PROGRAMMERINGSTEORI I

Erik Meineche Schmidt

Indhold

0 Introduktion **1**

1 Fundamentale begreber og notation **2**
1.1 Mængder 2
1.2 Følger 3
1.3 Funktioner 3
1.4 Prædikater 4

2 Elementære algoritmer **8**
2.1 Notation 8
2.2 Potensopløftning 10
2.3 Søgning 11
2.4 Fletning 15
2.5 Flytning 20

3 Kompleksitet **23**
3.1 Notation for størrelsesorden 24
3.2 Definition af tidskompleksitet 25
3.3 Direkte analyse 26
3.4 Analyse af nogle konkrete algoritmer 29
3.5 Analyse af en binær tæller 30
3.6 Tidskompleksitet af procedurer 32
3.7 Tidskompleksitet af rekursive procedurer 34

4 Specifikationer **38**

5 Referencer **41**

0 Introduktion

Dette hæfte indeholder sammen med hæftet **Programmeringsteori II** vigtige elementer af den del af programmeringsteorien, der beskæftiger sig med konstruktion af korrekte, effektive og let forståelige programmer.

Det konkrete indhold af dette hæfte er dels en introduktion til *algoritmer*, som er en slags abstrakte programstumper med formaliserede egenskaber og dels en introduktion til algoritmers og programmers kvantitative egenskaber, dvs. først og fremmest deres udførelsestid.

Efter en kort repetition af begreberne gyldighed og korrekthed, præsenteres de mest fundamentale algoritmer til søgning, fletning og sortering. Derefter etableres grundlaget for analyser af tidskompleksitet, og dette anvendes på en række simple eksempler. Endelig introduceres *specifikationer* som et værktøj til at beskrive egenskaber ved ubestemte "algoritmestumper".

1 Fundamentale begreber og notation

I dette afsnit repeteres/introduceres de grundlæggende matematiske begreber (med tilhørende notation), der vil blive brugt i dette hæfte.

1.1 Mængder

Vi bruger den sædvanlige mængdenotation (M_1 og M_2 er mængder og e er et element).

- $e \in M$: e tilhører M
- $e \notin M$: e tilhører ikke M
- $M_1 \cup M_2$: foreningsmængde
- $M_1 \cap M_2$: fællesmængde
- $M_1 \setminus M_2$: differens
- $M_1 \subseteq M_2$: inklusion
- $M_1 \subsetneq M_2$: ægte inklusion
- $|M|$: antal elementer i M
- $\emptyset$: den tomme mængde

De mængder, vi betragter, vil altid være delmængder af en mere eller mindre eksplicit defineret grundmængde G (G er ofte de hele tal, mængden af tegnfølger fra et givet alfabet o.l.). Enhver delmængde M af en grundmængde G er entydigt bestemt af sin *karakteristiske funktion*, som er en boolsk funktion (vi bruger $\underline{\mathrm{t}}$ og $\underline{\mathrm{f}}$ for de to Boolske værdier $\underline{\mathrm{true}}$ og $\underline{\mathrm{false}}$)

$$\chi_M : G \to \{\underline{\mathrm{t}}, \underline{\mathrm{f}}\}$$

defineret ved

$$\chi_M(e) = \begin{cases} \underline{\mathrm{t}} \text{ hvis } e \in M \\ \underline{\mathrm{f}} \text{ ellers} \end{cases}$$

Vi benytter følgende betegnelser for de mest almindelige mængder

$I\!N$: de naturlige tal, $I\!N = \{1, 2, 3, \ldots\}$
$I\!N_0$: de ikke-negative hele tal, $I\!N_0 = \{0, 1, 2, \ldots\}$
$\mathbf{Z}$: de hele tal, $\mathbf{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$

Da vi ofte får brug for at tale om mængder af heltal, hvori elementerne udgør et “interval”, indfører vi følgende betegnelse

$$n..m = \begin{cases} \emptyset \text{ hvis } n \geq m \\ \{n, n+1, \ldots, m-1\} \text{ ellers} \end{cases}$$

Bemærk, at $n..m$ er “halvåbent”, idet n er med, men m er ikke med.

1.2 Følger

Hvis M er en mængde, betegner M^* mængden af følger af elementer fra M. Følger er allerede introduceret i [TRINE] side 30 – vi bruger følgende yderligere notation

- den tomme følge ( ) betegnes også med λ
- hvis $f = (m_0, m_1, \ldots, m_{n-1})$ er en følge, betegner vi med $\mathrm{hd}(f)$ (head) *elementet* m_0, og med $\mathrm{tl}(f)$ (tail) *følgen* $(m_1, \ldots, m_{n-1})$. Såvel $\mathrm{hd}(f)$ som $\mathrm{tl}(f)$ er udefineret, hvis $f = \lambda$
- hvis $f' = (m'_0, m'_1, \ldots, m'_{n-1})$ og $f'' = (m''_0, \ldots, m''_{k-1})$ er to følger, betegnes deres sammensætning (eller *konkatenation*) med

  $$f = f' \cdot f'' = ((m'_0, \ldots, m'_{n-1}, m''_0, \ldots, m''_{k-1})$$

1.3 Funktioner

Givet mængder A og B, betegner

$$A \to B$$

mængden af funktioner fra A til B. Vi angiver, at f er en sådan funktion ved enten at skrive

$$f \in A \to B$$

eller det mere konventionelle

$$f : A \to B$$

Hvis $A' \subseteq A$, betegner vi *billedet af* A' med

$$f(A') = \{f(a) \mid a \in A'\}$$

og hvis $B' \subseteq B$, er *urbilledet af* B'

$$f^{-1}(B') = \{a \in A \mid f(a) \in B'\}$$

1.4 Prædikater

Begrebet (program) udsagn er introduceret uformelt i kapitel 8 i [TRINE]. Formelt er et udsagn et *prædikat*, dvs. en Boolsk funktion skrevet på "ligningsform". Prædikatet

$$x + y = z$$

angiver den Boolske funktion F

$$F(x, y, z) = \begin{cases} \underline{\mathrm{t}} \text{ hvis } x + y = z \\ \underline{\mathrm{f}} \text{ ellers} \end{cases}$$

Prædikater bygges op v.hj.a.

- konstanter $0, 1, \ldots, \underline{\mathrm{t}}, \underline{\mathrm{f}}$
- variable $x, y, z, \ldots$
- funktionssymboler $+, -, \leq, =, \ldots$
- konnektiver $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$
- kvantorer $\forall,\ \exists$

Vi skal ikke formalisere de præcise regler for prædikaters udseende, men nøjes med at angive og diskutere følgende eksempler

1) $P_1 : (x - 2 * y) > 0$
2) $P_2 : (x \leq y) \wedge (z = 0)$
3) $P_3 : (5 < 3) \Rightarrow \forall x \in 0..1 : x = y$
4) $P_4 : \forall x \in \mathbf{Z} : x * x > 0$
5) $P_5 : \exists x \in \mathbf{Z} : (\forall y \in \mathbf{Z} : x * y = z)$
6) $P_6 : \forall x \in 0..10 : (\exists y \in 0..11 : x < y)$
7) $P_7 : (x \leq y) \vee ((\forall x \in \mathbb{N} : x > 0) \wedge (y = 3))$

De variabler, der forekommer i et prædikat, deles op i *frie* og *bundne* variabler. De bundne variabler er alle de variabler, der er “knyttet til” en kvantor ($\forall$ eller $\exists$). De spiller nogenlunde samme rolle som lokale variabler (i indskudte sætninger) i et TRINE program, og de har også samme virkefelter. De variabler, der ikke er bundet, kaldes frie, og et prædikat er en Boolsk funktion af sine frie variabler.

Vi angiver de frie variabler i eksemplerne ovenfor ved for hvert prædikatsymbol P_i eksplicit at angive dets argumenter på den sædvanlige form $P_i(x, y, \ldots)$

1) $P_1(x, y)$
2) $P_2(x, y, z)$
3) $P_3(y)$
4) $P_4(\ )$
5) $P_5(z)$
6) $P_6(\ )$
7) $P_7(x, y)$

P_4 og P_6 er altså Boolske funktioner af 0 variabler, dvs. de er *konstanter*. En konstant Boolsk funktion er enten lig med $\underline{t}$ eller $\underline{f}$ (fordi der ikke er andre Boolske værdier).

P_4 siger, at “kvadratet på ethvert heltal er positivt”, hvilket er forkert, fordi $0 * 0 = 0$. Altså er $P_4 = \underline{f}$.

P_6 siger, at “for ethvert element i 0..10 findes der et større element i 0..11, hvilket unægteligt er rigtigt. Altså er $P_6 = \underline{t}$.

For de øvrige prædikater gælder, at værdierne afhænger af værdierne af

deres argumenter. Vi viser nogle eksempler

$$
\begin{array}{lcl}
P_1(6,2) & = & (6-4) > 0 = \underline{\mathrm{t}} \\
P_3(2) & = & (5 < 3) \Rightarrow \forall x \in 0..1 : x = 2 \\
 & = & \underline{\mathrm{f}} \Rightarrow (0 = 2) \wedge (1 = 2) \\
 & = & \underline{\mathrm{t}} \\
 & & \text{(fordi } \underline{\mathrm{f}} \text{ implicerer hvad som helst)} \\
P_5(0) & = & \exists x \in \mathbf{Z} : (\forall y \in \mathbf{Z} : x * y = 0) \\
 & = & \underline{\mathrm{t}} \\
 & & \text{(fordi } x = 0 \text{ giver } \forall y \in \mathbf{Z} : 0 * y = 0, \\
 & & \text{hvilket er rigtigt)} \\
P_7(5,y) & = & (5 \leq y) \vee ((\forall x \in I\!N : x > 0) \wedge (y = 3)) \\
 & = & (5 \leq y) \vee (y = 3) \\
 & & \text{(fordi alle naturlige tal er større end 0)}
\end{array}
$$

Som disse eksempler viser, finder man værdien af et prædikat på givne argumenter ved at erstatte forekomster af de frie variabler med de tilsvarende værdier, og derefter regne v.hj.a. de sædvanlige aritmetiske og logiske regler. Regnereglerne for $\wedge, \vee, \neg$ er som bekendt givet ved følgende tabel:

P	Q	$P \wedge Q$	$P \vee Q$	$\neg P$
$\underline{\mathrm{t}}$	$\underline{\mathrm{t}}$	$\underline{\mathrm{t}}$	$\underline{\mathrm{t}}$	$\underline{\mathrm{f}}$
$\underline{\mathrm{t}}$	$\underline{\mathrm{f}}$	$\underline{\mathrm{f}}$	$\underline{\mathrm{t}}$	$\underline{\mathrm{f}}$
$\underline{\mathrm{f}}$	$\underline{\mathrm{t}}$	$\underline{\mathrm{f}}$	$\underline{\mathrm{t}}$	$\underline{\mathrm{t}}$
$\underline{\mathrm{f}}$	$\underline{\mathrm{f}}$	$\underline{\mathrm{f}}$	$\underline{\mathrm{f}}$	$\underline{\mathrm{t}}$

Betydningen af $\Rightarrow$ og $\Leftrightarrow$ er nu som følger:

- $P \Rightarrow Q = \neg P \vee Q$
- $P \Leftrightarrow Q = (P \Rightarrow Q) \wedge (Q \Rightarrow P)$

Betydningen af kvantorerne $\forall$ og $\exists$ kan defineres på flere måder – vi vælger at gøre det rekursivt:

$$
\forall x \in X : P(x) = \begin{cases} \underline{\mathrm{t}} \text{ hvis } X = \emptyset \\ P(x_0) \wedge (\forall x \in X \backslash \{x_0\} : P(x)) \text{ for } x_0 \in X \end{cases}
$$

$$
\exists x \in X : P(x) = \begin{cases} \underline{\mathrm{f}} \text{ hvis } X = \emptyset \\ P(x_0) \vee (\exists x \in X \backslash \{x_0\} : P(x)) \text{ for } x_0 \in X \end{cases}
$$

Heraf følger den nyttige regneregel for negation

$$\neg(\forall x \in X : P(x)) = \exists x \in X : \neg P(x)$$

Bemærk også, at universel (eksistentiel) kvantificering over den tomme mængde altid er sand (falsk).

2 Elementære algoritmer

I dette kapitel konstruerer vi et antal simple algoritmer, som i forskellige iklædninger optræder igen og igen i praktisk programmering.

2.1 Notation

Vi skal anvende en notation for algoritmer, der er lidt mere abstrakt end fuldt færdige TRINE programmer og som illustreres af følgende udgave af Euklids algoritme (jfr. [TRINE] s. 93).

Algoritme : Euklid

Stimulans : n, m: $(n \geq 1) \wedge (m \geq 1)$
Respons : r: største fælles divisor af m og n
Metode : $p, q := n, m$
do $\{I\}$
$p > q \rightarrow p := p - q$
$\mid q > p \rightarrow q := q - p$
od
$r := p$

Denne notation vil blive brugt systematisk i det følgende til beskrivelse af abstrakte programmer, eller dele heraf, og er i det generelle tilfælde af formen

Algoritme : [ ]

Stimulans : [ ]
Respons : [ ]
Metode : [ ]

hvor stimulans og respons beskriver omgivelsernes påvirkninger af algoritmen og dennes reaktion herpå. Metodedelen er en abstrakt beskrivelse af algoritmen, normalt i stil med kroppen af en proces. Når vi bruger ordet *algoritme* i stedet for program, proces eller lignende, skyldes det at sidstnævnte indeholder en del elementer i form af erklæringer, initialiseringer,

ind- og udlæsning osv., som vi ofte vil være interesseret i at abstrahere væk fra. Der vil også være situationer, hvor vi ønsker at udtage en mindre del af en proces eller et system og betragte denne i isolation, og i sådanne tilfælde er det også hensigtsmæssigt med et nyt navn for beskrivelsen.

Vi anvender begreberne udsagn (som nu er prædikater), invariant og termineringsfunktion på præcis samme måde som i [TRINE]. Vi kan dermed tale om at (dekorerede) algoritmer er gyldige og korrekte på følgende måde.

En tilstand i en algoritme er (i lighed med en TRINE tilstand) en tilknytning af værdier til algoritmens variabler. En tilstand σ opfylder et prædikat P (hvis frie variabler er programvariabler), såfremt P har værdien $\underline{t}$ anvendt på de frie variablers værdier i σ.

Et udsagn i en algoritme er *gyldigt for en udførelse*, såfremt udsagnet opfyldes af algoritmens tilstand, hver gang det nås under den pågældende udførelse.

En *algoritme er gyldig*, såfremt alle dens udsagn er gyldige for alle de udførelser der starter med en tilstand, som opfylder algoritmens stimulansprædikat.

Vi kan nu præcisere kravene til algoritmers korrekthed på følgende måde.

> En algoritme er *korrekt* såfremt enhver udførelse, der starter i en tilstand der opfylder stimulansprædikatet, er endelig, og slutter i en tilstand der opfylder responsprædikatet.

Med disse definitioner skulle det være klart, at hvis invarianten i Euklid ovenfor er

$$I : (sfd(p, q) = sfd(m, n)) \wedge (n \geq 1) \wedge (m \geq 1) \wedge (p \geq 1) \wedge (q \geq 1)$$

så er algoritmen både gyldig og korrekt. Termineringsfunktionen (som vi skal betegne med μ) er

$$\mu(p, q) = p + q$$

2.2 Potensopløftning

Potensopløftningsprogrammet side 96 i [TRINE] kan formuleres som følgende algoritme

Algoritme : Potensopløftning

Stimulans : $n, p : p \geq 0$
Respons : $r : r = n^p$
Metode : $\ll$Initialiser r og $q \gg$
do $\{r * n^q = n^p\}$
$q \neq 0 \rightarrow \ll$Opdater r og $q \gg$
od

og vi har allerede set, at følgende konkretiseringer giver en gyldig algoritme, som også standser.

$\ll$Initialiser r og $q \gg$ **is** $r, q := 1, p$
$\ll$Opdater r og $q \gg$ **is** $r, q := r * n, q - 1$

Det er også nævnt, at denne metode involverer et antal multiplikationer, der er proportionalt med p. Vi kan finde en væsentlig mere effektiv potensopløftningsmetode ved at ændre invarianten på følgende måde (h er en ny variabel)

$\ll$Initialiser r, q og $h \gg$
do $\{r * h^q = n^p\}$
$q \neq 0 \rightarrow \ll$Opdater r, q og $h \gg$
od

og samtidig observere, at når q er *lige*, kan invarianten opretholdes ved at *kvadrere* h og samtidigt *halvere* q. Den konkrete algoritme kan nu skrives på følgende måde

Algoritme : Logaritmisk potensopløftning

Stimulans : $n, p : p \geq 0$
Respons : $r : r = n^p$
Metode : $r, q, h := 1, p, n$
do $\{r * h^q = n^p\}$
$q \neq 0 \rightarrow$ **if** q ulige $\rightarrow r, q := r * h, q - 1$
| q lige $\rightarrow h, q := h * h, q/2$
fi
od

Argumentet for at invarianten er gyldig kan sammenfattes i følgende to ligninger, som viser, at hvis invarianten er opfyldt inden et gennemløb af løkken, så er den også opfyldt efter dette gennemløb.

q ulige: $r' * h'^{q'} = (r * h) * h^{q-1}$
q lige : $r' * h'^{q'} = r * (h^2)^{q/2}$

Læseren opfordres til at undersøge, hvor mange multiplikationer der udføres af denne algoritme.

2.3 Søgning

Enhver søgning er karakteriseret ved, at man blandt en samling af elementer leder efter ét eller flere med bestemte karakteristika. Samlingen af elementer der ledes i, kalder vi *søgedomænet*, og de elementer der ledes efter kalder vi *målgruppen*.

Vi skal betragte det på samme tid simple og forholdsvis generelle tilfælde, hvor søgedomænet er en *funktion* fra en indeksmængde X ind i en elementmængde E, og hvor målgruppen består af et enkelt element af type E. Vi ønsker at afgøre om dette element forekommer i funktionens billedmængde og i så fald ønsker vi at returnere (et af) dets indeks.

I følgende algoritme er S søgedomænet, m er målelementet, og K afgrænser den del af S's indeksmængde, hvis funktionsværdier stadig er målgruppekandidater. Resultatet angives i mængden R, som enten er tom eller indeholder et indeks x, for hvilket $S(x) = m$.

Algoritme : Søgning

Stimulans : $S : X \to E$, søgedomæne
$m : E$, målelement

Respons : $R : S(R) = S(S^{-1}(m)) \wedge (|R| \leq 1)$

Metode : $\ll$Initialiser R og $K \gg$
do $\{(R \subseteq S^{-1}(m) \subseteq K) \wedge (|R| \leq 1)\}$
$(K \neq \emptyset) \wedge (R = \emptyset) \to \ll$Opdater R og $K \gg$
od

Denne algoritme er forholdsvis abstrakt og dens anvendelse i konkrete tilfælde forudsætter realiseringer af variablerne S, K og R. Det er imidlertid interessant, at vigtige egenskaber ved algoritmen kan vises selv på dette abstraktionsniveau. Mere præcist kan vi argumentere for, at hvis algoritmen er gyldig og den standser, så er den også korrekt. Argumentet er som følger.

Hvis algoritmen standser, er den kontrollerende betingelse falsk, dvs.

$$(K = \emptyset) \vee (R \neq \emptyset)$$

og der gælder yderligere at invarianten er opfyldt. Men så er responsudsagnet også opfyldt, fordi

$K = \emptyset$: Responsudsagnet følger trivielt fra invarianten.

$R \neq \emptyset$: Da $R \subseteq S^{-1}(m)$ gælder der $S(R) \subseteq S(S^{-1}(m)) \subseteq \{m\}$, men da $R \neq \emptyset$, må $S(R)$ indeholde mindst ét element, hvorfor der også gælder, at $\{m\} \subseteq S(R)$.

Vi viser nu to nyttige realiseringer af denne algoritme. I begge tilfælde realiseres de abstrakte variabler på følgende måde

- S er en liste af elementer af type E
- K repræsenteres som et interval lav..høj hvor lav og høj er to *konkrete* variable
- R repræsenteres som en variabel r af type Int. Sammenhængen mellem R og r angives af $R = \alpha(r)$

hvor α er følgende såkaldte *abstraktionsfunktion*

$$\alpha(r) = \begin{cases} \emptyset \text{ hvis } r = ?\text{–Int} \\ r \text{ ellers} \end{cases}$$

Den første realisering er som følger

Algoritme : Lineær søgning

Stimulans : S : **List**(E), søgedomæne
$m : E$, målelement

Respons : $r : S(\alpha(r)) = S(S^{-1}(m))$

Metode : lav,høj,r := 0, $|S|$, ?–Int
do $\{I\}$
(lav < høj) $\wedge(r =$?–Int$) \rightarrow$
if $S.$(lav) $= m \rightarrow r :=$ lav
& true $\rightarrow$ lav := lav + 1
fi
od

Invarianten, I, *den konkrete invariant*, fås ved at indsætte $\alpha(r)$ for R og lav..høj for K i den abstrakte invariant ovenfor. Dette giver

$$I \;:\; \alpha(r) \subseteq S^{-1}(m) \subseteq \text{lav..høj}$$

(idet det jo er klart, at $|\alpha(r)| \leq 1$).

Det er nemt at se, at de to konkretiseringer af

$$\ll\text{Initialiser } K \text{ og } R \gg \text{ og } \ll\text{Opdater } K \text{ og } R \gg$$

sørger for at holde I opfyldt, dvs. at algoritmen er gyldig. At algoritmen terminerer indses ved at betragte følgende termineringsfunktion

$$\mu(\text{lav,høj,r}) = \text{høj-lav-}|\alpha(r)|$$

Pointen er nu, at når den konkrete algoritme således er gyldig og terminerer, så følger det af argumentet ovenfor (for den abstrakte algoritme), at den konkrete også er korrekt.

I det næste eksempel antager vi, at værditypen E er ordnet og at listen S er ordnet i ikke-aftagende orden, dvs. at S opfylder prædikatet $\forall i, j \in 0..|S| : i \leq j \Rightarrow S.(i) \leq S.(j)$. I dette tilfælde kan vi anvende *binær*

søgning, dvs. en søgestrategi der består i at udvælge det midterste element i kandidatintervallet lav..høj for derefter at bruge ordningen til at halvere kandidaterne, hvis det inspicerede element ikke er det, der ledes efter.

Idet repræsentationen er den samme som før (og vi for simpelheds skyld antager, at E er heltallene), får vi følgende algoritme, hvor invarianten I er den samme som ovenfor.

Algoritme : Binær søgning

Stimulans : S : **List** (Int), søgedomæne
m : Int, målelement
Respons : $r : S(\alpha(r)) = S(S^{-1}(m))$
Metode : lav,høj,r := $0, |S|,$?–Int
do $\{I\}$
(lav < høj) $\wedge$ (r = ?–Int) $\rightarrow$
midt := (lav + høj)/2
if S.(midt) $= m \rightarrow r$:= midt
| S.(midt) $< m \rightarrow$ lav := midt + 1
| $m < S$.(midt) $\rightarrow$ høj := midt
fi
od

Det er nemt at vise, at denne algoritme også er gyldig. Med hensyn til terminering kan vi også genbruge termineringsfunktionen fra før. Her er det imidlertid ikke helt oplagt, at den altid aftager. Det detaljerede bevis herfor er som følger

- S.(midt) $= m$: $\mu(\text{lav}', \text{høj}', r') = \text{høj}' - \text{lav}' - |\alpha(r')| = \text{høj} - \text{lav} - 1 < \text{høj} - \text{lav} = \text{høj} - \text{lav} - |\alpha(r)| = \mu(\text{lav}, \text{høj}, r)$
- S.(midt) $< m$: $\mu(\text{lav}', \text{høj}', r') = \text{høj}' - \text{lav}' = \text{høj} - (\text{midt} + 1) = \text{høj} - ((\text{lav} + \text{høj})/2 + 1) \leq \text{høj} - (\text{lav} + 1) < \text{høj} - \text{lav} = \mu(\text{lav}, \text{høj}, r)$ fordi lav $\leq$ (lav + høj)/2 når lav < høj
- $m < S$.(midt) : $\mu(\text{lav}', \text{høj}', r') = \text{høj}' - \text{lav}' = \text{midt} - \text{lav} = (\text{lav} + \text{høj})/2 - \text{lav} < \text{høj} - \text{lav} = \mu(\text{lav}, \text{høj}, r)$ fordi (lav + høj)/2 < høj når lav < høj.

2.4 Fletning

Vi betragter nu en anden problemstilling, hvis løsning til en vis grad kan "standardiseres". Problemet består i at *flette* to ordnede følger og kan illustreres v.hj.a. følgende eksempel. Fletningen af følgerne

(3,5,9,10,15)

og

(1,5,8,13,13,16)

er følgen

(1,3,5,5,8,9,10,13,13,15,16)

I almindelighed er fletningen af to ordnede følger en ordnet følge, som indeholder præcis de elementer, der findes i de to følger.

Vi betegner fletningen af følgerne f_1 og f_2 med

$$\text{FLET}(f_1, f_2)$$

Vi får nu følgende algoritme for fletning, hvis idé er, at elementerne fra de to følger behandles ét for ét ved et symmetrisk *sekventielt* gennemløb af de to følger.

Algoritme: Fletning
Stimulans : $f_1, f_2 : E^*$, ordnede følger
Respons : $f : E^*,\ f = \text{FLET}(f_1, f_2)$
Metode : $f, g_1, g_2 := \lambda, f_1, f_2$

$$
\begin{array}{l}
\textbf{do}\ \{f \cdot \text{FLET}(g_1, g_2) = \text{FLET}(f_1, f_2)\} \\
\quad\ \ g_1 \neq \lambda \wedge g_2 = \lambda \rightarrow \ll \mathit{Flyt}_1 \gg \\
\quad |\ g_1 = \lambda \wedge g_2 \neq \lambda \rightarrow \ll \mathit{Flyt}_2 \gg \\
\quad |\ g_1 \neq \lambda \wedge g_2 \neq \lambda \rightarrow\ \textbf{if}\ \text{hd}(g_1) \leq \text{hd}(g_2) \rightarrow \ll \mathit{Flyt}_1 \gg \\
\qquad\qquad\qquad\qquad\qquad\quad |\ \text{hd}(g_1) \geq \text{hd}(g_2) \rightarrow \ll \mathit{Flyt}_2 \gg \\
\qquad\qquad\qquad\qquad\qquad\ \ \textbf{fi} \\
\textbf{od}
\end{array}
$$

hvor $\ll \mathit{Flyt}_1 \gg$ og $\ll \mathit{Flyt}_2 \gg$ er

$$
\begin{array}{l}
\ll \mathit{Flyt}_1 \gg\ \textbf{is}\ f, g_1 := f \cdot \text{hd}(g_1), \text{tl}(g_1) \\
\ll \mathit{Flyt}_2 \gg\ \textbf{is}\ f, g_2 := f \cdot \text{hd}(g_2), \text{tl}(g_2)
\end{array}
$$

Vi overlader det til læseren at argumentere for at algoritmen er korrekt (argumentet består hovedsageligt i at observere, at invarianten er gyldig).

Følgende Trine program viser en konkretisering af algoritmen, hvor de indgående følger er lister af heltal. `Sequence.tri` indeholder Boxen Sequence(T) (jfr. afsnit 10.5 i [TRINE]).

```
Process Fletning

(* Programmet indlæser to lister af heltal og kontrollerer
   at de er ordnede i ikke-aftagende orden. Derefter flettes
   de under anvendelse af fletteálgoritmen. *)

   (+ @"sequence.tri"

      Box ISq
         Sequence(Int)
      end ISq

      Proc Indlæs[f: Vector]
         (+ Proc check[vec: Vector] → (Bool)
               (+ Var i: Int (* Søgning efter i med vec.(i-1) > vec.(i) *)
                  i:= 1
                  do i < | vec | →
                        if vec.(i-1) ≤ vec.(i) → i:= i+1
                        & true → return false
                        fi
                  od
                  return true
               +)
            end check

            write("Liste: ")
            read[f]
            do ¬ check[f] →
                  write(eol, "Om igen: ")
                  read[f]
            od
         +)
      end Indlæs
```

```
        Var f1, f2: Vector

        Indlæs[f1]
        Indlæs[f2] (* f1 og f2 er ordnede *)
        (+ Var f, g1, g2: ISq'Seq
           ISq'Con[f](Vector())
           ISq'Con[g1](f1)
           ISq'Con[g2](f2)
           do  (* fletteealgoritme *)
                 (ISq'Len[g1] > 0) ∧ (ISq'Len[g2] = 0 ) → ≪ Flyt1 ≫
             | (ISq'Len[g1 verb:]: = 0) ∧ (ISq'Len[g2ěrb:]: > 0) → ≪ Flyt2 ≫
             | (ISq'Len[g1 verb:]: > 0) ∧ (ISq'Len[g2] > 0) →
                 if ISq'Ltop[g1] ≤ ISq'Ltop[g2] → ≪ Flyt1 ≫
                  | ISq'Ltop[g1] ≥ ISqěrb:':Ltop[g2] → ≪ Flyt2 ≫
                 fi
           od
           write(eol, "Resultat: ")
           ISq'Print[f]
        +)
    +)
end Fletning
where ≪ Flyt1 ≫ is
          ISq'Rpush[f](ISq'Ltop[g1])
          ISq'Lpop[g1]

where ≪ Flyt2 ≫ is
          ISq'Rpush[f](ISq'Ltop[g2])
          ISq'Lpop[g2]
```

Der findes en del algoritmer, som "næsten" er fletninger, dvs. de to følger gennemløbes på samme måde som ovenfor, men det er ikke sikkert, at alt kopieres "råt". Her er fletteealgoritmen også nyttig, fordi det er nemt at modificere den i det konkrete tilfælde. Et eksempel er en algoritme, der givet to ordnede følger uden "dubletter" finder præcis de elementer, der optræder i dem begge. I dette tilfælde skal algoritmen kun "producere" noget, når hovedet af de to følger er ens, og resten skal ignoreres.

```
Process Snit

(* Programmet indlæser to lister af heltal og kontrollerer
   at de er ordnede i voksende orden. Derefter findes de
   elementer der forekommer i dem begge. *)

   (+ @"sequence.tri"

      Box ISq
         Sequence(Int)
      end ISq

      Proc Indlæs[f: Vector]
         (+ Proc check[vec: Vector] → (Bool)
               (+ Var i: Int (* Søgning efter i med vec.(i-1) > vec.(i) *)
                  i := 1
                  do i < | vec | →
                        if vec.(i-1) < vec.(i) → i := i+1
                        ff true → return false
                        fi
                  od
                  return true
               +)
            end check

            write("Liste: ")
            read[f]
            do ¬ check[f] →
                  write(eol, "Om igen: ")
                  read[f]
            od
         +)
      end Indlæs
```

```
        Var f1, f2: Vector

        Indlæs[f1]
        Indlæs[f2] (* f1 og f2 er voksende *)
        (+ Var f, g1, g2: ISq'Seq
           ISq'Con[f](Vector())
           ISq'Con[g1](f1)
           ISq'Con[g2](f2)
           do  (* flettealgoritme *)
                (ISq'Len[g1] > 0) ∧ (ISq'Len[g2] = 0 ) → ≪ Flyt1 ≫
             |  (ISq'Len[g1 verb:]: = 0) ∧ (ISq'Len[g2ěrb:]: > 0) → ≪ Flyt2 ≫
             |  (ISq'Len[g1 verb:]: > 0) ∧ (ISq'Len[g2] > 0) →
                if ISq'Ltop[g1] < ISq'Ltop[g2] → ≪ Flyt1 ≫
                 | ISq'Ltop[g1] > ISq'Ltop[g2] → ≪ Flyt2 ≫
                 | ISq'Ltop[g1] = ISq'Ltop[g2] → ≪ Flyt12 ≫
                fi
           od
           write( "Resultat: ")
           ISq'Print[f]
        +)
    +)
end Snit
where ≪ Flyt1 ≫ is
          ISq'Lpop[g1]

where ≪ Flyt2 ≫ is
          ISq'Lpop[g2]

where ≪ Flyt12 ≫ is
          ISq'Rpush[f](ISq'Ltop[g1])
          ISq'Lpop[g1]
          ISq'Lpop[g2]
```

2.5 Flytning

Som den sidste problemstilling i dette kapitel betragter vi diverse former for *omflytningsalgoritmer*. En omflytningsalgoritme er karakteriseret ved, at den manipulerer elementerne i en struktur v.hj.a. *ombytninger*, dvs. den hverken fjerner eller tilføjer elementer, men nøjes med at flytte rundt på dem. Denstruktur, der er relevant for omflytningsalgoritmer er en liste w, som indeholder elementer fra en ordnet mængde E, og som kun kan manipuleres v.hj.a. følgende operationer

$\lvert w\rvert$	: antal elementer i w
$w.(i)$	: det i'te element i w
$w.(i) < w.(j)$	: sammenligning af i'te og j'te element i w
$w.(i) :=: w.(j)$	: ombytning af i'te og j'te element i w

De vigtigste eksempler på omflytningsalgoritmer er sorteringsalgoritmer, som på grund af deres udbredte anvendelse i praksis er et af de områder inden for datalogien, der har været undersøgt mest intenst. Vi skal i dette afsnit nøjes med at betragte to af de simpleste og mest ligefremme sorteringsmetoder samt en omflytningsalgoritme, der bl.a. kan bruges som hjælpeprocedure i en mere interessant sorteringsalgoritme.

Det første eksempel er såkaldt *indsættelsessortering*, som kan beskrives på følgende måde (i resten af dette afsnit betyder *sorteret*, at vektoren er sorteret i ikke-aftagende orden)

Algoritme: Indsættelsessortering
Stimulans: w: vektor
Respons : w: sorteret
Metode : $i := 0$
  do $\{(w(0..i)$ er sorteret$)\wedge(0 \leq i \leq |w|)\}$
    $i < |w| \rightarrow$ $\ll$indsæt $w.(i)$ i $w(0..i)\gg$
      $i := i + 1$
  od

Det skulle være klart, at hvis denne algoritme er gyldig, så er den også korrekt. Selve indsættelsen kan beskrives som en (baglæns) søgning (ledsaget af en "skubning") efter det sidste element i $w(0..i)$, som er mindre end eller lig med $w.(i)$. Dette kan gøres på følgende måde.

Algoritme : Indsættelsessortering

Stimulans: w: vektor
Respons : w: sorteret
Metode : $i := 0$

do $\{(w(0..i)$ er sorteret$) \wedge (0 \leq i \leq |w|)\}$
  $i < |w| \rightarrow$ j, fortsæt := $i - 1$,true
    do {søgning efter $j : w.(j) \leq w.(j+1)\}$
      $(0 \leq j) \wedge$ fortsæt $\rightarrow$ **if** $w.(j) \leq w.(j+1) \rightarrow$ fortsæt := false
        | $w.(j) > w.(j+1) \rightarrow$ $w.(j) :=: w.(j+1)$
          $j := j - 1$
      fi
    od
    $i := i + 1$
od

Det andet eksempel på sortering er udvalgssortering, som kan beskrives som følger.

Algoritme : Udvalgssortering

Stimulans : w: vektor
Respons : w: sorteret
Metode : $i := 0$

do $\{(w(0..i)$ er sorteret$) \wedge (w(0..i) \leq w(i..|w|) \wedge (0 \leq i \leq |w|)\}$
  $i < |w| \rightarrow$ ≪find det mindste element i $w(i..|w|)$ og
    ombyt det med $w.(i)$ ≫
    $i := i + 1$
od

Vi overlader det til læseren at skrive ≪find det mindste …≫, og nøjes med at pege på, at uligheden $w(0..i) \leq w(i..|w|)$ betyder, at alle elementer i $w(0..i)$ er mindre end eller lig med alle elementer i $w(i..|w|)$.[1]

Det sidste eksempel på omflytning er en algoritme, som deler en vektor i to dele på en sådan måde, at alle elementer i den nedre del er mindre eller lig med en værdi e, og alle elementer i den øvre del er større end eller lig med e. Udover at foretage omflytningen, giver algoritmen også besked om, hvor delepunktet ligger.

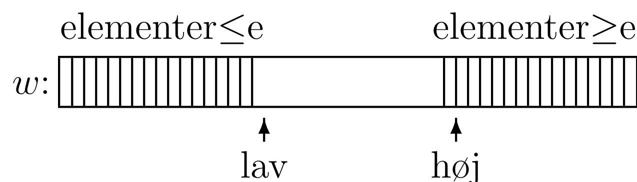
Algoritmen ser ud som følger.

[1] Den præcise betydning af $w(0..i) \leq w(i..|w|)$ er $\forall j \in 0..i : (\forall k \in i..|w| : w.(j) \leq w.(k))$.

Algoritme : Opdel
Stimulans : w, e: w vektor, e element
Respons : $w, r : (0 \leq r \leq |w|) \wedge (w(0..r) \leq e \leq w(r..|w|))$
Metode : $\text{lav}, \text{høj} := 0, |w|$

$$
\begin{array}{l}
\textbf{do}\ \{(w(0..\text{lav}) \leq e \leq w(\text{høj}..|w|)) \wedge (0 \leq \text{lav} \leq \text{høj} \leq |w|)\} \\
\quad \text{lav} < \text{høj} \rightarrow \textbf{if}\ \ w.(\text{lav}) \leq e \rightarrow \text{lav} := \text{lav} + 1 \\
\qquad\qquad\qquad\quad \mid w.(\text{høj} - 1) < e < w.(\text{lav}) \rightarrow w.(\text{lav}) :=: w.(\text{høj} - 1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{lav}, \text{høj} := \text{lav} + 1, \text{høj} - 1 \\
\qquad\qquad\qquad\quad \mid e \leq w.(\text{høj} - 1) \rightarrow \text{høj} := \text{høj} - 1 \\
\qquad\qquad\qquad\quad \textbf{fi} \\
\textbf{od} \\
r := \text{lav}
\end{array}
$$

Vi kan illustrere invarianten for algoritmen v.hj.a. følgende tegning



Det skulle være nemt at se, at algoritmen er gyldig. Læseren opfordres til at overbevise sig om, at den også er korrekt.

3 Kompleksitet

I de foregående kapitler har vi beskæftiget os med de *kvalitative* aspekter af programmering, dvs. spørgsmål som korrekthed og simpelhed i de anvendte algoritmer. Algoritmer har som bekendt også *kvantitative* aspekter, som det er nødvendigt at beskæftige sig med, og blandt disse er først og fremmest spørgsmålet om, hvor lang tid det tager at udføre dem. Selv om moderne datamater er særdeles hurtige, er der en mangfoldighed af problemer, hvor det er afgørende, at det program, der løser problemet, er effektivt. Der er naturligvis mange eksempler på sådanne programmer, blandt hvilke såkaldte *realtidsprogrammer* er nogen af de bedste. Et sådant program har til formål at reagere på påvirkninger *inden for en bestemt maksimal tid*, og det er stærkt generende, måske ligefrem meningsløst, hvis det ikke sker. Et eksempel på et meningsløst program af denne type er et lønudbetalingssystem for ugelønnede, som bruger en måned på at gennemføre de nødvendige beregninger, eller (måske mere realistisk) en simulator til træning af piloter, som er flere minutter om at beregne konsekvenserne af en pilotreaktion, hvis resultat indfinder sig på sekunder i en rigtig flyvemaskine.

Omend programmers *pladsforbrug* ofte er lige så afgørende en faktor som deres tidsforbrug, skal vi i dette afsnit indskrænke os til at betragte sidstnævnte. Vi definerer *tidskompleksiteten* af et program eller en algoritme, som det antal *primitive operationer*, der udføres fra programmet (eller algoritmen) startes i en eller anden starttilstand og til det standser (hvis det da overhovedet standser).

Ved en primitiv operation vil vi forstå en operation, som med rimelighed kan siges at udgøre en tidsmæssig enhed, når et program afvikles på en datamaskine. Der er flere muligheder for at lægge sig fast på en sådan enhed, men det viser sig (jfr. kapitel 17 i [TRINE]), at disse alle mere eller mindre svarer til at definere en primitiv operation som en læsning, skrivning eller kopiering af enten en *simpel værdi* eller af en *reference* til en variabel (standardværdier og pointerværdier opfattes i denne sammenhæng også som simple værdier). Tidskompleksiteten af f.eks. et TRINE program defineres således som det samlede antal læsninger, skrivninger og kopieringer af standardværdier, pointerværdier, værdier fra Unit, Int, Bool, Char, Real eller af variabelreferencer under programmets udførelse.

At opgøre denne tidskompleksitet eksakt er imidlertid, selv for små programmer, en overvældende sag, og vi skal derfor indskrænke os til at betragte dens *størrelsesorden*. Betimeligheden heraf er der argumenteret indgående for i bl.a. [Bentley] – her skal vi præcisere den notation, der bruges til at tale om størrelsesordener, ligesom vi skal se lidt på teknikker til at vurdere tidskompleksitet.

3.1 Notation for størrelsesorden

I det følgende betegner $\mathbb{R}_0$ ($\mathbb{R}_+$) de ikke-negative (positive) reelle tal. Vi betragter funktioner af formen

$$f : \mathbb{N}_0 \to \mathbb{R}_0$$

og definerer for en vilkårlig sådan funktion følgende klasse af funktioner

$$O(f) = \{g : \mathbb{N}_0 \to \mathbb{R}_0 \mid \exists k \in \mathbb{R}_+, n_0 \in \mathbb{N} : \forall n \geq n_0 : g(n) \leq kf(n)\}$$

$O(f)$ er klassen af funktioner, der *asymptotisk* (dvs. for store værdier af n) er opadtil begrænset af f "pånær en konstant faktor". Følgende er eksempler på anvendelse af notationen

$$4n^2 + 3n - 13 \in O(n^2)$$
$$\frac{1}{3}n + \sqrt{n} \in O(n)$$
$$35n + n(\sqrt{n} + 7) \in O(n^2)$$
$$2^n + 4n^4 \in O(2^n)$$

Det er klart, at vi altid vil være interesseret i at angive den mindst mulige begrænsende funktion. I den forbindelse skal vi lejlighedsvis også anvende følgende notation, som i analogi med $O(f)$ angiver funktioner, der er nedadtil begrænset.

$$\Omega(f) : \{g : \mathbb{N}_0 \to \mathbb{R}_0 \mid \exists k \in \mathbb{R}_+, n_0 \in \mathbb{N} : \forall n \geq n_0 : g(n) \geq kf(n)\}$$

Vi kan også tale om klassen af funktioner, der begrænses af f fra begge sider

$$\Theta(f) = O(f) \cap \Omega(f)$$

Vi skal ikke beskæftige os særlig meget med regneregler for disse funktionsklasser, men følgende er nyttige at kende.

Lad $g_1 \in O(f_1), g_2 \in O(f_2)$. Der gælder da

- $kg_1 \in O(f_1)$ for alle $k \geq 0$
- $g_1 \cdot g_2 \in O(f_1 \cdot f_2)$
- $g_1 + g_2 \in O(f_1 + f_2)$
- $f_1 \in O(f_2) \Rightarrow g_1 \in O(f_2)$

3.2 Definition af tidskompleksitet

Det fremgår af indledningen, at når tidskompleksiteten af et program eller en algoritme skal vurderes, skal man vurdere antallet af simple værdier og referencer, der "behandles" under programmets (algoritmens) udførelse. Det er nemt at indse, at denne "behandling" sker i forbindelse med

- tilordningssætninger
- betingelser
- kommunikationssætninger
- procedurekald
- variabelerklæringer

Vi vil derfor være godt hjulpet med at analysere omkostningerne for hver af disse programelementer. Her kan der være variationer fra sprog til sprog, og følgende diskussion handler derfor om TRINE.

TRINE's implementation på en fysisk datamat er beskrevet i kapitel 17 i [TRINE], og der er her argumenteret for at det vi er interesseret i, er summen af *størrelserne* af de værdier, der **udregnes** under afviklingen af programmet eller algoritmen. Størrelsen af en TRINE værdi defineres som følger (jfr. igen [TRINE])

- størrelsen af en simpel værdi er 1
- størrelsen af en sekvens er 2 + summen af størrelserne af dens elementer

Følgende eksempler illustrerer denne definition

- størrelsen af Int-konstanten 17 er 1
- størrelsen af List-konstanten Vector (0|100) er 102
- størrelsen af Prod-konstanten Point(7,0) er 4
- størrelsen af Sum-konstanten Res(x:7) er 4

Baseret på dette størrelsesmål, og på den tidligere nævnte rimelighed af at betragte manipulation af en simpel værdi eller reference som en primitiv operation, kan vi nu præcisere tidskompleksiteten af et TRINE program eller algoritme på følgende måde.

Tidskompleksiteten af et program (en algoritme) i TRINE er summen af størrelserne af de værdier, der udregnes under udførelsen af programmet (algoritmen) med en starttilstand, der opfylder stimulansprædikatet. Såfremt der findes flere sådanne udførelser, er tidskompleksiteten maksimum af summen af størrelserne over samtlige udførelser.

3.3 Direkte analyse

Vi betragter nu nogle eksempler på vurdering af tidskompleksitet. Af bekvemmelighedsårsager skal vi benytte notationen $T[\![A]\!]$ til at angive tidskompleksiteten af algoritmen (programmet A).

Betragt algoritmen til lineær potensopløftning fra side 10.

Algoritme : Lineær potensopløftning

Stimulans	:	$n, p : p \geq 0$
Respons	:	$r : r = n^p$
Metode	:	$r, q := 1, p$
		do $\{r * n^q = n^p\}$
		$\quad q \neq 0 \rightarrow r, q := r * n, q - 1$
		od

Det er klart, at alle værdier, der udregnes under udførelse af denne algoritme, har størrelse 1, og da løkken gennemløbes p gange, følger det at

$$T[\![\text{Lineær potensopløftning}]\!] \in \Theta(p)$$

Det er også klart, at af de to stimulans-variabler til Lineær potensopløftning, n og p, er det p der betyder noget for kompleksiteten, medens værdien af n er ligegyldig. I sådanne situationer skal vi fremhæve denne afhængighed eksplicit ved at tilføje den relevante parameter til $T[\![]\!]$ på følgende måde

$$T[\![\text{Lineær potensopløftning}]\!](p) \in \Theta(p)$$

Betragt nu som et andet eksempel på denne parametrisering følgende algoritmestump

```
A1:  i := 1
     do
        i ≤ n →  j := 1
                 do
                    j ≤ i → j := j + 1      A2
                 od
                 i := i + 1
     od
```

Det er klart, at kompleksiteten af A_1 afhænger af n, og at kompleksiteten af A_2 afhænger af i. Dette kan tydeliggøres på følgende måde

$$\begin{aligned}
T[\![A_1]\!](n) &\in O(T[\![A_2]\!](1) + \ldots + T[\![A_2]\!](i) + \ldots + T[\![A_2]\!](n)) \\
&= O(\sum_{i=1}^{n} T[\![A_2]\!](i)) \\
&= O(\sum_{i=1}^{n} i) \\
&= O(n^2)
\end{aligned}$$

Antag nu, at tilordningssætningen $i := i+1$ erstattes af sætningen $i := 2*i$. I så fald udføres der et antal iterationer i den yderste **do**-sætning, der er lig med det antal gange man skal multiplicere 1 med 2 for "at nå" n, dvs. $\log(n)$ gange[2]. Vi får da

$$
\begin{aligned}
T[\![A_1]\!] &\approx T[\![A_2]\!](1) + \ldots + T[\![A_2]\!](2^{\log(n)}) \\
&\approx 1 + \ldots + 2^i + \ldots + 2^{\log(n)} \\
&\approx 2n \\
&\in O(n)
\end{aligned}
$$

hvor vi har brugt $\approx$ til at angive, at vi regner med de tilnærmelser $O(\cdot)$ tillader.

Vi kan præcisere denne brug af parametre i kompleksitetsmålet på følgende måde. Hvis A er en (stump af) en algoritme, betegner vi med

$$T[\![A]\!](n)$$

tidskompleksiteten af A, når algoritmen startes i en tilstand, der er bestemt af den *karakteristiske parameter* n. Mere præcist skal der gælde, at $T[\![A]\!](n)$ skal være den mindste øvre grænse for summen af størrelserne af de værdier, der udregnes under udførelser, der starter i tilstande, som er karakteriseret af n. Vi skal ikke forsøge at formalisere hvad det i almindelighed betyder, at en tilstand er karakteriseret af en parameter, idet dette vil blive præciseret i hvert enkelt tilfælde. I de fleste tilfælde angiver n værdien af en af algoritmens variabler (som p og i ovenfor), og i så fald karakteriserer den enhver tilstand i hvilken den pågældende variabel har værdien n.

Det skal understreges, at kompleksitetsmålet er pessimistisk, idet $T[\![A]\!](n)$ dominerer tidskompleksiteten for *alle* udførsler af A i alle starttilstande, der karakteriseres af n. Dette kompleksitetsmål kaldes også *worst-case kompleksitet*, fordi det giver udførelsestiden for den værst tænkelige beregning. Et alternativt kompleksitetsmål er *gennemsnitlig* tidskompleksitet, hvor

[2]Alle logaritmer i disse noter har, med mindre andet nævnes eksplicit, grundtal 2, dvs. $\log(n) = \log_2(n)$.

man under mere eller mindre realistiske antagelser om fordelingen af data, udregner det forventede antal primitive operationer, der udføres i en beregning. Vi skal dog hovedsageligt interessere os for worst-case kompleksitet.

3.4 Analyse af nogle konkrete algoritmer

Vi anfører nu et antal eksempler på simple analyser af nogle af algoritmerne fra de foregående afsnit.

I algoritmen Logaritmisk potensopløftning er det igen klart, at alle forekommende værdier har størrelse 1, hvorfor vi kan vurdere dens tidskompleksitet ved antallet af iterationer i **do**-løkken. Her er det nemt at se, at variablen q halveres mindst hver anden gang løkken gennemløbes, hvorfor antallet af iterationer ligger mellem $\log(p)$ og $2\log(p)$. Altså gælder der

$$T[\![\text{Logaritmisk potensopløftning}]\!](p) \in \Theta(\log(p))$$

I algoritmen til binær søgning side 14 sker der en halvering af længden af søgeintervallet i hver iteration, hvorfor vi igen har (n er antallet af elementer i søgedomænet)

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

For algoritmen Indsættelsessortering side 21 gælder der (n er længden af w og "søgning" angiver den inderste **do**-løkke).

$$\begin{aligned} T[\![\text{Indsættelsessortering}]\!](n) &\approx \sum_{i=1}^{n} T[\![\text{søgning}]\!](i) \\ &\in \sum_{i=1}^{n} O(i) \\ &= O(n^2) \end{aligned}$$

Vi viser nu, at disse grænser er "skarpe", dvs. at man kan erstatte O med Θ. For indsættelsessortering skal vi altså vise, at

(*) $T[\![\text{Indsættelsessortering}]\!](n) \in \Omega(n^2)$

Iflg. definitionen på tidskompleksitet skal vi finde en starttilstand for algoritmen, som er karakteriseret af n, og hvor der under udførelsen udregnes af størrelsesordenen n^2 (simple) værdier. Da n betegner længder af w, skal vi finde en liste af længde n, for hvilken algoritmens udførelsestid er ca. n^2. Men det er en simpel sag at indse, at indsættelsessortering af den omvendt sorterede liste

$$w = (n, n-1, n-2, \ldots, 2, 1)$$

involverer $\frac{(n-1)*n}{2}$ ombytninger, og derfor er det klart, at (*) er opfyldt.

Det overlades til læseren på tilsvarende måde at vise, at

$$T[\![\text{Udvalgssortering}]\!](n) \in \Omega(n^2)$$

3.5 Analyse af en binær tæller

I dette afsnit vil vi analysere en algoritme, der simulerer optælling i et tælleregister R, der svarer til en *binær* kilometertæller. Vi vil benytte en metode til vurdering af tidskompleksitet, som kan give skarpere analyser end "lige ud ad landevejen" regningerne fra de foregående afsnit. Metoden består i en slags debet-kredit regnskab, og vil i hæftet *Programmeringsteori II* blive formaliseret og generaliseret.

Algoritme: Optælling
Stimulans : $n : n > 0$
Respons : $R : \tilde{R}$ er den binære repræsentation af n
Metode : $i, R := 0$, Vector( )
  do $\{\tilde{R}$ er den binære repræsentation af i og $(0 \leq i \leq n)\}$
    $i < n \rightarrow p := 0$
      do $(p < |R|) \wedge (R.(p) = 1) \rightarrow$
        $R.(p), p := 0, p + 1$
      od
      if $p = |R| \rightarrow R := R + +\text{Vector}(0)$ **fi**
      $R.(p) := 1$

$i := i + 1$

od

Hvis f.eks. n har værdien 14, vil algoritmen standse med følgende værdi for R

$$R = (0, 1, 1, 1)$$

hvilket passer med, at den binære repræsentation af 14 er 1110.

En grov analyse af algoritmen giver, at den yderste **do**-løkke gennemløbes præcis n gange. For hvert gennemløb

- udføres der højst $|R|$ gennemløb af den inderste **do**-løkke
- udføres sætningen $R := R + +\text{Vector}(0)$ eventuelt.

Da $R := R + +\text{Vector}(0)$ er den eneste sætning i algoritmen, hvor der udregnes en værdi hvis størrelse er forskellig fra 1, og da denne størrelse er $|R| + 1$, er det klart at

$$T[\![\text{Optælling}]\!](n) \in O(n \log(n))$$

idet $|R|$ højst kan blive $\log(n)$.

M.h.t. den "dyre" sætning $R := R + +\text{Vector}(0)$ er dette imidlertid alt for pessimistisk, fordi den faktisk udføres ret sjældent. Man kan godt udregne hvor sjældent, men tilbage står stadig at estimere det samlede antal gennemløb af den inderste **do**-løkke. Her anvender vi nu debet-kredit teknikken på følgende måde.

Vi vedtager at det koster 1 krone hver gang vi ser på et element i R, og at det koster $|R|$ kroner, hver gang R udvides (ved udførelse af sætningen $R := R + +\text{Vector}(0)$). Vi kan nu "finansiere" hele algoritmen ved for hver optælling at *betale* 2 kroner samt ved for hver forlængelse af R at *låne* $|R|$ kroner.

De 2 kroner der betales for hver optælling er nemlig tilstrækkelige til at overholde følgende *bogførings-invariant*:

"Hvert element i R med værdi 1 er i besiddelse af 1 krone."

Dette indses ved at observere, at en optælling består i at gennemløbe et antal elementer i R, der alle har værdien 1, indtil man når et 0. 1'erne sættes til 0 og 0'et til 1. Hvis nu R allerede overholder bogføringsinvarianten, kan alle 1'erne *selv* betale prisen for at besøge dem. De 2 kroner kan så bruges til dels at betale for at besøge 0'et, og dels til at give denne (som jo nu bliver en 1'er) den krone, den skal have for at overholde bogføringsinvarianten.

Når algoritmen standser, har de samlede udgifter været

- 2n kroner for optællingerne
- den gæld der er opbygget i forbindelse med forlængelserne af R.

R er imidlertid vokset med 1 element ad gangen, fra længde 0 op til slutværdien $\log(n)$, hvorfor gælden andrager

$$\left(\sum_{i=1}^{\log(n)} i\right) \approx \log(n)^2$$

De samlede udgifter beløber sig da til $\approx\ 2n + \log(n)^2$ kroner, og da antallet af kroner er proportionalt med summen af de udregnede værdier, er tidskompleksiteten givet ved

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

dvs. algoritmen er lineær.

Det skal understreges, at denne bogføringsinvariant er en tænkt invariant, som kun anvendes i forbindelse med analysen, og som ikke direkte har noget at gøre med invarianter der anvendes til korrekthedsargumenter.

3.6 Tidskompleksitet af procedurer

I dette og næste afsnit vil vi se på teknikker til at vurdere tidskompleksiteten af procedurer. Vi starter med ikke-rekursive procedurer.

Hvis P er en procedure, betegner vi med

$$T[\![\textbf{proc}\ P]\!](n)$$

tidskompleksiteten af at udføre P's *krop*[3], startende i en tilstand, der er karakteriseret af n.

Betragt som første eksempel følgende to procedurer til binær søgning (jfr. side 14), hvor det antages at argumentet S er en ordnet liste af længde n.

```
Proc Binær₁(S: Vector, m: Int) → (Int)
    (+ Var lav, høj, midt: Int
       lav, høj := 0, | S |
       do lav<høj →
            midt := (lav+høj)/2
            if S.(midt) = m → return midt
             | S.(midt)<m → lav: raisebox-1pt= midt+1
             | m<S.(midt) → høj: = midt
            fi
       od
       return ?-Int
    +)
end Binær₁

Proc Binær₂[S: Vector, m: Int] → (Int)
    (+ Var lav, høj, midt: Int
       lav, høj := 0, | S |
       do lav<høj →
            midt := (lav+høj)/2
            if S.(midt) = m → return midt
             | S.(midt)<m → lav: raisebox-1pt= midt+1
             | m<S.(midt) → høj: = midt
            fi
       od
       return ?-Int
    +)
end Binær₂
```

[3] Omkostningerne ved parametermanipulation tælles altså ikke med her.

Det følger af analysen af Binær søgning, at

$$T[\![\textbf{proc}\ \text{Binær}_1]\!](n) = T[\![\textbf{proc}\ \text{Binær}_2]\!](n) \in O(\log(n))$$

Hvis vi derimod ser på kompleksiteten af *kaldene*

$$\text{Binær}_1(S, x) \text{ og } \text{Binær}_2[S](x)$$

er der forskel, fordi der gælder

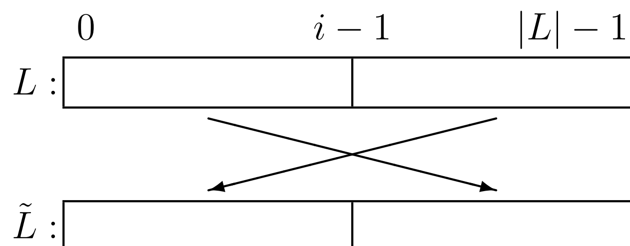
$$\begin{aligned} T[\![\text{Binær}_1(S, x)]\!] &\approx |S| + T[\![\textbf{proc}\ \text{Binær}_1]\!](|S|) \in O(|S|) \\ T[\![\text{Binær}_2[S](x)]\!] &\approx 1 + T[\![\textbf{proc}\ \text{Binær}_2]\!](|S|) \in O(\log(|S|)) \end{aligned}$$

Dette skyldes, at for en *værdiparameter* (S) er det som bekendt selve værdien (af størrelse $|S|$) der skal udregnes, medens det for *referenceparameteren* $[S]$ kun er referencen til den relevante variabel (af størrelse 1), der skal findes.

Vi ser altså et eksempel på, hvordan en uhensigtsmæssig parametermekanisme kan "ødelægge" en effektiv søgemetode.

3.7 Tidskompleksitet af rekursive procedurer

Når procedurer er rekursive, kan vi ikke udregne deres tidskompleksitet helt på samme måde som ovenfor. Betragt som eksempel følgende to procedurer, som spejler en liste L. Ideen i begge procedurer er, at der er følgende sammenhæng mellem L og dens spejlbillede $\tilde{L}$ (i er en vilkårlig værdi med $0 \leq i \leq |L|$).



Man kan spejle L ved at dele den i to, spejle hver del for sig og sætte dem sammen i omvendt rækkefølge – eller udtrykt mere kompakt

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

Forskellen på de to procedurer er valget af i

```
Type liste = List(Char)

Proc Reverse₁ [L: liste]
    (+ Var hlp: liste
       if | L |>1 →
            hlp, L := L(0 .. 1), L(1 .. | L  |)
            Reverse₁ [L]
            L := L++hlp
       fi
    +)
end Reverse₁

Proc Reverse₂ [L: liste]
    (+ Var hlp: liste
       if | L |>1 →
            hlp, L := L(0 .. | L |/2), L  (| L |/2 .. | L |)
            Reverse₂ [hlp]
            Reverse₂ [L]
            L := L++hlp
       fi
    +)
end Reverse₂
```

Da begge procedurer er rekursive, resulterer deres analyse i *rekursionsligninger* på følgende måde. For Reverse_1 gælder der, idet n betegner længden af parameteren L,

$$\begin{aligned}
T[\![\textbf{proc}\ \text{Reverse}_1]\!](1) &\in O(1) \\
T[\![\textbf{proc}\ \text{Reverse}_1]\!](n) &\approx n + T[\![\text{Reverse}_1[L]]\!](n-1) + n \\
&\approx n + T[\![\textbf{proc}\ \text{Reverse}_1]\!](n-1)
\end{aligned}$$

hvilket følger af, at størrelsen af værdierne af $L(1..|L|)$ og L++liste(hlp) begge tilhører $O(n)$.

Det er nemt at se, at løsningen til denne rekursionsligning er

$$\begin{aligned} T[\![\textbf{proc}\ \text{Reverse}_1]\!](n) &\in \sum_{i=1}^{n} O(i) \\ &= O(n^2) \end{aligned}$$

For proceduren Reverse$_2$ gælder der følgende analyse (for $n > 1$)

$$\begin{aligned} T[\![\textbf{proc}\ \text{Reverse}_2]\!](n) &\approx n + \\ & \quad T[\![\text{Reverse}_2[hlp]]\!](\frac{n}{2}) + \\ & \quad T[\![\text{Reverse}_2[L]]\!](\frac{n}{2}) + \\ & \quad n \\ &\approx 2(n + T[\![\textbf{proc}\ \text{Reverse}_2]\!](\frac{n}{2})) \\ &\approx \underbrace{2(n + 2(\frac{n}{2} + 2(\frac{n}{4} + \ldots + 2(1)\ldots)))}_{\log n} \\ &= 2(n + n + \ldots + n) \\ &\in O(n \log(n)) \end{aligned}$$

idet n "højst kan halveres $\log(n)$ gange". Bemærk, at det følger af analysen, at for store værdier af n er Reverse$_2$ væsentlig mere effektiv end Reverse$_1$ (der er stor forskel på n^2 og $n\log(n)$, når n er stor). Det kan altså betale sig at opdele problemet i to problemer, som hver er af halv størrelse, i stedet for at dele det i et stort og et lille.

Lad os for god ordens skyld også analysere proceduren KochLine fra afsnit 12.3 i [TRINE]. Idet den karakteristiske parameter denne gang er ordenen af den kurve der skal tegnes, skulle det være klart, at vi får følgende ligning

$$T[\![\text{KochLine}(n, r, l)]\!] \approx T[\![\textbf{proc}\ \text{KochLine}]\!](n)$$

samt at

$$T[\![\textbf{proc}\ \text{KochLine}]\!](0) \in O(1)$$
$$T[\![\textbf{proc}\ \text{KochLine}]\!](n) \approx 4 * T[\![\textbf{proc}\ \text{KochLine}]\!](n-1)$$

hvilket giver at

$$T[\![\textbf{proc}\ \text{KochLine}]\!](n) \in O(4^n)$$

4 Specifikationer

Når algoritmer opbygges v.hj.a. ubestemte stumper, herunder når der optræder procedurer i algoritmerne, bliver der behov for at få *egenskaber* ved sådanne ubestemte stumper ind i korrekthedsbeviserne.

Betragt som eksempel følgende situation, hvor K er en mængde af elementer af en eller anden type E.

Algoritme: Permutation

Stimulans: f: $\mathbf{List}(E)$, $\forall i, j \in 0..|f| : f.(i) \neq f.(j)$

Respons : f: permutation af listen

Metode : $K, i := \emptyset, 0$

$\quad$ **do** $i < |f| \rightarrow e, i := f.(i), i + 1$

$\qquad\qquad \ll$indsæt e i $K \gg$

$\quad$ **od**

$\quad i := 0$

$\quad$ **do** $i < |f| \rightarrow \ll$udvælg e fra $K \gg$

$\qquad\qquad \ll$fjern e fra $K \gg$

$\qquad\qquad f.(i), i := e, i + 1$

$\quad$ **od**

Hvis vi skal vise, at denne algoritme er korrekt, er det klart, at vi skal bruge stumpernes egenskaber. Disse udtrykker vi nu v.hj.a. *specifikationer* på følgende måde

(*) $\ll$indsæt e i $K \gg$ **sat** $K' = K \cup \{e\}$

sat er en forkortelse for **satisfies** og angiver, at stumpen $\ll$indsæt e i $K \gg$ *opfylder* udsagnet $K' = K \cup \{e\}$. Dette udsagn adskiller sig fra de udsagn vi hidtil har beskæftiget os med ved at indeholde såvel mærkede (K') som umærkede (K, e) variabelnavne. Meningen er den samme som i [TRINE, kap. 8], at de umærkede (mærkede) navne refererer til variablernes værdier *før* (*efter*) udførelsen af stumpen. (*) siger således, at værdien af variablen K efter udførelsen er lig med foreningsmængden af $\{e\}$ og værdien af K før kaldet.

Specifikationen af $\ll$fjern e fra $K \gg$ er analog, hvorimod $\ll$udvælg e fra $K \gg$ har udseendet

$\ll$udvælg e fra $K \gg$ **sat** $K \neq \emptyset \rightarrow (e' \in K) \wedge (K' = K)$

Her har vi tilføjet en *betingelse* ($K \neq \emptyset$), som angiver under hvilke omstændigheder det giver mening at udføre stumpen.

Vi angiver nu den præcise betydning af en specifikation. En sådan er i almindelighed af formen

(**) $\ll$Stump$\gg$ **sat** $P \rightarrow U$

hvor der gælder, at P kun indeholder umærkede variabelnavne og U (kan) indeholde såvel umærkede som mærkede navne. Vi kalder U et *blandet udsagn*.

I afsnit 2.1 blev det defineret, at en tilstand σ opfylder et prædikat U, hvis U (som jo er en logisk funktion) udregnet i tilstanden σ, har værdien $\underline{\text{t}}$. I analogi hermed definerer vi, at et par af tilstande (σ, σ') opfylder et blandet prædikat U, hvis U har værdien $\underline{\text{t}}$, når det udregnes i tilstandene σ og σ', hvilket mere præcist betyder, at umærkede (mærkede) variabelnavne refererer til σ (σ').

Den formelle betydning af (**) er nu som følger

For enhver tilstand σ som opfylder P gælder, at enhver udførelse af $\ll$Stump$\gg$ med σ som starttilstand terminerer, og for enhver sluttilstand σ' gælder, at (σ, σ') opfylder U.

Med denne definition skulledet være klart, at

$\ll$Stump$\gg$ **sat** $P \rightarrow U$

mekanismen ligger meget tæt på den hidtidige algoritmespecifikationsmekanisme. Ved at betragte definitionen på korrekthed af en algoritme side 9, er det nemt at se, at en algoritme af formen

<u>Algoritme: Noget</u>

Stimulans	: P
Respons	: U
Metode	: $\ll$Noget$\gg$

er korrekt hvis og kun hvis der gælder

$\ll$Noget$\gg$ **sat** $P \to U'$

hvor U' er identisk med U bortset fra, at alle variabelnavne er blevet mærket.

Specifikationernes teknik til at udtale sig om sammenhængen mellem værdierne af variabler før og efter udførelsen af algoritmestumper muliggør opstillingen af følgende *bevisprincip*, der giver grundlaget for såvel konstruktion som korrekthedsbeviser for den type algoritmer, vi har beskæftiget os med i de foregående to kapitler. Her er $\bar{x} = x_1, \ldots, x_n$ programmets variabler, I er invarianten og μ termineringsfunktionen.

Bevisprincip for do-sætninger

Hvis der for sætningen

do $B \to\ll$ Krop $\gg$ **od**

gælder

$\ll$ Krop $\gg$ **sat** $(B(\bar{x}) \wedge I(\bar{x})) \to I(\bar{x}') \wedge (\mu(\bar{x}) > \mu(\bar{x}') \geq 0)$

så gælder der også

do $B \to\ll$ Krop $\gg$ **od sat** $I(x) \to I(\bar{x}') \wedge \neg B(\bar{x}')$

Præcis ligesom vi udtrykker ubestemte stumpers egenskaber ved specifikationer, kan vi også udtrykke egenskaber ved procedurer. Dette beskrives nærmere i **Programmeringsteori II**.

5 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.