi$$

Vi ser nu på hvert af de to tilfælde .

$\underline{k > 0}$
Så er $n' = n$, $m' = m+1$ og $k' = k-1$, hvorfor

$$\Delta\Phi = \alpha(3(m+1) - (n+k-1)) - \alpha(3m - (n+k)) = 4\alpha$$

Men så er

$$\begin{aligned} T_A[\![\textbf{proc Rpush}]\!](n,m,k) &\leq 5\alpha \\ &\in O(1) \end{aligned}$$

[†] 1'tallet skyldes, at udsagnet også skal gælde, når $n = m = 0$.

$\underline{k = 0}$
I dette tilfælde fordobles listen, dvs. $n' = n$, $m' = m+1$ og $k' = n+m-1$. Vi har da

$$\begin{aligned} \Delta\Phi &= \alpha(3(m+1) - (2n+m-1)) - \alpha(3m-n) \\ &= \alpha(4-(n+m)) \end{aligned}$$

Så er

$$\begin{aligned} T_A[\![\textbf{proc Rpush}]\!](n,m,k) &= T[\![\textbf{proc Rpush}]\!](n,m,k) + \Delta\Phi \\ &\leq \alpha(n+m+1) + \alpha(4-(n+m)) = 5\alpha \\ &\in O(1) \end{aligned}$$

Hvis vi nu også kan vise, at $\Phi(n,m,k) \geq 0$ har vi altså, at

$$T_A[\![\textbf{proc Rpush}]\!] \in O(1)$$

Det er klart, at der tilsvarende gælder, at

$$T_A[\![\textbf{proc Lpush}]\!] \in O(1)$$

For at vise, at $\Phi(n,m,k) \geq 0$, bemærker vi først, at initialværdien er lig med 0, fordi

$$\Phi(0,0,0) = 0$$

Dernæst observerer vi, at potentialfunktionens værdi *vokser*, når Rpush udføres for $k > 0$. Tilbage er da kun at vise, at Φ's værdi også er ikke-negativ efter en udførelse af Rpush, hvor $k = 0$. Efter en sådan udførelse er værdien af Φ

$$\Phi(n, m+1, n+m-1) = 2(m-n) + 4$$

At den er ikke-negativ følger nu af den simple observation, at det *midterste* element i listen aldrig er spild. Når $k = 0$ betyder dette, at $m \geq n$, hvorfor $2(m-n)+4 > 0$, dvs.

$$\Phi(n, m+1, m+n-1) > 0.$$

1.4 Datatypen Sequence

Hvis vi betragter den "totale" box Sequence(T), dvs. den box vi får ved at tilføje Lpop og Rpop til SS, er det intuitivt klart, at de amortiserede udførelsestider for Lpush og Rpush vedbliver at være konstante. Begrundelsen er, at anvendelser af Lpop og Rpop blot betyder, at der går endnu længere tid mellem de "dyre" operationer Rpush og Lpush. Heraf følger det imidlertid ikke formelt, at Rpush og Lpush også har konstante amortiserede udførelsestider i box'en Sequence(T). Problemet er, at det ikke længere er rigtigt, at potentialfunktionen (*) er positiv, når vi også må anvende Lpop og Rpop.

Da vi kun kan have én potentialfunktion tilknyttet en datastruktur, skal vi derfor finde et alternativ til Φ ovenfor, som på samme tid er ikke-negativ og fører til konstante amortiserede udførelsestider for både Rpush og Lpush samt Rpop og Lpop. Betragt nu følgende inddeling af listen

hvor det skraverede stadig er spild, men hvor det nu er inddelt i "jomfrueligt" spild (m' og m'') og "almindeligt" spild (n og k). Forskellen på de to typer spild er, at jomfrueligt spild aldrig har indeholdt et "rigtigt" element, hvorimod almindeligt spild på et tidspunkt har indeholdt noget, som senere er blevet pop'et. Hvis vi nu som alternativ definerer potentialfunktionen ved

$$\Phi(n, m', m, m'', k) = \alpha(3(m + m' + m'') - (n + k))$$

kan vi stort set gentage argumenterne ovenfor, dvs. vi kan dels vise, at

$$\Phi(n, m', m, m'', k) \geq 0$$

og dels at de amortiserede udførelsestider for såvel Rpush (Lpush) som Rpop (Lpop) er konstante.

Som afslutning på dette kapitel bemærker vi, at vi både her og i forrige afsnit har en oplagt analogi med "opsparingsfilosofien" fra bogføringsinvarianten

(jfr. [Programmeringsteori I]). Når vi udfører en Rpush i en situation, hvor $k > 0$ (og $m'' = 0$), fører dette til en *stigning* i potentialfunktionens værdi på 4α. Dette svarer til at spare 4 kr. op til brug for at betale kommende "dyre" Rpush'er.

Bemærk også, at vi med potentialfunktioner har indført endnu en målemekanisme, der hjælper os med at ræsonnere om algoritmer og datastrukturer. En potentialfunktion er, i lighed med termineringsfunktioner og udsagn, en funktion, hvis domæner er algoritmers eller datastrukturers tilstand, og som bruges i forbindelse med argumentation for korrekthed og/eller effektivitet. Bemærk også, at disse funktioner kun anvendes til at *måle* på tilstande, de påvirker dem ikke, dvs. hverken udsagn eller termineringseller potentialfunktioner har nogen indflydelse på beregningerne.

2 Korrekthed af procedurer

Når man har behov for at argumentere for korrekthed af algoritmer, der indeholder procedurekald, rejser der sig spørgsmålet om, hvordan man kan få procedurekaldenes egenskaber passet ind i korrekthedsbeviserne.

2.1 Ikke-rekursive procedurer

Vi kan igen betragte algoritmen Permutation fra kapitel 3 i [Programmeringsteori I], men denne gang skrevet v.hj.a. procedurer på følgende måde. (S er en mængdetype med elementer fra E).

Algoritme: Permutation
Stimulans : f: **List** (E), $\forall i, j \in 0..|f| : f.(i) \neq f.(j)$
Respons : f: permuteret
Metode :

```
proc insert [s : S](e : E)
    ≪indsæt e i s ≫
end insert

proc delete [s : S](e : E)
    ≪fjern e fra s ≫
end delete

proc select [s : S, e : E]
    if s ≠ ∅ →≪ e := element i s ≫
    & true →abort
    fi
end select

K, i := ∅, 0
do i < |f| →  insert [K](f.(i))
              i := i + 1
od
i := 0
do i < |f| →  select [K, k]
              delete [K](k)
              f.(i), i := k, i + 1
od
```

Procedurernes (eller rettere procedurekaldenes) egenskaber vil igen blive angivet v.hj.a. specifikationer på følgende måde

insert $[s](e)$ **sat** $s' = s \cup \{e\}$
delete $[s](e)$ **sat** $s' = s \backslash \{e\}$
select $[s, e]$ **sat** $s \neq \emptyset \rightarrow (e' \in s) \wedge (s = s')$

og betydningen er præcis den samme som i kapitel 4 i [Programmeringsteori I].

Der kan også blive behov for at specificere en værdiprocedure som følgende alternative udgave af select

proc select$[s : S] \rightarrow (E)$
  if $s \neq \emptyset \rightarrow \ll$returner et e i $s \gg$
  & true $\rightarrow$ **abort**
  fi
end select

I sådanne tilfælde bruger vi selve procedurenavnet til at angive værdien af procedurekaldet, dvs. vi får en specifikation af følgende form

select $[s]$ **sat** $s \neq \emptyset \rightarrow (\text{select}' \in s) \wedge (s = s')$.

Procedure*kaldenes* egenskaber beskrives altså v.hj.a. specifikationer, og disse kan så på sædvanlig vis anvendes i ræsonnementer om de algoritmer eller programmer, hvori kaldene optræder. Spørgsmålet er nu, hvordan man bærer sig ad med at vise, at et procedurekald opfylder en specifikation. Svaret er, at det gør man ved at vise, at procedure*kroppen* opfylder den givne specifikation. Dette forudsætter imidlertid, at specifikationen er udtrykt i termer af procedurens *formelle* parametre, så når den skal bruges i forbindelse med et kald, skal specifikationens formelle parametre erstattes af de *aktuelle* parametre fra kaldet. Vi kan udtrykke dette mere præcist på følgende måde.

Betragt en proceduredefinition af formen

proc $p[r : R](v : V) \rightarrow (E)$
  $\ll$Krop$\gg$
end p

Hvis man kan vise, at kroppen opfylder specifikationen

$$\ll\text{Krop}\gg \textbf{ sat } P(r,v) \to U(r,r',v,p')$$

hvor $P(r,v)$ ($U(r,r',v,p')$) er prædikater med frie variabler r,v (r,r',v,p'), så gælder der, at $p[r](v)$ opfylder specifikationen

$$p[r](v) \textbf{ sat } P(r,v) \to U(r,r',v,p')$$

Herudover gælder der for et vilkårligt andet kald

$$p[a](e)$$

hvor a er en variabel af type R og e er et værdiudtryk af type V, at dette kald opfylder den specifikation, der opnås ved i P og U at erstatte de formelle parametre med de aktuelle, dvs.

$$p[a](e) \text{ sat } P(a,e) \to U(a,a',e,p')$$

Vi kan angive dette mere skematisk v.hj.a. følgende såkaldte bevisprincip

Bevisprincip for ikke-rekursive procedurer

Hvis der for proceduren

proc $p[r : R](v : V) \to (E)$
  $\ll$Krop$\gg$
end p

gælder

$\ll\text{Krop}\gg$ **sat** $P(r,v) \to U(r,r',v,p')$

så gælder der for ethvert kald $p[a](e)$ at

$p[a](e)$ **sat** $P(a,e) \to U(a,a',e,p')$

Princippet gælder også for procedurer med flere parametre *forudsat* at ingen aktuel [ ]-parameter er identisk med eller delvariabel af andre aktuelle [ ]-parametre.

Betragt som eksempel på en anvendelse af bevisprincippet følgende algoritme til såkaldt *geometrisk søgning*, der anvender en kvadratrodsalgoritme. Algoritmen er næsten identisk med Binær Søgning side 14 i [Programmeringsteori I]. Forskellen er kun, at i stedet for at beregne kandidatmængdens midtpunkt som det *aritmetiske* gennemsnit (lav+høj)/2, bruges nu det *geometriske* gennemsnit $\sqrt{\text{lav} * \text{høj}}$. Kvadratroden udregnes v.hj.a. en procedure, der returnerer heltalskvadratroden af sit argument.

Algoritme : Geometrisk søgning
Stimulans : S : **List** (Int), ordnet søgedomæne
m : Int, målelement
Respons : $r : S(\alpha(r)) = S\ (S^{-1}(m))$
Metode : **proc** *sqrt*(n : Int) → (Int)
(+**Var** r : Int
$r := n$
do $n < r^2 \to r := r - 1$ **od**
return r
+)
end *sqrt*

lav,høj,r := 0, $|S|$, Res(tom : #)
do
(lav < høj) ∧ **is** (r, tom) →
midt := *sqrt*(lav ∗ høj)
if $S.(\text{midt}) = m \to r :=$ Res(x : midt)
| $S.(\text{midt}) < m \to$ lav := midt + 1
| $m < S.(\text{midt}) \to$ høj := midt
fi
od

Vi beviser nu, at proceduren sqrt tilfredsstiller specifikation

(*) sqrt(n) **sat** $n \geq 0 \to (\text{sqrt}')^2 \leq n < (\text{sqrt}' + 1)^2$

Iflg. bevisprincippet er det nok at vise, at

$$\left.\begin{array}{l} r := n \\ \textbf{do}\ n < r^2 \to r := r - 1\ \textbf{od} \\ \textbf{return}\ r \end{array}\right\} \textbf{sat}\ n \geq 0 \to (\text{sqrt}')^2 \leq n < (\text{sqrt}' + 1)^2$$

hvilket – da sqrt′ betegner den returnerede værdi og det er r der returneres – er det samme som at vise

$$\left.\begin{array}{l} r := n \\ \textbf{do } n < r^2 \rightarrow r := r - 1 \textbf{ od} \end{array}\right\} \textbf{sat } n \geq 0 \rightarrow (r')^2 \leq n < (r'+1)^2$$

At dette er rigtigt, indses ved at observere, at nedenstående invariant er gyldig og at beregningen standser

$$\begin{array}{l} r := n \\ \textbf{do } \{n < (r+1)^2\} \\ \quad n < r^2 \rightarrow r := r - 1 \\ \textbf{od} \end{array}$$

Altså kan vi konkludere, at (*) er opfyldt.

I et bevis for korrekthed af Geometrisk Søgning skal vi bruge kvadratrodsprocedurens egenskaber til at bevise terminering. Termineringsfunktionen er høj − lav − $|\alpha(r)|$, og for at sikre at den aftager, skal vi vise, at der efter udførelse af tilordningen

$$\text{midt} := \text{ sqrt}(\text{lav} * \text{høj})$$

gælder

$$\text{lav} \leq \text{midt} < \text{høj}$$

Beviset går som følger. Fra (*) og bevisprincippet fås

$$\text{midt}^2 \leq \text{lav} * \text{høj} < (\text{midt}+1)^2$$

og vi ved yderligere at

$$\text{lav} < \text{høj}$$

for ellers var vi ikke kommet ind i løkken.

Vi skal derfor blot vise, at

$$
\begin{array}{ll}
 & (\text{midt}^2 \leq \text{lav} * \text{høj} < (\text{midt}+1)^2) \wedge (\text{lav} < \text{høj}) \\
(**) & \Downarrow \\
 & \text{lav} \leq \text{midt} < \text{høj}
\end{array}
$$

Vi fører beviset indirekte. Antag først at midt $<$ lav. Så er midt $+1 \leq$ lav, hvoraf

$$(\text{midt}+1)^2 \leq \text{lav}^2 \leq \text{lav} * \text{høj}$$

hvilket giver modstriden. Hvis vi omvendt antager, at høj $\leq$ midt fås analogt

$$\text{midt}^2 \geq \text{høj}^2 > \text{lav} * \text{høj}$$

som igen giver modstrid. Herefter er (**) vist, og vi konkluderer, at Geometrisk Søgning er korrekt.

2.2 Rekursive procedurer

Når man skal bevise, at en rekursiv procedure tilfredsstiller en specifikation, er situationen på en vis måde analog til at analysere dens tidskompleksitet. Man må også her udtrykke en sammenhæng mellem proceduren selv og de rekursive kald. I forbindelse med korrekthedsbeviser har denne sammenhæng dog ikke form af en rekursionsligning med derimod af en *implikation*, hvor præmissen er en antagelse om, at de rekursive kald allerede har den ønskede egenskab.

Mere præcist gælder der følgende bevisprincip for rekursive procedurer[1] Funktionen μ er en sædvanlig termineringsfunktion, som skal bruges til at sikre, at rekursionen ikke bliver ved i det uendelige.

[1]Udvidelsen til at betragte værdiprocedurer er helt analog til sidste afsnit.

Bevisprincip for rekursive procedurer

Hvis der for den rekursive procedure

proc $p[r : R](v : V)$
⋮
$p[s](u)$ ≪Krop≫
⋮
end p

gælder

- $P(r, v) \Rightarrow \mu(r, v) > \mu(s, u) \geq 0$ for alle rekursive kald $p[s](u)$
- $p[s](u)$ **sat** $P(s, u) \rightarrow U(s, s', u)$
  $\Downarrow$
  ≪Krop≫ **sat** $P(r, v) \rightarrow U(r, r', v)$

så gælder der for ethvert kald $p[a](e)$

$p[a](e)$ **sat** $P(a, e) \rightarrow U(a, a', e)$

Princippet gælder igen for procedurer med flere parametre og under samme forudsætning som for de iterative procedurer, dvs. der må ikke være overlap mellem forskellige aktuelle [ ]-parametre.

Lad os som eksempel på en anvendelse af princippet betragte proceduren Reverse_1. Hvis w er en Text-variabel, vil vi gerne vise at

$\text{Reverse}_1[w]$ **sat** $w' = \tilde{w}$

Iflg. princippet ovenfor skal vi så

- finde en termineringsfunktion μ
- bevise at

  $\text{Reverse}_1[L]$ **sat** $L' = \tilde{L}$
  $\Downarrow$
  ≪Krop≫ **sat** $L' = \tilde{L}$

Termineringsfunktionen er simpel, idet vi blot tager

$$\mu(L) = |L|$$

Med hensyn til at vise implikationen er sagen klar i det tilfælde, hvor $|L| \leq 1$. Hvis $|L| \geq 1$ skal vibevise følgende

$$\left.\begin{array}{l} hlp, L := L(0..1), L(1..|L|) \\ \text{Reverse}_1\ [L] \\ L := L{+}{+}\ hlp \end{array}\right\} \textbf{sat}\ L' = \tilde{L}$$

Hvis vi bruger indicering til at betegne værdierne af variablerne før og efter de enkelte sætninger, får vi

$$\begin{array}{llcl} hlp_0 = ? & L_0 & = & L \\ hlp_1 = L_0(0..1) & L_1 & = & L_0(1..|L|) \\ hlp_2 = hlp_1 & L_2 & = & \tilde{L}_1 \text{ (p.gr.a. antagelsen om det lokale kald)} \\ hlp' = hlp_2 & L' & = & L_2{+}{+}hlp_2 \end{array}$$

som ved simpel udregning giver $L' = \tilde{L}$.

Det skal afslutningsvist bemærkes, at der er en nøje sammenhæng mellem det der i analyseafsnittet hedder *karakteristisk parameter* og den her anvendte termineringsfunktion. De har begge til formål at finde et aftagende *mål* for de tilstande, hvori proceduren kaldes rekursivt, og det vil ofte være sådan, at karakteristisk parameter og termineringsfunktion er identiske.

3 Eksempel: Erathostenes Si

I dette kapitel gennemgås et eksempel, som illustrerer de fleste af de begreber, der er introduceret i det foregående og i [Programmeringsteori I].

Eksemplet er den klassiske algoritme til at finde samtlige primtal mindre eller lig med et givet tal $n \geq 2$, som går under navnet Erathostenes Si.

3.1 Den abstrakte algoritme

Fremgangsmåden kan illustreres på følgende måde for $n = 9$.

Vi opererer med to mængder P og K, som begge er delmængder af $\{2, \ldots, n\}$. P består af de hidtil fundne primtal, og K består af tal der er kandidater til at blive indlemmet i P. Når beregningen starter, er situationen

$$P_0 = \emptyset$$
$$K_0 = \{2, 3, 4, 5, 6, 7, 8, 9\}$$

Det er klart, at K's mindste tal er et primtal, og det er også klart, at alle tal i K, der har dette tal som divisor, ikke er primtal. Første skridt består derfor i at flytte 2 over i P og at fjerne alle lige tal fra K, dvs. vi får

$$P_1 = \{2\}$$
$$K_1 = \{3, 5, 7, 9\}$$

K's mindste tal er igen et primtal, hvorfor vi får

$$P_2 = \{2, 3\}$$
$$K_2 = \{5, 7\}$$

Gentagelse af dette skridt giver

$$P_3 = \{2, 3, 5\}$$
$$K_3 = \{7\}$$

og endelig får vi

$P_4 = \{2, 3, 5, 7\}$
$K_4 = \emptyset$

hvorefter vi konkluderer, at 2, 3, 5 og 7 er primtallene blandt de første 9 tal.

Eksemplet antyder, at algoritmen skal bestå af en løkke med en invariant, som siger at P består af primtal, K af tal som ikke har divisorer fra P, samt at P og K tilsammen indeholder alle primtal blandt de n første tal. Dette kan udtrykkes formelt, hvis vi med $PRIM(n)$ betegner primtallene blandt $\{2, \ldots, n\}$, dvs.

$$PRIM(n) = \{p \mid (2 \leq p \leq n) \text{ og } p \text{ er et primtal}\}$$

og ligesom i Euklids Algoritme betegner største fælles divisor af to tal p og q med $sfd(p, q)$. Vi får følgende abstrakte algoritme

Algoritme : Erathostenes Si
Stimulans : $n : n \geq 2$
Respons : $P : P = PRIM(n)$
Metode : $P, K := \emptyset, \{2, 3, \ldots, n\}$
do $\{(P \subseteq PRIM(n) \subseteq P \cup K) \wedge (sfd(P, K) = 1)\}$[2]
$K \neq \emptyset \rightarrow h := min(K)$
$P := P \cup \{h\}$
$K := K - \{k \in K \mid h \text{ går op i } k\}$
od

Vi interesserer os nu lidt nærmere for denne algoritmes egenskaber, såvel de kvalitative som de kvantitative. Vi viser gyldighed, terminering og korrekthed på sædvanlig vis:

Terminering
Følger direkte ved at betragte termineringsfunktionen $\mu(K) = |K|$.

Gyldighed
Det er klart, at invarianten tilfredsstilles af initialiseringen af P og K. At invarianten bevares, reduceres til at bevise at

[2]) $sfd(P, K)$ er et kompakt udtryk for prædikatet $\forall p \in P : \forall k \in K : sfd(p, k) = 1$.

$$(P \subseteq PRIM(n) \subseteq P \cup K) \wedge (sfd(P,K) = 1) \wedge (K \neq \emptyset)$$

$$\Downarrow$$

$$(P' \subseteq PRIM(n) \subseteq P' \cup K') \wedge (sfd(P',K') = 1)$$

hvor $P' = P \cup \{min(K)\}$
$K' = K - \{k \in K \mid min(K) \text{ går op i } k\}$

Denne implikation følger af at

a) $min(K)$ må være et primtal, for ellers ville det have en mindre primtalsfaktor, men denne måtte være med i P (fordi $PRIM(n) \subseteq P \cup K$), hvilket er i modstrid med at $sfd(P,K) = 1$

b) hvis $min(K)$ indsættes i P og det selv og alle dets multipla fjernes fra K, indeholder P' og K' de samme primtal som $P \cup K$, og vi har stadig, at $sfd(P',K') = 1$.

Korrekthed
Når algoritmen terminerer, gælder der

$$(P \subseteq PRIM(n) \subseteq P \cup K) \wedge (K = \emptyset)$$

hvoraf det direkte følger, at $P = PRIM(n)$.

3.2 Den konkrete algoritme

Vi kan ikke finde algoritmens udførelsestid ved at ræsonnere direkte på den foreliggende version. Problemet er, at de operationer der udføres på variablerne P og K *ikke* er primitive i kompleksitetsforstand. Vi må derfor realisere P og K (og deres operationer) på en sådan måde, at manipulationerne kommer tættere på de anerkendte primitiver. Dette kan gøres på følgende måde.

Variablen P skal kunne initialiseres til den tomme mængde, og derefter skal man kunne tilføje et element ad gangen. Begge disse operationer understøttes automatisk, hvis vi realiserer P som en heltals-sekvens på følgende måde

Box ISq
  Sequence(Int)
end ISq

Variablen K er mere interessant. Vi kan se, at den skal kunne initialiseres til $\{2, 3, \ldots, n\}$, man skal kunne finde dens mindste element, og man skal kunne fjerne "hvert h'te" element. Da K er monoton – mængden bliver aldrig større – kan vi realisere den effektivt v.hj.a. en bitvektor (jfr. [Datastrukturer]), hvor vi yderligere vedligeholder en tæller, der indeholder antallet af elementer i K, samt én der peger på det sidst fundne primtal. Følgende tegning (B betegner bitvektoren, a antal elementer og p peger på det sidst fundne primtal) viser repræsentationen af mængden K_1 ovenfor.

B : $\boxed{\underline{\text{f}} \mid \underline{\text{f}} \mid \underline{\text{f}} \mid \underline{\text{t}} \mid \underline{\text{f}} \mid \underline{\text{t}} \mid \underline{\text{f}} \mid \underline{\text{t}} \mid \underline{\text{f}} \mid \underline{\text{t}}}$

a : 4

p : 2

Med disse realiseringer af P og M får den konkrete algoritme følgende udseende.

Algoritme : Erathostenes
Stimulans : $n : n \geq 2$
Respons : $P : P = PRIM(n)$
Metode : ISq' Init$[P]$
$B, a, p :=$ **List**$(\underline{\text{f}} \mid 2) + +$**List**$(\underline{\text{t}} \mid n-1), n-1, 0$
do $a > 0 \rightarrow$
L_1: **do** $\neg B.(p) \rightarrow p := p + 1$ **od**
ISq' Rpush$[P](p)$
$i := p$
L_2: **do** $i \leq n \rightarrow$
if $B.(i) \rightarrow B.(i), a :=$ false$, a - 1$ **fi**
$i := i + p$
od
od

3.3 Kompleksiteten

Analysen af denne algoritme er mere kompliceret end de eksempler, vi hidtil har set, bl.a. fordi det ikke er klart, hvor mange gange den yderste løkke gennemløbes. Dette antal kan vi imidlertid finde i følgende vigtige sætning fra talteorien.

Primtalssætningen
Lad $\pi(n)$ være antallet af primtal i intervallet $2..n+1$.
Der gælder[3]

$$\pi(n) \in \frac{n}{ln(n)} + O(\frac{n}{ln(n)^2})$$

Vi kan nu finde algoritmens udførelsestid som summen af udførelsestiderne for

a) initialiseringen

b) løkken L_1

c) løkken L_2

d) resten af algoritmen

Vi påstår, at disse er som følger (p_i betegner det i'te primtal)

a) $\Theta(n)$

b) $\Theta(n)$

c) $\sum_{i=1}^{\pi(n)} \Theta(\frac{n}{p_i})$

d) $\Theta(\frac{n}{\log(n)})$

[3])$ln(x)$ er den naturlige logaritme.

Med hensyn til a) og b) skulle analysen være oplagt. Med hensyn til c) følger resultatet af, at hver gang der er fundet et nyt primtal p, gennemløbes K i "spring af længde p". Endelig følger d) af primtalssætningen og det faktum, at den *amortiserede* kompleksitet af proceduren Rpush i Box'en Sequence tilhører $O(1)$ (jfr. 1.3).

Det er klart, at det dominerende led blandt a)-d) er c), som vi kan omskrive til

$$\Theta(n \sum_{i=1}^{\pi(n)} \frac{1}{p_i})$$

Med hensyn til værdien af denne sum, skal vi igen appellere til et resultat fra talteorien, som siger

$$\sum_{i=1}^{\pi(n)} \frac{1}{p_i} \approx ln(ln(n))$$

Vores analyse giver altså følgende samlede resultat

$$T[\![\text{Erathostenes Si}]\!](n) \in \Theta(n \log(\log(n)))$$

3.4 TRINE programmet

Lad os slutte med at præsentere et fuldt færdigt TRINE program, der finder primtal v.hj.a. Erathostenes Si. Her har vi, for at øge læseligheden og modificerbarheden, realiseret variablen K som en box, hvor hjælpevariablerne B, a, p osv. er skjult, og hvor det blot er de nødvendige procedurer, der stilles til rådighed. Boxen skal levere følgende fire procedurer

Init$[K](n)$
Size$[K]$
Min$[K]$
Delete$[K](k)$

Selve mængden K realiseres som et produkt af type

Type Set = **Prod**$(B :$ **List** (Bool)$, a, m :$ Int)

hvor B angiver bitvektoren, a angiver antallet, men m nu angiver det *mindste* element i den mængde K repræsenterer. Dette er en lille forskel fra før, hvor p angav det sidst fundne primtal. Denne ændring er en konsekvens af at realisere K som box, fordi boxen skal have selvstændig mening uafhængigt af, hvad den bliver brugt til.

Denne selvstændige mening udtrykker vi v.hj.a. en *box-invariant* (jfr. afsnit 10.2 i [TRINE]) på følgende måde

$$\begin{aligned} I(B,a,m) \quad : \quad & (a = |\{i \in 0..|B|) \mid B.(i) = \underline{\mathrm{t}}\}|) \wedge (0 \leq m \leq |B|) \wedge \\ & (B(0..m) = \underline{\mathrm{f}}) \wedge ((B.(m) = \underline{\mathrm{t}}) \vee (m = |B|)) \end{aligned}$$

Vi skal så sørge for, at boxens initialiseringsprocedure etablerer I, og at de øvrige procedurer holder I invariant.

Vi ønsker imidlertid også at angive specifikationer for Init, Size, Min og Delete, og her får vi igen brug for en abstraktionsfunktion (jfr. afsnit 2.3 i [Programmeringsteori I]). Dette skyldes, at medens box-invarianten udtaler sig om de konkrete variabler (B, a og m), så skal specifikationerne udtale sig om den tilsvarende abstrakte variabel, som jo stadig skal angive en *mængde*. Det er egenskaber ved denne, vi ønsker at se udefra, og hele pointen er jo netop at holde egenskaberne ved B, a og m skjult. Korrespondancen mellem abstrakte og konkrete variabler udtrykkes ved følgende abstraktionsfunktion α, som afbilder variabler af type Set ind i mængder:

$$\alpha(K) = \{k \in 0..|K.B| \mid K.B.(k) = \underline{\mathrm{t}}\}$$

Ved hjælp af α kan vi nu angive følgende specifikationer af boxens operationer

$$\begin{aligned} &\mathrm{Init}[K](n) \ \textbf{sat} \ n \geq 2 \rightarrow \alpha(K') = \{2,\ldots,n\} \\ &\mathrm{Size}[K] \ \textbf{sat} \ (\mathrm{Size}' = |\alpha(K)|) \wedge (\alpha(K) = \alpha(K')) \\ &\mathrm{Delete}[K](p) \ \textbf{sat} \ \alpha(K') = \alpha(K) \backslash \{p\} \\ &\mathrm{Min}[K] \ \textbf{sat} \ \alpha(K) \neq \emptyset \rightarrow (\mathrm{Min}' = \min(\alpha(K))) \wedge (\alpha(K') = \alpha(K)) \end{aligned}$$

Det færdige TRINE program ser nu ud som følger.

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

    Box PS
      Type Blist = List(Bool)
      Type Set = Prod(B: Blist, a, m: Int)

      { I(B, a, m) }

      Proc Init[K: Set](n: Int)
        K := Prod(Blist(false|2)++Blist(true|n-1), n-1, 2)
      end Init

      Proc Size[K: Set] → (Int)
        return K.a
      end Size

      Proc Min[K: Set] → (Int)
        return K.m
      end Min

      Proc Delete[K: Set](k: Int)
        if 0 ≤ k < |K.B| →
            if K.B.(k) → K.B.(k), K.a := false, K.a-1 fi
            do (K.m < |K.B|) ∧ ( ¬ K.B.(K.m)) → K.m := K.m+1 od
        fi
      end Delete
    end PS

    Box ISq
      Sequence(Int)
    end ISq

    Var P: ISq'Seq
    Var K: PS'Set
    Var n: Int

    write("n=")
```

```
    read[n]
    do  n < 2  →
          write("n=")
          read[n]
    od
    ISq'Init[P]
    PS'Init[K](n)
    do  { Erathostenes-invariant }
       PS'Size[K] > 0  →
          (+ Var  p, i: Int
             p := PS'Min[K]
             ISq'Rpush[P](p)
             i := p
             do  i ≤ n  →
                   PS'Delete[K](i)
                   i := i+p
             od
          +)
    od
    write("Primtal:  ")
    ISq'Print[P]
  +)
end  Erathostenes
```

3.5 Kompleksiteten af TRINE programmet

Efter at have introduceret datastrukturen PS skal vi naturligvis argumentere for, at analysen fra afsnit 3.3 stadig holder.

Det er nemt at se, at dette er tilfældet såfremt såvel Rpush som Delete har konstante *amortiserede* udførelsestider.

Dette ved vi allerede for Rpush, medens vi for Delete er nødt til at finde en nyttig potentialfunktion, der kan knyttes til PS.

Det følger af box invarianten, at alle værdier i $B(0..m)$ er lig med $\underline{\mathrm{f}}$, medens $B.(m) = \underline{\mathrm{t}}$ (med mindre $m = |B|$). B har altså udseendet

B: $\boxed{\underline{\text{f}} \mid \underline{\text{f}} \mid \ldots \mid \underline{\text{f}} \mid \underline{\text{t}} \mid ? \mid ? \mid \ldots \mid ?}$

$\uparrow$

m

Det er klart, at udførelsestiden for Delete er konstant, men mindre det er elementet m, der fjernes; i så fald er den proportional med længden af den længste ubrudte sekvens af $\underline{\text{f}}$'er, der står umiddelbart til højre for m. Inspireret heraf definerer vi følgende potentialfunktion

$$\Phi(B, m) = \alpha * (\text{ antal } \underline{\text{f}}\text{'er i } B(m..|B|))$$

hvor α er den "sædvanlige" proportionalitetsfaktor.

Betragt nu kaldet Delete$[K](m)$ og antag, at B har udseendet

B: $\boxed{\underline{\text{f}} \mid \underline{\text{f}} \mid \ldots \mid \underline{\text{f}} \mid \underline{\text{t}} \mid \underline{\text{f}} \mid \underline{\text{f}} \mid \ldots \mid \underline{\text{f}} \mid \underline{\text{t}} \mid ? \mid ? \mid \ldots \mid ?}$

$\uparrow$ $\quad$ $\uparrow$

m $\quad$ x

Efter kaldet vil vi da have

B': $\boxed{\underline{\text{f}} \mid \underline{\text{f}} \mid \ldots \mid \underline{\text{f}} \mid \underline{\text{f}} \mid \underline{\text{f}} \mid \underline{\text{f}} \mid \ldots \mid \underline{\text{f}} \mid \underline{\text{t}} \mid ? \mid ? \mid \ldots \mid ?}$

$\uparrow$

$x+1$ $\quad$ m

hvoraf det følger, at

$$T[\![\text{Delete}[K](m)]\!] \leq \alpha(x+1)$$

Samtidig gælder der

$$\Delta\Phi = -\alpha x$$

hvorfor

$$\begin{aligned} T_A[\![\text{Delete}[K](m)]\!] &= T[\![\text{Delete}[K](m)]\!] + \Delta\Phi \\ &\leq \alpha(x+1) - \alpha x = \alpha \\ &\in O(1) \end{aligned}$$

Alt i alt har Delete altså konstant amortiseret udførelsestid.

4 Referencer

[**TRINE**] E.M. Schmidt & M.I. Schwartzbach: Programmering og programmeringssproget TRINE. DAIMI FN-36, Datalogisk Afdeling, Aarhus Universitet, august 1993.

[**Bentley**] J. Bentley: Programming Pearls. Addison-Wesley Publishing Company, 1986.

[**Datastrukturer**] M.I. Schwartzbach: Datastrukturer. DAIMI FN-38, Datalogisk Afdeling, Aarhus Universitet, februar 1994.

[**Programmeringsteori I**] E.M. Schmidt: Programmeringsteori I. DAIMI FN-51, Datalogisk Afdeling, Aarhus Universitet, januar 1994.