Martin Dietzfelbinger, Kurt Mehlhorn,
Peter Sanders

Algoritmer og datastrukturer

Grundværktøjerne (i uddrag, kladde)

27. februar 2021

Springer

[0] Udgave 6bb96eb, Sat Feb 27 16:29:46 2021 +0100, Thore Husfeldt.

Indhold

1

Forret: Plus og gange

Forrettens funktion er at vække appetit for resten af menuen. Det er netop formålet med dette kapitel: For at vække læserens interesse for algoritmiske[1] *teknikker præsenterer vi et overraskende resultat: At folkeskolemetoden for at gange to naturlige tal sammen er ikke den bedste multiplikationsalgoritme. Der findes nemlig hurtigere metoder, hvis effektivitet gør sig gældende, når tallene bliver meget store, dvs. består af tusinder af cifre. Vi skal betragte sådan en metode i dette kapitel.*

Aritmetik på »lange« (dvs. mangecifrede) heltal bruges i kryptografi, geometriske beregninger og computeralgebra. En forbedret multiplikationsalgoritme er derfor ikke bare en intellektuelt højdepunkt, men finder også praktisk anvendelse. Undervejs til denne algoritme skal vi – i denne meget simple kontekst – stifte bekendskab med nogle grundlæggende teknikker fra algoritmekonstruktion, -analyse og -implementation.

I det følgende går vi ud fra, at naturlige tal[2] som sædvanligt er repræsenteret som følger af cifre. I et talsystem med grundtal B, hvor B er et naturligt tal større end 1, findes cifrene 0, 1, ..., $B-1$, og en følge $a_{n-1}a_{n-2}\ldots a_1a_0$ af cifre repræsenterer tallet $\sum_{0\leq i<n} a_i B^i$. De vigtigste systemer med et lille gundtal B er det binære talsystem (grundtal 2, cifrene 0 og 1), decimalsystemet (grundtal 10, cifrene 0 til 9) og det heksadecimale talsystem (grundtal 16, cifrene 0 til 15, ofte skrevet som 0, ... , 9 , A, B, C, D, E, F. Større grundtal som fx 2^8, 2^{16}, 2^{32} og 2^{64} forekommer også. For eksempel repræsenterer

[1] Frimærket fra det tidligere Sovjetunionen viser den persisike matematiker og astronom *Muhammad ibn Musa al-Chwarizmi* (* omtr. 780; † mellem 835 og 850) fra provinsen Khorasan i området af det nuværende Usbekistan. Ordet »algoritme« kommer fra hans navn.

[2] I dette kapitel betyder »tal« altid »naturligt tal«.

$$\begin{aligned} \text{»10101« i basis } 2 \quad &\text{værdien } 1\cdot 2^4+0\cdot 2^3+1\cdot 2^2+0\cdot 2^1+1\cdot 2^0 &&= 21\,,\\ \text{»924« i basis } 10 \quad &\text{værdien } 9\cdot 10^2+2\cdot 10^1+4\cdot 10^0 &&= 924\,. \end{aligned}$$

Vi vil gå ud fra, at vi har to elementaroperationer til rådighed: addition af tre cifre med et tocifret resultat (sommetider kaldt en heladderer) og multiplikationen af to cifre, ligeledes med et tocifret resultat.[3] I tilfældet $B = 10$ gælder for eksempel

$$\begin{array}{r} 3\\ 5\\ +\ 5\\ \hline 13 \end{array} \qquad \text{og} \qquad 6\cdot 7 = 42\,.$$

Vi vil måle effektiviten af vores algoritmer som antallet af brugte elementaroperationer.

Hvert tal bestående af n cifre kan man skrive med m cifre for $m \geq n$ ved at foranstille nuller. For eksempel repræsenterer »425« og »000425« samme tal. I det følgende går vi ud fra, at additionens eller multiplikationens operander, som vi kalder a og b, begge består af n cifre. Antagelsen, at de to tal har samme længde, gør vores analyse nemmere, uden at vi går glip af kapitlets væsentlige indsigter. Vi vender tilbage til denne bemærkning til sidst. Cifrene i a kalder vi $a_{n-1}, \ldots, a_0$, hvor a_{n-1} er det mestbetydende og a_0 er det mindstbetydende ciffer. Vi skriver $a = (a_{n-1} \ldots a_0)$, hvor det mestbetydende ciffer kan være 0. På samme måde lader vi tallet b bestå af cifrene $b_{n-1}, \ldots, b_0$ og skriver $b = (b_{n-1} \ldots b_0)$.

1.1 Addition

Vi ved alle, hvordan man lægger $a = (a_{n-1} \ldots a_0)$ og $b = (b_{n-1} \ldots b_0)$ sammen. Vi skriver blot tallene under hinanden, forskudt efter de mindstbetydende cifre, og lægger de enkelte cifre sammen fra højre til venstre. Herved flyttes et enkelt ciffer, kaldt *menten*, fra en position til den næste. Resultatet er et $(n+1)$-cifret tal $s = (s_n \ldots s_0)$. Metoden kan fremstilles grafisk på følgende måde:

$$\begin{array}{rr} a_{n-1} \ldots a_1\ a_0 & \text{første operand}\\ b_{n-1} \ldots b_1\ b_0 & \text{anden operand}\\ c_n\ c_{n-1} \ldots c_1\ \ 0 & \text{menter}\\ \hline s_n\ s_{n-1} \ldots s_1\ s_0 & \text{sum} \end{array}$$

Her betegner $c_0, \ldots, c_n$ følgen af menter, og $s = (s_n \ldots s_0)$ er den resulterende sum. Der gælder $c_0 = 0$, $c_{i+1} \cdot B + s_i = a_i + b_i + c_i$ for $0 \leq i < n$ og $s_n = c_n$. Som program kan vi skrive algoritmen på følgende måde:

[3] Læg mærke til, at summen af tre cifre højst er $3(B-1)$ og produktet af to cifre højst er $(B-1)^2$, og at begge disse tal er højst $(B-1)\cdot B^1 + (B-1)\cdot B^0 = B^2 - 1$, og derfor kan repræsenteres med to cifre.

$c = 0 : \mathit{Ciffer}$ *// Variabel for menten*
for $i{:=}0$ **til** $n-1$ **udfør**
  læg a_i, b_i, c sammen, resultatet er s_i og den nye mente c
$s_n{:=}c$

Algoritmen bruger en elementaroperation per position og derfor n elementaroperationer i det hele.

Sætning 1.1. *Addition af to heltal med n cifre kræver præcis n elementaroperationer. Resultatet er et $(n+1)$-cifret heltal.*

1.2 Multiplikation: Skolemetoden

Vi ved også alle, hvordan man ganger to tal sammen. I dette afsnit skal vi gense »skolemetoden« for multiplikation. Derefter vil vi kigge på en anden metode, som er betydligt hurtigere for store tal.

Vi begynder langsomt. Først skal vi forstå, hvordan man ganger et n-cifret tal a med et etcifret tal b_j. Vi bruger betegnelsen b_j, fordi det bliver praktisk længere fremme i argumentet. For hvert ciffer a_i i tallet a udregner vi produktet $a_i \cdot b_j$. Resultatet er et tocifret tal $(c_i d_i)$, dvs.

$$a_i \cdot b_j = c_i \cdot B + d_i \qquad \text{for } i \in \{0, \ldots, n-1\}\,.$$

Ud fra alle cifrene c_i og d_i danner vi to tal $c = (c_{n-1} \ldots c_0\, 0)$ og $d = (d_{n-1} \ldots d_0)$. Idet c_i-cifrene er mestbetydende, hægter vi et ekstra 0 på c. Nu lægger vi c og d sammen og får produktet $p_j = a \cdot b_j$. Grafisk ser det ud på følgende måde:

$$(a_{n-1} \ldots a_i \ldots a_0) \cdot b_j \quad \longrightarrow \quad \begin{array}{r} c_{n-1}\ c_{n-2}\ \ldots\ c_i \quad\ c_{i-1} \ldots c_0\ 0 \\ \underline{\quad d_{n-1} \ldots d_{i+1}\ d_i \quad \ldots d_1\ d_0} \\ \text{Summen af } c \text{ og } d \end{array}$$

Hvor mange elementaroperationer har vi brugt? For hvert $i \in \{0, \ldots, n-1\}$ kræves en elementaroperation for at danne produktet $a_i \cdot b_j$. For at lægge $(c_{n-1} \ldots c_0\, 0)$ til $(d_{n-1} \ldots d_0)$ kan vi nøjes med at kopiere d_0 uden at betale nogen operation, de andre position kræver en elementaraddition. Totalt kræver det $2n$ elementaroperationer. Læg mærke til, at menten i den sidste elementaraddition er 0, fordi $a \cdot b_j \leq (B^n - 1)(B-1) < B^{n+1}$. Resultatet består altså af $n+1$ cifre.

Lemma 1.1. *Multiplikation af et n-cifret og et etcifret tal kan gøres med $2n$ elementaroperationer. Resultatet er et $(n+1)$-cifret tal.*

I praksis vil man nok være lidt mere snedig. Man sammenfatter nemlig udregningen af cifferproduktet $a_i \cdot b_j$ og additionen af c og d i samme fase,[4]

[4] I oversætterkonstruktion kaldes denne transformation, som hører til kodegenerering og kodeoptimering, for *løkkefusion*.

dvs. at de enelte cifre i c og d skabes i samme øjeblik, som de skal bruges i den afsluttende addition. Vores algoritme danner i stedet samtlige cifferprodukter $a_i \cdot b_j$ for $i \in \{0, \ldots, n-1\}$ i en fase for sig. Det tjener kun til at gøre beskrivelsen mere letforståelig.

Opgave 1.1. Skriv et program, der ganger a og b_j i samme fase.

Nu er vi parate til at gange to n-cifrede tal. Skolemetoden hertil virker som følger: Vi danner delprodukterne p_j (med repræsentationen $(p_{j,n} \ldots p_{j,0})$), ved at gange a med det $j$te ciffer b_j i b, og adderer derefter produkterne $p_j \cdot B^j$, passende forskudt under hinanden, for at danne produktet af a og b. Grafiskt ser det sådan ud:

$$\begin{array}{cccccccc}
 & & p_{0,n} & p_{0,n-1} & \cdots & p_{0,2} & p_{0,1} & p_{0,0} \\
 & p_{1,n} & p_{1,n-1} & p_{1,n-2} & \cdots & p_{1,1} & p_{1,0} & \\
p_{2,n} & p_{2,n-1} & p_{2,n-2} & p_{2,n-3} & \cdots & p_{2,0} & & \\
 & & \cdots & & & & & \\
\underline{p_{n-1,n} \cdots p_{n-1,3}} & & \underline{p_{n-1,2}} & \underline{p_{n-1,1}} & & \underline{p_{n-1,0}} & &
\end{array}$$

Summen af de n delprodukter

Beskrivelsen i pseudokode er vældig kompakt. Vi initialiserer produktet p til 0 og lægger delprodukterne $a \cdot b_j \cdot B^j$ til p et efter et for $j \in \{0, \ldots, n-1\}$:

$p = 0 : \mathbf{N}$
for j:=0 **til** $n-1$ **udfør** p:=$p + a \cdot b_j \cdot B^j$

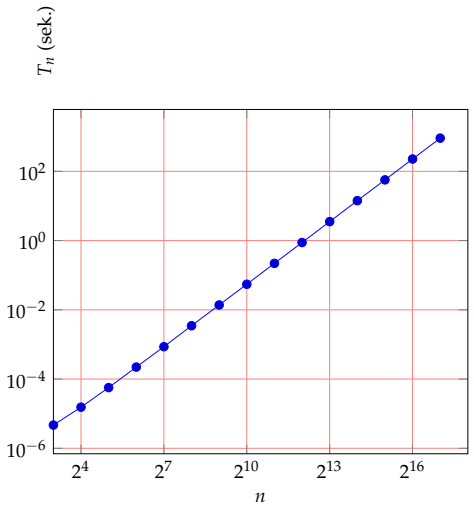
Nu skal vi bestemme, hvor mange elementaroperationer skolemetoden bruger. For hvert delprodukt p_j bruges $2n$ elementaroperationer, og derfor kræver beregningen af samtlige delprodukter $2n^2$ elementaroperationer i det hele. Produktet $a \cdot b$ består af $2n$ cifre, og derfor har de tal, der indgår i en addition på formen $p + a \cdot b_j \cdot B^j$, højst $2n$ cifre. Hver af disse $n-1$ additioner koster højst $2n$ elementaroperationer, sammenlagt højst $2n^2 - 2n$ mange. I det hele bruges altså højst $4n^2$ elementaroperationer.

En enkel observation tillader os at forbedre denne øvre grænse. Tallet $a \cdot b_j \cdot B^j$ består af $n+1+j$ cifre, hvoraf de sidste j har værdien 0. Vi kan altså begynde med additionen på position $(j+1)$. Desuden gælder der i det øjeblik, som $a \cdot b_j \cdot B^j$ lægges til p, at p har værdien $a \cdot (b_{j-1} \ldots b_0)$, som består af højst $n+j$ cifre. Derfor går additionen af p og $a \cdot b_j \cdot B^j$ ud på att lægge to $(n+1)$-cifrede tal sammen til et $(n+1)$-cifret resultat, hvilket kun behøver $n+1$ elementaroperationer. Sammenlagt kræver alle $n-1$ additioner ikke mere end $(n-1)(n+1) < n^2$ elementaroperationer. Vi får følgende resultat:

Sætning 1.2. *Skolemetoden udregner produktet af to n-cifrede tal ved brug af højst* $3n^2$ *elementaroperationer.*

Vi har analyseret antallet af elementaroperationer for addition of multiplikation med skolemetoden. Lad M_n betegne antallet af elementaroperationer,

n	T_n (sek.)	$T_n / T_{n/2}$
8	0,000 004 69	
16	0,000 015 4	3,29
32	0,000 056 7	3,68
64	0,000 222	3,91
128	0,000 860	3,88
256	0,003 47	4,04
512	0,0138	3,98
1024	0,0547	3,96
2048	0,220	4,02
4096	0,880	4
8192	3,53	4,01
16 384	14,2	4,01
32 768	56,7	4,00
65 536	227	4,01
131 072	910	4,00



Figur 1.1. Udførelsestiden for skolemetoden til multiplikation af to n-cifrede tal. De tre søjler i tabellen til *venstre* angiver n, udførelsestiden T_n af C++-implementationen fra afsnit 1.7 og forholdet $T_n / T_{n/2}$. Grafen til *højre* tegner $\log T_n$ som funktion af $\log n$; stort set som en ret linje. Læg mærke til, at når $T_n = \alpha n^\beta$ for konstante faktorer α og β, så gælder $T_n / T_{n/2} = 2^\beta$ og $\log T_n = \beta \log n + \log \alpha$. Med andre ord er $\log T_n$ en lineær funktion af $\log n$ med hældning β. I vores tilfælde er hældningen 2, hvilket man kan efterprøve med en lineal.

som skolemetoden bruger til multiplikation af to n-cifrede tal. Vi har vist, at M_n er omtrent $3n^2$. Vi vil sige at »M_n vokser *kvadratisk*«. Læg mærke til, at

$$\frac{M_n}{M_{n/2}} = \frac{3n^2}{3(n/2)^2} = 4\,.$$

Kvadratisk vækst betyder altså, at antallet af elementaroperationer i alt væsentligt bliver 4 gange større, hver gang størrelsen af probleminstansen fordobles.

Antag nu, at vi implementerer multiplikationsalgoritmen i vores yndlingsprogrammeringssprog (det gør vi faktisk forneden) og måler den tid, det tager på vores yndlingscomputer at gange par n-cifrede tal a og b for forskellige n. Hvilke udførelsestider kan vi regne med? Vi påstår, at eksperimentet vil vise kvadratisk vækst i n. Grunden hertil er, at *elementaroperationerne er repræsentative for programmets udførelsestid*. Betragt hertil først additionen af to n-cifrede tal. Hvad sker der under programmets udførelse? For hver cifferposition i flyttes cifrene a_i og b_i fra hovedlageret til processoren, summen $a_i + b_i + c$ bliver regnet ud, sumcifret s_i kopieres over i hovedlageret, menten c aktualiseres, index i tælles op og løkkeafslutningsbetingelsen prøves. I vores analyse har vi talt præcis 1 elementaroperation for hvert i, derfor er antallet af elementaroperationer repræsentativt for antallet af udførte maskincykler. På en moderne

processor opstår desuden særlige fænomener som pipelining af operationer og den meget vanskelige mekanisme for datastransport mellem hovedlageret og processoren, men disse påvirker alle cifferpositioner nogenlunde lige. Derfor er antallet af elementaroperationer også repræsentativt for udførelsestiden af en faktisk implementation på en virkelig computer. Samme overvejelser kan gennemføres for multiplikationsalgoritmen, idet etcifret multiplikation ligner etcifret addition, og skolemetodens anden fase går ud på en følge af additioner.

Vi vil nu udsætte disse overvejelser for et eksperiment. I figur 1.1 vises udførelsestiderne for en C++-implementation af skolemetoden; programmet findes i afsnit 1.7. For hvert n har vi gennemført vældigt mange [5] multiplikationer af tilfældige n-cifrede tal for at bestemme den gennemsnitlige udførelsestid T_n. Denne finder man i tabellens anden søjle. Figur 1.1 viser desuden datapunkterne grafisk.[6] Målepunkterne antyder kvadratisk vækst, hvilket man kan aflæse på forskellige måder. Forholdet $T_n / T_{n/2}$ er altid tæt på 4, og tegningen i dobbeltlogaritmisk skala viser stort set et ret linje med hældning 2. Eksperimentet er altså ganske lovende: *Vores teoretiske analyse tillader forudsigelser af algoritmens faktiske opførsel. Analysen kom frem til, at antallet af elementaroperationer vokser kvadratisk. Vi har argumenteret for, at udførelsestiden burde ligge tæt op ad antallet af elementaroperationer, og den målte udførelsestid vokser vitterligt kvadratisk.* Alligevel observerer vi også systematiske afvigelser. For små n vokser tiden fra en række i tabellen til den næste med mindre end en faktor 4. Det hænger sammenn med, at lineære og konstante termer stadig spiller en betragtelig rolle for udførelsestiden. For større n er forholdet meget tæt på 4. For meget store n (så store, at det bliver vanskeligt at bestemme udførelsestiden eksperimentelt) ville forholdet nok blive større end 4, fordi lageradgangstiden afhænger af størrelsen af datamængden. Vi vender tilbage til dette punkt i afsnit 2.2.

Opgave 1.2. Skriv programmer for addition og multiplikation af lange tal. Antag, at tallene er givet som følger (rækker eller lister eller hvad dit programmeringssprog nu måtte bruge af datastrukturer) af decimalcifre, og anvend den forhåndenværende aritmetik for at implementere elementaroperationerne. Skriv så funktioner `plus`, `gange1` og `gange` til henholdsvis addition af to tal, multiplikation af et tal med et etcifret tal og multiplikation af n-cifrede tal. Lav din egen udgave af figur 1.1. Prøv også at bruge et større grundtal end 10, fx 2^{16}.

Opgave 1.3. Beskriv og analyser skolemetoden for division af to naturlige tal.

---

[5] Det ur, som bruges til at måle cpu-tiden, leverer sit svar i en vis enhed, fx millisekunder, og den nødvendige afrundingsfejl medfører en fejl på op til halvdelen af enheden. Eksperimentet skal derfor vare meget længere en denne enhed for begrænse effekten af denne afrundingsfejl.

[6] I denne bog angiver $\log x$ logaritmen af x med grundtal 2, altså $\log_2 x$.

1.3 Resultatkontrol

Vores additions- og multipækationsalgoritmer er temmelig simple, så vi kan trygt gå ud fra at være i stand til at implementere dem korrekt i det valgte programmeringssprog. Alligevel er konstruktionen af programmer kendt for at være behæftet med fejl,[7] så vi bør altid interessere os for metoder for at efterprøve resultatet af en beregning. For multiplikation lærte forfatterne i grundskolen følgende metode, som kaldes »ni-prøve« (på engelsk »*casting out nines*«, på fransk »*preuve par neuf*«.)

Find tværsummen af a, dvs. summen af de enkelte cifre. Hvis tværsummen har mere end et ciffer, tag tværsummen af tværsummen. Gentag, indtil der kun er et ciffer tilbage, kaldt *prøvesummen* af a og betegnet s_a. Her er et eksempel:

$$4528 \to 19 \to 10 \to 1\,.$$

Gør ligeså med b og c for at skabe prøvesummerne s_b og s_c. Beregn nu produktet $s_a \cdot s_b$ og dets prøvesum s. Hvis s og s_c er forskellige, så er c ikke produktet $a \cdot b$. Denne prøve blev beskrevet af al-Chwarizmi i sin algebrabog.

Lad os gennemføre niprøven på et enkelt eksempel. Givet $a = 429$, $b = 357$ og $c = 154153$. Da gælder $s_a = 6$, $s_b = 6$ og $s_c = 1$. Endivedere er $s_a \cdot s_b = 36$ og derfor $s = 9$. Derfor har vi $s_c \neq s$, så c kan ikke være produktet af a og b, som nemlig er $c' = 153153$. Prøvesummen af c' er 9, så det består niprøven. Prøven er finder nogle, men selvfølgeligt ikke alle fejl, fx bestås den af 135153.

For at forstå den matematiske baggrund, betragter vi en mere generel metode. Betragt et positivt heltal q (niprøven er tilfældet $q = 9$). Lad s_a være resten ved heltalsdivision af a med q a durch q, dvs. $s_a = a - \lfloor a/q \rfloor \cdot q$. Da gælder $0 \leq s_a < q$. I matematisk notation skriver man dette som $s_a = a \bmod q$.[8] På tilsvarende måde gælder $s_b = b \bmod q$ og $s_c = c \bmod q$. Endelig er $s = (s_a \cdot s_b) \bmod q$. Hvis der gælder $c = a \cdot b$, så må der også gælde $s = s_c$. Derfor medfører $s \neq s_c$, at $c \neq a \cdot b$, hvilket afslører en fejl ved multiplikationen (eller niprøven). Hvad ved vi, når $s = s_c$ gælder? I så fald må q dele differencen mellem c og $a \cdot b$. Hvis denne difference ikke er 0, opdages fejlen af hvert q, som ikke deler differencen.

I vores eksempel forsøger vi nu med $q = 7$. Da gælder $a \bmod 7 = 2$ og $b \bmod 7 = 0$, og derfor $s = (2 \cdot 0) \bmod 7 = 0$. Men $135153 \bmod 7 = 4$, og vi har opdatet, at 135153 og $429 \cdot 357$ er forskellige.

Opgave 1.4. Forklar, hvorfor den niprøve, forfatterne lærte i skolen, svarer til $q = 9$. *Vink*: $10^k \bmod 9 = 1$ for alle $k \geq 0$.

[7] Fejlen i divisionsalgoritmen i Pentium-processorens divisionsalgoritme for kommatal var notorisk. Den var resultatet af en enkelt fejlagtig indgang i en tabel i algoritmen.

[8] Den metode, man måske har lært i skolen, bruger rester i området $1, \ldots, 9$ i stedet for $0, \ldots, 8$; svarende til formlen $s_a = a - (\lceil a/q \rceil - 1) \cdot q$.

Opgave 1.5 (*Elleveprøve*)**.** Det er nemt at finde resten modulo 11 af tipotenser: Der gælder nemlig $10^k \bmod 11 = (-1)^k$ for alle $k \geq 0$, dvs. $1 \bmod 11 = 1$, $10 \bmod 11 = -1$, $100 \bmod 11 = +1$, $1000 \bmod 11 = -1$ osv.

1.4 En rekursiv udgave af skolemetoden

Vi vil nu udvikle en rekursiv udgave af skolemetoden for multiplikation. Det er vores første møde med det algoritmiske princip »*del og hersk*«, et af de grundlæggende paradigmer for algoritmekonstruktion.

Vi betragter igen to n-cifrede tal a og b, som vi ønsker at multiplicere. Nu bruger vi følgende fremgangsmåde: Lad $k = \lfloor n/2 \rfloor$. Opdel a i to tal a_{H} og a_{L}, således at a_{L} består af de k mindstbetydende cifre i a, og a_{H} består af de $n-k$ mestbetydende cifre. På samme måde opdeles b i b_{H} og b_{L}. Vi har altså

$$a = a_{\mathrm{H}} \cdot B^k + a_{\mathrm{L}} \quad \text{og} \quad b = b_{\mathrm{H}} \cdot B^k + b_{\mathrm{L}}\,,$$

og derfor

$$a \cdot b = a_{\mathrm{H}} \cdot b_{\mathrm{H}} \cdot B^{2k} + (a_{\mathrm{H}} \cdot b_{\mathrm{L}} + a_{\mathrm{L}} \cdot b_{\mathrm{H}}) \cdot B^k + a_{\mathrm{L}} \cdot b_{\mathrm{L}}\,.$$

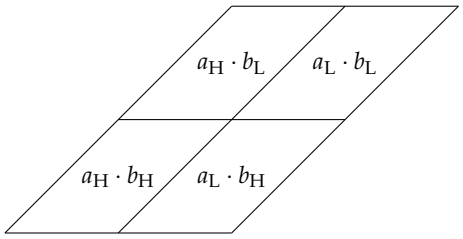
Denne formel giver anledning til følgende algoritme for at bestemme $a \cdot b$:

(a) Opdel a og b i a_{H}, a_{L}, b_{H} og b_{L}.
(b) Beregn de fire produkter $a_{\mathrm{H}} \cdot b_{\mathrm{H}}$, $a_{\mathrm{H}} \cdot b_{\mathrm{L}}$, $a_{\mathrm{L}} \cdot b_{\mathrm{H}}$ og $a_{\mathrm{L}} \cdot b_{\mathrm{L}}$.
(c) Læg produkterne sammen, efter passende justering, for at danne $a \cdot b$.

Læg mærke til, at tallene a_{H}, a_{L}, b_{H} og b_{L} hver består af kun $\lceil n/2 \rceil$ cifre, og multiplikationen i skridt (b) derfor er enklere end den oprindelige n-cifrede multiplikation så snart $\lceil n/2 \rceil < n$ eller ækvivalent $n > 1$. Vi er parate til at beskrive den fuldstængide algoritme. For at multiplicere etcifrede tal sammen, benytter vi elementaroperationerne for multiplikation. For at multiplicere tal bestående af $n \geq 2$ cifre, bruger vi metoden foroven med skridtene (a) til (c).

Metoden hedder »del og hersk«, fordi den reducerer problemet at gange a og b til nogle *enklere* delopgaver af samme slags. En del og hersk-algoritme består altid af tre skridt: I første skridt opdeles den oprindelige instans i mindre delinstanser (vores skridt (a)); i næste skridt løses problemet på de mindre instanser med den samme metode, det vil sige *rekursivt* (vores skridt (b)); i tredje skridt konstrueres løsningen til den oprindelge instans ud fra løsningerne til delinstanserne (vores skridt (c)).

Hvordan hænger den rekursive multiplikationsalgoritme og skolemetoden sammen? Metoderne er nært beslægte i den forstand, at de begge beregner samtlige n^2 cifferprodukter $a_i \cdot b_j$, hvilket resulterer i $2n^2$ cifre, som (forskudt efter deres cifferposition) lægges sammen for at danne produktet $a \cdot b$. Forskellen mellem metoderne ligger i rækkefølgen, hvori disse summering foregår. Figur 1.2 viser skematisk cifferprodukterne og deres position som parallellogram. I skolemetoden lægges cifferprodukterne samme linje for linje, mens



Figur 1.2. Illustration af den rekursive udgave af skolemetoden for multiplikation. Hele parallellogrammet repræsenterer de n^2 ciffervise delprodukter, som optræder ved multiplikationen af a og b. De fire mindre parallellogrammer svarer til produkterne $a_H \cdot b_H$, $a_H \cdot b_L$, $a_L \cdot b_H$ og $a_L \cdot b_L$. I den rekursive udgave beregnes først cifferprodukterne i de fire delområder og siden, i et separat skridt, de fire resulterende summer.

den rekursive metode løser de fire delproblemer i hver af de fire dele, og lægges dem sammen i tre afsluttene additioner. Denne betragtning gør det desuden åbenbart, at antallet af elemantaroperationer er kvadratisk også i den rekursive udgave. Alligevel vil vi også udlede udførelsestiden direkte ud fra en analyse af den rekursive algoritme. I den sammenhæng vil vi støde på en såkaldt *rekursionsligning*, et slags udtryk, som ofte optræder ved analysen rekursive algoritmer.

Lemma 1.2. *Lad $T(n)$ være det maskimal antal elementaroperationer, som skal bruges til den rekursive multiplikationsalgoritme for to n-cifrede tal. Da gælder:*

$$T(n) \leq \begin{cases} 1\,, & \text{når } n = 1\,; \\ 4 \cdot T(\lceil n/2 \rceil) + 2 \cdot 2 \cdot n\,, & \text{når } n \geq 2\,. \end{cases}$$

Bevis. Multiplikationen af to etcifrede tal kræver en elementaroperation. Dermed gælder påstanden for $n = 1$. Antag $n \geq 2$. Opdelingen af tallene a og b i de fire dele a_H, a_L, b_H kræver b_L ingen elementaroperationer. [9] Hvert af de fire tal har højst $\lceil n/2 \rceil$ cifre, så de fire rekursive multiplikationer bruger sammenlagt ikke mere end $4 \cdot T(\lceil n/2 \rceil)$ elementaroperationer. At lægge $a_L \cdot b_L$ og $a_H \cdot b_H \cdot B^{2k}$ sammen koster ingen elementaroperationer, idet $a_L \cdot b_L$ kun består af $2k$ cifre og $a_H \cdot b_H \cdot B^{2k}$ ender med $2k$ nuller. Afslutningsvist bruges to additioner til at beregne det endelige resultat. Hver af disse additioner behandler to tal med højst $2n$ cifre og bruger derfor højst $2n$ elementaroperationer. Derfor gælder også uligheden for $n \geq 2$. □

I afsnnit 2.6 skal vi se, hvordan man løser rekursionsligningen i lemma 1.2 for at etablere den formodede kvadratiske udførelsestid for den rekursive algoritme.

Lemma 1.3. *Lad $T(n)$ være det maskimal antal elementaroperationer, som skal bruges til den rekursive multiplikationsalgoritme for to n-cifrede tal. Da gælder $T(n) \leq 5n^2$, når n er en topotens, og $T(n) \leq 20n^2$ for vilkårlige naturlige tal n.*

[9] Selvfølgeligt kræver dette skridt beregningstid, men denne tages altså ikke hensyn til i analysen.

Bevis. Se afsnit 1.8. □

1.5 Karatsubamultiplikation

År 1962 opdagede denn sovjetiske matematiker A. Karatsuba [30] en hurtigere metode til multiplikation af store naturlige tal. Hans algoritmes udførelsestid vokser omtrent som $n^{\log 3} \approx n^{1{,}585}$. Metoden er overraskende nem. Karatsuba lagde mærke til, at man kan spare en af multiplikationerne i del og herskudgaven væk ved hjælp af en enkel enkel omskrivning. Resultatet er, at man kan gange to n-cifrede tal sammen ved at gange blot *tre* tal af den halve længde.

Vi vil nu beskrive hans metode i detaljer. Lad $k = \lfloor n/2 \rfloor$. Som før kalder vi multiplikationens to operander for a og b og deler a i to tal a_{H} og a_{L}, hvor a_{L} er de k mindstbetydende cifre i a, og a_{H} er de resterene $n-k$ mestbetydende cifre. På samme måde opdeles b i b_{L} og b_{H}. Vi har altså

$$a = a_{\mathrm{H}} \cdot B^k + a_{\mathrm{L}} \quad \text{og} \quad b = b_{\mathrm{H}} \cdot B^k + b_{\mathrm{L}}\,,$$

og derfor (trikket er i den anden ligning!)

$$\begin{aligned} a \cdot b &= a_{\mathrm{H}} \cdot b_{\mathrm{H}} \cdot B^{2k} + (a_{\mathrm{H}} \cdot b_{\mathrm{L}} + a_{\mathrm{L}} \cdot b_{\mathrm{H}}) \cdot B^k + a_{\mathrm{L}} \cdot b_{\mathrm{L}} = \\ &a_{\mathrm{H}} \cdot b_{\mathrm{H}} \cdot B^{2k} + ((a_{\mathrm{H}} + a_{\mathrm{L}}) \cdot (b_{\mathrm{H}} + b_{\mathrm{L}}) - (a_{\mathrm{H}} \cdot b_{\mathrm{H}} + a_{\mathrm{L}} \cdot b_{\mathrm{L}})) \cdot B^k + a_{\mathrm{L}} \cdot b_{\mathrm{L}}\,. \end{aligned}$$

Umiddelbart ser det jo ud, som om vi bare har gjort tingene mere komplicerede. Men ved nærmere eftersyn ser man, at det sidste udtryk kun bruger tre multiplikationer af kortere tal, nemlig beregningen af $a_{\mathrm{H}} \cdot b_{\mathrm{H}}$, $a_{\mathrm{L}} \cdot b_{\mathrm{L}}$ og $(a_{\mathrm{H}} + a_{\mathrm{L}}) \cdot (b_{\mathrm{H}} + b_{\mathrm{L}})$. Godtnok kræves der nu fem additioner, [10] hvilket er tre mere end i den rekursive udgave af skolemetoden. Men det afgørende er, at additionerne (som udføres umiddelbart) er meget billige i forhold til multiplikationerrne (som kræver rekursion), hvorfor det er giver mening at spare en multiplikation for prisen af tre additioner. Vi har altså følgende algoritme for at beregne $a \cdot b$:

(a) Opdel a og b i a_{H}, a_{L}, b_{H} og b_{L}.
(b) Beregn de tre produkter

$$p_2 = a_{\mathrm{H}} \cdot b_{\mathrm{H}}, \quad p_0 = a_{\mathrm{L}} \cdot b_{\mathrm{L}}, \quad p_1 = (a_{\mathrm{H}} + a_{\mathrm{L}}) \cdot (b_{\mathrm{H}} + b_{\mathrm{L}}).$$

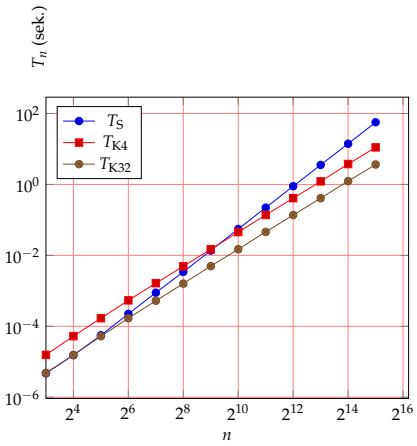
(c) Læg produkterne fra (b) sammen, passende forskudte, for at danne produktet $a \cdot b$, dvs. som

$$a \cdot b = (p_2 \cdot B^{2k} + p_0) + (p_1 - (p_2 + p_0)) \cdot B^k\,.$$

(Den første sum er bare en sammenføjning af to tal, ikke en rigtig addition.)

[10] For at være nøjagtig, skal man bruge fire additioner og en subtraktion. Vi overlader det til læseren at overbevise sig om, at subtraktion ikke er dyrere end addition.

Tallene a_H, a_L, b_H, b_L, $a_H + a_L$ og $b_H + b_L$ består hvert af højst $\lceil n/2 \rceil + 1$ cifre. Derfor er multiplikationen i skridt (b) enklere end den oprindelige opgave, såfremt $\lceil n/2 \rceil + 1 < n$, dvs. $n \geq 4$. Den fuldstændige algoritme kan nu beskrives på følgende måde: For at multiplicere tal med tre cifre eller mindre, benytter man skolemetoden. For tal med $n \geq 4$ cifre bruges tretrinsmetoden foroven.



Figur 1.3. Eksperimentel sammenligning af udførelsestiden for to implementationer af Karatsubas algoritme og skolemetoden for multiplikation af naturlige tal. Implementationen K4 skifter til skolemetoden for tal bestående af færre end fire cifre, K32 for tal med færre end 32 cifre. Hældningen for de tilsvarende linjer er omtrent 1,58. K32 er tre gange hurtigere end K4.

Figur 1.3 viser udførelsestiderne $T_S(n)$, $T_{K4}(n)$ og $T_{K32}(n)$ for implementationer i C++ af skolemetoden og Karatsubas algoritme for multiplikation af n-cifrede tal. Varianten K4 (med udførelsestid $T_{K4}(n)$) skifter til skolemetoden ved færre end fire cifre. Forbedringen i varianten K32 (med udførelsestid $T_{K32}(n)$) beskrives i afsnit 1.6. Begge akser bruger en logaritmisk skala. Groft sagt viser figuren tre linjer med forskellig hældning. Udførelsestiden T_S for skolemetoden vokser som n^2, og derfor har den tilsvarende linje hældning 2. Linjen for Karatsubas algoritme T_{K4} har en svagere hældning, hvilket tyder på, at udførelsestiden vokser som n^β for en konstant $\beta < 2$. Kvotienterne [11] $T_{K4}(n)/T_{K4}(n/2)$ og $T_{K32}(n)/T_{K32}(n/2)$ ligger tæt på 3, hvilket antyder, at β opfylder ligningen $2^\beta = 3$, dvs. at der gælder $\beta = \log 3 \approx 1{,}585$. Omtrent samme hælding kan også aflæses af figur 1.3, og forneden beviser vi formelt, at $T_{K4}(n)$ vokser som $n^{\log 3}$. For at sammenfatte disse betragtninger siger vi, at Karatsubas algoritme er *asymptotisk hurtigere* end skolemetoden. Men vi indser også, at instanserne skal være meget store, inden karatsubaalgoritmens bedre asymptotiske opførsel gør sig gældende i form af et faktisk mindsket tidsforbrug. Læg mærke til, at skolemetoden stadig her hurtigere end K4 for $n = 2^8$, at algoritmerne er lige hurtige for $n = 2^9$, og at Karatsubas algoritme vinder fra og med $n = 2^{10}$. Disse betragtninger fortæller os, at

[11] $T_{K4}(1024) = 0{,}0455$, $T_{K4}(2048) = 0{,}1375$ og $T_{K4}(4096) = 0{,}41$.

- bedre asymptotisk opførsel vinder i sidste ende,
- en asymptotisk langsommere algoritme kan være hurtigere på små instanser.

I næste afsnit skal vi se, hvordan man kan forbedre Karatsubas algoritme for små n og derved opnå en algoritme, som er mindst lige så god som skolemetoden.

Nu kommer vi endelig til den grundige analyse af Karatsubas algoritme.

Lemma 1.4. *Lad $T_K(n)$ være det maksimale antal elementaroperationer, som Karatsubas algortime bruger på n-cifrede tal. Da gælder:*

$$T_K(n) \leq \begin{cases} 3n^2 & \text{når } n \leq 3, \\ 3 \cdot T_K(\lceil n/2 \rceil + 1) + 8n & \text{når } n \geq 4. \end{cases}$$

Bevis. For $n \leq 3$ bruger vi sætning 1.2, som siger, at multiplikationen af to n-cifrede tal med skolemetoden bruger højst $3n^2$ elementaroperationer. Betragt nu tilfældet $n \geq 4$. Opdelingen af a og b i fire dele a_{H}, a_{L}, b_{H} und b_{L} bruger ingen elementaroperationer (jf. fodnoten i beviset for lemma 1.2). Alle dele samt summerne $a_{\mathrm{L}} + a_{\mathrm{H}}$ og $b_{\mathrm{L}} + b_{\mathrm{H}}$ har hver højst $\lceil n/2 \rceil + 1$ cifre; så de tre rekursive multiplikationer bruger højst $3 \cdot T_K(\lceil n/2 \rceil + 1)$ elementaroperationer. For at udregne summerne $a_{\mathrm{L}} + a_{\mathrm{H}}$ og $b_{\mathrm{L}} + b_{\mathrm{H}}$ skal vi bruge to additioner, hvis resultatet hver har færre end n cifre, så vi bruger $2n$ elementaroperationer. Endelig bruges tre additioner af højst $2n$-cifrede tal for at udregne resultatet fra de tre produkter, hvilket kræver højst $3 \cdot 2n$ yderligere elementaroperationerer. Sammenlagt giver dette den øvre grænse i lemmaet for $n \geq 4$. □

I afsnit 2.6 viser vi, hvordan man kan løse rekursionsligninger af denne type generelt.

Sætning 1.3. *Lad $T_K(n)$ være det maksimale antal elementaroperationer, som Karatsubas algortime bruger til multiplikation af to n-cifrede tal. Da gælder $T_K(n) \leq 153n^{\log 3}$ for alle n.*

Bevis. Se afsnit 1.8. □

1.6 Algoritmeoptimering

Udeladt

1.7 Programmerne

Udeladt

1.8 Bevis for lemma 1.3 og sætning 1.3

For fuldstændighedens skyld beviser vi lemma 1.3 og sætning 1.3. Vi begynder med analysen af den rekursive udgave af skolemetoden. I lemma 1.2 har vi set, at det maksimale antal elementaroperationer, som skal bruges til multiplikation af to n-cifrede tal, opfylder rekursionsligningen

$$T(n) \leq \begin{cases} 1\,, & \text{når } n = 1\,; \\ 4 \cdot T(\lceil n/2 \rceil) + 4n\,, & \text{når } n \geq 2\,. \end{cases}$$

Ved induktion i n viser vi, at $T(n) \leq 5n^2 - 4n$, når n er en topotens. For $n = 1$ har vi $T(1) \leq 1 = 5n^2 - 4n$. For topotens n med $n > 1$ gælder

$$T(n) \leq 4 \cdot T(n/2) + 4n \leq 4(5(n/2)^2 - 4n/2) + 4n = 5n^2 - 4n\,;$$

hvor den anden ulighed gælder ved induktion. For vilkårlig n observerer vi, at multiplikationsalgoritmen på to n-cifrede tal ikke kan være dyrere end på to $2^{\lceil \log n \rceil}$-cifrede tal, idet $n = 2^{\log n} \leq 2^{\lceil \log n \rceil}$. Derfor gælder $T(n) \leq T(2^{\lceil \log n \rceil})$. Idet $2^{\lceil \log n \rceil} \leq 2^{1+\log n} = 2n$, får vi den øvre grænse $T(n) \leq 20n^2$ for alle n.

Opgave 1.6. Givet rekursionsligningen $T(1) \leq 1$ und $T(n) \leq 4 \cdot T(n/2) + 9n$. Vis en øvre grænse på $T(n)$ når n er en topotens.

Hvordan kunne vi vide i forvejen, at »$5n^2 - 4n$« var den grænse, som fik induktionsbeviset til at holde? Det kræver ikke, at man er synsk. For $n = 2^k$ får vi ved gentagen substitution:

$$\begin{aligned} T(2^k) &\leq 4 \cdot T(2^{k-1}) + 4 \cdot 2^k \leq 4^2 \cdot T(2^{k-2}) + 4 \cdot (4^1 \cdot 2^{k-1} + 2^k) \\ &\leq 4^3 \cdot T(2^{k-3}) + 4 \cdot (4^2 \cdot 2^{k-2} + 4^1 \cdot 2^{k-1} + 2^k) \leq \cdots \\ &\leq 4^k \cdot T(1) + 4 \cdot \sum_{0 \leq i \leq k-1} 4^i 2^{k-i} \leq 4^k + 4 \cdot 2^k \cdot \sum_{0 \leq i \leq k-1} 2^i \\ &\leq 4^k + 4 \cdot 2^k (2^k - 1) = n^2 + 4n(n-1) = 5n^2 - 4n\,. \end{aligned}$$

Nu til beviset for sætning 1.3. Fra lemma 1.4 ved vi, at T_K opfylder følgende rekursionsligning:

$$T_K(n) \leq \begin{cases} 3n^2\,, & \text{når } n \leq 3\,; \\ 3 \cdot T_K(\lceil n/2 \rceil + 1) + 8n\,, & \text{når } n \geq 4\,. \end{cases}$$

Rekursionsligningen for skolemetoden havde den behagelige egenskab, at hvis n var en topotens, så var argumenterne på højresiden af udtrykket også topotenser. Det gælder ikke for T_K. Men når n er på formen $n = 2^k + 2$ og $k \geq 1$, så gælder $\lceil n/2 \rceil + 1 = 2^{k-1} + 2$. Vores induktionsbevis bør altså føres for tal på formen $n = 2^k + 2$ med $k \geq 0$. Vi vil vise, at der for $k \geq 0$ gælder

$$T_K(2^k + 2) \leq 51 \cdot 3^k - 16 \cdot 2^k - 8\,.$$

For $k = 0$ har vi

$$T_K(2^0 + 2) = T_K(3) \leq 3 \cdot 3^2 = 27 = 51 \cdot 3^0 - 16 \cdot 2^0 - 8\,.$$

I induktionsskridtet for $k \geq 1$ gælder

$$\begin{aligned} T_K(2^k + 2) &\leq 3 \cdot T_K(2^{k-1} + 2) + 8 \cdot (2^k + 2) \\ &\leq 3 \cdot \left(51 \cdot 3^{k-1} - 16 \cdot 2^{k-1} - 8\right) + 8 \cdot (2^k + 2) \\ &= 51 \cdot 3^k - 16 \cdot 2^k - 8\,. \end{aligned}$$

Igen fandt vi den givtige induktionsantagelse uden heksekunster, med ved gentagen substitution:

$$\begin{aligned} T_K(2^k + 2) &\leq 3 \cdot T_K(2^{k-1} + 2) + 8 \cdot (2^k + 2) \\ &\leq 3^k \cdot T_K(2^0 + 2) + 8 \cdot (3^0 \cdot (2^k + 2) + 3^1 \cdot (2^{k-2} + 2) + \ldots + 3^{k-1} \cdot (2^1 + 2)) \\ &\leq 27 \cdot 3^k + 8 \cdot \left(2^k \cdot \frac{(3/2)^k - 1}{3/2 - 1} + 2 \cdot \frac{3^k - 1}{3 - 1}\right) \\ &= 51 \cdot 3^k - 16 \cdot 2^k - 8\,. \end{aligned}$$

Endelig skal vi generalisere udtrykket til alle n. Hertil vælger vi det mindste k, som opfylder $n \leq 2^k + 2$. Åbenbart gælder $k \leq 1 + \log n$. Multiplikationen af n-cifrede tal kan ikke være dyrere end for tal med $2^k + 2$ cifre. Heraf følger:

$$T_K(n) \leq 51 \cdot 3^k - 16 \cdot 2^k - 8 \leq 153 \cdot 3^{\log n} \leq 153 \cdot n^{\log 3}\,,$$

hvor vi har brugt $3^{\log n} = 2^{(\log 3)\cdot(\log n)} = n^{\log 3}$.

1.9 Implementationsaspekter

Udeladt

1.10 Historiske anmærkninger og yderligere resultater

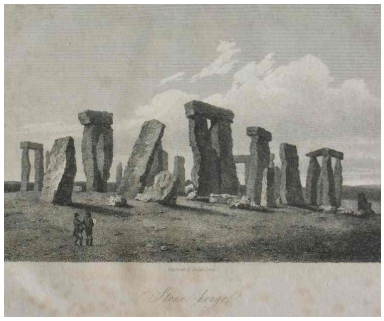
Er Karatsubas algoritme den hurtigst kendte multiplikationsalgoritme? Nej, man kender meget hurtigere metoder. Karatsubas algoritme deler n-cifrede tal op i to dele og klarer sig med blot tre multiplikationer af halvt så lange tal. En naturlig generalisering er at opdele tallene i k bidder af længde omtrent n/k. Hvis det rekursive skridt kan nøjes med l multiplikationer af tal af længde n/k, så vokser udførelsestiden af algoritmen som $n^{\log_k \ell}$. Denne observation nblev brugt af Toom [55] og Cook [9] for at opnå udførelsestiden [12] $O(n^{1+\epsilon})$ for

[12] Vi definerer $O(\cdot)$-notationen i afsnit 2.1.

vilkårligt positivt ϵ. I mange år var algoritmerne af Schönhage og Strassen [46] og Schönhage [45] de asymptotisk bedst kendte fremgangsmåder. Den første algoritme ganger to n-bitstal med $O(n \log n \log\log n)$ bitoperationer og kan implementeres på en turingmaskine i det tilsvarende antal operationer. Den anden algoritme kører i lineær tid $O(n)$ på den maskinmodel, som præsenteres i afsnit 2.2. Denne model tillader multiplikation af to heltal bestående af $\log n$ bit i konstant tid. År 2007 og 2009 blev den første algoritme forbedret til $O((n \log n)2^{c \log^* n})$ bitoperationer af Fürer [17] og af De, Kurur, Saha und Saptharishi [11] Her betegner $\log^* n$ det mindste tal $k \geq 0$ med egenskaben, at $\log(\log(\ldots \log(n) \ldots))$ (k-foldigt gentagen anvendelse) er højst 1. Funktionen $\log^* n$ vokser altså yderst langsomt. Den øvre grænse på $O(n \log n)$ blev etableret i 2019 af Harvey og van der Hoeven.

2

Indledning



Hvis man vil blive billedhugger,[1] *skal man lære en masse grundlæggende teknikker: Hvor finder man passende sten? Hvordan flytter man dem, hvordan virker mejslen, hvordan bygger man et stillads, osv. Når de grundlæggende teknikker er på plads, er man langtfra nogen berømt kunstner, men selv en sjældent talent kan ikke blive berømt uden at beherske grundlaget. Wer die Grundtechniken beherrscht, Man behøver ikke kende det hele, inden man går i gang med sin første skulptur Men man skal være parat til at gå tilbage til grundteknikkerne for at blive bedre og bedre.*

Dette indledende kapitel spiller en lignende rolle i bogen. Vi præsenterer de grundlæggende begreber og metoder for at bedre kunne beskrive og analysere algoritmer i de senere kapitler. Man behøver ikke at arbejde sig igennem dette kapitel fra A til Z inden man giver sig i kast med de følgende kapitler. Vi anbefaler ved første læsning at studere materialet til og med afsnit 2.3 grundigt, og skimme de følgende afsnit. Afsnit 2.1 beskriver notation og terminologi for at beskrive algoritmers kompleksitet kort og præcist. I afsnit 2.2 præsenteres en enkel beregningsmodel, som gør det muligt at abstrahere bort fra mange af de komplikationer, der ville opstå ved at tage hensyn til egenskaberne ved moderne maskinarkitektur. Modellen er tilstrækkelig konkret til at levere nyttige forudsigelser, men tilstrækkelig abstrakt til at tillade elegante overvejelser. I afsnit 2.3 introduceres et notation for pseudokode, som minder om et højniveausprogrammeringssprog og tillader en bekvem beskrivelse af algoritmer end maskinmodellens kode. Desuden giver pseudokoden mulighed for at benytte notation fra matematikken, uden at vi behøver at bekymre os om, hvordan denne ville skulle oversættes til en ægte maskine. Vi vil gøre hyppig brug af kommentarer i programmerne, både for at øge deres læsbarhed og for at gøre det letter at føre formelle korrekthedsbeviser. Teknikker for den slags beviser er genstant for afsnit 2.4.

[1] Billedet af stenkredsen ved Stonehenge er taget fra [42].

Afsnit 2.5 indeholder det første omfattende eksempel: Binærsøgning i en sorteret række. I afsnit 2.6 beskrives matematiske teknikker for programmers kompleksitetsanalyse med vægt på indlejrede løkker og rekursive procedurekald. For analysen af gennemsnitligt tidsforbrug har vi brug for yderligere teknikker; disse beskrives i afsnit 2.7. Randomiserede algoritmer, præsenteret i afsnit 2.8, gør brug af tilfældighed ved under udførelsen at kunne slå plat og krone. Afsnit 2.9 handler om grafer, et begreb som spiller en stor rolle i resten af bogen. I afsnit 2.10 diskuteres spørgsmålet, hvornår man skal betegne en algoritme som effektiv, og kompleksitetsklasserne P og NP samt den vigtige klasse af NP-fuldstændige problemer. Som alle andre kapitler slutter kapitlet med implementationsaspekter (afsnit 2.11) og historiske anmærkninger og videre resultater (afsnit 2.12).

2.1 Asymptotisk notation

Algoritmeanalysens formål er primært at skabe tilforladelige udsagn om algoritmers opførsel, bl.a. deres kørselstid, som er både præcise, kortfattede, almentgyldige og begribelige. Det er selvfølgeligt vanskeligt at opfylde alle disse krav samtidigt. For fx at beskrive algoritmens tidsforbrug T kan man opfatte T som en funktion, der afbilder mængden $\mathcal{I}$ af alle mulige probleminstanser (eller *input*) til mængden $\mathbf{R}_+$ af positive reelle tal. For hver instans I til problemet er da $T(I)$ kørselstiden på I. Denne detaljeringsgrad fører dog til så overvældende meget information, at det ville være håbløst at udvikle en brugbar teori. I stedet skal vi betragte algoritmens opførsel i et mere mere almengyldigt perspektiv.

Vi vil opdele mængden af instanser i klasser af »lignende« instanser og så sammenfatte algoritmens opførsel på instanser fra samme klasse som et eneste tal. Det mest gængse kriterium for klassedelingen er instansens *størrelse*. Sædvanligvis er der en naturlig måde for at bestemme instansens størrelse. Størrelsen på et heltal er antallet af cifre i dens binærrepræsentation; størrelsen af en mængde er dens kardinalitet, dvs. antallet af elementer. Instansstørrelsen er altid et naturligt tal. Sommetider bruger man mere end én parameter for at angive størrelsen på en instans; fx er det gængs at karakteriser størrelsen på en graf i termer af både antal knuder og antal kanter. Vi vil i første omgang se bort fra de komplikationer, der optræder herved. Mængden af alle instanser af størrelse n skrives som $\mathcal{I}_n$ for $n \in \mathbf{N}$. For instanser af størrelse n kan vi interessere os for maksimale, minimale og gennemsnitlige kørselstider, defineret på følgende måde: [2]

[2] Vi vil altid sikre, at mængden $\{ T(I) \colon I \in \mathcal{I}_n \}$ har både maksimum og minimum, og at mængden $\mathcal{I}_n$ er endelig, når vi beregner gennemsnit.

$$T(n) = \begin{cases} \max\{\, T(I) \colon I \in \mathcal{I}_n \,\} & \text{»i værste fald«,} \\ \min\{\, T(I) \colon I \in \mathcal{I}_n \,\} & \text{»i bedste fald«,} \\ \dfrac{1}{|\mathcal{I}_n|} \displaystyle\sum_{I \in \mathcal{I}_n} T(I) & \text{»i gennemsnit«.} \end{cases}$$

Den mest interessante af disse størrelse er kørselstiden i værste fald, fordi den udgør den mest omfattende garanti for algoritmens opførelse. Sammenligningen af opførelsen i bedste og værste fald fortæller os, hvor stor variation i kørselstid der kan forekomme mellem instanser i samme størrelsesklasse. Når afvigelsen mellem bedste og værste fald er meget stor, kan sommetider en analyse af den gennemsnitlige kørselstid give nærmere indsigt i algoritmens faktiske tidsforbrug. Vi skal se nærmere på en eksempel i afsnit 2.7.

Vi går endnu et skridt videre i vores forsøg på at informationen mere overskuelig ved at gøre analysen grovere: Vi koncentrerer os på kørselstidens *vækstrate* ved at bruge *asymptotisk analyse*. To funktioner $f(n)$ og $g(n)$ har samme *vækstrate*, hvis der eksisterer positive konstanter c og d, så uligheden $c \leq f(n)/g(n) \leq d$ gælder der for alle tilstrækkeligt store n. Funkionen $f(n)$ *vokser hurtigere* end $g(n)$, hvis der gælder for hver positive konstant c, at uligheden $f(n) \geq c \cdot g(n)$ for alle tilstrækkeligt store n. For eksempel har funktionerne n^2, $n^2 + 7n$, $5n^2 - 7n$ og $\frac{1}{10}n^2 + 10^6 n$ alle samme vækstrate. Desuden vokser disse funktioner alle hurtigere end funktionen $n^{3/2}$, som selv vokser hurtigere end funktionen $n \log n$. Læg mærke til, at vækstraten fokuserer på opførslen for store n, hvilket også er meningen med begrebet »asymptotisk« i »asymptotisk analyse«.[3]

Hvad er grunden til, at vi kun interesserer os for vækstrate og opførslen for store n? Hovedårsagen for at udvikle effektive algoritmer er netop ønsket om at kunne håndtere store instanser. Når algoritmen A har en lavere vækstrate end algoritmen B for samme problem, vil A typisk være B overlegen for store n. Desuden er vores maskinmodel i forvejen en abstraktion af de faktiske kørselstider og nøjes med at bestemme en konkret maskines opførelse inden for en maskinafhængig konstant faktor. Derfor vil vi ikke skelne mellem algoritmer, hvis kørselstider har samme vækstrate. Vores indskrænkning til vækstrater har desuden den glædelige sideeffekt, at algoritmers kørselstider kan karakteriseres af meget enkle funktioner. Vi vil dog i bogens implementationsafsnit regelmæssigt se lidt nærmere på de maskinnære detaljer, som den asymptotiske analyse er blind for. Generelt bør læseren ved studiet og anven-

[3] Ovs. anm.: Ordet »asymptotisk« er kendt fra matematisk analyse, hvor det betegner opførslen af en funktion, som nærmer sig en grænseværdi uden at antage den. Funktionen $x \mapsto 1/x$ har fx y-aksen som asymptote for $x \to 0$. Ordet er græsk og betyder »ikke-sammenfaldende« og betoner altså, at funktionen nærmer sig, men aldrig helt når, en bestemt linje. I modsætning hertil vil man lede forgæves efter en god forklaring for, hvorfor man i algoritmeanalysen bruger »asymptotisk« for at betegne noget i retning af »opførsel for store n«.

delsen af algoritmer i denne bog altid spørge sig selv, om det asymptotiske perspektiv er relevant.

Vi skal nu indføre den gængse notation for funktioners *asymptotiske opførsel*. Her betegner $f(n)$ og $g(n)$ funktioner, som afbilder naturlige tal til ikke-negative reelle tal. Vi definerer

$$\begin{aligned}
O(f(n)) &= \{\, g(n) \colon \exists c > 0 \colon \exists n_0 \in \mathbf{N}_+ \colon \forall n \geq n_0 \colon g(n) \leq c \cdot f(n) \}\,, \\
\Omega(f(n)) &= \{\, g(n) \colon \exists c > 0 \colon \exists n_0 \in \mathbf{N}_+ \colon \forall n \geq n_0 \colon g(n) \geq c \cdot f(n) \}\,, \\
\Theta(f(n)) &= O(f(n)) \cap \Omega(f(n))\,, \\
o(f(n)) &= \{\, g(n) \colon \forall c > 0 \colon \exists n_0 \in \mathbf{N}_+ \colon \forall n \geq n_0 \colon g(n) \leq c \cdot f(n) \}\,, \\
\omega(f(n)) &= \{\, g(n) \colon \forall c > 0 \colon \exists n_0 \in \mathbf{N}_+ \colon \forall n \geq n_0 \colon g(n) \geq c \cdot f(n) \}\,.
\end{aligned}$$

Venstresiderne læses som »store-o af $f(n)$« og tilsvarende for »store-omega«, »theta«, »lille-o« og »lille-omega«. Læg mærke til, at »$f(n)$« i udtrykket »$O(f(n))$« og »$g(n)$« i udtrykket »$\{\, g(n) \colon \ldots\}$« betegner funktionerne f og g – notationen forsøger blot at tydeliggøre, at funktionen afhænger af variablen n. Derimod menes i betingelsen »$\forall n \geq n_0 \colon g(n) \leq c \cdot f(n)$« funktions*værdien* for n.

Lad os betragte nogle eksempler. Mængden $O(n^2)$ indeholder de funktioner, som vokser højst kvadratisk. Mængden $o(n^2)$ indeholder de funktioner ‚som vokser langsommere end kvadratisk. Mængden $o(1)$ indeholder de funktioner, som går mod 0 for voksende n, hvor strengt taget symbolet »1« betegner den konstante funktion $n \mapsto 1$, som altid har funktionsværdien 1. Dermed tilhører funktionen $f(n)$ mængden $o(1)$, hvis $f(n) \leq c \cdot 1$ for hvert positive c og tilstrækkeligt stort n, dvs. når $f(n)$ går mod 0 for voksende n. Generelt kan man tænke på $O(f(n))$ som mængden af funktioner, som »ikke vokser hurtigere end« $f(n)$; og på $\Omega(f(n))$ som mængden af funktioner, som »vokser mindst lige så hurtigt som« $f(n)$. For eksempel ligger den asymptotiske værstefaldstid for Karatsubas algoritme for heltalsmultiplikation i $O(n^{1{,}585})$, mens den asymptotiske kørselstid for skolemetoden ligger i $\Omega(n^2)$. Derfor kan vi sige, at Karatsubas algoritme er hurtigere end skolemetoden. Notationen $o(f(n))$ angiver mængden af funktioner, som »vokser skarpt langsommere end« $f(n)$. Dens modsætning, notationen $\omega(f(n))$, forekommer ganske sjældent i den grundlæggende algoritmeanalyse og er her kun medtaget for fuldstændighedens skyld.

De fleste algoritmer i denne bog har kørselstider, som kan skrives som polynomium eller som logaritmisk funktion, eller som produkt af den sådanne funktioner. Det næste resultat ser nærmere på polynomier i det asymptotiske perspektiv; beviset giver nogle eksempler på omgangen med notationen.

Lemma 2.1. *Lad $p(n) = \sum_{i=0}^{k} a_i n^i$ være et polynomium med reelle koefficienter, hvor $a_k > 0$. Da gælder $p(n) \in \Theta(n^k)$.*

Bevis. Vi skal vise $p(n) \in O(n^k)$ og $p(n) \in \Omega(n^k)$. Vi bemærker først, at der for $n > 0$ gælder

$$p(n) \leq \sum_{i=0}^{k} |a_i|\, n^i \leq n^k \sum_{i=0}^{k} |a_i| \ ,$$

hvilket medfører $p(n) \leq (\sum_{i=0}^{k} |a_i|)n^k$ for alle positive n. Derfor gælder $p(n) \in O(n^k)$.

Sæt $A = \sum_{i=0}^{k-1} |a_i|$. For alle $n > 0$ har vi nu

$$p(n) \geq a_k n^k - An^{k-1} = \frac{a_k}{2} n^k + n^{k-1} \left(\tfrac{a_k}{2} n - A\right) \ ,$$

og derfor $p(n) \geq (\frac{1}{2}a_k)n^k$ for $n > 2A/a_k$. Ved at vælge $c = \frac{1}{2}a_k$ og $n_0 = 2A/a_k$ i definitionen af $\Omega(n^k)$ ses nu, at $p(n)$ tilhører $\Omega(n^k)$. □ □

Opgave 2.1. Sandt eller falskt? (a) $n^2 + 10^6 n \in O(n^2)$; (b) $n \log n \in O(n)$; (c) $n \log n \in \Omega(n)$; (d) $\log n \in o(n)$.

Asymptotisk notation er så udbredt i algoritmeanalysen, at man af bekvemmelighedsgrunde ofte anvender den præcise notation på en mere fleksibel måde. Ikke mindst benytter man ofte betegnelser for funktionsmængder (fx $O(n^2)$) som om de selv var en enkelt funktion. Især plejer man at skrive $h(n) = O(f(n))$ i stedet for $h(n) \in O(f(n))$ og $O(h(n)) = O(f(n))$ is stedet for $O(h(n)) \subseteq O(f(n))$, som fx:

$$3n^2 + 7n = O(n^2) = O(n^3) \, .$$

Følger af »ligninger« med O-notation skal strengt taget opfattes som udsagn om tilhørsforhold og mængdeinklusioner, og de giver kun mening læst fra venstre til højre.

For en funktion $h(n)$, funktionsmængder F og G og en operator $\diamond$ (fx $+$, $\cdot$ eller /) lad $F \diamond G$ være en forkortelse for $\{\, f(n) \diamond g(n) \colon f(n) \in F, g(n) \in G \,\}$, og $h(n) \diamond F$ være en forkortelse for $\{h(n)\} \diamond F$. Med denne konvention betegner $f(n) + o(f(n))$ altså mængden af funktioner $f(n) + g(n)$ med den egenskab, at $g(n)$ vokster stærkt langsommere end als $f(n)$, dvs. at kvotienten $(f(n) + g(n))/f(n)$ går mod 1 for $n \to \infty$. Ækvivalent skrives $(1 + o(1))f(n)$. Vi bruger denne notation, når vi vil understrege $f(n)$s rolle som »*førende term*«, i forhold til hvilken »*termer af lavere orden*« kan ignoreres.

Lemma 2.2 (regneregler). *Der gælder:*

$$\begin{aligned} cf(n) &= \Theta(f(n)), \textit{for hver positive konstant } c, \\ f(n) + g(n) &= \Omega(f(n)) \, , \\ f(n) + g(n) &= O(f(n)), \textit{når } g(n) = O(f(n)) \, , \\ O(f(n)) \cdot O(g(n)) &= O(f(n) \cdot g(n)) \, . \end{aligned}$$

Opgave 2.2. Bevis lemma 2.2.

Opgave 2.3. Skærp lemma 2.1 ved at vise $p(n) = a_k n^k + o(n^k)$.

Opgave 2.4. Bevis, at der gælder $n^k = o(c^n)$ for heltal k og vilkårligt $c > 1$. Hvor står $n^{\log \log n}$ i forhold til n^k og c^n?

2.2 Maskinmodellen

År 1946 foreslog John von Neumann (fig. 2.1) en enkel, men magtfuld beregningsarkitektur [56]. Den tids begrænsede hardwaremuligheder førte ham til en elegant model, som indskrænkede sig til det væsentlige; ellers havde realiseringen ikke været mulig. I årene efter 1945 har hardwareteknologien godtnok videreudviklet sig dramatisk, men von Neumanns programmeringsmodel , er så elegant og kraftfuld, at den den dag i dag udgør grundlaget for en stor del af programmeringen. Normal virker de programmer, som er udviklet i von Neumann-modellen også i nutidens meget mere komplekse hardware.

Figur 2.1. John von Neumann, * 28.12.1903 i Budapest, † 8.2.1957 i Washington, DC.

Til analyse af algoritmer benytter man en variant av von Neumann-maskinen, som hedder *registermaskinen* eller ram (*random access machine*), en maskine med vilkårlig lageradgang). Denne model blev foreslået i 1963 af Shepherdson og Sturgis [49]. Det drejer det sig om en *sekventiel* regner med uniformt lager, dvs. at der findes bare én centralenhed (CPU), og hver lageradgang koster samme tid. Lageret består af uendeligt mange lagerceller $S[0]$, $S[1]$, $S[2]$, . . ., hvoraf kun et endeligt antal er i brug til enhver tid. Derudover råder registermaskinen over et lille, konstant antal *registre* $R_1, \ldots, R_k$.

Lagercellerne og registrene har plads til »små« heltal, som også kaldes (*maskin*)*-ord*. I vores overvejelser om heltalsaritmetik i kap. 1 gik vi ud fra, at »lille« betyder »etcifret«. Det er dog mere fornuftigt og bekvemt at antage, at definitionen af »lille« afhænger af inputstørrelsen. Vores standardantagelse skal være, at et tal kan gemmes i en lagercelle, hvis dets størrelse er begrænset af et polynomium i inputstørrelsen. Binærrepræsentationen af disse tal behøver et antal bit, der er logaritmisk i inputstørrelsen. Denne antagelse er fornuftig, fordi vi altid kan opdele indholdet af en lagercelle i logaritmisk mange celler, som kun kan gemme en enkelt bit, uden at tid- eller pladsbehovet øges med mere end en logaritmisk faktor. Idet registre bruges til at adressere lagerceller, vil vi forlange, at hver adresse, der forekommer i vores beregninger, finder plads i et register. Hertil rækker det med en polynomiel begrænsning af størrelsen.

På den anden side er antagelsen om en logaritmisk grænse på bitlængden af de gemte tal også nødvendig: Hvis man tillod, at en lagercelle kunne gemme tal af vilkårlig størrelse, ville det i mange tilfælde lede til algoritmer med absurd lille tidsforbrug. Fx ville man men n på hinanden følgende kvadreringer af tallet 2 (som fylder 2 bit) skabe et tal med 2^n bit Man begynder

med $2 = 2^1$, kvadrerer en gang for at få $2^2 = 4$, kvadrer igen for at få $16 = 2^{2\cdot 2}$, osv. Efter n kvadreringer opnår man tallet 2^{2^n}.

Vores model tillader en bestemt, ubegrænset form for parallelisme: enkle operationer på logaritmisk mange bit kan gennemføres i konstant tid.

En registermaskinen kan udføre et (maskin)program. Sådan et *program* udgøres af en liste af maskinkommandoer, nummereret fra 1 til et tal l. Indgangene i listen hedder *linjer* i programmet. Programmet står i programlageret. Vores registermaskine kan udføre følgende maskinkommandoer:

- $R_i := S[R_j]$ *læser* indholdet af den lagercelle, hvis indeks står i register R_j, ind i register R_j.
- $S[R_j] := R_i$ *skriver* indholdet af register R_i ind i den lagercelle, hvis indeks står i register R_j.
- $R_i := R_j \odot R_h$ udfører en binær operation $\odot$ på indholdet af R_j og R_h og gemmer resultatet i register R_i. Der er en række muligheder for, hvad »$\odot$« kan være. De *aritmetiske* operationer er som sædvanligt $+$, $-$ og $*$; disse fortolker registerindholdet som heltal. Operationerne $\div$ og mod , ligeledes på heltal, giver kvotienten hhv. resten ved heltalsdivision. *Sammenlignings-operationerne* $\leq$, $<$, $>$ og $\geq$ giver deres sandhedsværdier som resultat, dvs. *sand* ($= 1$) og *falsk* ($= 0$). Derudover findes de bitvise operationer | (logisk *eller*), & (logisk *og*) og $\oplus$ (eksklusiv *eller*). disse fortolker registerindholdet som bitstreng. Operationerne » (højreskift) og « (venstreskift) tolker sit første argument som bitstreng og sit andet som ikkenegativ forskydningsværdi. De logiske operationer $\wedge$ og $\vee$ bearbejder på *sandhedsværdierne* 1 og 0. Vi kan også antage, at der findes operationer, som tolker registerindholdet som et tal med flydende komma, dvs. som endelig tilnærmelse af et reelt tal.
- $R_i := \odot R_j$ udfører en *unær* operation på register R_j og gemmer resultatet i register R_i. Hertil findes operationerne $-$ (for heltal), $\neg$ (logisk negation af sandhedsværdier) og ~ (bitvis negation af bitstrenge).
- $R_i := C$ tildeler register R_i den *konstante* værdi C.
- JZ k, R_i fortsætter beregningen på programlinje k, hvis register R_i indeholder 0 (*forgrening* eller *betinget hop*), ellers fortsættes der på næste programlinje.[4]
- J k fortsætter beregningen på programlinje k (*ubetinget hop*).

Et program af denne slags udføres skridt for skridt på givet input. Input står i begyndelsen i lagercellerne $S[1]$, ..., $S[R_1]$, udførelsen begynder på programlinje 1. Med undtagelse af forgreningerne JZ og J følges udførelsen af en programlinje altid af udførelsen af den næste programlinje. Udførelsen af programmet ender, hvis den skal udføre en linje, hvis nummer ligger uden for området $\{1, \ldots, l\}$.

[4] Hopmålet k kan, hvis det er nødvendigt, også være angivet som indhold $S[R_j]$ af en lagercelle.

Vi bestemmer tidsforbruget for at udføre et program på givet input på den enklest tænkelige måde: *Det tager præcis en tidsenhed at udføre en maskinkommando*. Et programs totale *kørselstid* er summen af alle udførte kommandoer.

Det vigtigt at klargøre, at registermaskinen er en abstraktion; man må ikke forveksle modellen med en virkelig beregner. Især har ægte elektroniske beregner et endeligt lager og et fast antal bit per register og lagercelle (fx 32 eller 64). I modsætning hertil vokser (»skalerer«) ordlængde og lagerstørrelse i en registermaskine med inputstørrelsen. Man kan betrage dette som en abstraktion af den historiske udvikling: mikroprocessorer har i løbet af tiden haft maskinord af længde 4, 8, 16, 32 og 64 bit Med ord af længde 64 kan man adressere et lager af størrelse 2^{64}. På grund af prisen for dagens lagermedier begrænses størrelsen af lageret i dag derfor af omkostningerne og ikke af længden af adresserne. Interessant nok var dette også sandt ved indførelsen af 32-bit ord!

Vores model for beregningskompleksitet er groft forenklet på den måde, at moderne processorer prøver at gennemføre mange operationer samtidigt. I hvilket omfang dette fører til en tidsbesparelse, afhænger af faktorer som dataafhængighed mellem påhinandenfølgende operationer. Derfor er det ikke muligt at knytte en fast omkostning til hver operation. Denne effekt gør sig ikke mindst gældende ved lageradgang. I værste fald kan lageradgangen bruge mange hundrede gange så lang tid som det bedste fald! Dette hænger sammen med at moderne processorer prøver at holde hyppigt benyttet data i *skyggelageret*, hvilket er en forholdsvis lille, hurtig lagerenhed som ligger tæt forbundet til processoren. Hvilken tidsforbedring opnås af skyggelageret afhænger stærkt af maskinarkitekturen, programmet og det konkrete input.

Vi ville kunne forsøge at udvikle en mere nøjagtig omkostningsmodel, men det skulle være at skyde over målet. Resultatet er en megen kompleks model, som er vanskelig at arbejde med. Selv en vellykket analyse leder til monstrøse formler, som afhænger af mange parametre og som desuden skulle ændres ved hver ny generation af processorer. Selvom informationen i sådan en formel ville være meget præcis, ville den være ganske ubrugelig allerede i kraft af sin kompleksitet. Derfor går vi til den andet yderpunkt ved at eliminere samtlige modelparametre og blot antage, at hver maskinkommando tager én tidsenhed. Det leder til, at konstante faktorer ikke spiller nogen rolle i vores model – endnu en grund til at for det meste holde sig til asymptotisk analyse af algoritmer. Som udligning finder der i hvert kapitel et afsnit om implementationsaspekter, i hvilket vi diskuterer implementationsvarianter og afvejninger som afhængigheder mellem behovet af forskellige resurser som tid og lagerplads.

2.2.1 Baggrundslager

Den mest dramatiske forskel mellem registermaskinen og en virkelig regnemaskine er lagerstrukturen: regnemaskinens uniforme lager er en utilstrækkelig

beskrivelse af den virkelige maskines komplekse lagerhierarki. I afsnit 5.7, **??** og **??** skal vi betrage algoritmer, som er skræddersyede til store datamængder, som skal holdes i det langsomme baggrundslager som fx på et eksternt hårddrev. For at undersøge den slags algoritmer benytter vi *baggrundslagermodellen*.

Yderlagermodellen ligner registermaskinen, men adskiller sig i lagerstrukturen. Det (hurtige) indre lager (sommetider kaldt »hovedlageret«) består af kun M ord; det (langsomme) ydre lager (sommetider kaldt »baggrundslageret«) har ubegrænset størrelse. Der findes en *blokflytningsoperation*, , som flytter B på hinanden følgende lagerceller mellem det langsomme og det hurtige lager. Hvis yderlageret fx er et hårddrev, vil M betegne størrelsen af hovedlageret og B betegne blokstørrelsen for dataoverførsel mellem hovedlager og hårddrevet, som repræsenterer et godt kompromis mellem den store ventetid (latens) og den store båndbredde ved overførslen fra et lagermedium til det andet. I skrivende stund er $M = 2$ Gbyte og $B = 2$ MByte realistiske værdier. En blokflytning varer omtrent 10 ms, hvilket svarer til $2 \cdot 10^7$ klokcykler i en 2 GHz-processor. Med andre fastlæggelser for parametrene M og B kan vi modellere den mindre forskel mellem maskinens skyggelager og hovedlageret.

2.2.2 Parallelbehandling

I moderne regnemaskiner findes mange slags parallelbehandling. Mange processorer råder over *SIMD*-registre med en bredde mellem 128 og 256 bit, som muliggør parallel udførelse af en kommando på en hel række dataobjekter (SIMD står for *single instruction, multiple data*: enkel operation – multiple data).

Simultan flertråding gør det muligt for processer at bedre udnytte deres resurser ved at flere tråde (delprocesser) udføres samtidigt på en processorkerne. Sågar mobile enheder indeholder ofte flerkerneprocessorer, som kan udføre programmer uafhængigt af hinanden, og de fleste værtsmaskiner har flere af den slags flerkerneprocessorer, som har adgang til et *fælles lager*.

Koprocessorer, især dem, der bruges i computergrafik, indeholder endnu flere parallelt arbejdende komponenter på en og samme chip. Højpræstationsregnere består af flere værtssystemer, som er sammenkoblet i et eget hurtigt netværk. Endeligt findes muligheden for at ganske løst sammenkoble al slags computere i et netværk (internettet, radionetværk, osv.), som herved danner et *fordelt system*, i hvilken millioner af knuder arbejder sammen. Det er klart, at ingen simpel model rækker til at beskrive parallelle programmer, som kører på så mange forskellige niveauer af parallelitet I denne bog vil vi derfor indskrænke os til af og til (og uden formel argumentation) at forklare, hvorfor en bestemt sekventiel algoritme er mere eller mindre velegnet til at blive omarbejdet til parallel udførelse. For eksempel ville man kunne bruge SIMD-instruktioner til algoritmerne for aritmetik for lange heltal i kapitel 1.

2.3 Pseudokode

Registermaskinen er en abstrakion og forenkling af maskinprogrammer, som skal udføres på en mikroprocessor, der muliggør en præcis definition af begrebet »udførelsestid«. Modellen er samtidigt netop på grund af sin enkelhed uegnet til formuleringen af komplekse algoritmer, fordi de tilsvarende registermaskineprogrammer bliver alt for lange og stort set ulæselige for mennesker. I stedet vil vi formulere vores programmer i *pseudokode*, som danner en passende abstraktion og forenkling af imperative programmeringssprog som C, C++, Java, C# og Pascal, kombineret med rundhåndet brug af matematisk notation. Vi skal nu beskrive vores konventioner for pseudokodenotation i denne bog, og præsentere en tidsmålingsmodel for pseudokodeprogrammer. Denne er meget enkel: *Grundlæggende pseudokodeinstruktioner behøver konstant tid; procedure- og funktionskald behøver konstant tid plus tiden for at udføre deres krop.* Vi skal retfærdiggøre dette ved at skitsere, hvordan man kan oversætte pseudokode til ækvivalent registermaskinkode, som netop har denne tidsopførsel. Vi skal nøjes med at forklare denne oversættelse så langt, at tidsmålingsmodellen bliver forståelig. Vi bland andet enhver overvejelse om optimeringsteknikker for oversættere, idet konstante faktorer ikke spiller nogen rolle i vores teori. Læseren er dog velkommen til at bladre forbi de næste afsnit og acceptere tidsmålingsmodellen for pseudokode som aksiomatisk. Den her benyttede syntax ligner programmeringssproget Pascal [29], fordi denne notation virker typografisk mere behagelig i bogform end den velkendte syntax fra C og dette sprogs videreudviklinger som C++ og Java.

2.3.1 Variabler og elementære datatyper

En *variabelerklæring* »$v = x : T$« introducerer en variabel v af typen T og initialiserer den med værdien x. For eksempel skaber »$svar = 42 : \mathbf{N}$« en variabel *svar*, som kan antage ikke-negative heltalsværdier, og giver den værdien 42. Sommetider undlader vi at nævne typen af en variabel, hvis den fremgår af sammenhængen. Som typer forekommer elementartyper som fx heltal, booleske værdier, pegere (referencer) og sammensatte typer. Vi fordefinerer nogle sammensatte typer, nemlig rækker og visse anvendelsesspecifikke klasser (se nedenfor). Når typen af en variabel er irrelevant for fremstillingen, benytter vi den uspecificerede type *Element* som pladsholder for en vilkårlig type. Sommetider udvides de numeriske typer med værdierne $+\infty$ og $-\infty$, når det er bekvemt. Ligeledes udvider vi sommetider typer med den udefinerede værdi (betegnet med symbolet $\perp$), som skal kunne skelnes fra »egentlige« objekter af typen T. Især ved pegertyper er en udefineret værdi nyttig. En værdi af pegertypen »**Peger på** T« kaldes et *greb* om et objekt af typen T. I registermaskinen er sådan et greb bare indeks (kaldt »adressen«) af første celle af det lagersegment, som indeholder et objekt af typen T.

Erklæringen »a: *Række*$[i..j]$ **af** T« introducerer *rækken* a, som består af $j - i + 1$ *indgange* af typen T, hvilke er gemt i lagercellerne $a[i], a[i+1], \ldots, a[j]$.

Rækker er registermaskinen realiserede som sammenhængende lagersegmenter. For at finde indgangen ved $a[k]$ er det nok at kende startadressen af a og størrelsen af objekter af typen T. Hvis fx registeret R_a indeholder startadressen for rækken a, registeret R_i indeholder indeks 42, og hvis hver indgang fylder et lagerord, så læser kommandosekvensen »$R_1 := R_a + R_i; R_2 := S[R_1]$« inholdet af $a[42]$ ind i register R_2. Størrelsen af rækken ligger fast, når erklæringen er udført; den slags rækker hedder *statiske*. I afsnit 3.2 viser vi, hvordan man implementerer *dynamiske rækker*, som kan vokse og krympe i løbet af programudførelsen.

Erklæringen »c: **Klasse** *alder*: **N**, *indkomst*: **N**« stiller en variabel c til rådighed, hvis værdier er heltalspar. Komponenterne kaldes sommetider felter og angives som $c.alder$ og $c.indkomst$. Udtrykket »**adressen på** c« giver et greb om c (dvs. adressen på c). Når p er en variabel af en passende pegertype, så kan vi gemme grebet i p med kommandoen »$p :=$ **adressen på** c«; tilsvarende giver udtrykket »$*p$« objektet c tilbage. De to felter tilhørende c kan også betegnes som $p \rightarrow alder$ og $p \rightarrow indkomst$. Den alternative skrivemåde $(*p).alder$ og $(*p).indkomst$ giver mening, men bliver sjældent brugt.

Pladsen for rækker og objekter stilles til rådighed med kommandoen **alloker** og frigøres igen med kommandoen **afalloker**. For eksempel stiller kommandoen »$p :=$ **alloker** *Række*$[1..n]$ **af** T« en række med n objekter af typen T til rådighed, dvs. at der reserveres et sammenhængende lagersegment med plads til akkurat n objekter af typen T, og variablen ps værdi sættes til rækkens greb (dvs. lagersegmentets startadresse). Kommandoen **afalloker** p frigiver lagerområdet, så det kan anvendes til andet brug. Med operationerne **alloker** og **afalloker** kan vi opdele rækken S af registermaksinens lagerceller i disjunkte segmenter, som kan tilgås enkeltvis. Begge funktioner kan implementeres på en sådan måde, at de kun kræver konstant tid, fx på følgende ekstremt nemme måde: Adressen til den første ledige lagercelle i S holdes i specialvariablen *fri*. Et kald til **alloker** reserverer et lagerafsnit, som begynder ved *fri*, og øger derefter *fri* med omfanget af det reserverede afsnit. Et kald til **afalloker** har ingen effekt. Denne implementation er tidsbesparende men ikke pladsbesparende: Godtnok kræver både **alloker** og **afalloker** kun konstant tid, men det totale lagerforbrug er summen af længderne af samtlige nogenside reserverede segmenter i stedet for maksimum af den til et bestemt tidspunkt brugte (dvs. reserverede, men endnu ikke frigivne) plads. Det er et åbent problem at realisere en vilkårlig følge af **alloker**- og **afalloker**-operationer både lagereffektivt og med konstant tidsforbrug per operation. For de algoritmer, som præsenteres i denne bog, kan **alloker** og **afalloker** dog realiseres både tids- og pladseffektivt på samme tid.

Vi vil gøre brug af nogle sammensatte datatyper fra matematiken. Især vil vi benytte tupler, endelige følger og mængder. Ordnede *par*, *tripler* og andre *tupler* skriver vi i parenteser som fx $(3,1)$, $(3,1,4)$ og $(3,1,4,1,5)$. Idet antallet af komponenter i en tupel er fastlagt af typen, kan operationer på tupler brydes ned i operationer på de enkelte komponenter på den oplagte måde. En *følge*, skrevet i spidse parenteser, gemmer objekter af samme type i

en bestemt orden; i modsætning til tupler er antallet af indgange ikke fastlagt af typen. For eksempel erklærer tildelingen »$s = \langle 3, 1, 4, 1 \rangle$: *Følge* **af** **Z**« en følge s af heltal og initialiserer den med følgen $\langle 3, 1, 4, 1 \rangle$ af tallene 3, 1, 4 og 1 i den rækkefølge. Den tomme følge skrives som $\langle\ \rangle$. Følgen er en naturlig abstraktion af mange andre datatyper som fx arkivet (eng. *file*), strengen (eller »tegnfølgen«), listen, stakken og køen. I kap. 3 skal vi se grundigt på mange forskellige måder at repræsentere følger. I de senere kapitler vil vi så bruge følger som matematisk abstraktion uden at tage videre hensyn til implementationsdetaljer.

Mængder spiller en vigtig rolle i matematiske overvejelser, og vi vil ofte bruge dem i vores pseudokode. Der vil altså forekomme erklæringer som fx »$M = \{3, 1, 4\}$: *Mængde* **af** **N**«, som skal forstås på samme måde som erklæringer af rækker eller mængder. Implementationen af datatypen »mængde« sker ofte ved hjælp af følger.

2.3.2 Kommandoer

Den enkleste anvisning er *tildelningen* »$x := E$«, hvor x er en variabel og E er et udtryk. En tildeling kan transformeres til en følge af operationer i registermaskinen af konstant længde. For eksempel bliver tildelingen $a := a + b \cdot c$ oversat til »$R_1 := R_b * R_c$; $R_a := R_a + R_1$«, hvor R_a, R_b og R_c angiver de registre, som indholder a, b og c. Fra programmeringssproget C låner vi forkortelserne `++` og `--` for øgning og mindskning af en variabel med 1. Vi vil også bruge *simultan tildeling* af flere variable. Hvis fx a og b er variabler af samme type, så bytter kommandoen »$(a, b) := (b, a)$« indholdet af a og b.

Den betingede kommando **hvis** C **så** I **ellers** J, hvor C er et boolsk udtryk og I og J er anvisninger, oversættes til kommandofølgen

$$\mathit{eval}(C);\ \texttt{JZ}\ \mathit{sElse},\ R_C;\ \mathit{oversæt}(I);\ \texttt{J}\ \mathit{sEnd};\ \mathit{oversæt}(J)$$

De enkelte dele skal læses som følger: $\mathit{eval}(C)$ er en følge af kommandoer, som evaluerer udtrykket C og gemmer resultatet i registret R_C; $\mathit{oversæt}(I)$ er en følge af anvisninger, som implementerer kommandoen I; $\mathit{oversæt}(J)$ implementerer tilsvarende kommandoen J; *sElse* er nummeret på den første kommando i $\mathit{oversæt}(J)$; endelig er *sEnd* nummeret på den første kommando som følger efter $\mathit{oversæt}(J)$. Instruktionerne beregner først værdien af C. Hvis denne er *falsk* (dvs. lig med 0), så hopper registermaskinen til den første kommando i oversættelsen af J. Hvis resultatet derimod er *sandt* (dvs. lig med 1), så udfører programmet oversættelsen af I og hopper derefter videre til den første kommando, som følger efter oversættelsen af J. Vi skriver »**hvis** C **så** I« som forkortelse for »**hvis** C **så** I **ellers** ;«, dvs. en betinget udførelse med tom ellers-gren.

Man kan gruppere enkelte kommandoer ved at sætte dem efter hinanden som $I_1; \ldots; I_r$, og kan opfatte denne gruppe som én (mere kompleks) kommando. Ved oversættelsen til et maskinprogram bliver oversættelserne af de enkelte dele sat efter hinanden.

Vores pseudokodenotation for programmer er tiltænkt en menneskelig læser og følger derfor ikke en lige så streng syntaks som programmeringssprog normalt skal gøre. Blandt andet vil vi bruge indrykninger for at gruppere anvisninger, hvilket sparer os for en masse parenteser sammenlignet med fx C eller Pascal. Vi vil kun bruge parenteser til gruppering, hvis programmet ellers var flertydigt. På samme måde bruger vi linjeskift i stedet for semikolon for at dele to anvisnigner.

Løkken »**gentag** I **indtil** C« oversættes til $\mathit{oversæt}(I)$; $\mathit{eval}(C)$; JZ sI, R_C, hvor sI er nummeret på den første kommando i $\mathit{oversæt}(I)$. Vi skal bruge mange andre former for løkker, som man kan opfatte som forkortelser for forskellige **gentag** -løkker. I følgende tabel angives til venstre nogle eksempler på den slags forkortelser, og til højre den fuldstændige formulering.

mens C **udfør** I	**hvis** C **så gentag** I **indtil** $\neg C$
for $i := a$ **til** b **udfør** I	$i := a$; **mens** $i \leq b$ **udfør** I; i++
for $i := a$ **to** ∞ **mens** C **udfør** I	$i := a$; **mens** C **udfør** I; i++
for hvert $e \in s$ **udfør** I	**for** $i := 1$ **til** $\lvert s \rvert$ **udfør** $e := s[i]$; I

Ved oversættelsen af løkker til registermaskinen er mange maskinnære forbedringer mulige, som vi dog ikke tager hensyn til her. For vores formål er det kun af betydning, at tiden for udførelsen af en løkke er summen af de enkelte gennemløbstider samt udførelsestiden for at evaluere løkkens iterationsbetingelse.

2.3.3 Procedurer og funktioner

Et underprogram ved navn *foo* erklæres som »**Procedure** $\mathit{foo}(D)$ I«. Her er I prodedurens krop, og D en liste af variabelerklæringer, som angiver de formelle parametre af proceduren *foo*. Et kald til *foo* har formen $\mathit{foo}(P)$, hvor P er en liste af *argumenter* eller *egentlige* parametre [5] som har samme længde som listen af variabelerklæringer. Argumenterne er enten værdier eller referencer. Medmindre andet er sagt, går vi ud fra, at elementære objekter som fx heltal og boolske værdier overføres som værdier, mens komplekse objekter som fx rækker overføres som reference. Denne konvention svarer til moderne sprog som C og sikrer, at parameteroverførelsen tager konstant tid. Parameteroverførelsens semantik er følgende: Når x er en værdiparameter af typen T, så skal det tilsvarende kaldsargument være et udtryk E, som leverer en værdi af typen T. Parameteroverførelsen bliver derved ækvivalent til erklæringen af en lokal variabel ved navn x, som initialiseres med værdien af E. Når derimod x er en referenceparameter af typen T, så skal det tilsvarende kaldsargument være en variabel af samme type; i løbet af udførelsen af procedurekroppen er x blot et alterantivt navn til denne variabel.

Ganske som ved variabelerklæringer undlader vi sommetider at skrive typen i parametererklæringen, når vi finder den uvæsentlig eller åbenlyst ud fra

[5] Ovs. anm.: Man kan sommetider støde på begrebet »aktuel paremeter«, som er en misforståelse af det engelske udtryk »*actual parameter*«.

sammenhængen. Sommetider vil vi bruge matematisk notation til en implicit erklæring af en parameter. For eksempel erklærer »**Procedure** $bar(\langle a_1, \ldots, a_n \rangle)$ I« en procedure, hvis argument er en følge af elementer af ikke nærmere specificeret type.

I procedurekroppen kan den særlige anvisning **returner** forekomme, som afslutter udførelsen af procedurekroppen og fortsætter udførelsen ved den første anvisning efter kaldsstedet. Det samme sker, når udførelsen når enden af procedurekroppen.

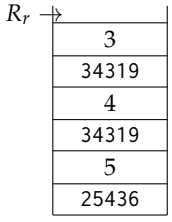
De fleste procedurekald kan oversættes ved blot at erstatte kaldet med procedurekroppen og tage hensyn til parameteroverførelsen. Denne fremgangsmåde kaldes »linjeindlejring«. Overførelsen af argumentet E til den formelle parameter $x\colon\ T$ realiseres som en følge af kommandoer, som evaluerer udtrykket E og tildeler resultatet til den lokale variabel x. Overførelsen til en formel referenceparameter $x\colon\ T$ sker ved, at den lokale variabel x får typen **Peger på** T og alle forekomster af x i procedurekroppen erstattes af $(*x)$. Ved indgangen til procedurekroppen udføres desuden tildelingen $x :=$ **adressen på** y, hvor y er kaldsargumentet. Brugen af linjeindlejring giver oversætteren rige muligheder for alskens optimeringer, så denne fremgangsmåde fører til den mest effektive kode, når proceduren er lille eller kun kaldes fra et enkelt sted.

Funktioner minder meget om procedurer, bortset fra at de deres **returner** -anvisning også skal returnere en værdi. , I figur 2.2 ses erklæringen af en funktion *fakultet*, som på argument n returnerer resultatet $n!$, samt dens oversættelse til registermaskinen. Linjeindlejringen virker ikke for *rekursive* funktioner og procedurer som fx *factorial*, som kalder sig selv enten direkte eller indirekte – kodeerstatningen ville aldrig blive færdig. For at realisere rekursive underprogrammer i registermaskinkode, har man brug for en *rekursionsstak*. Udtrykkelige kald og rekursionsstakken bruges også for omfattende underprogrammer, som kaldes fra flere steder i programmet, fordi linjeindlejringen ville føre til for meget kode. Rekursionsstakken, betegnet med RS, placeres i en til formålet reserveret del af lageret. Den indeholder en følge af *aktiveringsposter*, en for hvert aktivt underprogramskald. Et særligt register R_r peger altid på den første ledige plads på rekursionsstakken. Aktiveringsposten for et underpogram med k parametre og ℓ lokale variable har størrelse $1 + k + \ell$. Første position indeholder *returadressen*, dvs. nummeret på den programlinje, ved hvilken programmets udførelse skal fortsætte efter kaldets afslutning; de næste k positioner er reserverede til parametre, og de resterende ℓ positioner til de lokale variable. Kaldet af et underpogram realiseres på følgende måde som maskinkode: Først skriver den kaldende procedure *kalder* returadressen og argumenterne på stakken, øger værdien af R_r med $1 + k$ og hopper til den første programlinje underprogrammet *kaldt*. Det kaldte underprogram begynder med at reservere plads til sine ℓ lokale variabler ved at øge R_r med ℓ. Derefter udføres kroppen af *kaldt*. I løbet af dennne udførelse findes den $i$te formelle parameter (for $0 \leq i < k$) i celle $RS[R_r - \ell - k + i]$ og den $j$te lokale variable (for $0 \leq j < \ell$) i celle $RS[R_r - \ell + j]$. Anvisningen **returner** i

Funktion *fakultet*(n): **N**
  hvis $n = 0$ **så returner** 1 **ellers returner** $n \cdot$ *fakultet*$(n-1)$

```
fakultet → 34313  R_arg := RS[R_r − 1]          //indlæs n til register R_arg
           34314  HN 34323, R_arg               //hop til så -grenen, hvis n = 0
ellers     34315  RS[R_r] := 34319              //returadresse for rekursionen
           34316  RS[R_r + 1] := R_arg − 1      //parameteren er n − 1
           34317  R_r := R_r + 2                //øg stakpegeren
           34318  H 34313                       //rekursivt kald fakultet(n − 1)
           34319  R_res := RS[R_r − 1] ∗ R_res  //gem n · fakultet(n − 1)
           34320  R_r := R_r − 2                //frigør aktiveringsposten
           34321  H RS[R_r]                     //hop retur
           34322  H 34326
så         34323  R_res := 1                    //gem 1 i resultatregisteret
           34324  R_r := R_r − 2                //frigør aktiveringsposten
           34325  H RS[R_r]                     //hop retur
           34326
```

Figur 2.2. Rekursiv funktion *fakultet* til beregning af fakultetsfunktionen og den tilsvarende maskinkode. Maskinkoden returnerer resultatet $n!$ i register R_{res}. Koden svarer nogenlunde til dette kapitels konventioner for *oversæt*. For læselighedens skyld er maskinens repertoire af kommandoer udvidet med et ubetinget indirekte hop til adressen $RS[R_r]$ i linje 34322 og 34325. Desuden sammenfatter linje 34319 udregningnen højresiden, som egentlig skulle kræve registeroperationer som fx $R_0 := R_r$; $R_1 := 1$; $R_0 := R_0 - R_1$; $R_0 := RS[R_0]$; $R_{res} := R_0 * R_{res}$. På den anden side ville en moderne oversætter fjerne linje 34322, som jo alrig nås, og muligvis slå de to identiske programstumper for **returner** i linje 34320–34321 og 34324–34325 sammen.



Figur 2.3. Rekursionsstakken for kaldet *factorial*(5), når rekursionen er nået til kaldet *factorial*(3) Nederst ligger returadressen til den kaldende kode.

kaldt oversættes til en følge af maskinkommandoer, som mindsker R_r med $1 + k + \ell$ (*kaldt* kender naturligvis tallene k og ℓ) og hopper så retur til den adresse, som er gemt i $RS[R_r]$. Udførelsen fortsætter nu i proceduren *kalder*. Læg mærke til, at rekursion med denne metode ikke længere er et problem, fordi hvert kald (eller »inkarnation«) af underprogrammet har sit eget om-

råde på rekursionsstakken for sine parametre og lokale variable. Figur 2.3 viser rekursionsstakkens indhold for et kald af $\mathit{factorial}(5)$ i det øjeblik, hvor rekursionen er nået til kaldet af $\mathit{factorial}(3)$. Værdien `25436` nederst i stakken står for nummeret på den programlinje, som følger efter kaldet $\mathit{factorial}(5)$.

Opgave 2.5 (Eratostenes' si)**.** Oversæt nedenstående pseudokode, som finder og udskriver alle primtal op til tallet n, til maskinkode. (Vi er ligeglade med at oversætte udskriftskommandoen på sidste linje.) Vis først, at algoritmen er korrekt.

$a = \langle 1, \ldots, 1 \rangle : \mathit{Række}[2..n]$ **af** $\{0, 1\}$
// referenceparameter; til sidst gælder $a[i] = 1 \Leftrightarrow$ *i er primsk*
for i:=2 **til** $\lfloor \sqrt{n} \rfloor$ **udfør**
  hvis $a[i]$ **så for** j:=2i **til** n **skridt** i **udfør** $a[j] := 0$
  // Hvis $a[i] = 1$ *så er i primsk, men multipler af i er ej*
for i:=2 **til** n **udfør** **hvis** $a[i]$ **så** udskriv$(i, »$ er primsk$«)$

2.3.4 Objektorientering

Vi vil holde os til en enkel form for objektorientering med formålet at adskille datastrukturers specifikation (også kaldt *grænseflade*) fra implementationen. Vi vil forklare den relevante notation med et eksempel. Definitionen

Klasse $\mathit{Komplekst}(x, y\colon \mathit{tal})$ **af** tal
  $\mathit{re} = x\colon \mathit{tal}$
  $\mathit{im} = y\colon \mathit{tal}$
  Funktion $\mathit{abs}\colon$ **R returner** $\sqrt{\mathit{re}^2 + \mathit{im}^2}$
  Funktion $\mathit{addér}(c'\colon \mathit{Komplekst})\colon \mathit{Komplekst}$
    returner $\mathit{Komplekst}(\mathit{re} + c'.\mathit{re}, \mathit{im} + c'.\mathit{im})$

stiller en (delvis) implementation af en taltype for komplekse tal til rådighed, som kan bruge vilkårlige taltyper som **Z**, **Q** eller **R** til real- og imaginærdelen. (»*Komplekst*« er også et eksempel på, at vi ofte vil skrive vores klassenavne med stort begyndelsesbogstav.) Real- og imaginærdel holdes i *instansvariablerne re* og *im*. Deklarationen »$c\colon \mathit{Komplekst}(2, 3)$ **af R**« skaber et komplekst tal c, som initialiseres til $2 + 3i$, hvor i angiver den imaginære enhed. Udtrykket $c.\mathit{im}$ tilgår imaginærdelen af c, og metoden $c.\mathit{abs}$ bestemmer absolutværdien af c, som er et reelt tal.

Angivelsen af en type efter **af** i variabeldeklarationen gør det muligt at parameterisere klasser på lignende måde som *template*-mekanismen i C++ og de generiske typer i Java. Læg mærke til, at vores tidligere deklarationer »*Mængde* **af** *Element*« og »*Følge* **af** *Element*« med denne notation definerer helt almindelge parameteriserede klasser. Objekter i en klasse intialiseres ved at tildele instansvariablerne de værdier, som er angivet i variabeldeklarationen.

2.4 Konstruktion af korrekte algoritmer og programmer

2.5 Eksempel: Binærsøgning

Binærsøgning er en nyttig teknik til at søge efter en element i en sorteret række. Vi kommer til at bruge binærsøgning meget i de senere kapitler.

Det enkleste scenario ser ud på følgende måde. Givet er en sorteret række $a[1..n]$ af parvist forskellige indgange, dvs. at $a[1] < a[2] < \cdots < a[n]$, samt et element x. Vi skal afgøre, om x forekommer i rækken og finde det indeks $k \in \{1, \ldots, n+1\}$, som opfylder $a[k-1] < x \leq a[k]$. Her skal man opfatte $a[0]$ og $a[n+1]$ som fiktive indgange med værdierne $-\infty$ hhv. $+\infty$. Vi bruger disse fiktive indgange i invarianter og beviser for at forenkle notationen, men de forekommer ikke i programmet.

Binærsøgning er et eksempel på det algoritmisk princip, som hedder »del og hersk«. Vi vælger et index $m \in \{1, \ldots, n\}$ og sammenligner x med $a[m]$. Hvis $x = a[m]$, så er vi færdige og returnerer $k = m$. Hvis derimod $x < a[m]$, så indskrænker vi søgningen til delrækken før $a[m]$. Hvis $x > a[m]$, så indskrænker vi søgning til delrækken efter $a[m]$. Vi skal nu præcisere, hvad vi mener med »at indskrænke søgningen til en delrække«. Vi vedligeholder to indeks v og h (for »venstre« og »højre«) og invarianten

$$0 \leq v < h \leq n+1 \quad \text{og} \quad a[v] < x < a[h]\,. \tag{I}$$

Invarianten gælder klart i begyndelsen, hvor $v = 0$ og $h = n+1$. Hvis v og h er på hinanden følgende indeks, dvs. $v+1 = h$, så kan vi konkludere, at x ikke forekommer i rækken. Figur 2.4 viser det fuldstændige program.

Lad os vise programmets korrekthed. Vi viser først, at løkkeinvarianten I er gyldig, hver gang iterationsbetingelsen »$l+1 < r$« bliver prøvet. Dette gøres ved induktion i antallet af iterationer. Vi har allerede etableret, at

```
Funktion binærsøgning(x: Element, a: Række[1..n] af Element): 1..n+1
    (v, h) := (0, n+1)
    så længe v + 1 < h udfør
        antag I ∧ v + 1 < h      //løkkeinvarianten og iterationsbetingelsen gælder
        m := ⌊(v + h)/2⌋         //v < m < h
        s := sammenlign(x, a[m])     //s = −1 hvis x < a[m], s = 0 hvis x = a[m], s = +1 hvis x > a[m]
        hvis s = 0 så returner m     //a[m] = x
        hvis s < 0 så h := m         //a[v] < x < a[m] = a[h]
        hvis s > 0 så v := m         //a[v] = a[m] < x < a[h]
        antag I
    antag I ∧ v + 1 = h
returner h       //a[h − 1] < x < a[h]
```

Figur 2.4. Binærsøgning efter x i en sorteret række $a[1..n]$. Algoritmen returnerer et indeks k, som opfylder $a[k-1] < x \leq a[k]$.

invarianten gælder i begyndelsen, altså når $v = 0$ og $h = n + 1$. Dette er basis for induktionen. For induktionsskridtet skal vi vise, at et gennemløb af løkkens krop opretholder invarianten. Lad os altså antag, at invarianten gælder, og at iterationsbetingelsen $v + 1 < h$ er opfyldt. Idet v og h er heltal, har vi $v + 2 \leq h$. Derfor har vi både

$$m = \lceil \frac{v+h}{2} \rceil = \lceil \frac{v+2+h-2}{2} \rceil =\leq \lceil \frac{2h-2}{2} \rceil = h - 1 < h$$

og på samme måde $m > v$, dvs. $v < m < h$. Især er m et gyldigt indeks og det giver mening at evaluere $a[m]$. Hvis $x = a[m]$, standser vi. Ellers sætter vi enten $h = m$ eller $v = m$, og i begge tilfælde gælder $h < v$. Hvis $x < a[m]$ så sætter vi $h = m$, således at $x < a[h]$, mens $a[l] < x$ gælder fra invarianten. Tilfældet $x > a[m]$ behandles på samme måde. Invarianten er nu genetableret og er derfor gyldig hver gang, løkkens iterationsbetingelse bliver evalueret.

For at færdiggøre beviset betragter vi endeligt situationen $h + 1 \geq v$, hvor løkkens krop ikke bliver besøgt. Fra invarianten fås $h < v$, og idet h og v er heltallige, har vi $h + 1 = v$. Ifølge invarianten har vi nu $a[h - 1] < x < a[h]$. Hermed er vi færdige: enten retunerer programmet et indeks m med $a[m] = x$, eller et indeks h med $a[h - 1] < x < a[h]$.

Vi mangler at argumentere for, at løkken terminerer. Vi lægger først mærke til, at alle løkkegennemløb bortset fra det sidste enten forøger v eller formindsker h, dvs. at $h - v$ er skarpt aftagende positive heltal. Derfor terminerer løkken. Vi vil dog vise noget stærkere, nemlig at løkken terminerer allerede efter et logaritmisk antal gennemløb. Hertil betragter vi værdien $h - v - 1$. Det er netop antallet af indeks i, som opfylder $v < i < h$ og derfor et naturligt mål for størrelsen af det aktuelle delproblem. Vi vil nu vise, at hvert løkkegennemløb på nær det sidste gør størrelsen af delproblemet halvt så stort eller endnu mindre. I hvert af disse gennnemløb minsker $h - v - 1$ til en værdi, der er højst

$$\begin{aligned}
&\max \left\{ h - \left\lfloor \frac{h+v}{2} \right\rfloor - 1, \left\lfloor \frac{h+v}{2} \right\rfloor - v - 1 \right\} \\
&\quad \leq \max\{h - ((h+v)/2 - \tfrac{1}{2}) - 1, (h+v)/2 - v - 1 \\
&\quad = \max\{(h - v - 1)/2, (h - v)/2 - 1\} = (r - v - 1)/2\,,
\end{aligned}$$

hvilket udgør mindst en halvering. Vi begynder med $h - v - 1 = n + 1 - 0 - 1 = n$. Efter $k \geq 0$ gennnemløb gælder derfor uligheden $r - v - 1 \leq n/2^k$. Lad gennemløb nummer $k + 1$ være det sidste gennemløb, i hvilket x bliver sammenlignet med indgangen $a[m]$. (Hvis søgningen lykkes, viser det sig, at $x = a[m]$, og søgningen afsluttes. Hvis søgningen mislykkes, fører sammenligningen i runde $k + 2$ til, at $h = v + 1$, og søgningen afsluttes uden yderligere sammenligninger.) Da må uligheden $h - v - 1 \geq 1$ have været opfyldt i runde k. Det følger, at $1 \leq n/2^k$, og dermed $2^k \leq n$, dvs. $k \leq \log n$. Idet k er et heltal, kan uligheden skærpes til $k \leq \lfloor \log n \rfloor$.

Sætning 2.1. *Binærsøgning finder et element i en sorteret række af længde n i højst $1 + \lfloor \log n \rfloor$ sammenlignigner mellem indgange. Udførelsestiden er $O(\log n)$.*

Opgave 2.6. Vis, at denne grænse er skarp, dvs. at der for hvert n findes instanser af størrelse n, som leder til $1 + \lfloor \log n \rfloor$ sammenligninger i algoritmen.

Opgave 2.7. Formuler binærsøgning i termer af tovejssammenligninger, dvs. at sammenligningen kun afgør, om $x \leq a[m]$ eller $x > a[m]$.

Vi skal nu betragte to vigtige udvidelser af binærsøgning. For det første behøver værdierne $a[i]$ ikke at være gemt i en række. Vi skal bare være i stand til at beregne værdien $a[i]$ for givet i. Hvis vi fx får givet en strengt monotont voksende funktion f og argumenter i og j med $f(i) < x \leq f(j)$, så kan vi benytte binærsøgning for at bestemme det index $k \in \{i+1, \ldots, j\}$, som opfylder $f(k-1) < x \leq f(k)$. I denne sammenhæng kaldes binærsøgning ofte »bisektion«.

For det andet kan vi også benytte binærsøgning i situationer, hvor rækken er uendelig lang. Antag, at vi har et sorteret række $a[1..\infty]$ og vil finde det index $k \in \{1, 2, 3, \ldots\}$, som opfylder $a[k] < x \leq a[k+1]$. Hvis x er større end alle indgange i rækken, må proceduren gerne tage uendelig lang tid. Fremgangsmåden er at sammenligne x med $a[2^0]$, $a[2^1]$, $a[2^2]$, $a[2^3]$, ..., indtil vi finder det første i, som opfylder $x \leq a[2^i]$. Denne fremgangsmåde kaldes *eksponentiel søgning* Hvis $i = 0$, så returneres $k = 1$, ellers afsluttes søgnningen med sædvandlig binærsøgning i den endelige række $a[(2^{i-1}+1)..2^i]$.

Sætning 2.2. *Kombinationen af eksponentiel og binærsøgning finder x i en ubegrænset sorteret række med højst $(2 \log k) + 3$ sammenligninger, hvor k er givet ved betingelsen $a[k-1] < x \leq a[k]$.*

Bevis. Vi behøver $i+1$ sammenligninger, for at finde det mindste i med $x \leq a[2^i]$, og (givet at $i > 0$) derefter højst $\log(2^i - 2^{i-1}) + 1 = i$ sammenligninger for binærsøgningen. Sammenlagter er dette $2i + 1$ sammenligninger. Idet $k \geq 2^{i-1}$, får vi $i \leq 1 + \log k$, hvoraf påstanden følger. □

Binærsøgning er en certificerende algoritme. Den returnerer et indeks k som opfylder $a[k-1] < x \leq a[k]$. Hvis $x = a[k]$, så bekræfter indeks k, at x forekommer i rækken. Hvis $a[k-1] < x < a[k]$, og rækken er sorteret, bekræfter indeks k, at x ikke forekommer i rækken. Læg mærke til, at hvis rækken ikke opfylder sorteringsforudsætningen, så har vi ingen information i dette tilfælde. Det er umuligt at kontrollere sorteringsforudsætningen i logaritmisk tid.

2.6 Grundlæggende algoritmeanalyse

Vi kan sammenfatte de hidtil betragtede principper for algoritmeanalyse på følgende måde. Vi abstraherer bort fra komplikationerne i en virkelig datamat ved at betragte registermaskinen. I denne model måles kørselsstid

ved at tælle antallet af udførte kommandoer. Vi forenkler analysen yderligere ved at inddele problemer efter instansstørrelse og fokusere på værstefald. Den asympotiske notation gør det muligt at ignorere konstante faktorer og lavereordens termer. Dette grovkornige perspektiv tillader os også at betragte øvre grænser for kørselstiden i stedet for eksakte vurderinger, så længe det asymptotiske resultater forbliver uændret. Samlet leder disse forenklinger til, at vi kan analysere kørselstiden af pseudokode direkte, dvs. uden at behøve at oversætte programmet til maskinsprog.

Vi skal nu præsentere et sæt enkle regler for analyse af pseudokode. Lad $T_n(I)$ være den maksimale kørselstid for en programstump I på instanser af størrelse n. For kørselstiden af større programmer gælder da følgende:

- For på hinanden følgende kommandoer: $T_n(I;J) \leq T_n(I) + T_n(J)$.
- For betingede kommandoer: $T_n(\textbf{hvis } C \textbf{ så } I \textbf{ ellers } J) = T_n(C) + \max\{T_n(I), T_n(J)\}$.
- For løkker: $T_n(\textbf{gentag } I \textbf{ indtil } C) = \sum_{i=1}^{k(n)} T'_n(I, C, i)$, hvor $k(n)$ angiver det maksimale antal løkkegennemløb på instanser af størrelse n, og $T'_n(I, C, i)$ er kørselstiden for det $i$te gennemløb af løkken, inklusive betingelsen C.

Underprogrammer betragtes i afsnit 2.6.2. Bland reglerne foroven er det kun løkkereglen, der kan give anledning til vanskeligheder, idet man skal vurdere summer.

2.6.1 Summer

Vi viser nu nogle grundlæggende teknikker til evaluering af summer. Summer dukker op i forbindelse med analyse af løkker samt ved analysen af gennemsnitlige og forventede kørselstider.

For eksempel består algoritmen *indsættelsessortering*, som forklares i afsnit 5.1, af to indlejrede løkker. Den ydre løkke tæller variablen i fra 2 op til n. Den indre løkke gennemløbes højst $(i-1)$ gange. Derfor er det samlede antal gennemløb gennem den indre løkke højst

$$\sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)\,,$$

hvor den anden ligning følger af (A.11). Idet et enkelt gennemløb af den indre løkke kræver tid $O(1)$, bliver kørselstid i værste fald $\Theta(n^2)$. Alle indlejrede løkker, hvis antal iterationer er let at bestemme, kan analyseres på en tilsvarende måde: man arbejder »indfra og ud«, og prøver at finde et estimat for kørselstiden af den aktuelle »indre løkke« som et lukket udtryk. Med simple omformninger som $\sum_i ca_i = c\sum_i a_i$, $\sum_i(a_i + b_i) = \sum_i a_i + \sum_i b_i$ eller $\sum_{i=2}^{n} a_i = -a_1 + \sum_{i=1}^{n} a_i$ kan man ofte få summerne på en pæn form, som man kan finde i et katalog over summeformler. Et lille udvalg af den slags formler findes i appendix A. Idet vi normal kun interesserer os for asympototisk opførsel, kan vi se bort fra præcise formler for disse summer og i stedet nøjes

med estimater. For eksempel kan vi i stedet for at udregne summen foroven eksakt, begrænse den opad- og nedadtil meget nemmere (for $n \geq 2$):

$$\sum_{i=2}^{n}(i-1) \leq \sum_{i=1}^{n} n = n^2 = O(n^2)\,,$$
$$\sum_{i=2}^{n}(i-1) \geq \sum_{i=\lceil n/2\rceil+1}^{n} n/2 = \left\lfloor \frac{n}{2} \right\rfloor \cdot \frac{n}{2} = \Omega(n^2)\,.$$

2.6.2 Rekursionsligninger

I vores regler for algoritmeanalyse har vi hidtil ikke bekymret os om kald til underprogrammer. Ikke-rekursive underprogrammer er lette at håndtere, fordi vi kan analysere underprogrammet for sig og sætte den resulterende grænse ind i udtryk for kørselstiden for det kaldende program. Men når vi bruger denne fremgangsmåde på rekursive underprogrammer, fører den ikke til en lukket formel, men til en rekursionsligning.

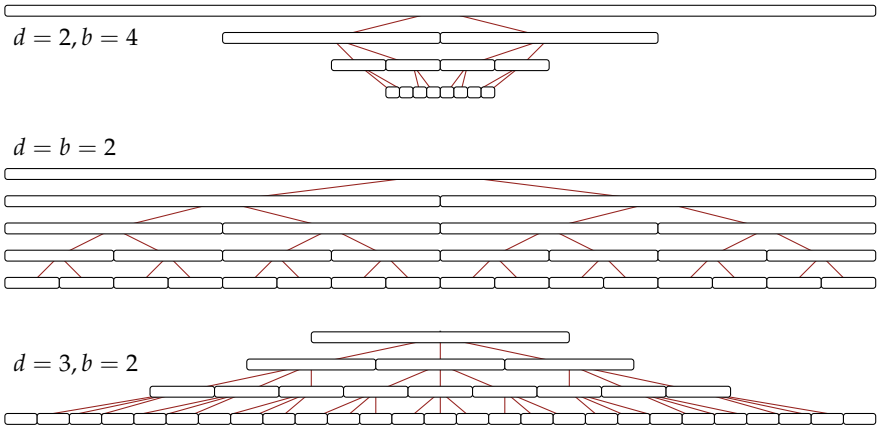
I den rekursive udgave af skolemetoden for multiplikation optræder ved analysen af antallet af elementaroperationer for eskempel ligningerne $T(1) = 1$ og $T(n) = 4T(\lceil n/2\rceil) + 4n$. De tilsvarnde ligninger for Karatsubas algoritme er $T(n) = 3n^2$ for $n \leq 3$ og $T(n) = 3T(\lceil n/2\rceil + 1) + 8n$ for $n > 3$. Generelt definerer en *rekursionsligning* som en funktion i termer af funktionens værdier på mindre argumenter. Definition fuldstændiggøres ved udtrykkelige definitioner af små parameterværdier (»basis«). Løsningen af rekursionsligninger, dvs. at finde en ikke-rekursiv, lukket formel for en rekursivt definieret funktion, er en interessant og omfattende del af matematikken. I vores kontekst skal vi hovedsageligt koncentere os om den slags rekursionsligninger, som optræder i forbindelse med analysen af del-og-hersk-algoritmer. Vi begynder med et enkelt specialtilfælde for at gøre hovedidéen forståelig. Givet en instans af størrelse $n = b^k$ for et naturligt tal k. For $k \geq 1$ benytter vi lineært meget arbejde for at opdele instansen i d mindre instanser af størrelse n/b og for at kombinere resultaterne af de rekursive kald på de enkelte delinstanster. For $k = 0$ sker ingen rekursive, og vi bruger a meget arbejde for at løse opgaven for instansen direkte.

Sætning 2.3 (Klassifikation af del-og-hersk-ligninger, enkel form)**.** *For positive konstanter a, b, c og d, og $n = b^k$ for et naturligt tal k, lad r være en rekursionsligning givet ved*

$$r(n) = \begin{cases} a & \text{for } n = 1\,, \\ d \cdot r(n/b) + cn & \text{for } n > 1\,. \end{cases}$$

Da gælder

$$r(n) = \begin{cases} \Theta(n) & \text{for } d < b\,, \\ \Theta(n \log n) & \text{for } d = b\,, \\ \Theta(n^{\log_b d}) & \text{for } d > b\,. \end{cases}$$



Figur 2.5. Illustration af de tre fald i klassifikationen af del-og-hersk-ligninger. Instanser fremstilles som rektangler, hvis længde repræsenterer instansens størrelse. Delinstanserne, som opstår ud fra en instans, er vist et niveau længere nede. Figurens øvre del viser tilfældet $d = 2$ og $b = 4$, i hvilket en instans leder til to delinstanser med en fjerdedel af størrelsen – den samlede størrelse af delinstanserne udgør altså halvdelen af den oprindelige instans. Figurens midterste del viser tilfældet $d = b = 2$, og den nederste del $d = 3$ og $b = 2$.

Figur 2.5 illustrerer den centrale indsigt, som ligger til grund for sætning 2.3. Vi betragter tidsforbruget på hvert niveau af rekursionen. Vi begynder med en instans af størrelse n. På nivau i af rekursionen har vi d^i delinstanser af størrelse n/b^i hver. Det samlede størrelse af alle instanser på niveau i er derfor

$$d^i \frac{n}{b^i} = n \left(\frac{d}{b}\right)^i .$$

Tidsforbruget for en instans (bortset fra de rekursive kald, den selv måtte lede til) er c gange instansstørrelse. Derfor er det samlede tidsforbrug på et rekursionsniveau proportionalt med den samlede størrelse af alle instanser på dette niveau. Afhængigt af, om d/b er mindre end 1, lig med 1, eller større end 1, opstår der forskellige opførsler.

For $d < b$ er tidsforbruget *geometrisk aftagende* med rekursionsniveauet, og det *første* niveau i rekursionen er allerede ansvarlig for en konstant til af det samlede tidsforbrug.

For $d = b$ bruges på *hvert* rekursionsniveau *præcis den samme* tid. Idet der findes logaritmisk mange niveauer, er det samlede tidsforbrug $\Theta(n \log n)$.

For $d > b$ er tidsforbruget *geometrisk voksende* i rekursionsniveauet, og en konstant del af den samlede tid bruges på rekursionens *sidste* niveau. Vi skal nu gennemgå disse overvejelser i detaljer.

Bevis. Vi begynder med en enkelt instans af størrelse $n = b^k$. Lad os kalde den for niveau 0 i rekursionen.[6] På niveau 1 findes d instanser af størrelse $n/b = b^{k-1}$ hver. På niveau 2 findes d^2 instanser af størrelse $n/b^2 = b^{k-2}$ hver. Generelt findes der på det $i$te niveau d^i mange instanser af størrelse $n/b^i = b^{k-i}$ hver. I bunden af rekursionen, på niveau k, findes d^k instanser af størrelse $n/b^k = b^{k-k} = 1$ hver. Hver sådan instans forårsager omkostning a, så den samlede omkostning på rekursionens sidste niveau er ad^k.

Hernæst beregner vi den samlede omkostning for del-og-hersk-skridtene på nivau i for $i \in \{0, \ldots, k-1\}$. På niveau i forekommer d^i rekursive kald, hver til én instans af størrelse b^{k-i}. Hvert sådant kald forårsager omkostning $c \cdot b^{k-i}$; den samlede omkostning på niveau i er derfor $d^i \cdot c \cdot b^{k-i}$. Ved at summere bidraget fra alle niveauer fra 0 bis $k-1$ fås

$$\sum_{i=0}^{k-1} d^i \cdot c \cdot b^{k-i} = c \cdot b^k \cdot \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i = cn \cdot \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i .$$

Vi deler op i tilfælde afhængigt af størrelseforholdet mellem d og b:

$d = b$: Vi får omkostning $ad^k = ab^k = an = \Theta(n)$ for rekursionens sidste niveau k og samlet omkostning $cnk = cn \log_b n = \Theta(n \log n)$ for del-og-hersk-skridtene.

$d < b$: Vi får omkostning $ad^k < ab^k = an = O(n)$ for rekursionens sidste niveau. Til del-og-hersk-skridtene benytter vi ligning (A.13) for summen af en geometrisk række, nemlig $\sum_{i=0}^{k-1} q^i = (1-q^k)/(1-q)$ for $q > 0$ og $q \neq 1$, og opnår grænserne

$$cn \cdot \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i = cn \cdot \frac{1-(d/b)^k}{1-d/b} < cn \cdot \frac{1}{1-d/b} = O(n)$$

og

$$cn \cdot \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i = cn \cdot \frac{1-(d/b)^k}{1-d/b} > cn = \Omega(n) .$$

$d > b$: Der gælder

$$d^k = (b^{\log_b d})^k = b^{k \log_b d} = (b^k)^{\log_b d} = n^{\log_b d} .$$

Derfor har det sidste niveau af rekursionen omkostning $an^{\log_b d} = \Theta(n^{\log_b d})$. For del-og-hersk-skridtene benytter vi igen formlen (A.13) for den geometriske række og får

[6] I beviset holder vi os til terminologien af rekursive programmer med henblik på at støtte intuitionen for, hvad der foregår. De matematiske overvejelser kan dog bruges for alle ligninger af sætning 2.3, uanset om de stammer fra et rekursivt program eller ej.

$$cb^k \frac{(d/b)^k - 1}{d/b - 1} = c\frac{d^k - b^k}{d/b - 1} = cd^k \frac{1 - (b/d)^k}{d/b - 1} = \Theta(d^k) = \Theta(n^{\log_b d}) . \tag{2.1}$$

□

Vi kommer til at bruge del-og-hersk-klassifikationen i sætning **??** mange gange i denne bog.[7] Desværre dækkes fx rekursionsligningen $T(n) \leq 3n^2$ for $n \leq 3$ og $T(n) \leq 3T(\lceil n/2 \rceil + 1) + 8n$ for $n > 3$, som optræder i analysen af Karatsubas algoritme, ikke af sætningen i sin enkle form, fordi den ikke tager hensyn til hverken uligheder eller afrundinger i de indgående udtryk. Det kan også forekomme, at den additive term i rekursionsligningen ikke er lineær. Vi skal nu vise, hvordan man kan udvide sætningen til at gælde for mere generelle utryk i form af en *rekursionsrelation*.

$$r(n) \leq \begin{cases} a , & \text{hvis } n \leq n_0 ; \\ cn^s + d \cdot r(\lceil n/b \rceil + e_n) , & \text{hvis } n > n_0 . \end{cases} \tag{2.2}$$

Her er $a > 0$, $b > 1$, $c > 0$, $d > 0$ og $s \geq 0$ konstante reelle tal, og værdierne e_n for $n > n_0$ er reelle tal med $-\lceil n/b \rceil < e_n \leq e$ for noget heltal $e \geq 0$.

Sætning 2.4 (Klassifikation af del-og-hersk-relationer)**.** *Lad r være en rekursionsrelation som opfylder (2.2). Da gælder*

$$r(n) = \begin{cases} O(n^s) & \text{for } d < b^s, \text{ dvs. } \log_b d < s ; \\ O(n^s \log n) & \text{for } d = b^s, \text{ dvs. } \log_b d = s ; \\ O(n^{\log_b d}) & \text{for } d > b^s, \text{ dvs. } \log_b d > s . \end{cases}$$

Bevis. Udeladt. □

Der findes mange yderligere generaliseringer af sætning **??**: Man kan afbryde rekursionen tidligt, variere delinstansernes størrelse mere, lade antallet af delinstanser afhænge af instansstørrelsen osv. For mere information henvises læseren til bøgerne [21, 47].

Opgave 2.8. Betragt rekursionsligningen

$$C(n) = \begin{cases} 1 & \text{for } n = 1, \\ C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + cn & \text{for } n > 1. \end{cases}$$

Vis, at $C(n) = O(n \log n)$.

[7] Ovs. anm.: Klassifikationen lader til at fylde så meget i mange grundlæggende fremstillinger af algoritmeanalysens matematiske værktøjer, at den i mange kredse er kendt under det noget højttravende navn »mestersætningen«.

Opgave 2.9. Angiv en del-og-hersk-algoritme, hvis kørselstid er givet af rekursionsligningen $T(1) = a$ og

$$T(n) = \lceil \sqrt{n} \rceil \cdot T\left(\left\lceil \frac{n}{\lceil \sqrt{n} \rceil} \right\rceil\right) + cn \quad \text{for } n > 1\,.$$

Vis, at $T(n) = O(n \log\log n)$.

Opgave 2.10. Tidsforbruget for opslag i en datastruktur beskrives ofte af rekursionsligningen $T(1) = a$, $T(n) = T(n/2) + c$. Vis, at $T(n) = O(\log n)$.

2.6.3 »Globale« betragtninger

De teknikker for algoritmeanalyse, vi har introduceret foroven, er »syntaktiske« i følgende forstand: For at analysere et større program, betragter vi i første omgang de enkelte dele og kombinerer derefter deres analyseresultater til en samlet analyse af hele programmet. Ved kombinationsskridtet benyttes summer og rekursionsligninger.

Vi skal dog også bruge en helt anden fremgångsmåde, som man kunne kalde »semantisk«. Hertil associerer vi dele af programmets kørsel til dele af en kombinatorisk struktur og argumenter siden i termen af denne struktur. For eksempel kan vi observere, at en bestemt programstump udføres højst én gang per kant i en graf, og at derfor den samlede omkostning svarer til antallet af kanter i grafen. Eller vi kan observere, at udføreslsen af en vis del af programmet fordobler størrelsen af en vis struktur; hvis man derudover ved, at strukturens størrelse begynder med 1 og ender som højst n i slutningen af programmet, kan den pågældende del af programmet højst udføres $\log n$ gange.

2.7 Gennemsnitsanalyse

I denne afsnit præsenterer vi gennemsnitsanalysen af algoritmer. Vi gør dette ved hjælp af tre eksempler af voksende kompleksitet. Vi går ud fra, at læseren er fortrolig med sandsynlighedsregningens grundlæggende begreber, fx udfaldsrum, middelværdi, indikatorvariabel og middelværdiens linearitet. Disse begreber er sammenfattede i afsnit A.3.

2.7.1 Tælle én op

Vi begynder med et enkelt eksempel. Givet er en række $a[0..(n-1)]$ at nuller og ettere. Opgaven er at forøge (modulo 2^n) det binære tal beskrevet af cifrene $a[n-1],\ldots,a[0]$.

$i := 0$
så længe $(i < n$ og $a[i] = 1)$ **udfør** $a[i] := 0$; i++
hvis $i < n$ **så** $a[i] := 1$

Hvor ofte bliver løkkens krop udført? I værste fald (nemlig når $a = \langle 1, \ldots, 1 \rangle$) klart n gange, og i bedste fald (nemlig når $a[0] = 0$) klart 0 gange. Men hvad er gennemsnitstiden? For at spørgsmålet giver mening, skal vi først definere en stokastisk model, dvs. det udfaldsrum, som ligger til grund for analysen. Lad os fastlægge os på følgende stokastiske model: Hvert ciffer er 0 eller 1 med sandsynlighed $\frac{1}{2}$, og cifrene er uafhængige. Ækvivalent kunne vi sige, at hver bitfølge med n bit har samme sandsynlighed, nemlig $(\frac{1}{2})^n$. Antal afviklinger af løkkekroppen er k, hvis og kun hvis $k \in \{0, \ldots, n-1\}$ og $a[0] = \cdots = a[k-1] = 1$ samt $a[k] = 0$, eller hvis $k = n$ og $a = \langle 1, \ldots, 1 \rangle$. Den første hændelse sker med sandsynlighed $2^{-(k+1)}$, den anden med sandsynlighed 2^{-n}. Middelværdien for antallet af løkkegennemløb er altså

$$\sum_{k=0}^{n-1} k2^{-(k+1)} + n2^{-n} \leq \sum_{k=0}^{n-1} (k+1)2^{-(k+1)} < \sum_{k \geq 1} k2^{-k} = 2\,,$$

hvor den sidste ligning kan findes i appendix som (A.14).

2.7.2 Finde maksimum

Vores næste eksempel er mere krævende. Vi betragter følgende programstump, som finder den største ingang i rækken $a[1..n]$:

$m := a[1];$ **for** $i := 2$ **til** n **udfør hvis** $a[i] > m$ **så** $m := a[i]$

Hvor ofte udføres tildelingen $m := a[i]$? I værste fald (fx når $a = \langle 1, \ldots, n \rangle$) sker tildelingen ved hvert løkkegennemløb, dvs. $(n-1)$ gange. I bedste fald (fx når $a = \langle 1, \ldots, 1 \rangle$) slet ikke. For at bestemme gennemsnittet skal vi igen begynde med at bestemme udfaldsrummet. Lad os antage, at rækken består af n forskellige indgange, og at hver rækkefølge er lige sandsynlig. Med andre ord består udfaldsrummet af de $n!$ forskellige permutationer af rækkens indgange. Hver permutation er lige sandsynlig, nemlig $1/n!$. Det er for analysen ligegyldig, hvilke konkrete værdier, der står i rækken (bare de er forskellige); vi kan altså antage, at a indeholder tallene fra 1 til n i en eller anden rækkefølge. Vi interesserer os nu for det forventede antal af »foreløbige maksima«. Her kaldes en indgang $a[i]$ for *foreløbigt maksimum*, hvis den er større end alle tidligere indgange. For eksempel har følgen $\langle 1, 2, 4, 3 \rangle$ tre foreløbige maksima, og følgen $\langle 3, 1, 2, 4 \rangle$ har to foreløbige maksima. Antallet af gange, tildelingen $m := a[i]$ bliver udført, er 1 mindre end antallet af foreløbige maksima. For en permutation π af tallene fra 1 til n lad $M_n(\pi)$ være antallet af foreløbige maksima i π. Det står tilbage at bestemme middelværdien $E[M_n]$. Vi skal vise to måder at gøre det på. For små n er det nemt at udregne middelværdien direkte. For $n = 1$ findes der kun én permutation, nemlig $\langle 1 \rangle$, med ét foreløbigt maksimum, så $M_1 = 1$ og derfor $E[M_1 = 1]$. For $n = 2$ findes der to permutationer, nemlig $\langle 1, 2 \rangle$ og $\langle 2, 1 \rangle$. De har henholdsvis 2 og 1 foreløbige maksima, hvorfor $E[M_2] = \frac{3}{2}$. For større n skal vi bruge følgende analyse.

Skriv M_n som sum af indikatorvariablerne I_1 til I_n, dvs. $M_n = I_1 + \cdots + I_n$, hvor $I_k = 1$ for en permutation π, hvis den $k$te indgang i π er et foreløbigt maksimum. For eksempel er $I_3(\langle 3,1,2,4\rangle) = 0$ og $I_4(\langle 3,1,2,4\rangle) = 1$. Da gælder

$$\begin{aligned} E[M_n] &= E[I_1 + \cdots + I_n] \\ &= E[I_1] + \cdots + E[I_n] \\ &= \Pr(I_1 = 1) + \cdots + \Pr(I_n = 1)\,, \end{aligned}$$

hvor vi har brugt middelværdiens linearitet for den anden ligning (A.2), og tredje ligning gælder fordi I_k'erne er indikatorvariable. Vi skal stadig beregne sandsynligheden for hændelsen $I_k = 1$. Den $k$te indgang i en tilfældig permutation er et foreløbigt maksimum hvis og kun hvis den er størst blandt de k første indgange. I en tilfældig permutation står den største af de k første indgange på hver af pladserne med samme sandsynlighed. Derfor har vi $\Pr(I_k = 1) = 1/k$, og der gælder

$$E[M_n] = \sum_{k=1}^{n} \Pr(I_k = 1) = \sum_{k=1}^{n} \frac{1}{k}\,.$$

Hermed kan man fx udregne $E[M_4] = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} = \frac{1}{12}(12 + 6 + 4 + 3) = \frac{25}{12}$. Summen $\sum_{k=1}^{n} 1/k$ vil dukke up igen senere i bogen. Den kaldes det »$n$te harmoniske tal« og betegnes H_n. Man kan begrænse den som $\ln n \le H_n \le 1 + \ln n$, dvs. $H_n \approx \ln n$, se (A.12). Vi konkluderer, at antallet af foreløbige maksima i gennemsnit er meget mindre end i værste fald.

Opgave 2.11. Bevis $\displaystyle\sum_{k=1}^{n} \frac{1}{k} \le \ln n + 1$. *Vink*: Begynd med at vise $\displaystyle\sum_{k=2}^{n} \frac{1}{k} \le \int_1^n \frac{1}{x}\,dx$.

Nu skal vis beskrive en anden metode for at bestemme det forventede antal af foreløbige maksima. Vi vil forkorte $A_n = E[M_n]$ for $n \ge 1$ og sætter $A_0 = 0$. Enhver permutations første indgang er et foreløbigt maksimum, og hvert tal har sandsynlighed $1/n$ for at stå på første plads. Når første tal er i, kan kun tallene fra $i + 1$ til n være foreløbige maksima længere til højre. Disse $n - i$ står i resten af permutationen i tilfældig rækkefølge; derfor er det forventede antal yderligere foreløbige maksima givet ved A_{n-i}. Vi har altså rekursionsligningen

$$A_n = 1 + \frac{1}{n} \cdot \sum_{i=1}^{n} A_{n-i} \qquad \text{dvs.} \qquad nA_n = n + \sum_{i=1}^{n-1} A_i\,.$$

Denne ligning kan forenkles med et standardtrick. Samme ligning for $n - 1$ i stedet for n giver $(n-1)A_{n-1} = n - 1 + \sum_{i=1}^{n-2} A_i$. Vi trækker denne ligning fra ligningen for n og opnår

$$nA_n - (n-1)A_{n-1} = 1 + A_{n-1} \qquad \text{dvs.} \qquad A_n = 1/n + A_{n-1}\,,$$

som giver $A_n = H_n$.

2.7.3 Lineær søgning

Vi er nået til det tredje og og mest krævende eksempel. Det drejer sig om følgende søgningsproblem. Vi begynder med at placere indgange med nummer 1 til n på en eller anden måde, indgang i står fx på position ℓ_i. Efter placeringen gennemføres nogle søgninger. For at lede efter en indgang med nøgle x, gennemgås følgen fra venstre til højre, til vi støder på nøgle x. På denne måde koster det ℓ_i skridt at få adgang til den $i$te indgang.

Vi antager nu, at sandsynligheden for at søge efter en bestemt indgang er den samme for hver søgning. Vi vedtager, at indgang nummer i søges med sandsynlighed p_i. Der gælder $p_i \geq 0$ for alle $i \in \{1, \ldots, n\}$, og $\sum_i p_i = 1$. I denne situation er de *forventede* eller *gennemsnitlige* omkostninger ved en søgning lig med $\sum_i p_i \ell_i$, fordi vi leder efter den $i$te indgang med sandsynlighed p_i, og denne søgning koster ℓ_i skridt.

Hvad er nu den »bedste« måde at arrangere de n indgange på, dvs. den ordning, som minimerer gennemsnitssøgetiden? Intuitivt er det klart, at indgangene nok bør sorteres efter faldende søgningssandsynlighed. Lad os vise dette.

Lemma 2.2. *En ordning er optimal med hensyn til den forventede søgetid, hvis der gælder $\ell_i < \ell_j$ for alle $i, j \in \{1, \ldots, m\}$ med $p_i > p_j$. For søgningssandsynligheder med $p_1 \geq p_2 \geq \cdots \geq p_n$ fører ordningen $\ell_1 = 1$, …, $\ell_n = n$ til den optimale forventede søgetid $\sum_i p_i i$.*

Bevis. Betragt modsætningsvist en optimal ordning, hvor der findes i og j med $p_i > p_j$ og $\ell_i > \ell_j$, dvs. at indgang i er mere sandsynlig end indgang j men alligvel ordnet længere bagved. Hvis vi bytter runndt på de to ingange, er ændringen i gennemsnitlig søgetid givet ved

$$-(p_i\ell_i + p_j\ell_j) + (p_i\ell_j + p_j\ell_i) = (p_j - p_i)(\ell_i - \ell_j) < 0\,.$$

Med andre ord er den nye ordning bedre end den gamle, i modstrid med antagelsen.

Betragt først tilfældet $p_1 > p_2 > \cdots > p_n$. Idet der kun findes endelig mange ordninger (nemlig $n!$), eksisterer der en optimal ordning. Hvis $i < j$ og indgang i er ordnet efter indgang j, så kan ordningen ifølge afsnittet foroven ikke være optimal. Derfor placerer den optimale ordning indgang i på plads $\ell_i = i$, og den resulterende forventede søgetid er derfor $\sum_i p_i i$.

Hvis $p_1 \geq p_2 \geq \cdots \geq p_n$, så er ordningen $\ell_i = i$ for alle i stadig optimal. Hvis nogle af sandsynlighederne er ens, findes mere end en optimal ordning. Men ordningen af indgange med samme sandsynlighed er irrelevant. □

Kan vi også gøre noget snedigt, hvis vi ikke kender sandsynlighederne p_i? Svaret er ja, og den enkle strategi er den såkalde »flyt-forrest«-heuristik. Den virker på følgende måde. Antag, at en søgning leder efter indgang i og finder den på plads ℓ_i. Hvis $\ell_i = 1$, så er vi godt tilfredse og gør ikke mere. Ellers flytter vi ingang i til første plads og forskyder indgangene på pladsern $1, \ldots, \ell_i - 1$ en position bagud. På denne måde håber vi, at hyppigt søgte indgange står tidligt i ordningen, mens sjældent søgte indgange befinder sig længere bagud. Lad os analysere den forventede opørsel af flyt-forrest-heuristiken.

Sætning 2.5. *Ser man bort fra den første søgning efter hvert element, så er den gennemsnitlige omkostning ved søgning med flyt-forrest-heuristikken højst dobbelt så stor som den gennemsnitlige omkostning ved en optimal, fast ordning.*

Bevis. Vi går ud fra, at de n indgange i listen oprindeligt står i en bestemt, men vilkårlig rækkefølge. Vores stokastiske model er en følge af søgningsrunder, hvor hver runde *uafhængigt af tidligere runder* søger efter en indgang, og denne indgang er valgt sådan, at indgang i vælges med sandsynlighed p_i. Den første søgning efter indgang i afhænger hovedsagligt af den oprindelige rækkefølge og har derfor ikke særlig meget med heuristikken at gøre. Derfor ignorerer vi den første søgning efter indgang i og vedtager helt enkelt, at den har omkostning 1. [8]Betragt søgerunde t. Lad C_{ff} betegne omkostningen i runde t. For hvert i lad ℓ_i angive indgang is plads i listen i begyndelsen af runde t. Betragt de stokastiske variable $\ell_1, \ldots, \ell_n$; de afhænger kun af, hvad der er sket i de første $t-1$ runder. Hvis t søger efter indgang i, så er omkostningen $1 + Z_i$, hvor

$$Z_i = \begin{cases} \ell_i - 1 & \text{hvis } i \text{ blev søgt før runde } t\,, \\ 0 & \text{ellers.} \end{cases}$$

De stokastiske variable $Z_1, \ldots, Z_n$ afhænger kun af, hvad der er sket i de første $t-1$ runder. Vi har altså $C_{ff} = \sum_i p_i(1 + E[Z_i]) = 1 + \sum_i p_i E[Z_i]$. Vi skal nu bestemme $E[Z_i]$ for givet i. Definér hertil for alle $j \neq i$ indikatorvariablen

$$I_{ij} = \begin{cases} 1 & j \text{ står før } i \text{ i starten af runde } t, \\ & \text{og minst en af de to indgange blev søgt før runde } t \\ 0 & \text{ellers.} \end{cases}$$

Da gælder $Z_i \leq \sum_{j:\, j \neq i} I_{ij}$. (Hvis runde t er den første runde, som søger efter i, så er $Z_i = 0$. Hvis runde t ikke er den første runde, som søger efter i, så er $I_{ij} = 1$ for hvert j, som står før i i listen, dvs. $Z_i = \sum_{j:\, j \neq i} I_{ij}$.) Derfor gælder også $E[Z_i] \leq \sum_{j:\, j \neq i} E[I_{ij}]$. Vi skal altså bestemme middelværdien $E[I_{ij}]$ for hvert $j \neq i$.

[8] Hermed ignorer vi omkostninger på højst $n(n-1)$. Man kan vise, at den forventede afvigelse ved denne forenkling af argumentet for den $t$te søgning er $n \cdot \sum_i p_i(1-p_i)^{t-1}$. Lad $t > n$. Idet funktionen $p \mapsto p(1-p)^{t-1}$ har sit maksimum i $p = 1/t$, bliver summen maksimeret ved at sætte $p_1 = \cdots = p_{n-1} = 1/t$ og $p_n = 1 - (n-1)/t$. Afgivelsen er altså $n(n-1)(1/t)(1-1/t)^{t-1} < n^2/et$.

Hvis der ikke forekom nogen søgning efter hverken i eller j for runde t, har vi $I_{ij} = 0$. Ellers betragter vi den sidste runde $t_{ij} < t$, som søgte efter enten i eller j. Den betingede sandsynlighed for, at denne søgning var efter j i stedet for efter i, dvs. at $I_{ij} = 1$, er $p_j/(p_i + p_j)$. Samlet får vi: $E[I_{ij}] = \Pr I_{ij} = 1 \leq p_j/(p_i + p_j)$, dvs. $E[Z_i] \leq \sum_{j:\, j \neq i} p_j/(p_i + p_j)$.

Ved at summere over alle i får vi nu:

$$C_{ff} = 1 + \sum_i p_i E[Z_i] \leq 1 + \sum_{i,j:\, i \neq j} \frac{p_i p_j}{p_i + p_j} .$$

Bemærk nu, at termen $p_i p_j/(p_i + p_j) = p_j p_i/(p_j + p_i)$ optræder to gange i summen for hvert par (i, j) med $j < i$. For at forenkle notationen antager vi, at $p_1 \geq p_2 \geq \cdots \geq p_n$. Med $\sum_i p_i = 1$ fås

$$\begin{aligned} C_{ff} &\leq 1 + 2 \sum_{i,j:\, j<i} \frac{p_i p_j}{p_i + p_j} = \sum_i p_i \left(1 + 2 \sum_{j:\, j<i} \frac{p_j}{p_i + p_j} \right) \\ &\leq \sum_i p_i \left(1 + 2 \sum_{j:\, j<i} 1 \right) < \sum_i p_i 2i = 2 \sum_i p_i i = 2\, Opt \ . \end{aligned}$$

□

2.8 Randomiserede algoritmer

Lykke deltager i et underholdningsprogram på tv, som drejer sig om 100 æsker, nummereret fra 1 til 100, som hun har lov til at åbne i vilkårlig rækkefølge. Æske i indeholder m_i kr. Lykke kender ikke m_i inden hun har åbnet æske i. Forskellige æsker indeholder forskellige beløb. Reglerne er:

- I begyndelsen af spillet får Lykke 10 jetoner fra studieværten.
- Når hun åbner en æske, som indeholder et større beløb end alle tidligere åbnede æsker, betaler hun en jeton. [9]
- Når Lykke skal betale en jeton, men ikke har flere, ender spillet og Lykke har tabt.
- Hvis Lykke kan åbne samtlige æsker, får hun lov til at beholde alle pengene.

Æskerne er dekorerede med pudsige billeder, og studieværten forsøger sommetider at påvirke Lykke ved at foreslå, hvilken æske hun burde åbne hernæst. Lykkes tante, som aldrig går glip af et eneste afsnit at tv-programmet, kan fortælle hende, at spillet kun sjældent har en vinder. Lykke spørger sig selv, om det overhovedet kan betale sig at deltage i programmet. Findes der en

[9] Den først åbnede æskes beløb er per definition større end alle tidligerede åbnede æsker. Det koster altså Lykke under alle omstændigheder 1 jeton at åbne den første æske.

strategi, der giver en god gevinstchance? Og er studieværtens forslag værd at følge?

Lad os begynde med at analysere den åbenlyse algoritme, nemlig at Lykke altid følger studieværtens forslag. I værste fald vil han pege på æsker med voksende beløb. Så skal hun aflevere en jeton ved hver æske og har tabt, når spillet er nået til den 11. æske. Både kandidater og tv-publikum ville ærge sig over den fæle studievært, seertallene ville falde, og studieværten blive fyret. Værstefaldsanalysen hjælper os ikke meget videre. I bedste fald foreslår studieværten straks den bedste æske, dvs. den med det største beløb. Lykke ville være godt tilfreds, men programmet ville være for kort til en reklamepause og den venlige studievært ville også blive fyret. Bedstefaldsanalysen hjælper os altså heller ikke videre.

Efter en smule overvejelse indser vi, at spillet egentlig bare er en omformulering af problemet at finde foreløbige maksima fra foregående afsnit. Lykke bliver af med en jeton, hver gang hun støder på et nyt maksimum. Fra sidste afsnit ved vi, at det forventede antal foreløbige maksima i en tilfældig permutation er er det $n$te harmoniske tal H_n. For $n = 100$ gælder $H_n < 6$. Hvis studieværten altså foreslår æskerne i (uniformt) tilfældig orden, kommer Lykke af med lidt under 6 jetoner i gennemsnit. Men hvorfor skulle studienværten følge netop denne strategi? Der er jo fra hans perspektiv ingen grund til at nærmest garantere deltagerens sejr.

Løsningen for Lykke er at tage skæbnen i egen hånd ved at *åbne æskerne i tilfældig rækkefølge.* Hun vælger en æske tilfældigt og åbner den, vælger næste æske tilfældigt bland de uåbnede, åbner denne, og så videre. Hvordan vælger hun en tilfældig æske? Hvis der er k uåbnede æsker tilbage, vælger hun et tilfældigt heltal i området fra 1 til k, fx ved at dreje en snurretop, hvis rand er en regelmæssig k-kant, eller ved at bruge et lykkehjul med k muligheder. På denne måde skaber Lykke en tilfældig permutation af æskerne, og vi kan genbruge analysen fra foregående afsnit. I gennemsnit skal hun betale færre end 6 jetoner, og de 10 jetoner er næsten altid nok. Vi har netop beskrevet en *randomiseret algoritme*. Det er værd at understrege, at selvom den matematiske analyse er identisk med den situation, hvor studieværten anbefaler æskerne i tilfældig rækkefølge, så er konklusionen helt forskellig. I gennemsnitsscenariet er vi afhængige af, at studienværten vitterlig benytter en tilfældig rækkefølge. I så fald gælder analysen, men hvis han følger en anden strategi, gælder analysen ikke. Hvad der virkelig er foregået i programmet, kan man i bedste fald sige noget om efter rigtig mange udsendelser og kun i bakspejlet. Situationen i scenariet »Lykke bruger en randomiseret algoritme« er helt anderledes. Hun udfører nemlig selv de tilfældige hændelser og skaber derved en tilfældig permutation. Analysen gælder, uanset hvad studieværten foretager sig.

2.8.1 Formel model

Formelt udstyrer vi registermaskinenn med yderligere en instruktion: $R_i := \mathit{tilfældigtTal}(R_j)$ tildeler register R_i en tilfældig værdi i mængden $\{0, \ldots, k-1\}$, hvor k er indholdet af R_j. Alle k værdier har samme sandsynlighed, og udfaldet er uafhængigt at samtlige tildigere tilfældige valg.

I pseudokode skriver vi $v := \mathit{tilfældigtTal}(C)$, hvor v er en heltalsvariabel og C en heltalsværdi. Vi vedtager, at det tager konstant tid at vælge et tilfældigt tal. Algoritmer uden den slags tilfældige tildelinger kaldes *deterministiske*.
.

Kørselstiden for en randomiseret algoritme afhænger af dens tilfældige valg. Dermed er kørselstiden for instans i ikke længere et tal, men en stokastisk variabel, som afhænger af algoritmens tilfældige valg. Vi kan undgå at lade kørselstiden afhænge af de tilfældige valg ved at udstyre maskinen med et »stopur«. I begyndelsen af programkørslen sætter vi stopuret til værdien $T(n)$, hvor n angiver størrelsen af instansen. Under kørslen tæller vi uret ned skridt for skridt, og standser efter $T(n)$ skridt. Skulle programmet standses »i utide« på denne måde, kommer det ikke til at producere sit sædvanlige svar og kørslen bedømmes som mislykket.

En randomiseret algoritmes svar kan ligeledes afhænge af de tilfældige valg. Man kan naturligvis sætte spørgsmålstegn ved værdien af en algoritme, hvis svar på instans i afhænger af tilfældigheder – det kan fx være »ja« den ene dag og »nej« den anden. Hvis de to svar er lige sandsynlige, er algoritmens svar ganske rigtig værdiløs. Men hvis det rigtige svar er meget mere sandsynligt end det forkerte, så er svaret af nytte. Vi forklarer det med et eksempel.

Ane og Bo er forbundet over en langsom telefonledning. Ane kender et n-bitstal x, Bo kender et n-bitstal y. De ønsker at finde ud af, om de to tal er ens. Fordi deres kommunikationskanal er langsam, vil de kommunikere mindst mulig information. På den anden side spiller det ingen rolle, hvor store beregninger de hver især skal gennemføre.

Den oplagte løsning er, at Ane sender sit tal til Bo, som undersøger, om de to tal er ens, og sender resultatet tilbage. Dette kræver, at n bit information kommunikeres over ledningen. Alternativt kan Ane formidle sit tal ciffer for ciffer, Bob kan så sammenligne cifrene et ad gangen og annoncere resultatet, så snart han kender det, dvs. så snart to cifre på samme plads er forskellige. I værste fald skal alle n cifre kommunikeres, nemlig når begge tal er det samme. Vi skal nu se, at randomisering fører til en dramatisk forbedring. Vi vil blot kommunikere $O(\log n)$ bit inden resulatet om lighed eller ulighed kan annonceres, og svaret er korrekt med høj sandsynlighed.

Ane og Bo bruger følgende protokol. Hver av dem skaber en ordnet liste $p_1, \ldots, p_L$ af primtal, nemlig de L mindste primtal større end 2^k. Vi skal se forneden, hvordan værdierne for k og L skal bestemmes (i afhægighed af n). På denne måde sikres, at Ane og Bo skaber samme lister. Nu vælger Ane et tilfældigt indeks $i \in \{1, \ldots, L\}$ og sender i samt x_A mod p_i til Bo. Han

beregner $x_B \bmod p_i$. Hvis $x_A \bmod p_i \neq x_B \bmod p_i$, så melder han, at tallene er forskellige. Ellers, at de er ens. Hvis $x_A = x_B$, så er det klart, at Bos svar bliver »ens«. Hvis derimod $x_A \neq x_B$, men alligevel $x_A \bmod p_i = x_B \bmod p_i$, så vil Bo fejlagtigt erklære det to tal for ens. Hvor stor er sandsynligheden for denne fejl?

Fejlen opstår, når $x_A \neq x_B$ og $x_A \equiv x_B \pmod{p_i}$. Det sidste betingelse er ensbetydende med, at p_i deler differensen $D = x_A - x_B$. Absolutværdien af denne differens er højst 2^n. Idet hvert p_i er større end 2^k, indeholder Anes liste højst n/k primtal, som deler D. (Lad d være antallet af primtal i listen, som deler D. Da gælder $2^n \geq |D| \geq (2^k)^d = 2^{kd}$, altså $d \leq n/k$.) Derfor er fejlsandsynligheden højst $(n/k)/L$. Denne sandsynlighed kan gøres vilkårlig lille ved at lade L være tilstrækkelig stor. Hvis vi fx vil have fejlsandsynlighed højst $0{,}000\,001 = 10^{-6}$, sætter vi $L = 10^6(n/k)$.

Men hvordan vælges k? For tilstrækkeligt store k er der bland tallene i $\{2^k, \ldots, 2^{k+1} - 1\}$ omtrent $2^k / \ln(2^k) \approx 1{,}4427 \cdot 2^k/k$ mange primtal.[10] Hvis altså $2^k/k \geq 10^6 n/k$, består listen udelukkende af tal med $k+1$ bit. Betingelsen $2^k \geq 10^6 n$ er ensbetydende med $k \geq \log n + 6 \log 10$. Dette valg af k medfører, at Ane sender $\log L + k + 1 = \log n + 12 \log 10 + 1$ bit. *Dette udgør en eksponentiel forbedring i forhold til den naive protokol!*

Hvordan kan vi reducere fejlsandsynligheden yderligere, fx til højst 10^{-12}? Vi kan selfølgeligt bare sætte $L = 10^{12} n/k$ i argumentet foroven. En alternativ løsning er at køre protokollen to gange, og annoncere, at de to tal er ens, hvis begge udførsler erklærer tallene for ens. Denne totrinsprotokol tager fejl, kun hvis begge trin tog fejl; derfor er fejlsandsynligheden højst $10^{-6} \cdot 10^{-6} = 10^{-12}$.

Opgave 2.12. Sammenlign kørselstiden for de to metoder, som opnår fejlsandsynlighed 10^{-12}.

Opgave 2.13. Det virker fjollet, at Ane og Bo skal skabe enormt lange lister af primtal. Diskuter i stedet følgende variation af protokollet: Ane vælger et tilfældigt tal p med $k+1$ bit (og foranstillet 1) og afgør, om p er primsk. Hvis dette ikke er tilfældet, gentager hun forsøget. Hvis p er et primtal, sender hun p og p und $x_A \bmod p$ til Bo. Bo udregner $x_B \bmod p$ og sammenligner.

Opgave 2.14. Antag, at du har en randomiseret algoritme, som giver det forkerte svar med sandsynlighed højst $\frac{1}{4}$. Gentag algoritmen k gange og returner flertalsafgørelsen, dvs. det svar, som optræder mere end $k/2$ gange, hvis det findes. Beregn fejlsandsynligheden som funktion af k. Find et præcist udtryk for $k = 2$ og $k = 3$, og en øvre grænse for større k. Hvordan skal man vælge k for at opnå en fejlsandsynlighed, som er mindre en et givet ϵ?

[10] For $x \geq 1$ lad $\pi(x)$ betegne antal primtal, som er mindre end eller lig med x. For eksempel er $\pi(10) = 4$, fordi der findes fire primtal (2, 3, 5 og 7), som er mindre end eller lig med 10. Ifølge talteoriens Primtalssætning gælder $x/(\ln x + 2) < \pi(x) < x/(\ln x - 4)$ for all $x \geq 55$.

2.8.2 Las-Vegas- og Monte-Carlo-algoritmer

Der findes i grunden to varianter af randomiserede algoritmer, nemlig Las-Vegas-algoritmer og Monte-Carlo-algoritmer. En *Las-Vegas-algoritme* giver altid det rigtige svar, men dens kørselstid på input i er en stokastisk variabel. Vores løsning til tv-programmet med Lykke er en Las-Vegas-algoritme, hvis vi fortsætter med at lade den lede efter den rigtige æske; men antallet af foreløbige maksima, dvs. antallet af betalte jetoner, er en stokastisk variabel. En Monte-Carlo-algoritme har altid den samme kørselstid, men der findes en positiv sandsynlighed for, at den giver et forkert svar. Sandsynligheden for et forkert svar er højst $\frac{1}{4}$. Algoritmen, som sammenligner to tal over en telefonforbindelse, er en Monte-Carlo-algoritme. Opgave 2.14 viser, at fejlsandsynligheden for en Monte-Carlo-algoritme kan gøres vilkårlig lille ved gentalse.

Opgave 2.15. Antag, at du har en Las-Vegas-algoritme med forventet kørselstid $t(n)$. Lad den køre i $4t(n)$ skridt. Skulle den give et svar inden for denne tid, returneres dette svar, ellers returneres en vilkårlig værdi. Vis, at den resulterende algoritme er en Monte-Carlo-algoritme.

Opgave 2.16. Antag, at du har en Monte-Carlo-algoritme med kørselstid $m(n)$, som giver det rigtige svar med sandsynlighed mindst p, og en deterministisk algoritme, som i tid $v(n)$ kan afgøre, om Monte-Carlo-algoritmens svar er korrekt. Beskriv, hvordan de to algoritmer kan kombineres til et Las-Vegas-algoritme med forventet kørselstid $(m(n) + v(n))/(1 - p)$.

Afslutningsvis vender vi tilbage til Lykkes tv-program. Hun har 10 jetoner og forventer at skulle bruge færre end 6 af dem. Hvor sikker kan hun være på at vinde? Vi skal altså vurdere sandsynligheden for, at M_n er mindst 11, idet Lykke taber hvis og kun hvis antallet af foreløbige maksima i følgen af åbnede æskebeløb er 11 eller mere. Hertil bruges *Markovs ulighed*: Lad X være en ikke-negativ stokastisk variabel og $c \geq 1$ en vilkårlig konstant. Da gælder $\Pr(X \geq c \cdot E[X]) \leq 1/c$, se (A.4). Vi bruger dette med $X = M_n$ og $c = \frac{11}{6}$ og får

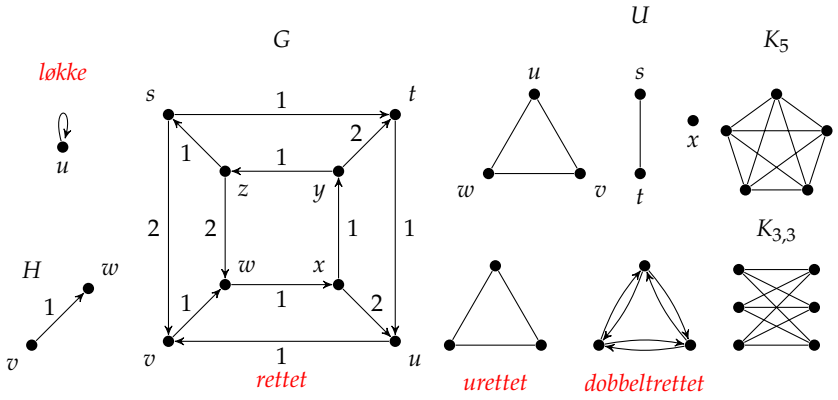
$$\Pr(M_n \geq 11) \leq \Pr(M_n \geq \tfrac{11}{6} E[M_n]) \leq \tfrac{6}{11}\,,$$

så Lykkes vinder med sandsynlighed større end $5/11$.

2.9 Grafer

Grafer er et overordentligt nyttigt koncept i algoritmikken. Vi bruger dem altid, når vi skal modelle objekter og deres indbyrdes relationer; i grafterminologien hedder objekterne *knuder*, og relationerne mellem par af knuder hedder *kanter*. Temmelig åbenlyse anvendelser af grafer er vejkort og kommunikationsnetværk, men der findes også mere abstrakte anvendelser. For eksempel kan knuderne repræsentere de delopgaver, som skal udføres, når

man bygger et hus, fx »byg murene« eller »sæt vinduerne i«, og kanterne kan beskrive præcedens- eller forrangsrelationer, fx »murene skal bygges, inden vinduerne kan sættes i«. Vi kommer også til at lære mange datastukturer at kende, i hvilke vi på en naturlig måde kan opfatte hvert objekte som en knude og hver peger som en rettet kant fra det objekt, den går ud fra, til det objekt, den peger på.



Figur 2.6. Nogle grafer.

Når mennesker gør sig overvejelser om grafer, finder de det normalt bekvemt at arbejde med billeder, som fremstiller knuder som små cirkler og kanter som lige eller bøjede linjer eller pile. For at bearbejde grafer algoritmisk har vi dog brug for en mere matematisk orienteret notation: en *rettet graf* $G = (V, E)$ er et par bestående af en mængde V af *knuder* (også kaldt *hjørner* eller *punkter*) og en mængde $E \subseteq V \times V$ af *kanter* (også kaldt *buer*). Sommetider støder man på ordet »*digraf*« for »rettet graf« (fra eng. *directed graph*).[11] Fig. 2.7 viser den rettede graf $G = (\{s,t,u,v,w,x,y,z\}, \{(s,t),(t,u),(u,v),(v,w),(w,x),(x,y),(y,z),(z,s),(s,v),(z,w),(y,t),(x,u)\})$. I hele bogen gælder $n = |V|$ og $m = |E|$, medmindre n og m er tildelt en anden betydning. En kant $e = (u,v) \in E$ repræsenterer en forbindelse fra u til v. Knuderne u og v kaldes for es endeknuder eller endepunkter, u kaldes es *startknude* og v for es *målknude*. Vi siger at e er *incident med* eller *hosliggende til* u og v, at u og v *ligger på* e og at u og v er *naboer*. Envidere er e *udgående fra* v og *indkommende til* u. Sommetider siger vi at e er *udkant* til v hhv. *indkant* til u. Ofte kalder vi også

[11] Ovs. anm.: Man kan desuden støde på »orienteret graf« brugt i betydningen »rettet graf«. Desværre er »orienteret graf« dog et veletableret begreb i grafteori og betyder noget andet: En rettet graf er *orienteret*, hvis den ikke indeholder et par af knuder, som er forbundet af to kanter i hver sin retning. Orienterede grafer er altså rettede, men rettede grafer er ikke nødvendigvis orienterede, idet de kan indeholde modsatrettede kanter.

u for en (umiddelbar) *forgænger* til v og v for en (umiddelbar) *efterfølger* for u. Vi udelukker for det meste specialtilfældet hvor (v, v) er en *løkke*, medmindre andet er sagt.

En knudes *udgrad* v er antallet af kanter, som udgår fra v, dens *indgrad* er antallet af kanter, der indgår til v. Formelt har vi $\mathit{udgrad}(v) = |\{\, u \in V : (v,u) \in E \,\}|$ og $\mathit{indgrad}(v) = |\{\, u \in V : (u,v) \in E \,\}|$. Fx har knuden w i grafen G i fig. 2.7 indgrad 2 og udgrad 1.

En *dobbeltrettet graf* er en rettet graf, som for hver rettede kant (u, v) indeholder modkanten (v, u). En *urettet graf* kan man opfatte som en strømlinet udgave af en dobbeltrettet graf, hvor vi skriver kantparret (u, v), (v, u) som parmængden $\{u, v\}$. Figur **??** viser en urettet graf med tre kunder og dens dobbeltrettede modpart. De fleste grafteoretiske begreber defineres for urettede grafer præcis som for rettede grafer; derfor koncentrerer vi os i dette afsnit om rettede grafer og nævner urettede grafer kun, når der findes en afvigelse. Fx har en urettet graf kun halvt så mange kanter som sin dobbeltrettede modpart. Når $\{u, v\}$ er en kant, kalder vi u for vs *nabo*; i den dobbeltrettede udgave er u både forgænger og efterfølger til v. En knudes udgrad og indgrad i en urettet graf er altid ens, så den kaldes bare knudens *grad* (eller *valens*). Urette grafer er vigtige, fordi retninger ofte ikke spiller nogen rolle, og mange problemer i urettede grafer kan løses enklere end i almene rettede grafer; visse problemer giver overhovedet kun mening i urettede grafer.

En graf $G' = (V', E')$ er en *delgraf* af grafen $G = (V, E)$, hvis $V' \subseteq V$ og $E' \subseteq E$. Givet $G = (V, E)$ og en knudedelmængde $V' \subseteq V$ defineres den *inducerede delgraf* G' som $G' = (V', E \cap (V' \times V'))$. I fig. 2.7 inducerer knudemængden $\{u, v\}$ i G delgrafen $H = (\{v, w\}, \{(v, w)\})$. Grafen G' kaldes *knudeinduceret*. En kantdelmængde $E' \subseteq E$ inducerer delgrafen (V, E').

Ofte knyttes yderligere information til knuder og kanter. Især har vi ofte behov for *kantvægte* eller *kantomkostninger* $c : E \to \mathbf{R}$, som tildeler en talværdi til hver kant. Fx har kanten (z, w) i grafen G i fig. 2.7 vægten $c(\{z, w\}) = 2$. Læg mærke til, at en kant $\{u, v\}$ i en urettet graf kun kan have én vægt, mens der i en dobbeltrettet graf kan gælde $c((u, v)) \neq c((v, u))$.

De sidste par afsnit indeholdt mange tørre definitioner på i stram form. Hvis man efterlyser definitionerne »i aktion«, kan finde algoritmer på grafer i kapitel 8. Men selv her skal materialet straks blive lidt mere interessant.

Et vigtigt højere begreb i grafteorien er *vejen*. En *vej* (eller *sti*) $p = \langle v_0, \ldots, v_k \rangle$ i en rettet graf $G = (V, E)$ er en følge af knuder, således at på hinanden følgende knuder er forbundet med kanter i E, dvs. at der gælder $(v_0, v_1) \in E$, $(v_1, v_2) \in E$, $\cdots$, $(v_{k-1}, v_k) \in E$. Tallet $k \geq 0$ kaldes for *længden* af p; vi siger at vejen *går* (eller *leder*) fra v_0 til v_k. Sommetider angiver vi vejen som en følge af dens kanter. Fx er $\langle u, v, w \rangle = \langle (u, v), (v, w) \rangle$ i fig. 2.7 en vej af længde 2. I en urettet graf er $p = \langle v_0, \ldots, v_k \rangle$ en vej, hvis p er en vej i den tilsvarende dobbeltrettede graf. I grafen U i fig. 2.7 er $\langle u, w, v, u, w, v \rangle$ en vej af længde 5. En vej er simpel [12], hvis dens knuder bortset måske fra v_0 og v_k

[12] I mange fremstillinger bruges »vej« for »simpel vej« og »vandring« for »vej«.

er parvist forskellige. I grafen G i fig. 2.7 er $\langle z, w, x, u, v, w, x, y\rangle$ en vej, som ikke er simpel. Det ses nemt, at hvis der findes en vej fra u til v, så findes der også en simpel vej fra u til v.

En *kreds* (eller *cykel*) er en vej af længde mindst 1, hvis første og sidste knude er den samme. I urettede grafer kræves yderligere, at den første og sidste kant er forskellige, hvorved en kreds har længde mindst 3. (I fig. 2.7 indeholder G kredsen $\langle u, v, w, x, y, z, w, x, u\rangle$ og U kredsen $\langle u, w, v, u, w, v, u\rangle$.) En *simpel* kreds er en kreds, hvis interne knuder er forskellige. Det ses let, at hvis G har en kreds, så har den også en simpel kreds. En simpel kreds, som besøger alle knuder i grafen, kaldes en hamiltonkreds. (I fig. 2.7 har G hamiltonkredsen $\langle s, t, u, v, w, x, y, z, s\rangle$ og U hamiltonkredsen $\langle w, u, v, w\rangle$.)

Begreberne »vej« og »kreds« tillader os at definere endnu mere komplekse begreber. En rettet graf er *stærkt sammenhængende*, hvis der fra hver knude u og til hver knude v findes en vej. Grafen G i fig. 2.7 er stærkt sammenhængende. En *stærk sammenhængskomponent* er en maksimal, knudeinduceret, stærkt sammenhængende delgraf af G. Hvis vi i fig. 2.7 fjerner kanten (w, x) i G, opstår en graf uden kredse. En graf uden kredse hedder *kredsfri* eller *acyklisk*. En rettet, acyklisk graf betegnes kort *rag*. Hver stærke sammenhængskomponent i en rag består af præcis en knude. En urettet graf kaldes *sammenhængende*, , hvis der for hvert par u, v af knuder findes en (urettet) vej fra u til v. Sammenhængskomponenterne er de maksimale, sammenhængdende delgrafer. Inden for en sammenhængskomponent er hvert par af knuder forbundet af en vej, mens der ikke fører nogen vej fra nogen sammenhængskomponent til en anden. I fig. 2.7 har graphen G sammenhængskomponenterne $\{u, v, w\}$, $\{s, t\}$ og $\{x\}$. Knudemængden $\{u, w\}$ inducerer en sammenhængende delgraf, men denne er ikke maksimal og udgør derfor ingen sammenhængskomponent.

Opgave 2.17. Beskriv mindst ti grundlæggende forskellige anvendelser, som kan modelleres med grafer. (Bil- og cykelveje gælder *ikke* som grundlæggende forskellige!) Mindst fem anvendelse bør ikke forekommme i bogen.

Opgave 2.18. En graf er *planær*, hvis den kan tegnes på et stykke papir på sådan måde, at ingen kanter krydser hinanden. Forklar, hvorfor vejnet ikke nødvendigvis er planære. Vis at K_5 og $K_{3,3}$ fra fig. 2.7 ikke er planære.

2.9.1 En første grafalgoritme

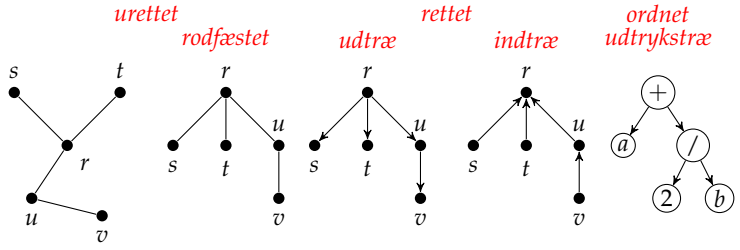
Det er tid til en eksempelalgoritme. Vi skal betragte en algoritme, som afgør, om en rettet graf G er acyklisk eller ej. Udgangspunktet er den enkle observation, at en knude v med udgrad 0 ikke kan ligge på en kreds. Hvis vi altså stryger v (og dets indgående kanter), opnår vi en graf G', som er acyklisk, hvis og kun hvis G er acyklisk. Denne transformation gentages nu, indtil en af to siutationer opstår: Enten ender vi i den tomme graf, dvs. grafen uden knuder, som jo sikkert er acyklisk. Eller vi ender i en graf G^*, hvor hver knude har udgrad mindst 1. I det sidste tilfælde er det let at finde en kreds: Begynd med en vilkårlig knude. Fra hver knude, som vi netop har nået, fortsætter vi

ad en vilkårlig kant til den næste knude, indtil vi møder en knude v', som vi har mødt før. Den herved konstruerede vej har formen $\langle v, \ldots, v', \ldots, v' \rangle$, hvis suffiks $\langle v', \ldots, v' \rangle$ danner en kreds. For eksempel har grafen G i fig. 2.7 ingen knude med udgrad 0. For at finde en kreds, kunne vi begynde i knude z og følge turen $\langle z, w, x, u, v, w \rangle$, vil vi støder på knude w for anden gang. Herved har vi fundet kredsen $\langle w, x, u, v, w \rangle$. Anderledes forløber det, når vi betragter grafen G uden kanten (w, x). Da findes ingen kreds, og algoritmen stryger samtlige knuder i rækkefølgen w, v, u, t, s, z, y, x. I kapitel 8 skal vi se, hvordan man kan repræsentere grafen på en måde, så den foroven skitserede algoritme kører i lineær tid, dvs. i tid $O(|V| + |E|)$. (Se hertil opgave 8.3.) Algoritmen kan let udvides til at være certificerende: Når algoritmen finder en kreds, er grafen sikkert cyklisk, så det er let at afgøre, om den fundne knudefølge udgør en kreds i G. Når algoritmen fjerner samtlige knuder i G, nummererer vi knuderne i den rækkefølge, de blev strøget. Når vi fjerner en knude v med udgrad 0, må alle kanter i G, som har v som startknude, gå til knuder, som blev fjernet tidligere, og som derfor har et lavere nummer end v. Nummereringne beviser altså, at G er acyklisk: Langs hver kant er knudenummeret faldende, hvilket igen er nemt at certificere.

Opgave 2.19. For vilkårligt n, find en rag med $\frac{1}{2}n(n-1)$ kanter. Vis, at der ikke findes en rag med flere end $\frac{1}{2}n(n-1)$ kanter.

2.9.2 Træer

En urettet graf er et *træ*, hvis der mellem hvert par af knuder findes netop én vej; fx som i fig. 2.8. En urettet graf er en *skov*, hvis der mellem hvert par af knuder findes højst én vej. Det ses let, at hver sammenhængskomponent i en skov er et træ.



Figur 2.7. Forskellige slags træer. Fra venstre til højre ses et urettet træ, et urettet, rodfæstet træ, et træ rettet væk fra og et rettet mod roden og et aritmetisk udtryk.

Lemma 2.1. *For en urettet graf G er følgende ækvivalent:*

(1) G er et træ

(2) G er sammenhængende og har præcis $n-1$ kanter.
(3) G er sammenhængende og acyklisk

Bevis. I et træ findes mellem hvert par af knuder én entydig bestemt vej. Derfor er træet sammenhængende og indeholder ingen kreds. Omvendt gælder: Hvis G indeholder to knuder, som er forbundet med forskellige veje, så indeholder G en kreds. (Betragt hertil u og v og to forskellige veje fra u til v sådan at summen af længderne af disse veje er minimal over alle valg af knudepar og veje. Ved minimalitet indses, at vejene må forlade u ad to forskellige kanter, og kan først mødes igen i v. Derfor danner vejene en enkel kreds.) Hermed har vi vist ækvivalens af (1) og (3). Vi viser nu, at (2) og (3) også er ækvivalente. Hertil betrager vi en sammenhængde graf $G = (V, E)$ med $m = |E|$. Vi gennemfører nu følgende tankeeksperiment: Vi begynder med den tomme graf med knudemængde V og ingen kanter og tilføjer en kant fra E efter den anden. Tilføjelsen af hver kant reducerer antallet af sammenhængskomponenter med højst 1. Idet vi begyndte med $n = |V|$ komponenter, og ender med en eneste komponent, må antallet af kanter være mindst $n-1$. Antag nu, at tilføjelsen af kanten $e = \{u, v\}$ ikke formindsker antallet af komponenter med 1. Det må betyde, at u og v allerede er forbundet med en vej, og tilføjelsen af e vil derfor danne en kreds. Hvis G er acyklisk, kan dette altså ikke ske, og vi har $m = n-1$. Derfor følger (2) af (3). Antag nu, at G er sammenhængende og har præcis $n-1$ kanter. Igen tilføjer vi disse kanter til en oprindeligt tom graf. Når kanten $e = \{u, v\}$ lukker en kreds, er u og v allerede forbundet, og tilføjelsen af e reducerer ikke antallet af komponenter. Vi skulle altså ende med mere end én komponent, hvilket er en modstrid. Derfor følger (3) af (2). □

Lemma 2.8 gælder ikke for rettede grafer. Især kan en rag have meget mere end $n-1$ kanter. En rettet graf kaldes *udtræ* med rod r, hvis der fra r findes netop én vej til hver knude. Den kaldes *indtræ*, hvis der findes netop én vej fra hver knude til r. Figur 2.8 viser nogle eksempler. *Dybden* af en knude v i et træ med rod er længden af vejen mellem v og roden. *Højden* (også kaldt *dybden*) af et træ med rod er den maksimale dybde av nogen knude.

Man kan rodfæste et urettet træ ved at erklære en vilkårlig knude til rod. Dataloger har den spøjse vane, at tegne træer så roden er foroven og alle kanter forløber oppefra og ned. For rodfæstede træer er det almindeligt, at betegne forholdet mellem knuder med ord, som er taget fra familjerelationer. Kanter forløber mellem en *forælder* og dets *børn*. Knuder med samme forælder hedder *søskende*. En knude uden børn kaldes et *blad*. En knude, om ikke er et blad, kaldes en *indre* knude. Når u ligger på vejen fra roden til en knude v, og u er forskellig fra v, kaldes u en *ane* til v, tilsvarende kaldes v en *efterkommer* til u. En knude u og alle dens efterkommere danner et *undertræ* med rod u. Vi betragter et eksempel: I de rettede træer i fig. 2.8 er r roden; s, t og v er blade; s, t og u er søskende, fordi de er børn til samme forælder r; u er en indre knude; r og u er ane til v; s, t, u og v er efterkommmerne til r; v og u danner et deltræ med rod u.

2.9.3 Ordnede træer

Træer egner sig til at repræsentere hierarkier. Betragt fx det aritmetiske udtryk $a + 2/b$. Vi ved godt, at dette skal betyde, at a og $2/b$ skal lægges sammen. Men det er slet ikke så let at udlæse fra tegnfølgen $\langle a, +, 2, b\rangle$. Fx skal man iagttage, at division udføres før addition. Oversætteren isolerer denne syntaktiske viden i en *parser*, som ud fra formelteksten skaber en struktureret, træbaseret repræsentation. Vores eksempeludtryk ville føre til træet til højre i fig. 2.8. Disse træer er rettede og – i modsætning til grafteoretiske træer – desuden *ordnede*, dvs. at rækkefølgen af hver knudes børn ligger fast. I vores eksempel er a-knuden det første barn af roden, /-kunden er det andet barn.

Funktion *eval*(r) : **R**
  hvis r er et blad **så returner** det i r gemte tal
  ellers
    v_1 := *eval*(første barn til r)
    v_2 := *eval*(andet barn til r)
    returner v_1 *operator*(r) v_2

Figur 2.8. Rekursiv evaluering af et utrykstræ med rod r.

Udtrykstræer kan enkelt evalueres med en rekursiv algoritme. Fig. 2.9 angiver en algoritme for evalueringen af udtrykstræer, hvis blade indeholder tal, og hvis indre knuder indeholder operationer som $+$, $-$, $\cdot$ eller /.

I denne bog vil vi møde mange andre eksempler på ordnede træer. I kapitel 6 og 7 bruges de til at illustrere grundlæggende datastrukturer, i kapitel 12 til den systematiske gennemgang af løsningsrum.

2.10 P og NP

Hvornår kalder vi en algoritme for »effektiv«? Findes der problemer, for hvilke der ikke findes effektive algoritmer? Selvfølgeligt er det lidt af en smagssag, hvor man præcis vil trække grænsen mellem »effektive« og »ueffektive« algoritmer. Men det har vist sig, at følgende adskillelse er nyttig: En algoritme $\mathcal{A}$ bruger *polynomiel tid* eller er en *polynomialtidsalgoritme*, hvis der findes et polynomium $p(n)$, således at udførelsestiden af $\mathcal{A}$ på instanser af størrelse n tilhører $O(p(n))$. Medmindre andet er sagt, vil vi måle instansstørrelsen i antal bit. Vi siger at et problem *kan løses i polynomiel tid*, hvis der findes en polynomialtidsalgoritme for det. Nu sætter vi lighedstegn mellem »kan løses effektivt« og »kan løses i polynomiel tid«. En af fordelene ved denne definition er, at implementationsdetaljer normalt ikke spiller nogen rolle. For eksempel er det underordnet, om en snedig datastruktur er i stand til at reducere udførelsestiden af en $O(n^3)$-algoritme med en faktor n. Alle kapitler i denne bog, med undtagelse af kapitel 12, drejer sig om effektive algoritmer.

Der findes mange problemer, for hvilke der findes algoritmer, uden at man kender en *effektiv* algoritmer for dem. Vi vil nævne seks eksempler:

- Hamiltonkredsproblemet: Givet en urettet graf. Afgør, om den indeholder en hamiltonkreds.
- Opfyldningsproblemet for boolske formler: Givet en boolsk formel på *konjunktiv normalform*. Afgør, om der findes en opfyldende tildeling. – Her er en boolsk formel på konjunktiv normalform en konjunktion $C_1 \wedge \cdots \wedge C_k$ af *klausuler*. En klausul er en disjunktion $(\ell_1 \vee \cdots \vee \ell_h)$ af *literaler*, og en literal er en boolsk variabel x_i eller en negeret boolsk variabel $\neg x_i$. Et eksempel på en klausul er $(x_1 \vee \neg x_3 \vee \neg x_9)$. En tildeling knytter en boolsk værdi til hver variabel, den er *opfyldende*, hvis den gør formlen sand.
- Klikeproblemet: Givet en urettet graf og et naturligt tal k. Afgør, om grafen indeholder en fuldstændig delgraf (også kaldt en *klike*) med k knuder. – En graf kaldes *fuldstændig*, hvis hvert par af knuder er forbundet med en kant; et eksempel er »5-kliken« K_5 i figur. **??**.
- Rygsæksproblemet: Givet n par (w_i, p_i) af naturliget tal for $i \in \{1, \ldots, n\}$ og to naturlige tal M og P. Afgør, om der findes en delmængde $I \subseteq \{1, \ldots, n\}$ af indeks, som opfylder $\sum_{i \in I} w_i \leq M$ og $\sum_{i \in I} p_i \geq P$. – Man skal tænke på dette som n givne objekter, hvor objekt i har volumen w_i og profit p_i, og som man prøver at pakke i en rygsæk med volumen M for at opnå en samlet profit på mindst P.
- Handelsrejsendeproblemet: Givet en kantvægtet urettet graf og det naturligt tal C. Afgør, om grafen indeholder en hamiltonkreds af længde højst C. (Se nærmere i afsnit **??**.)
- Graffarvningsproblemet: Givet en urettet graf og et naturligt tal $k > 0$. Afgør, om der findes en farving af knuderne i grafen med k farver, således hvert par af naboknuder har forskellige farver.

Bare fordi vi ikke kender nogle effektive algoritmer for noget af disse seks problemer, kan vi selvfølgeligt ikke udelukke, at den slags algoritmer alligevel findes. Man ved det helt enkelt ikke. Især har vi ingen matematiske beviser for, at der for disse problemer ikke kan findes en effektiv algoritme. Generelt er det meget vanskeligt at bevise, at der for et givet problem ikke kan findes en algoritme inden for en bestemt tidsgrænse. (Enkle eksempler på den slags nedre grænser skal vi dog lære at kende i kapitel 5.3.) De fleste algoritmikere tror dog ikke, at der findes effektive algoritmer for de seks problemer.

Teorien for beregningskompleksitet, bland dataloger ofte bare kaldt »kompleksitetsteorien«, giver en interessant måde at argumentere for disse problemers sværhed. Teorien sammenfatter algoritmiske problemer i store grupper, kaldt »kompleksitetsklasser«, således at problemer i samme klasse er ækvivalente med hensyn til deres kompleksitet. Især findes der en stor gruppe af ækvivalente problemer, som kaldes *NP-fuldstændige*. Her er »NP« en forkortelse for »nondeterministisk polynomiel tid«. Det er unødvendigt for forståelsen af resten af afsnittet, om læseren kender begrebet »nondeterministisk polynomiel tid« eller ej. De seks problemer foroven er allesammen NP-fuldstændige,

ligesom mange andre naturlige problemer. I resten af asnittet vil vi definere klassen NP og klassen af NP-fuldstændige problemer formelt. For en grundig fremstilling af teorien henviser vi til lærebøger i beregningsteori og kompleksitetsteori [3, 18, 50, 58].

Som sædvandligt i kompleksitetsteorien går vi ud fra, at instansen til problemet er kodet som streng over et fast, endeligt alfabet Σ med $|\Sigma| \geq 2$. (Man kan tænke sig ASCII- eller Unicode-alfabeterne, eller binærrepræsentationen af disse alfabeter. I det sidste tilfælde er $\Sigma = \{0,1\}$.) Mængen af strenge (eller »ord«) med bogstaver fra Σ betegnes Σ^*. For $x = a_1 \cdots a_n \in \Sigma^*$ bruger vi antallet $|x|$ af tegn i x som størrelsesmål, med konventionen $n = |x|$.[13] Et *beslutningsproblem* er en delmængde $L \subseteq \Sigma^*$. Med $\chi_L \colon \Sigma^* \to \{0,1\}$ (læs: »chi«, udtalt med k) betegner vi den *karakteristiske funktion* af L, definieret som

$$\chi_L(x) = \begin{cases} 1\,, & \text{hvis } x \in L\,, \\ 0\,, & \text{hvis } x \notin L\,. \end{cases}$$

Et beslutningsproblem kan *løses i polynomiel tid*, hvis dens karakteristiske funktion kan beregnes i polynomiel tid. Med P betegnes klassen af beslutningsproblemer, som kan løses i polynomiel tid. Et beslutningsproblemer L tilhører klassen NP, hvis der findes et prædikat $Q(x,y)$, (dvs. en mængde $Q \subseteq (\Sigma^*)^2$) og en polynomium p, således at der gælder følgende:

(1) For hvert $x \in \Sigma^*$ gælder $x \in L$, hvis og kun hvis der eksisterer $y \in \Sigma^*$ med $|y| \leq p(|x|)$ og således at $Q(x,y)$ gælder;
(2) den karakteristike funktion for Q kan beregnes i polynomiel tid.

Hvis $x \in L$ og y opfylder $Q(x,y) = \mathit{sand}$, kalder vi y for et *vidne* eller *certifikat* for x (y bevidner altså den omstændighed, at $x \in L$). For vores eksempelproblemer kan man let vise, at de tilhører NP. For hamiltonkredsproblemet er vidnet en hamiltonkreds i instansgrafen. Et vidne for en boolsk funktion er en tildeling af de variable, som gør formlen sand. At en instans rygsæksproblemet kan løses, bevidnes af en delmænge af objekterne, hvis samlede volumen får plads i rygsækken og som samlet udgør mindst den ønskede profit.

Opgave 2.20. Vis, at klikeproblemet , handelsrejsendeproblemet og graffarvningsproblemet tilhører NP.

Man formoder, at P er en ægte delmængde af NP. Selvom der findes gode argumenter for denne formodning, som vi straks skal se, er den ikke bevist. Et

[13] Undtagelsen er grafer. I denne bog, ligesom i resten af literaturen, bruges n for at betegne antallet af *knuder* i grafen, se afsnit **??**. Størrelsen af en graf med n knuder og m kanter er derimod $\Theta(n+m)$ eller $\Theta(n^2)$, afhængigt af den valgte repræsentation (se afsnit **??**). Graferne i denne bog er næsten udelukkende »simple« (der gælder $E \subseteq V \times V$, dvs. vores grafer har ingen multikanter i samme retning), så vi har $m \leq n^2$. Derfor er grafers størrelse polynomiel i n, og den uheldige sammenblanding af n og $n+m$ gør ingen forskel med hensyn til polynomialtidsreduktioner.

bevis for formodningen ville medføre, at der ikke findes effektive algoritmer for NP-fuldstændige problemer.

Beslutningsproblemet L kan *reduceres i polynomiel tid* (eller bare *reduceres*) til beslutningsproblemet L', hvis der findes en funtion g, som kan beregnes i polynomiel tid, og for hvilken der gælder for alle $x \in \Sigma^*$, at $x \in L$ hvis og kun hvis $g(x) \in L'$. Hvis L kan reduceres til L', og L' tilhører P, indses nemt at L også tilhører P. (Bevis: Givet en algoritme for reduktionensfunktionen g med polynomiel tidsgrænse $p(n)$ og en algoritme for $\chi_{L'}$ med polynomiel tidsgrænse $q(n)$. Så har vi følgende algoritme for χ_L. På input x beregner vi $g(x)$ i tid højst $p(|x|)$ med den første algoritme, hvorefter vi bruger den anden algoritme for at afgøre om $g(x) \in L'$. Afgørelsen tager tid $q(|g(x)|)$. For at begrænse længden af strengen $g(x)$ er det nok an indse, at den første algoritme i tid $p(|x|)$ højst kan skabe en streng af længde $p(|x|)$; strengt taget er dette et udsagn om maskinmodellen, som behøver mindst en tidsenhed for at skabe et tegn i den nye streng og fx gemme den i en lagercelle for sig – detaljer for forskellige repræsentationer af bl.a. tegnfølger findes i kapitel **??**. Vi har altså, at $|g(x)| \leq |x| + p(|x|)$, så den samlede udførelsestid er $p(|x|) + q(|x| + p(|x|))$, hvilket er et polynomium i $|x|$.) På lignende måde kan man indse, at reducibilitet er en transitiv relation. Beslutningsproblemetet L kaldes *NP-svært*, hvis *hvert* problem i NP kann reduceres til L i polyniel tid. Et problem kaldes *NP-fuldstændigt*, hvis det er NP-svært og tilhører NP. Umiddelbart virker det meget vanskeligt at bevise, at der overhovedet findes NP-fuldstændige problemer – blandt andet skal man jo vise, at *hvert* problem i NP kan reduceres til L. Men netop det gjordes i år 1971, da Cook og Levin viste (uafhænigt af hinanden), at opfyldningsproblemet for boolske formler er NP-fuldstændigt [10, 35]. Siden har det været »let« at etablere NP-fuldstændighed. Antag, at vi vil vise, at L er NP-fuldstændigt. Dertil skal vi vise to ting: (1) $L \in$ NP, og (2) der findes et andet problem L', hvis NP-fuldstændighed allerede er etableret, og som kan reduceres til L. Hvert nye NP-fuldstændige problem gør det lettere, at vise andre problemers NP-fuldstændighed. Vi ser nu nærmere på et eksempel på en reduktion.

Lemma 2.3. *Opfyldningsproblemet for boolske formler kan reduceres i polynomiel tid til klikeproblemet.*

Bevis. Lad $F = C_1 \wedge \cdots \wedge C_k$ være en boolsk formel på konjunktiv normalform, med $C_i = (\ell_{i1} \vee \ldots \vee \ell_{ih_i})$ og $\ell_{ij} = x_{ij}^{\beta_{ij}}$ for $i \in \{1, \ldots, n\}$ for $j \in \{1, \ldots, r\}$. Her er x_{ij} en boolsk variabel, og $\beta_{ij} \in \{0, 1\}$. Toptegnet 0 angiver en negeret variabel. Betragt nu følgende graf G. Knuderne svarer til formlens literaler, dvs. $V = \{ r_{ij} \colon 1 \leq i \leq k \text{ og } 1 \leq j \leq h_i \}$. Et par af knuder r_{ij} og $r_{i'j'}$ er naboer, hvis $i \neq i'$ og enten $x_{ij} \neq x_{i'j'}$ eller $\beta_{ij} = \beta_{i'j'}$. Med andre ord: Knuderne for to literaler er naboer, hvis og kun hvis literalerne hører til forskellige klausuler, og der findes en tildeling, som opfylder begge samtidigt. Vi påstår, at F kan opfyldes, hvis og kun hvis G har en klike af størrelse k.

Antag først, at der findes en tildeling α, som opfylder F. Denne tildeling opfylder mindst en literal per klausul, lad os sige literalen $\ell_{i,j(i)}$ i klausul C_i for $i \in \{1,\ldots,k\}$. Betragt nu den delgraf af G, som induceres af de k mange knuder $r_{i,j(i)}$ for $i \in \{1,\ldots,k\}$. Vi påstår, at delgrafen er en klike. Antag modsætningsvist, at $r_{i,j(i)}$ og $r_{i',j(i')}$ ikke er naboer. I så fald gælder $x_{i,j(i)} = x_{i',j(i')}$ og $\beta_{i,j(i)} \neq \beta_{i',j(i')}$. Men da er literalerne $\ell_{i,j(i)}$ og $\ell_{i',j(i')}$ hinandens komplement, og α kan ikke opfylde dem begge, i modstrid med konstruktionen.

Antag omvendt, at G har en klike K bestående af k knuder. Da kan vi konstruere en opfyldende tildeling α for F. Kliken K indeholder en knude for hvert $i \in \{1,\ldots,k\}$, som vi kalder $r_{i,j(i)}$. Lad nu α være givet ved $\alpha(x_{i,j(i)}) = \beta_{i,j(i)}$, for de øvrige variable kan α vælges vilkårligt. Læg mærke til, at α er veldefineret, idet $x_{i,j(i)} = x_{i',j(i')}$ medfører $\beta_{i,j(i)} = \beta_{i',j(i')}$; ellers skulle knuderne $r_{i,j(i)}$ og $r_{i',j(i')}$ ikke være naboer. Det er klart, at α opfylder F. □

Opgave 2.21. Vis, at hamiltonkredsproblemet kan reduceres til opfyldningsproblemet for boolske formler.

Opgave 2.22. Vis, at graffarvningsproblemet kan reduceres til opfyldningsproblemet for boolske formler.

De NP-fuldstædige problemer »sidder i samme båd«. Hvis nogen skulle være i stand til at løse *et eneste* af dem i polynomiel tid, så ville NP = P. Idet rigtig mange mennesker uden success har prøvet at udvikle den slags algoritmer, bliver det mere og mere vanskeligt at forestille sig, at det nogensinde skulle lykkes. De NP-fuldstædige problemer dokumenterer altså sammenn deres egen vanskelighed.

Kan man bruge NP-fuldstændighedsteorien også på optimeringsproblemer? Optimeringsproblemer (se kapitel 12) kan let omvandles til beslutningsproblemer. I stedet for at lede efter en optimal løsning, prøver vi bare at afgøre, om der *findes* en gyldig løsning med målfunktionsværdien mindst k, hvor k er del af input. Den omvende konstruktion gælder ligeledes: Hvis der findes en algoritme for at afgøre, om der findes en gyldig løsning med målfunktionsværdi mindst k, kan vi bruge kombinationen af eksponentiel og binær søgning (se afsnit 2.5) for at bestemme den optimale værdi af målfunktionen.

En algoritme for beslutningsproblemet svarer »ja« eller »nej«, afhængig af, om input tilhører det tilsvarende sprog eller ej. Den giver dog ikke noget vidne. Ofte kan man konstruere vidner ved at udføre beslutningsalgoritmen gentagne gange på lettere forandrede instanser. Lad os fx antage, at grafen G indeholder en klike af størrelse k, og at vi ønsker at finde sådan en klike. Antag, at har en algorime, der afgør, om en givet graf indeholder en klike af en givet størrelse. For $k = 1$ kan en vilkårlig knude i G bruges som svar på den eftersøgte klike. Eller vælger vi en vilkårlig knude v i G og spørger, og konstruerer grafen G' som $G' = G \setminus v$ (notationen beskriver grafen G uden knude v og dens hosliggende kanter) Vi spørger nu algoritmen, om G' indeholder en klike af størrelse k. I bekræftende fald, fortsætter vi søgningen efter en k-klike rekursivt i G'. I modsat fald må v indgå i samtlige k-kliker i G.

Da betragter vi mængden V' af naboer til v og fortsætter søgningen efter en $(k-1)$-klike K' rekursivt i den af V' inducerede delgraf. Mængden $\{v\} \cup K'$ er en k-klike G.

2.11 Implementationsaspekter

Vores pseudokode lader sig nemt oversætte til programmer i imperative programmeringssprog. For C++, Java og Python går vil lidt mere i detaljer. Programmeringssproget Eiffel [38] gør det muligt at arbejde med antagelser, invarianter, før- og efterbetingelser.

Vores specialværdier $\perp$, $-\infty$ og ∞ stilles til rådighed for kommatal af programmeringssproget. For andre datatyper skal man simulere disse værdier. For eksempel kan man bruge det mindste og største heltal for $-\infty$ hhv. ∞. Udefinerede pegere repræsenteres ofte af nulpegeren **null**. Sommetider bruger vi specialværdierne $\perp$, $-\infty$ og ∞ kun af bekvemmelighedsgrunde; en robust implementation skulle i forvejen undgå deres brug. Vi skal se nogle eksempler på dette i de senere kapitler.

Randomiserede algoritmer har brug for en tilfældighedskilde. Her har man valget mellem en maskinel tilfældighedsgenerator, som producerer ægte tilfældige tal, og en algoritmisk generator, der genererer pseudotilfældige tal.

2.11.1 C++

Man kan opfatte vores pseudokode som en kompakt notation for en delmængde af C++. Lagerforvaltningsoperationerne **alloker** og **afalloker** ligner operationerne *new* og *delete* i C++. Når en række bliver skabt i C++, kaldes den underliggende types standardkonstruktoren for hver indgang i rækken; det tager altså $\Omega(n)$ tid at skabe en række af længde n, hvorimod det kun tager konstant tid at skabe en række af n elementer af heltalstypen *int*. I bogens fremstilling foretages der ingen initialisering. Vi går derimod ud fra, at rækker uden udtrykkelig initialisering er fyldt med vilkårligt indhold, ofte kaldt »*affald*«. I C++ kan man opnå den samme opførsel ved at benytte de mere grundlæggende C-funktioner *malloc* og *free*. Denne fremgangsmåde der dog ildeset og bør kun bruges i undtagelsestilfælde, når initialisering af rækken udgør en flaskehals i kørslen. Hvis lagerforvaltningen for mange små objekter bliver kritisk for udførelsestiden, kan man tilpasse C++standardbibliotekets klasse *allocator* til respektive situation.

Vores parameterisering af klasser ved brug af **af** er et specialtilfælde af C++-sprogets mønstermekanisme. De parametre, der ved objektdeklarationen tilføjes klassenavnet i parenteser svarer til C++-konstruktorens parametre.

Antagelser implementeres som C-makroer i et inkluderet arkiv kaldt `assert.h`. Normalt udløser en brudt antagelse en køretidsfejl; fejlmeddelelsen indeholder antagelsen position i programteksten. Ved at definere makroen *NDEBUG* slår man den automatiske sikring af antagelser fra.

For mange af de datastrukturer og algoritmer, som diskuteres i denne bog, findes der fremragende implementationer i diverse standardbiblioteker. Gode biblioteker er *Standard Template Library* (STL) [43], BOOST [6] for C++ og LEDA [37, 34].

2.11.2 Java

I Java findes ingen udtrykkelig lagerforvaltning. Derimod findes et separat lagerrensningsprogram, som regelæssigt identificerer lagersegmenter, som ikke længere er i brug, og stiller dem til rådighed igen. Dette forenkler programmeringen i Java betydeligt, men kan også have meget negative effekter på tidsforbruget. Metoder, som afhjælper dette fænomen, ligger uden for bogens emner. Java tillader parameteriserede klasser i form af generiske typer. Antagelser implementeres som kommandoen *assert*.

Pakken *java.util* og datastrukturbiblioteket JDSL [20] indeholder fremragende implementationer af mange datastrukturer og algoritmer.

2.11.3 Python

Pythons abstraktionsniveau er endnu højere en Javas; på nogle punkter ligger syntaksen meget tæt på vores pseudokode, på andre punkter er sproget næsten for langt fra von Neumann-arkitekturen til at en oversættelse bliver meningsfuld. Som pædagogisk værktøj for at forstå koncepterne i denne bog, virker sproget ypperligt; vil man derimod skrive højtoptimerede programmer med fokus på effektivitet, gør man klogt i at vælge et andet sprog.

På samme måde som Java tillader Python heller ingen udtrykkelig lagerforvaltning, og oprydningsarbejdet sker i baggrundnen og uden for programmørens kontrol, med de samme fordele og ulemper. Pythons typesystem er dynamisk, så parameterising af klasser med **af** giver ikke meget mening. Antagelser implementeres med *assert*.

Mest bemærkelsesværdigt er det måske, at Python ikke stiller rækker til rådighed, men direkte bruger en liste-abstraktion, som vi først vil møde i kapitel **??**. Læseren aftale med sig selv, at fx udtrykket »A = [0] * N« skaber en række af heltal af længde N, initialiseret til 0, for at oversætte pseudokoden til fungerende python. Men i virkeligheden er A en pythonliste med langt mere magtfulde operationer. Listeoperatiorne i moderne pythonfortolker er selvfølgeligt selv baseret på meget effektive implementationer af operationer på rækker, skrevet i C, men dette abstraktionsniveau er altså ikke eksponeret til programmøren. Modulet *arrays* stiller dog »rigtige« rækker af vsse grundtyper til rådighed. Typen *Set* i vores pseudokode er på de anden side del sproget, og udtrykkene *None* og *float('inf')* kan bruges som vores $\perp$ og ∞.

2.12 Historiske anmærkninger og videre resultater

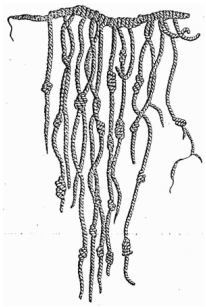
Definitionen af registermaskinen til algoritmeanalyse blev foreslået af Shepherdson og Sturgis [49]. Modellen begrænser størrelsen af de tal, der kan gemmes en i enkelt lagercelle, til logaritmisk mange bit. Uden denne indskrænkning opstår der uønskede konsekvenser, fordi modellen bliver alt for kraftig; fx kollapser kompleksitetsklasserne P og PSpace [22]. En meget detaljeret abstrakt maskinmodel er beskrevet af Knuth [32].

Invarianter som koncept i programsemantik blev introduceret af Floyd [15], og yderligere systematiseret af Hoare [23, 24]. Bogen [21] indeholder en meget omfattende gemmengang af arbejdet med summer og rekursionsligninger og mange andre emner i »tællende« kombinatorik, som kommer til anvendelse i algoritmeanalysen.

Bøger om oversætterkonstruktion (fx [39, 59]) forklarer detaljerne ved oversættelsen af højniveauprogrammeringssprog til maskinkode.

3

Repræsentation af følger med rækker og hægtede lister



De måske ældste datastrukturer i verden var kileskriftstavlerne[1]*, som for mere end 5000 år siden blev brugt af embedsmænd i sumeriske templer. Embedsmændene førte lister over varer – deres mængder, ejere og købere. Billedet til venstre viser sådan en tavle, muligvis det ældste eksempel på skriftsprog. Operationerne, som dengang skulle udføres på disse lister, er de samme som i dag: Indgange bliver tilføjet, informationen gemmes til senere brug, man vil lede efter en indgang og måske ændre dem, hele listen skal gennemgås for at skabe en sammenfatning, osv. Det peruanske quipu [**?**] tjente i inkariget lignende formål; hertil brugte man knuder på farvede snore, som en efter en var anbragt på en hovedsnor. Det var sandsynligvis lettere at gemme, pleje og organisere data på tavler end med knudesnore, men på den anden side er det ubekvemt slæbe stentavler over stejle stier i Andesbjergene. Åbenbart giver det mening at bruge forskellige repræsentationer af samme slags data alt efter behov.*

Det abstrakte begreb *følge* er meget enkelt og har i første omgang ikke noget med dens digitale repræsentation at gøre. Matematisk set er den eneste væsentlige egenskab ved følgen $s = \langle e_1, \ldots, e_n \rangle$, at en række af værdier (kaldt *indgange*) er anbragt i en lineær ordning – i modsætning graferne og træerne, som vi skal betragte i kapitel7 og 8, og som ikke er *lineært* ordnede, og hakketabellerne i kapitel 4, som slet ikke er ordnede.

Grundlæggende kan man tilgå indgangene i en følge på to måder. Den første mulighed består af at angive indgangens indeks. Det svarer til vores konvention for rækkeadgang, dvs. at vi med udtrykket $s[i]$ angiver den

[1] Den 4600 år gamle tavle oppe til venstre indeholder en liste af gaver til ypperstepræstinden af Adab (se `commons.wikimedia.org/wiki/Image:Sumerian_26th_c_Adab.jpg`).

*i*te indgang e_i i følgen s. Hertil stiller vores pseudokode i afsnit 2.3 allerede *statiske rækker* til rådighed. En datastruktur kaldes sommetider *statisk*, når dens størrelse er givet på forhånd og ikke kan ændres af tilføjelser og fjernelser.[2] Vi kalder datastrukturen *begrænset*, når en øvre grænse på dens størrelse er givet på forhånd. I afsnit skal vi betragte *ubegrænsede* (også kaldt *dynamiske*) rækker, som kan vokse og krympe, når indgange bliver tilføjet eller fjernet i enden af følgen. Til tidsanalysen for ubegrænsede rækker skal vi bruge *amortiseret analyse*.

Den anden mulighed for at tilgå indgangene i en følge er relativt til en anden indgang. For eksempel kunne man spørge om efterfølgeren til indgang e, forgængeren til indgang e' eller delfølgen $\langle e, \ldots, e' \rangle$ af indgange mellen e og e'. Selvom den slags relativ adgang kan simuleres med indicerede rækker, skal vi i afsnit 3.1 introducere en mere fleksibel repræsentation i form af hægtede lister. Især gør denne repræsentation det lettere at indføje og fjerne vilkårlige afsnit i en følge.

Mange algoritmer nøjes med at benytte følger på en meget indskrænket måde. Ofte bliver der kun læst og ændret i enden af følgen, forrest eller bagest. Følger, som benyttes på denne indskrænkede måde, kaldes *stakke*, *køer* og *dobbeltkøer*. Disse datastrukturer betragtes i afsnit 3.4. Afsnit 3.5 sammenfatter kapitlets resultater.

3.1 Hægtede lister

I dette afsnit betragter vi repræsentationen af følger ved hjælp af hægtede lister. Hægtede lister er opbygget af elementer kaldt *knuder*; hver knude består af en indgang fra følgen og en eller flere pegere. Man kan forestille sig en hægtet liste som en kæde, hvor hvert led indeholder en af følgens indgange. Når vi har greb om et kædeled, kan vi derfra nå andre led i kæden. I en dobbelthægtet liste peger hver knude på sin forgænger og sin efterfølger. I en enkelthægtet liste peger hver knude på sin efterfølger. Vi skal se, hvor let det er at foretage foretage forskellige ændringer i hægtede lister: vi kan indføje og slette knuder og dellister, og vi kan hægte lister efter hinanden. Ulempen ved hægtede lister er, at de ikke understøtter vilkårligt indiceret adgang i form at operatoren $[\cdot]$ effektivt. Vi betragter dobbelthægtede lister i afsnit 3.1.1 og enkelthægtede lister i afsnit 3.1.2. Enkelthægtede lister behøver mindre plads og er en smule hurtigere. Derfør er de at foretrække, medmindre men har brug for hele den dobbelthægtede listes funktionalitet.

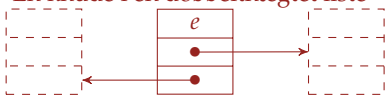
[2] Ovs. anm.: Værdierne i de enkelte indgange kan derimod ændres. En datastruktur, hvis værdier ikke kan ændres kaldes *uforanderlig*. De fleste programmeringssprogs strengtype er uforanderlig.

3.1.1 Dobbelthægtede lister

Den grundlæggende byggesten for en hægtet liste er vist i figur 3.1. En *knude* gemmer en indgang med to pegere til henholdsvis efterfølger og forgænger. En peger til en knude kaldes også et *greb* om denne knude, som i afsnit 2.3. Konstruktionen virker måske ligetil, men pegere er så kraftfulde, at vi vil behandle dem med største omhu. Hvilke konventioner (dvs. hvilken datastrukturinvariant) garanterer listestrukturens konsistens, dvs. dens linearitet? For det første skal man kræve, at forgængeren til efterfølgeren til knuden k er k selv, og at efterfølgeren til forgængeren til k ligeledes er k selv. Ved siden af den lokale linearitet skal vi desuden sikre, at der findes et indgangspunkt til listen, hvorfra alle indgange kan nås ved at følge efterfølgerpegere.

Klasse *Greb* = **Peger på** *Knude*

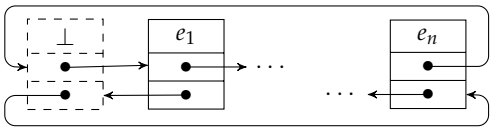
Klasse *Knude* **af** *Element* // En knude i en dobbelthægtet liste
  e: *Element*
  næste: *Greb*
  forrige: *Greb*
  invariant *næste* → *forrige* = *forrige* → *næste* = **dette**



Figur 3.1. En knude i en dobbelthægtet liste.

En følge $\langle e_1, \ldots, e_n \rangle$ med n indgange repræsenteres som en ring af $n + 1$ knuder. I ringen findes en særlig *attrapknude* h, som ikke indholder nogen følgeindgang. Attrapknudens efterfølger h_1 indeholder følgens første indgang e_1, knude h_1s efterfølger h_2 indeholder følgens anden indgang e_2, osv. Forgængeren til attrapknuden h indeholder følgens sidste indgang e_n, se figur 3.1. Den tomme følge repræsenteres af en ring, som kun består af attrapknuden h. I denne situation er altså h sin egen efterfølger og forgænger. Figur 3.3 viser en implementation af dobbelthægtede lister. Et objekt af klassen *Liste* indeholder en enkelt knude h som instansvariabel. Denne »hovedknude« h initialiseres ved at sætte indgangen til attrapværdien $\perp$ og lade knuden pege på sig selv som både forgænger og efterfølger. Dette repræsenterer en tom liste som begyndelsestilstand.

Alle grundlæggende listeoperationer bruger grundoperationen *splejs*, som er vist i figur 3.4. Splejsning klipper en delliste ud af listen og indføjer den bagved en given målknude. Dellisten er hertil givet ved greb a og b om sin første og sidste knude. For den korrekte udførelse af operationen er det væsentligt, at knuden med greb b kan nås fra knuden med greb a gennem at følge efterfølgerpegere uden undervejs møde attrapknuden. Målknuden t, som ligeledes er givet som et greb, kan sidde enten i den samme eller i en anden liste. I første fald må t selvfølgeligt ikke optræde i dellisten fra a til b.



Figur 3.2. Repræsentation af følgen $\langle e_1, \ldots, e_n \rangle$ som dobbelthægtet liste. Repræsentationen består af $n+1$ knuder anbragt i en ring: En attrapknude h, som ikke indeholder nogen indgang, og en knude for hver af følgens n indgange. Knuden med indgang e_i er efterfølger til knuden med indgang e_{i-1} og forgænger til knuden med indgang e_{i+1}. Attrapknuden sidder mellem de to knuder med indgang e_n og e_1.

Klasse *Liste* **af** *Element*
// Knude h er forgænger til den første og efterfølger til den sidste knude.
$h = \begin{pmatrix} \bot \\ \textbf{dette} \\ \textbf{dette} \end{pmatrix} : \mathit{Knude}$

// Funktioner for enkel adgang.
Funktion *hoved*: *Greb* **returner adressen på** h *// Position før alle indgange*

Funktion *tom*: $\{0,1\}$ **returner** $h.\mathit{next} = \textbf{dette}$
Funktion *første*: *Greb* **antag** $\neg\, \mathit{tom}$; **returner** $h.\mathit{næste}$
Funktion *sidste*: *Greb* **antag** $\neg\, \mathit{tom}$; **returner** $h.\mathit{forrige}$

// Flytning af enkelte indgange inden for samme følge.
// $\langle \ldots, a, b, c, \ldots, a', c', \ldots \rangle \mapsto \langle \ldots, a, c, \ldots, a', b, c', \ldots \rangle$.
Procedure *flytEfter*$(b, a'$: *Greb*$)$ *splejs*(b, b, a')
Procedure *flytForrest*$(b$: *Greb*$)$ *flytEfter*(b, hoved)
Procedure *flytBagest*$(b$: *Greb*$)$ *flytEfter*(b, sidste)

Figur 3.3. Nogle konstanttidsoperationer på dobbelthægtede lister.

Operationen *splejs* ændrer ikke på det totale antal knuder i systemet. Vi går ud fra, at der findes en speciel liste *friListe*, som stiller et forråd af ubenyttede knuder til rådighed. Når nye indgange skal optages i en følge, tages de nødvendige knuder fra listen *friListe*; når indgange fjernes fra listen, bliver det frigjorte knuder givet tilbage til *friListe*. Funktionen *sikrFriListe* reserverer lagerplads til nye knuder, når dette bliver nødvendigt. I opgave 3.3 spørges og i afsnit 3.6 diskuteres hvordan denne funktion kan implementeres.

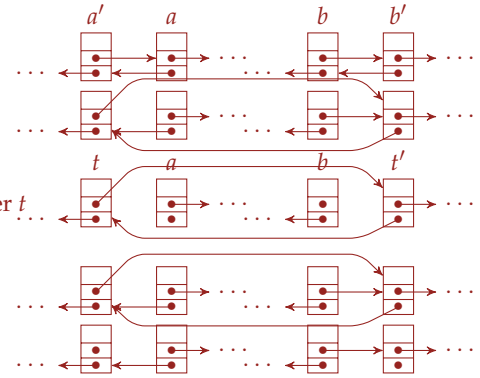
Baseret på disse konventioner kan vi nu implementere mange nyttige listeoperationer som funktioner bestående af en enkelt linje, som alle kræver konstant tid. Idet *splejs* er så kraftig, kan vi sågar behandle dellister af vilkårlig længde i konstant tid. Figur 3.5 viser mange eksempler på den slags operationer. For at afgøre om en liste er tom, behøver vi blot at undersøge, om h er sin egen efterfølger. Når en liste ikke er tom, finder vi dens første og sidste indgang som efterfølger hhv. forgænger til h. For at flytte knude b

// Fjern $\langle a, \ldots, b\rangle$ fra sin liste og indføj den efter t
// $\ldots, a', a, \ldots, b, b', \ldots + \ldots, t, t', \ldots \mapsto \ldots, a', b', \ldots + \ldots, t, a, \ldots, b, t', \ldots$

Procedure *splejs*$(a, b, t:$ *Greb*$)$
antag a og b tilhører samme liste, b står ikke før a, og $t \notin \langle a, \ldots, b\rangle$

// klip $\langle a, \ldots, b\rangle$ ud
$a' := a \rightarrow \mathit{forrige}$
$b' := b \rightarrow \mathit{næste}$
$a' \rightarrow \mathit{næste} := b'$
$b' \rightarrow \mathit{forrige} := a'$

// indføj $\langle a, ..., b\rangle$ efter t
$t' := t \rightarrow \mathit{næste}$
$b \rightarrow \mathit{næste} := t'$
$a \rightarrow \mathit{forrige} := t$
$t \rightarrow \mathit{næste} := a$
$t' \rightarrow \mathit{forrige} := b$



Figur 3.4. Operationen *splejs* på lister.

til positionen efter knude a' klipper man bare den delliste, som begynder og ender ved b, ud og indføjer den efter a'. Dette gøres med kaldet *splejs*(b, b, a'). For at placere en indgang forrest eller bagest i listen flyttes den tilsvarende knude bag ved hovedknuden h eller bag ved den sidste indgang. En indgang slettes ved at flyttes den tilsvarende knude til *friListe*. For at indsætte en ny indgang e udtages en ny knude fra *friListe*, indgangen e gemmes i knuden, og knuden anbringes på den ønskede plads.

Opgave 3.1 (Alternativ listeimplementation)**.** Diskutér en alternativ implementation af klassen *Liste*, som undgår brugen af en attrapknude. I stedet gemmer man i listeobjektet pegere på den første og den sidste knude i listen. Forgængeren til den første knude og efterfølgeren til den sidste knude er nulpegere. Grænsefladen og de asymptotiske kørselstider bør forblive uændrede. Nævn mindst en fordel og en ulempe ved denne implementation i sammenligning med den i teksten beskrevne implementation.

Attrapknuden kan også være nyttig for andre operationer. Betragt fx problemet at finde den næste forekomst af indgangen x fra en givet knude *fra*. Hvis x ikke forekommer, skal attrapknudens greb *hoved* returneres. Attrapknuden bruges her som en såkaldt *vogterknude*. En vogterknude i en datastruktur er en knude, der sørger for, at en vis løkke terminerer. Ved søgning i en liste gemmer vi den søgte nøgle x i attrapknuden. Herved opnår vi, at x findes i listestrukturen, så søgningen garanteret terminerer. Søgningen ender enten i en egentlig knude eller i attrapknuden, alt efter om x fandtes i den oprindelige liste eller ej. Dette trick gør, at man ikke i hvert skridt behøver at undersø-

```
// Indsættelse og fjernelse af enkelte indgange.
// ⟨...,a,b,c,...⟩ ↦ ⟨...,a,c,.....⟩.
Procedure fjern(b: Greb)   flytEfter(b, friListe.hoved)
Procedure fjernForrest   fjern(første)
Procedure fjernBagest   fjern(sidste)

// ⟨...,a,b,...⟩ ↦ ⟨...,a,e,b,.....⟩.
Funktion indsætEfter(x: Element, a: Greb): Greb
  sikrFriListe              // Sikr at friListe ikke er tom. Se opg. 3.3.
  a' := friListe.første     // Skaf ny knude a' til at indeholde x,
  flytEfter(a', a)          // indsæt den på rette plads,
  a'→e := x                 // og fyld den med det rette indhold.
  returner a'

Funktion indsætFør(x: Element; b: Greb): Greb
  returner indsætEfter(e, b→forrige)
Procedure tilføjForrest(x: Element)   indsætEfter(x, hoved)
Procedure tilføjBagest(x: Element)   indsætEfter(x, sidste)

// Behandling af hele lister.
// (⟨a,...,b⟩,⟨c,...,d⟩) ↦ (⟨a,...,b,c,...,d⟩,⟨⟩)
Procedure sammenføj(L': Liste)   splejs(L'.første, L'.sidste, sidste)

// ⟨a,...,b⟩ ↦ ⟨⟩
Procedure tøm   friListe.sammenføj(dette)
```

Figur 3.5. Flere konstanttidsoperationer på dobbelthægtede lister til klassen fra figur 3.3.

ge, om søgningen har nået enden af listen. Dette kan øge effektiviteten af søgningen betragteligt:

```
Funktion findNæste(x : Element; fra: Greb): Greb
  h.e := x      // Vogterknude
  så længe fra→e ≠ x udfør
    fra := fra→næste
  h.e := ⊥
returner fra
```

Opgave 3.2. Implementer en procedure *ombyt*, som i konstant tid ombytter to delfølger, dvs. transformerer følgerne

$$(\langle \ldots, a', a, \ldots, b, b', \ldots \rangle, \langle \ldots, c', c, \ldots, d, d', \ldots \rangle)$$

til følgerne

$$(\langle \ldots, a', c, \ldots, d, b', \ldots \rangle, \langle \ldots, c', a, \ldots, b, d', \ldots \rangle)\,.$$

Er *splejs* et specialtilfælde af *ombyt*?

Opgave 3.3 (Lagerhåndtering.)**.** Angiv en implementation af funktionen *sikrFriListe*, som kaldes fra *indsætEfter*, som angivet i figur 3.5. Funktionen skal sikre, at *friListe* ikke er tom, og i modsat fald tilføje nye knuder. Fordi den kan have en negativ indvirkning på kørselstiden at kalde den af programmeringssproget til rådighed stillede grundoperation **alloker** for hver knude separat, bør funktionen stille knuder til rådighed i større »portioner«. Kørselstiden for *sikrFriListe* i værste fald bør være uafhængig af portionernes størrelse. *Vink:* Benyt ved siden af *friListe* en lille række af ubenyttede (og endnu ikke initialiserede) knuder.

Opgave 3.4. Beskriv en algoritme, som i konstant tid udfører en højrerotation $\langle a, \ldots, b, c\rangle \mapsto \langle c, a, \ldots, b\rangle$ af en position i en følge. Beskriv en mere generel algoritme for rotationen $\langle a, \ldots, b, c, \ldots, d\rangle \mapsto \langle c, \ldots, d, a, \ldots, b\rangle$, givet et greb om b. Kørselstiden bør igen være konstant.

Opgave 3.5. Funktionen *findNæste* kører på grund af anvendelsen af en vogterknude hurtigere, end hvis vi i hvert løkkegennemløb skulle teste, om vi er nået til den sidste knude i listen. Men hvor stor er gevinsten i kørselstid? Hvilken relativ forskel i kørselstid ville du forvente i følgende situationer? (a) I en kort liste med 100 indgange gennemføres mange søgninger; (b) i en lang liste med 10 000 000 indgange gennemføres en enkelt søgning. Hvorfor afhænger den relative hastighedsforskel af listelængden?

Vedligeholdelse af listestørrelsen

I vores enkle listedatastruktur i figur 3.3 kan vi ikke bestemme listens størrelse (dvs. dens længde) i konstant tid. Til det formål kan man udvide klassen med en instansvariabel *størrelse*, som aktualiseres, når antallet af listeelementer bliver ændret. Hermed får nu alle operationer, som kan ændre flere lister, brug for at kende længderne af de berørte lister, i modsætning til vores konvention for grundoperationen *splejs*, som kunne nøjes med at kende til de berørte knuders greb. Betragt fx følgende kodestump, som fjerner en enkelt knude a fra liste L og flytter den bag ved knude a' i liste L' og aktualiserer de pågældende instansvariabler for størrelserne.

Procedure *flytEfter*$(a, a' :$ *Greb*$; L, L' :$ *Liste*$)$
  splejs(a, a, a'); $\quad L.$*størrelse*`--`; $\quad L'.$*størrelse*`++`

Vedligeholdelsen af listelængden kræver tilsvarende ændringer i implementationen af de andre listeoperationer. Et større problem end de udvidede grænseflader er dog, at vi ikke længere kan garantere konstante kørselstider. Hertil skulle operationer som fx splejsning, der flytter hele listeafsnit, jo også have brug for at kende til listeafsnittets størrelse. Den næste opgave beskriver et muligt kompromis.

Opgave 3.6. Konstruer på grundlag af en dobbelthægtet liste en listedatatype, som både tillader konstant-tids forskydning af dellister mellem liste og

bestemmelse af en listens størrelse, forudsat at listen ikke har været involveret i dellisteoperationer med andre lister siden det sidste kald til *størrelse*. Efter at den slags dellisteoperationer er blevet udført, skal værdien af *størrelse* først genberegnes, når den skal bruges næste gang.

Opgave 3.7. Forklar, hvordan operationerne *fjern*, *indsætEfter* og *sammenføj* skal modificeres for at vedligeholde listestørrelsen.

3.1.2 Enkelthægtede lister

At hver knude har både forlæns- og baglænsreferencer gør det ganske nemt at implementere operationer på dobbelthægtede lister. Vi skal nu se på deres lidt slankere søskende, de enkelthægtede lister. Knudetypen i enkelthægtede lister kalder vi *EElement*. Typen *EElement* har ingen forgængerpeger og nøjes med at gemme en peger på efterfølgeren. Det gør, at enkelthægtede lister kræver mindre plads og ofte også er hurtigere end dobbelthægtede. Ulempen er, at mange operationer ikke længere udføres i konstant tid eller ikke længere virker med samme almengyldighed. For eksempel bliver vi nødt til at kende en *EElement*-knudes forgænger for at fjerne den.

Vi følger den samme grundlæggende idé som for implementationen af dobbelthægtede lister. En enkelthægtet list skabes som en ring af knuder. Hvert objekt af typen *EListe* har en attrapknude h af typen *EElement*, som er forgænger til den første egentlige knude og efterfølger til den sidste egentlige knude. Mange listeoperationer kan stadig gennemføres med små ændringer af grænsefladen. For eksempel har *splejs*-operationen nu brug for at kende *forgængeren* til den første knude af den delliste, som skal flyttes:

$$// \left(\langle \ldots, a', a, \ldots, b, b', \ldots \rangle, \langle \ldots, t, t', \ldots, \rangle\right) \mapsto \left(\langle \ldots, a', b', \ldots \rangle, \langle \ldots, t, a, \ldots, b, t', \ldots, \rangle\right)$$

Procedure *splejs*$(a', b, t\colon\ EGreb)$

$$\begin{pmatrix} a' \rightarrow \textit{næste} \\ t \rightarrow \textit{næste} \\ b \rightarrow \textit{næste} \end{pmatrix} := \begin{pmatrix} b \rightarrow \textit{næste} \\ a' \rightarrow \textit{næste} \\ t \rightarrow \textit{næste} \end{pmatrix}$$

På en lignende måde skal operationen *findNæste* ændres til at returnere et greb om *forgængeren* til søgenøglen, hvis den fundne indgang skal kunne fjernes. Tilsvarende må funktionen *findNæste* begynde sin søgning ikke ved den angivne knude, men først ved dennes efterfølger. En nyttig tilføjelse til enkelthægtede lister er en peger på den sidste listeknude, hvilken gør det muligt at udføre operationen *indsætBagest* i konstant tid.

Opgave 3.8. Implementer klasserne *EGreb*, *EElement*, *EListe* på basis af enkelthægtede lister svarende til *Greb*, *Element* og *Liste*. Vis, at alle de følgende operationer kan implementeres, så de tager konstant tid. Hertil skal operationerne *hoved*, *første*, *sidste*, *tom*, *fjernForrest*, *indsætForrest*, *indsætForrest*, *indsætEfter*, *sammenføj* og *gørTom* have den samme grænseflade som for dobbelthægtede lister; hvorimod operationerne *flytEfter*, *flytTilForrest*, *flytTilBagest*, *fjern*, *fjernForrest* og *findNæste* har brug for ændringer i grænsefladen.

Vis skal i det videre forløb se nogle eksempler på brugen af enkelthægtede lister, fx hakketabeller i afsnit 4.1 og flettesortering i afsnit 5.2. Enkelthægtede lister egner sig også til at vedligeholde lister af frie knuder til lagerhåndtering – inklusive for knuder i dobbelthægtede lister.

3.2 Ubegrænsede rækker

Vi betragter nu en datastruktur for følger, som ved siden af den indeksbaserede adgangsoperation $[\cdot]$ tillader operationerne *indsætBagest*, *fjernBagest* og *størrelse* som følger:

$$\begin{aligned} \langle e_1, \ldots, e_n \rangle.\mathit{indsætBagest}(e) &= \langle e_1, \ldots, e_n, e \rangle\,, \\ \langle e_1, \ldots, e_n \rangle.\mathit{fjernBagest}(e) &= \langle e_1, \ldots, e_{n-1} \rangle\,, \qquad (\text{for } n \geq 1)\,, \\ \mathit{størrelse}(\langle e_1, \ldots, e_n \rangle) &= n\,. \end{aligned}$$

Hvorfor en den slags ubegrænsede rækkestrukturer vigtige? I mange situationer ved man ikke i forvejen, hvor stor en række skal blive. Et typisk eksempel er følgende: Lad os implementere unixkommandoen *sort*, som sorterer de enkelte linjer i en fil. Vi begynder med at læse filens enkelte linjer ind i en række av linjer, sorterer rækker in hovedlageret og udskriver til sidst den sorterede række af linjer. Med ubegrænsede rækker er dette ligetil, men med begrænsede rækker skulle vi derimod behøve at læse hele filen to gange: En første gang bare for at bestemme antallet af linjer, og en anden gang for at læse linjerne ind i den nu parate række af den rette størrelse.

Vi skal nu beskrive, hvordan man implementerer ubegrænsede rækker. Vi simulerer en ubegrænset række u af n indgange ved at dynamisk vedligeholde en begrænset række b med w pladser, hvor $w \geq n$. De første n pladser i b bruges til indgangene fra u. De resterende $w - n$ pladser i b er ubenyttede. Så længe $w > n$, kan operationen *indsætBagest* nøjes med at øge pejlevariablen n og placere den nye indgang på den første ubenyttede plads i b. Når $w = n$, fører det næste kald af *indsætBagest* til, at der stilles en ny begrænset række b' til rådighed, som er en konstant faktor (fx to gange) større end b. For at genetablere invarianten at u er gemt i b, bliver bs indhold kopieret til de første n pladser af b'. Hermed kan den gamle række b frigøres. Endelig flyttes pegeren på b til at pege på b'. Det er endnu nemmere at fjerne den sidste indgang (ved *fjernBagest*), fordi der aldrig er fare for, at b bliver for lille. Det kan dog tænkes, at vi spilder alt for meget lagerplads ved at tillade, at b er meget større end nødvendig. Den overskydende lagerplads kan holdes lille ved krympe b, når n bliver for lille i forhold til w. I figur 3.6 vises pseudokoden for en klasse, der realiserer ubegrænsede rækker. Den underliggende række vokser og krymper ved hjælp af proceduren *realloker*. I vores implementation har vi benyttet konstanterne α og β med $\beta = 2$ og $\alpha = 4$. Når den aktuelle begrænsede række bliver for lille, erstatter vi den med en ny række, som er β gange større; når den udnyttede del af rækken bliver α gange mindre end

rækkens størrelse, erstatter vi den med en ny række af størrelse n/β. Vi gør rede for vores valg af værdierne for α og β forneden.

```
Klasse URække af Element
  Konstant β := 2: R_{>0}
  Konstant α := 4: R_{>0}
  w := 1: N
  n := 0: N
  invariant n ≤ w ≤ αn eller n = 0 og w ≤ β
  b: Række[1..w] af Element

  Operator[i: N]: Element
    antag 1 ≤ i ≤ n
    returner b[i]

  Funktion størrelse: N    returner n

  Procedure tilføjBagest(e: Element)
    hvis n = w så
      realloker(βn)
    b[n + 1] := e
    n++

  Procedure fjernBagest
    antag n > 0
    n--
    hvis αn ≤ w ∧ n > 0 så
      realloker(βn)

  Procedure realloker(w': N)
    w := w'
    b' := alloker Række[1..w'] af Element
    (b'[1], ..., b'[n]) := (b[1], ..., b[n])
    afalloker b
    b := b'
```

Figur 3.6. Pseudokode for ubegrænsede rækker.

3.2.1 Amortiseret analyse af ubegrænsede rækker: det globale perspektiv

Vores implementation af ubegrænsede rækker følger princippet om at være »sædvanligvist hurtig«. Adgangen til en indgang ved brug af $[\cdot]$ er lige så hurtig som for en begrænset række. Intuitivt betragtet vil *indsætBagest* og *fjernBagest* ligeledes være hurtige »sædvanligvist« – datastrukturen skal jo bare aktualisere pejlevariablen n. Alligevel vil visse indsættelser og fjernelser

kræve tid $\Theta(n)$. Vi skal nu vise, at disse dyre operationer er sjældne, og at hver følge af m operationer begyndende fra en tom række kan udføres i tid $O(m)$.

Lemma 3.1. *Betragt en ubegrænset række u, som er tom i begyndelsen. En vilkårlig følge $\sigma = \langle \sigma_1, \ldots, \sigma_m \rangle$ af m* indsætBagest- *og* fjernBagest-*operationer på u bliver udført i tid $O(m)$.*

Lemma 3.1 er alt andet end trivielt. En lille og uskyldigt udseende ændring i implementationen gør påstanden falsk, som næste opgave viser.

Opgave 3.9. Hr. Karrig opfordrer dig til at ændre forvalget af α til $\alpha = 2$. Hans argumentet er, at det er spild af plads at vente med at erstatte rækken med en mindre, til den er kvartfuld. Han foreslår derfor at formindske rækken, så snart $n \leq \frac{1}{2}w$. Overbevis ham om, at det er en dårlig ide. Angiv hertil en følge af m *indsætBagest*- og *fjernBagest*-operationer, som skulle kræve tid $\Theta(m^2)$ med hans valg af α.

Lemma 3.1 er et udsagn om de såkaldt »amortiserede« omkostninger ved en følge af *indsætBagest*- og *fjernBagest*-operationer. Enkelte operationer kan være dyre, men hele følgen af m operationer koster $O(m)$. Hvis vi dividerer den samlede omkostning for alle operationer i følgen σ med antallet af operationer, får vi en konstant. Det udtrykker vi på følgende måde: Den *amortiserede omkostning* af hver enkel operation er konstant. Betydningen, som vi her tildeler ordet »amortiseret«, ligner den hverdagssproglige brug, men undgår en fælde, som ofte optræder i forbindelse med forestillinger om amortisering. Måske virker følgende ræsonnement bekendt? Bjarne siger: »Fra og med i dag cykler jeg på arbejde hver dag, så derfor har jeg råd til en luksuscykel. I det lange løb er udgiften per køretur meget lille – investeringen amortiserer sig altså.« Hvad kommer til at ske i virkeligheden? Bjarne køber den dyre cykel, vejret slår om i regnvejr, og alle gode forsæt er glemt; den dyre cykel står i et hjørne, og ingenting er blevet amortiseret. I modsætning hertil planlægger vi vores datastrukturer sådan, at alle store udgifter altid retfærdiggøres af opsparinger i fortiden i stedet for af håbet om fremtidig nøjsomhed. I vores eksempel om transport til og fra arbejde kunne det se sådan ud: Astrids langsigtede mål er at køre på arbejde i en luksuslimousine. Hun køber den dog ikke med det samme, men går derimod til fods og lægger hver dag et fast beløb til side. Efter et stykke tid har hun nu råd til at købe en cykel. Hun fortsætter med at spare og har snart råd til en lille bil, og efter endnu længere tid til den ønskede luksusbil. Hver udgift er dækket af tidligere opsparing og er amortiseret i samme øjeblik, den sker. Med dette amortiserede udgiftsbegreb kan vi formulere lemma 3.1 endnu mere elegant. Den elegante formulering tillader også en mere præcis sammenligning af forskellige datastrukturer.

Korollar 3.1. *Ubegrænsede rækker understøtter operationen $[\cdot]$ i konstant værstefaldstid og operationerne* indsætBagest *og* fjernBagest *i amortiseret konstant tid.*

Bevis for lemma 3.1. For at etablere lemmaet benytter vi den såkaldte *bankkonto-metode*. Denne er ækvivalent til *potentialemetoden*, som er nærmere forklaret forneden. Ved siden af vores datastruktur tænker vi os en bankkonto, som til hver tid indeholder et vist beløb, der aldrig må være negativt. For hver *tilføjBagest*- og *indsætBagest*-operation indbetaler vi et fast beløb på kontoen. Vi kalder vores valuta for *mønter*. Åbenbart er kopieringen af indgange i proceduren *realloker* den eneste aktion i programmet i figur 3.6, som medfører mere end en konstant omkostning. Ved nærmere betragtning indser vi, at proceduren altid kaldes med parameterværdien $w' = 2n$ og så skal flytte n indgange. Grundideen i analysen er nu at lade opsparingen på bankkontoen finansiere omkostningen ved denne dyre operation *realloker*. Lad os vedtage, at en enkelt mønt kan finansiere omkostningen ved at flytte en enkelt indgang fra b til b'. Det betyder, at vi kan nøjes med at hæve n mønter fra kontoen for et kald af *realloker*. Vi bestemmer os med forsyn for at indbetale 2 mønter for hvert kald af *indsætBagest* og 1 mønt for hvert kald af *fjernBagest*. Vi skal nu vise, at disse indbetalinger er nok til at dække udgifterne ved kaldene til *realloker*.

Det første kald til *realloker* sker, når rækken indeholder én indgang, og den næste indgang bliver tilføjet. For den allerede eksisterende indgang havde vi indbetalt to mønter, og det er rigeligt til at dække den ene mønt, som *realloker*-kaldet koster.

Efter hvert kald *realloker* i det videre forløb indeholder rækken w pladser, hvoraf $n = \frac{1}{2}w$ er benyttede og $\frac{1}{2}w$ står tomme. Ved det næste kald af *realloker* gælder enten $n = w$ eller $4n \leq w$. I det første tilfælde må der være tilkommet mindst $\frac{1}{2}w$ indgange i mellemtiden; for hver af disse havde vi indbetalt 2 mønter. Derfor må der være mindst w mønter på kontoen, hvilket er nok til at finansiere flytningen af n indgange, idet $n = w$. I det andet tilfælde er der blevet fjernet mindst $\frac{1}{2}w - \frac{1}{4}w = \frac{1}{4}w$ indgange fra rækken; for hver af disse operationer havde vi indbetalt 1 mønt. Derfor er der mindst $\frac{1}{4}w$ mønter på kontoen, hvilket er nok til at finansiere flytningen af n indgange, idet $n \leq \frac{1}{4}w$. Hermed er lemma 3.1 bevist. □

Opgave 3.10. Ændr beviset for lemma 3.1 således, at det gælder for generelle værdier af α og β. Hertil kræves, at et kald af *indsætBagest* koster $\beta/(\beta - 1)$ mønter, og et kald af *fjernBagest* koster $\beta/(\alpha - \beta)$ mønter. Når n' opfylder ligningen $w = \beta n'$, ender hvert kald af *realloker* med netop n' optagede pladser og $(\beta - 1)n' = ((\beta - 1)/\beta)w$ ledige pladser. Når næste kald af *realloker* sker, gælder enten $n = w$ eller $\alpha n \leq w$. Vis, at der i begge tilfælde er tilstrækkeligt mange mønter.

Amortiseret analyse er et kraftfuldt redskab med mange anvendelser. Derfor kan det betale sig at lære yderligere metoder til at bevise den slags udsagn. [3] Vi skal se nærmere på to variationer over beviset for lemma 3.1.

[3] Situationen minder om induktionsbeviser, som sommetider kan formuleres i termer af mindste modeksempler. Det er nyttigt at beherske begge metoder.

Foroven bestemte vi at anslå 2 mønter for hvert *indsætBagest*- og 1 mønt for hvert *fjernBagest*-kald. Alternativt kunne vi have regnet med 3 mønter for *indsætBagest* og 0 mønter for *fjernBagest*. Bogholderiet er ligetil: Af de 3 mønter bruges de to til at finansiere tilføjelsen af en indgang og den tredje til indgangens (eventuelle og fremtidige) fjernelse, som altså er blevet betalt lang tid i forvejen.

Opgave 3.11 (Fortsættelse af opgave 3.10)**.** Vis, at det er nok med en afgift på $\beta(\beta-1)+\beta)(\alpha-\beta)$ mønter for hver *indsætBagest*-operation. Bestem (afhængigt af β) en værdi for α, som sikrer $\beta/(\alpha-\beta) \leq 1/(\beta-1)$ hhv. $\beta/(\alpha-\beta) \leq \beta/(\beta-1)$.

3.2.2 Amortiseret analyse af ubegrænsede rækker: det lokale perspektiv

Vi skal nu betragte den anden variant af beviset for lemma 3.1. I overvejelserne foroven har vi anlagt et globalt perspektiv for at overbevise os om, at der altid fandtes tilstrækkeligt mange mønter på kontoen for hvert kald af *realloker*. Nu vil vi i stedet indtage et lokalt standpunkt. Vi husker, at rækken umiddelbart efter kaldet til *reallocate* består af w pladser, af hvilke $\frac{1}{2}w$ er optagede og $\frac{1}{2}w$ er ledige. Vi skal nu vise, at der for hvert tidspunkt efter første kald til *reallocate* gælder følgende invariant for antallet af mønter:

$$\text{Der er mindst } \max\{2(n-\tfrac{1}{2}w), \tfrac{1}{2}w-n\} \text{ mønter på kontoen.}$$

Det ses let, at denne værdi aldrig er negativ. Vi skal vise invarians ved induktion i antal udførte operationer. Umiddelbart efter første udførelse af *realloker* er kontostanden 1 mønt, hvilket opfylder invarianten, som jo kræver mindst 0 mønter i dette fald, hvor $n = w/2 = 1$. Hver udførelse af *indsætBagest* (uden hensyntagen til det muligvis udløste kald til *realloker*) øger n med 1 og indbetaler 2 mønter. Invarianten gælder altså stadig. Hver udførelse af *fjernBagest* (uden hensyntagen til det muligvis udløste kald til *realloker*) formindsker n med 1 og indbetaler 2 mønter. Invarianten gælder altså også i dette tilfælde stadig. Nu betrager vi et kald til *realloker*, som jo skal bruge n mønter. Der er to tilfælde, svarende til $n = w$ og $4w \leq n$ med $n \geq 0$. I det første tilfælde er kontostanden ifølge invarianten mindst $2(n-\frac{1}{2}w) = n$, hvilket dækker behovet. I det andet tilælde er kontostanden ifølge invarianten mindst $\frac{1}{2}w - n \geq n$, hvilket ligeledes dækker behovet. I begge tilfælde gælder ligningen $n = \frac{1}{2}w$ efter udførelsen af *reallocate*; invarianten er altså stadig gyldig.

Opgave 3.12. Lad os vedtage, at *tilføjBagest* koster 3 mønter, og *fjernBagest* ingen. Vis, at kontoen altid har en saldo på mindst $n + \max\{2(n-w/2), w/2-n\} = \max\{3n-w, w/2\}$ mønter.

Opgave 3.13 (Samtidig fjernelse af flere indgange)**.** Implementer operationen *fjernBagest*(k) for $0 < k \leq n$, som fjerner de sidste k elementer i rækken, i amortiseret konstant tid. Læg mærke til, at værdien k er en kaldsparameter, ikke en konstant.

Opgave 3.14 (Konstant værstefaldstid)**.** Antag, at vi har brug for ubegrænsede rækker med konstant *værstefalds*-tid per operation. Konstruér sådan en datastruktur. *Vink:* Begynd med at understøtte $[\cdot]$ og *indsætBagest*. Gem værdierne i op til to rækker. Påbegynd flytningen til en større række inden den mindre række er helt udnyttet. Hvordan kan man generalisere denne idé til også at understøtte *fjernBagest*?

Opgave 3.15 (Implicitte voksende rækker)**.** [TODO]

Opgave 3.16 (Tyndt besatte rækker)**.** Implementer en begrænset række b med konstant allokeringstid og konstant tid for $[\cdot]$. Alle rækkeindgange skal være (implicit) initialiserede til $\perp$. Du må ikke gøre nogen antagelser om indholdet i en nyallokeret række. *Vink:* Brug en ekstra række b' af samme størrelse og husk antallet t af indgange i b, som har fået tildelt en værdi. I begyndelsen er $t = 0$. Når indgangen i tildeles en værdi, gemmes denne værdi sammen med et indeks j med $1 \leq j \leq t$, og $b'[j]$ sættes til i.

3.2.3 Amortiseret analyse af binær tæller

[TODO]

3.3 Amortiseret analyse *

[TODO]

3.4 Stakke og køer

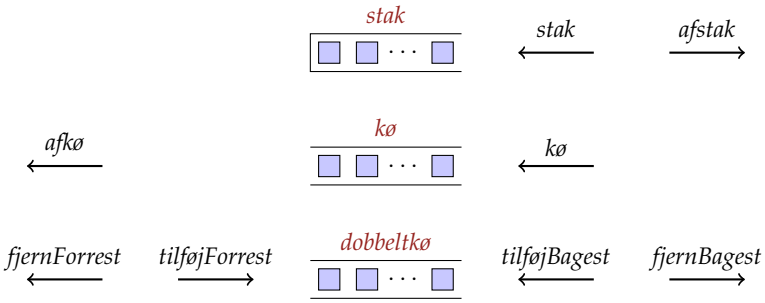
Følger bliver ofte brugt på en temmelig begrænset måde. Lad os begynde med to eksemper fra den førdigitale tidsalder. En sagsbehandler kan arbejde på følgende måde: Hun har en *stak* af ubehandlede sagsmapper på sit skrivebord. Nye sagsmapper bliver lagt på toppen af stakken. Når sagsbehandleren vil behandle næste sag, vælger hun den mappe, der ligger øverst på stakken. Denne »datastruktur« af sagsmapper er nem at bruge; selvom den kan ske, at visse mapper længere nede i stakken ender med at forblive ubehandlede i lang tid. I terminologien for klassen *Liste* fra afsnit 3.1 er en stak en følge, som kun bruger operationerne *tilføjBagest*, *fjernBagest*, *sidste* og *tom*. Vi kalder gerne de tre første operationer for *stak*, *afstak*[4] og *top*. Andre betegnelser for stak er *sifu-liste* eller *sifu-kø* (»sidst ind, først ud«) og *kælderlager*.

En anden opførsel kan man observere, når folk står pænt i kø hos bageren: Kunder stiller sig bagest i køen og forlader den forrest, når de når frem til

[4] O.a.: Vær opmærksom på, at mange lærebøger og programbiblioteker bruger *afstak* til at betegne en kombination af *sidste* og *fjernBagest*, som returnerer det sidste element (dvs. staktoppen) og samtidigt fjerner det fra stakken, svarende til operationerne $tmp := L.\mathit{sidste}$; $L.\mathit{fjernBagest}$; **returner** tmp.

disken. Sådan en følge kaldes *kø*, *fifu-liste* eller *fifu-kø* (»først ind, først ud«). I terminologien for klassen *Liste* fra afsnit 3.1 er en kø en følge, som kun bruger operationerne *tilføjBagest*, *fjernForrest*, *første* og *tom*, hvoraf de første kan omdøbes til *kø* og *afkø* [5].

En mere generel *dobbeltkø* stiller operationerne *første*, *sidste*, *tilføjForrest*, *tilføjBagest*, *fjernForrest*, *fjernBagest*, og *tom* til rådighed. Dobbeltkøen kan observeres hos bageren, når en uforskammet person stiller sig forrest i køen, eller den person, der står bagest i køen, bliver træt af at vente og forlader butikken. Figur 3.7 sammenfatter funktionaliteten af datastrukturerne stak, kø og dobbeltkø.



Figur 3.7. Operationer på stakke, køer og dobbeltkøer.

Opgave 3.17 (Hanois tårne)**.** *I det store brahmatempel i den hellige indiske by Varanasi, under den kuppel, som markerer verdens midtpunkt, ligger 64 skiver af det rene guld på en messingplade. Skiverne har forskellig størrelse, og hver skive har et hul i midten. Tempelpræsterne bærer disse skiver enkeltvis frem og tilbage mellem tre lange nåle i overensstemmelse med Brahmas urokkelige lov: ingen skive må anbringes på en mindre skive. Da jorden blev skabt, lå alle 64 skiver på første nål; die dannede Brahmas tårn. I vores tid er transporten af skiverne til den anden nål i fuld gang. Når endelig Brahmas tårn genopstår i sin helhed på den anden nål, så er verdens undergang kommet og alt bliver støv.* [25] [6]

Giv en formel beskrivning af problemet at flytte et vilkårligt antal k af skiver fra en nål til en anden ved hjælp af en tredje nål. Skriv et program, som realiserer de tre tårn som stakke $\langle u_1, \ldots, u_l \rangle$ med indgange fra $\{1, \ldots, k\}$, hvor der altid skal gælde $u_1 > \cdots > u_l$. Programmet skal udlæse en følge af stakoperationer, som transformerer tilstanden $(\langle k, \ldots, 1 \rangle, \langle\,\rangle, \langle\,\rangle)$ til tilstanden $(\langle\,\rangle, \langle k, \ldots, 1 \rangle, \langle\,\rangle)$. *Vink:* Løsningen er lettest at formulere rekursivt.

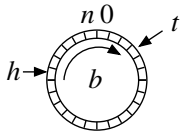
[5] O.a.: I mange fremstillinger og programbiblioteker dækker funktionen *afkø* over en kombination af *første* og *fjernForrest*, som både returnerer det første element i køen og fjerner det fra køen.

[6] I virkligheden blev denne historie fundet på i 1883 af den franske matematiker Édouard Lucas som matematisk gåde.

Opgave 3.18 (Kø med to stakke)**.** Forklar, hvordan man kan implementere en kø ved hjælp af to stakke, sådan at hver køoperation tager konstant amortiseret tid.

Hvorfor gør vi os overhovedet tanker om følgetyperne stak, kø og dobbeltkø, når nu *Liste*-datastrukturen allerede stiller samtlige deres operationer og flere til rådighed i konstant tid? Det er der mindst tre grunde til. For det første er programmer mere læsbare og indeholder færre fejl, hvis man udtrykkeligt indskrænker sig til snævrere anvendelsesmønstre. For det andet tillader smallere grænseflader en større fleksibilitet ved implementationen. På grund af deres enkelhed kan stakke og køer bruge specialiserede implementationer, som er mere pladsbesparende end implementationen af den almindelige type *Liste*. Vi vil se nærmere på dette algoritmiske aspekt i resten af afsnittet. Især vil vi stræbe efter implementationer, som er baserede på rækker i stedet for lister. For det tredje er lister dårligt egnede til brug af yderlageret, idet hver adgang til en listeknude kan udløse en blokflytning. Når man derimod repræsenterer stakke og køer med rækker, fører deres sekventielle adgangsmønster til gentagen adgang til samme blok i nærlageret, hvilket leder til en stor forbedring af effektiviteten.

Begrænsede stakke, hvis maksimale størrelse er kendt på forhånd, kan umiddelbart implementeres som begrænsede rækker. For ubegrænsede stakke kan vi bruge ubegrænsede rækker. Stakke kan også let repræsenteres som enkelthægtede lister; staktoppen svarer til listens begyndelse. Køer kan repræsenteres som enkelthægtede lister med en peger på den sidste kunde i listen. Dobbeltkøer kan derimod ikke repræsenteres som enkelthægtede lister på nogen effektiv måde.



Klasse *BegrænsetKø*(n: **N**) **af** *Element*
  b: *Række*[0..n] **af** *Element*
  $h = 0$: **N** *// Indeks på første indgang*
  $t = 0$: **N** *// Indeks på første ledige position*

  Funktion *tom*: $\{0,1\}$; **returner** $h = t$

  Funktion *første*: Element; **antag** $\neg$ *tom*; **returner** $b[h]$

  Funktion *størrelse*: **N**; **returner** $(t - h + n + 1) \bmod (n + 1)$

  Procedure *tilføjBagest*(x: *Element*)
    antag *størrelse* $< n$
    $b[t] := x$
    $t := (t + 1) \bmod (n + 1)$

  Procedure *fjernForrest* **antag** $\neg$ *tom*; $h := (h + 1) \bmod (n + 1)$

Figur 3.8. Implementation af en begrænset kø med en række.

Videre betragter vi nu en rækkebaseret implementationen af en begrænset kø, se figur 3.8. Vi opfatter hertil rækken som en cyklisk struktur, hvis sidste indgang efterfølges af indgang 0. Med andre ord er de mulige rækkeindeks tallene $0, \ldots, n$, og vi opfatter indicene modulo $n + 1$. Vi opretholder to indeks f og t, som begrænser køens gyldige område; køen omfatter rækkepositionerne med indeks i $f..(t-1)$. Indeksene f og t vandrer rundt i ring, efterhånden som indgange stilles i køen og fjernes fra den. Indgangenes cykliske opførsel opnår man ved at regne modulo rækkestørrelsen $n + 1$.[7] Til enhver tid forbliver mindst én rækkeindgang ubenyttet; ellers er det vanskeligt at skelne en fuld kø (med n indgange) fra en tom kø. Implementationen kan let overføres til begrænsede dobbeltkøer. Cykliske rækker støtter også indiceret adgang med $[\cdot]$:

$$\textbf{Operator}[i\colon \mathbf{N}]\colon \mathit{Element}; \textbf{returner}\ b[i + j \bmod n]$$

Rækkebaserede køer og dobbeltkøer kan gøres ubegrænsede ved hjælp af teknikkerne for ubegrænsede rækker fra afsnit 3.2.

Nu har vi set de væsentlige teknikker til implementationen af stakke, køer og dobbeltkøer. Disse teknikker kan kombineres for at opnå løsninger, som er specielt velegnede til meget lange følger og til eksterne lagermedier.

Opgave 3.19 (Lister af rækker)**.** TODO

Opgave 3.20 (Stakke og køer i fjernlageret)**.** TODO

3.5 Sammenligning af lister og rækker

Tabel 3.1 sammenfatter dette kapitels resultater. Rækker har fordele ved indeksbaseret adgang, hvorimod hægtede lister er mere velegnede til følger, som skal ændres (ved tilføjelse og fjernelse) på en vilkårlig plads. Begge tilgange kan realisere stak- og køoperationerne effektiv. Dog skal det siges, at rækker mere effektivt udnytter moderne lagerarkitektur, og at listers tidsgrænser gælder i værste fald.

Enkelthægtede lister kan konkurrere med dobbelthægtede lister på de fleste punkter, men ikke alle. Den eneste fordel ved cykliske rækker i forhold til ubegrænsede rækker er deres effektive implementation af *indsætForrest* og *fjernForrest*.

Spørgsmål om datastrukturens pladseffektivitet er ligeledes ikketrivielle. Hægtede lister er meget kompakte, når selve indgangene er betydeligt større end pegerne. For indgangstyper med lavt pladsbehov er rækker normalt en mere kompakt løsning, fordi ingen ekstra plads skal bruges til pegere. Dette gælder med sikkerhed altid, når rækkestørrelsen er kendt i forvejen, så vi kan bruge begrænsede rækker. Ubegrænsede rækker udviser en god balance

[7] På mange maskiner kan man opnå en mærkbar hastighedsforbedring af indeksberegningerne ved at vælge rækkestørrelsen som en to-potens og erstatte moduloperationen med bitoperationer.

Operation	*Liste*	*EListe*	*URække*	*CRække*	Forklaring af »∗«
$[\cdot]$	n	n	1	1	
størrelse	1^*	1^*	1	1	ikke med *splejs* for flere lister
første	1	1	1	1	
sidste	1	1	1	1	
indsæt	1	1^*	n	n	kun for *indsætEfter*
fjern	1	1^*	n	n	kun for *fjernEfter*
tilføjForrest	1	1	1^*	1^*	amortiseret
tilføjBagest	1	1	n	1^*	amortiseret
fjernForrest	1	n	1^*	1^*	amortiseret
fjernBagest	1	1	n	1^*	amortiseret
sammenføj	1	1	n	n	
splejs	1	1	n	n	
findNæste, ...	n	n	n^*	n^*	i praksis hurtigere pga. lagerblokke

Tabel 3.1. Kørselstider for operationer på følger med n indgange. Det er underforstået, at hver tidsangivelse er omgivet af »$O(\cdot)$«. *Liste* står for dobbelthægtet liste, *EListe* står for enkelthægtet liste, *URække* står for ubegrænset række, *CListe* står for cyklisk række.

mellem pladseffektivitet og ekstra tidsforbrug for kopiering af indgangene ved operationen *realloker*.

3.6 Implementationsaspekter

De fleste programmeringssprog stiller begrænsede rækker til rådighed som grunddatatype. Udover disse findes ubegrænsede rækker, lister, stakke, køer og dobbbeltkøer ofte tilgængeligt i standardbibliotekerne til de gængse imperative programmeringssprog. Alligevel kommer man ofte i en situation, hvor man selv skal implementere listelignende datastrukturer, fx når de behandlende objekter forekommer samtidigt som indgange i flere sammenkædede lister. I den slags implementationer udgør lagerforvaltningen ofte en betydelig udfordring.

3.6.1 C++

Klassen *vector*⟨*Element*⟩ i STL (Standard Template Library) realiserer ubegrænsede rækker. Alligevel tillader de færreste implementationer af STL, at rækker krymper. Klassens funktionalitet omfatter fastlæggelsen af rækkens maximale senere omfang ved det tidspunkt, den bliver skabt. Normalt angiver man hertil på dette tidspunkt et skøn over følgens længde n. På den måde kan man undgå mange udvidelsesoperationer. Sommetider ved man også, hvornår rækkens vækst er afsluttet, og kan så fremtvinge egenskaben $w = n$. På grund af disse raffinementer findes der egentlig ingen grund til at bruge $C++$s indbyggede rækker. En yderligere fordel ved datatypen *vector* er, at dens

instanser automatisk tilintetgøres, når de tilsvarende variables gyldighedsområde bliver forladt. I løbet af fejlfindingsfasen kan man desuden skifte over til varianter, som ved hver rækkeadgang foretager en kontrol af indeksgrænser.

Der findes yderligere nogle aspekter at tage hensyn til, når man ofte har at gøre med voksende om krympende rækker og vil opnå særligt gode kørselstider. I løbet af forstørrelsen og formindskningen skal klassen *vector* flytte indgange fra en række til en anden, hvortil den benytter konstruktøren *Copy* fra klassen *Element*. I de fleste tilfælde vil det være meget tidseffektivt at kalde den maskinnære grundoperation *memcpy*, som kopierer en nærmere betegnet blok af på hinanden følgende bytes. En anden maskinnær optimeringsmulighed betstår i at realisere operationen *realloker* ved hjælp af C-standardfunktionen *realloc*. Lagerforvaltningssystemet kan så muligvis helt undgå at kopiere data.

Snublestenen ved anvendelsen af ubegrænsede rækker er den omstændighed, at pegere på rækkeindgange bliver ugyldige, når en ny række stilles til rådighed. (Man siger, at »referenceintegriteten« er ødelagt.) Man må altså ubetinget sikre, at rækken ikke bliver skiftet ud, mens sådan en peger er i brug. Når den slags udskiftning ikke kan udelukkes, må man referere til indgange ved hjælp af indices i stedet for pegere.

Både STL og LEDA [34] stiller dobbbelthægtede lister til rådighed i form af klassen $list\langle Element\rangle$, og enkelthægtede lister i form af klassen $slist\langle Element\rangle$.

[TODO incomplete]

3.6.2 Java

Siden Java 6 indeholder pakker *util* klassen *ArrayList* for ubegrænsede rækker og klassen *LinkedList* for dobbelthægtede lister. Der findes også en grænseflade *Deque* for dobbeltkøer, som implementeres både af klassen *ArrayDeque* og af klassen *LinkedList*. Stakgrænsefladen *Stack* implementeres i Java som udvidelse af klassen *Vector*.

Mange javabøger forkynder stolt, at Java ikke indeholder pegere. Læseren kan altså spørge sig, hvordan man så kan implementere hægtede lister. Svaret er naturligvis, at referencer til objekter i al væsentlighed er pegere, og at man i hvert fald kan bruge dem som var de pegere. I en vis forstand kan man sågar sige, at Java *kun* har pegere, fordi den eneste adgang til objekter af ikke-elementære datatyper er via referencer – objekter gemmes aldrig i forælderobjektet.

Eksplicit lagerforvaltning er i Java frivillig, fordi køretidssystemets skraldhåndtering tager sig af alle objekter, som ikke længere refereres til.

3.6.3 Python

Python er ikke noget maskinnært sprog og råder ikke over begrænsede rækker som del af de grundlæggende datatyper. Den primære følgetype i Python er typen *List*, som i referenceimplementationen er implementeret som en

ubegrænset række. Standardmodulet *collections* indeholder datatypen *deque* for dobbbeltkøer, som er implementeret som en dobbelthægtet liste for hurtig manipulation i begge ender af følgen, men uden at understøtte hverken hurtig sammenføjning eller operationer på følgens interne indgange som fx *splejs*.

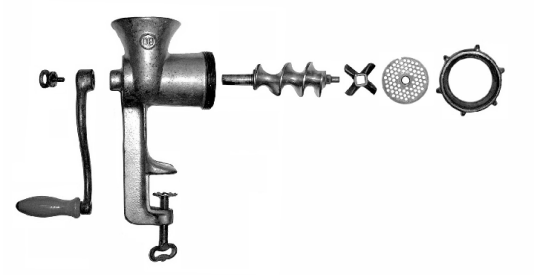
3.7 Historiske anmærkninger og videre resultater

Alle resultaterne i dette kapitler kan betragtes som »folklore« i den forstand, at de er kendt meget længe, og ingen gør krav på at have opfundet dem. Faktisk har vi set, at mange af de grundlæggende abstrakte ideer er ældre end den digitale databehandling. Amortisering er lige så gammel som selve algoritmeanalysen. *Bankkontometoden* og *potentialemetoden* blev foreslået i begyndelsen af 1980'erne af R. E. Brown, S. Huddlestone, K. Mehlhorn, D. D. Sleator og R. E. Tarjan [7, 27, 51, 52]. Oversigtsartiklen [54] gjorde betegnelsen *amortiseret analyse* alment anvendt; sætning **??** er fra [36].

Der findes en rækkelignende datastruktur for følger, som tillader indiceret adgang i konstant tid samt indføjelse og fjernelse på vilkårlig plads i amortiseret tid $O(\sqrt{n})$. Dette opnår man med et relativt enkelt kneb. Rækken realiseres som en følge af delrækker, hvoraf hver indeholder et antal $n' = \Theta(\sqrt{n})$ af indgange, borset fra den sidste, som muligvis indholder færre indgange. Hver delrække organiseres cyklisk som beskrevet i afsnit 3.4. Adgang $[i]$ til position i forskydes derfor med en værdi h. Indgangen med indeks i finder man på plads i mod n' af delrække nummer $\lfloor i/n' \rfloor$. En ny indgang tilføjes i tid $O(\sqrt{n})$ i den delrække, som bestemmes af indeks i. Denne delrække får nu en overskydende indgang. For at opretholde invarianten, at hver delrække har eksakt n' indgange, tilføjes den overskydende indgang som første element til den næste delrække. Denne proces af gentagen videresending af en overskydende indgang gentages nu $n/n' = O(\sqrt{n})$ gange, indtil den sidste delrække er nået. Fjernelser gøres på lignende måde. Sommetider skal der skabes en ny delrække, eller størrelsen n' skal ændres og alle indgange omfordeles. De amortiserede omkostninger ved disse ekstraoperationer kan holdes små. Med yderligere en modifikation kan desuden alle dobbeltkøoperationer gennemføres i konstant tid. I [31] kan den interesserede læser finde nogle snedige implementationer af dobbeltkøer og et implementationsstudie.

4

Hakning og associative rækker



I dette kapitel skal vi betragte en robust og effektiv implementation af en datastruktur til implementation af såkaldte ordbøger eller associative rækker. Denne struktur kaldes en *hakketabel*. [1] Den anden grundlæggende mulighed er en datastruktur baseret på træer, som vi skal se nærmere på i kap. 7. Man skal forestille sig associativ række som en række med potentielt uendelig – eller i det mindste meget stor – indeksmængde, hvoraf dog kun nogle få indeks faktisk er i brug. For eksempel udgøres de mulige indeks af mængden af strenge, og de faktisk benyttede indeks er alle navne i et konkret C-program. Eller de mulige indeks kunne være alle muligheder for at placere skakfigurer på et skakbræt, og de faktisk benyttede index er de stillinger, som optræder i et konkret parti. Associative rækker har mange anvendelser. Oversættere benytter dem til deres symboltabeller som knytter navne i det program, der skal oversættes, til deres tilhørende information. Programmer, som løser problemer inden for kombinatorisk søging, benytter dem ofte for at afgøre, om en bestem konfiguration allere er blevet undersøgt. I skakeksemplet, hvor samme stilling jo kan nås ad forskellige træk, bør programmet jo kun vurdere hver enkelt stilling én gang. Løsningen består i at gemme vurderingen i en associativ række, indiceret af stillingerne. Inden for relationelle databaser gemmer en af de mest benyttede implementationer af *join*-operationen de indgående relationer i en associativ række. Skriptsprog som AWK [1] og Perl [57] bruger associative rækker som deres primære datastruktur. I alle de nævnte eksempler er den associative række normalt implementeret som hakketabel. I opgaverne til dette afsnit betragtes yderligere anvendelser af associative rækker.

Formelt består en associativ række S af en mængde af *indgange*. Til hver indgang e knyttes en nøgle $nøgle(e) \in Nøgle$, hvor $Nøgle$ er mængden af mulige nøgler. Vi skal sikre, at nøglerne i S er entydige, dvs. at forskellige

[1] Foroven: En kødkværn.

indgange i den associative rækker har forskellige nøgler. Associative rækker stiller følgende operationer til rådighed:

$S.opbyg(\{e_1,\ldots,e_n\})$	$S{:=}\{e_1,\ldots,e_n\}$.
$S.indsæt(e : Element)$	Hvis der findes $e' \in S$ med $nøgle(e') = nøgle(e)$, så $S{:=}(S \setminus \{e'\}) \cup \{e\}$. Ellers $S{:=}S \cup \{e\}$.
$S.fjern(x : Nøgle)$	Hvis der findes $e \in S$ med $nøgle(e) = x$, så $S{:=}S \setminus \{e\}$.
$S.find(x : Nøgle)$	Hvis der findes $e \in S$ med $key(e) = x$ så returner e. Ellers returner $\perp$.

Når kun operationerne *opbyg* og *find* er i brug, taler man om en *statisk*, ellers en *dynamisk* datastruktur. Ved udførelsen af operationen $opbyg(\{e_1,\ldots,e_n\})$ skal man sikre, at indgangene $e_1,\ldots,e_n$ har parvist forskellige nøgler. Operationen $indsæt(e)$ opfylder to formål: Når $x = nøgle(e)$ ikke forekommer i S, så tilføjes e. Ellers erstattes indgangen med nøgle x af e. Det svarer til at *opdatere* den indgang i den associative række, som er givet af indeks x. Læg mærke til, at operationerne *find* hhv. *indsæt* og *fjern* i alt væsentligt realiserer læsning og skrivning med »vilkårlig adgang« til en række – heraf navnet associativ række.

Vi går ud fra, at der ved siden af de nævnte operationer findes en måde at gennemgå samtlige indgange i S. Sådan en *forAlle*-operation er normalt ganske enkelt at realisere; vi nøjes derfor med at behandle den i øvelsesopgaverne.

Mængden *Nøgle* indeholder samtlige potentielle rækkeindeks, hvorimod mængden $\{\, nøgle(e) \colon e \in S\}$ kun indeholder de indeks, som er i brug på et vist tidspunkt. I hele kaptitlet skal n betegne størrelsen af S og N betegne størrelsen af *Nøgle*. I typiske anvendelser af associative rækker er N så kæmpestor, at brugen af rækker af størrelse N er udelukket. I stedet vil vi kigge på løsninger, hvis pladsbehov er $O(n)$.

Den grundlæggende idé for at implementere associative rækker som hakketabeller er ganske enkel. En *hakkefunktion* h afbilder mængden *Nøgle* af mulige rækkeindeks til et lille område $\{0,\ldots,m-1\}$ af heltallene. Desuden findes en række t indiceret af $\{0,\ldots,m-1\}$, kaldt *hakketabellen*. For at holde pladsforbruget nede, bør m omtrent svare til antallet af elementer i S. Um den Platzbedarf niedrig zu halten, sollte m ungefähr der Anzahl der Elemente von S entsprechen.

Hakkefunktionen afbilder hver indgang e til en *hakkeværdi* $h(nøgle(e))$. For at forenkle notationen skriver vi $h(e)$ i stedet for $h(nøgle(e))$ for hakkeværdien af e. I det indledende bibliotekseksempel afbilder h hvert studienummer til sine sidste to cifre. Ideelt skulle vi kunne gemme indgangen e på position

$t[h(e)]$ i tabllen t. Når det virker, kan hver af de tre operationer *indsæt*, *fjern* og *find* gennemføres i konstant tid. [2]

Desværre vil det dog ikke altid være muligt, at gemme e i $t[h(e)]$, fordi forskellige indgange kan *kollidere*, dvs. have samme hakkeværdi. Bibliotekseksemplet indeholder allerede en af de mulige udveje: der tillades at stille de bestilte bøger til forskellige brugere i samme hylde. Det fører selvfølgeligt til, at man skal gennemse hele hylden, for at finde en bestemt bestilling. En mere generel form for denne idé fører til *separat kobling*. På hver plads i tabellen gemmes en *mængde* af indgange, hvor en mængde implementeres som enkelthægtet liste. I afsnit 4.1 analyseres separat kobling, hvor vi gør nogle ret optimistiske (og derfor urealistiske) antagelser om hakkefunktionens opførsel. I denne model opnår vi konstant forventet tid for alle tre ordbogsoperationer.

Vi kan holde hakketabellens struktur (men ikke dens analyse!) enkel ved at gå tilbage til den oprindelige idé om at gemme alle indgange i selve tabellen. Når en ny indgang e skal tilføjes, men tabelpladsen $t[h(e)]$ allerede er optaget, gennemgår man de følgende tabelpladser, indtil man finder en ledig plads. I bibliotekseksemplet skulle det svare til, at der højst kan stå én bog på hver hylde. Bibliotekaren ville så benytte de tilstødende hyldepladser for at placere de bøger, som har fået tildelt den samme plads af hakkefunktionen. Denne tilgang, som kaldes *hakning med åben adressering og lineær probering*, er nærmere beskrevet i afsnit 4.3.

Begrebet »hakning« skal minde om at skære, bikse, sønderdele, eller kværne. [3] Det er netop hvad en hakkefunktion gør med nøglerne. Når nøglerne fx er strenge, kan hakkefunktionen skære dem i dele af fast længde, fortolke hver del som heltal, og beregne én værdi, *hakkeværdien*, ud fra den resulterende talfølge. En god hakkefunktion skaber uorden og undgår derved kollisioner.

Opgave 4.1. Givet en mængde M af heltalspar, definieret ved et binær relation R_M. Anvend en associativ række for at afgøre, om R_M er symmetrisk.

Opgave 4.2. Skriv et program, som læser et tekstarkiv og udgiver de 100 mest hyppige ord ved brug af den associativ række.

Opgave 4.3 (Et regnskabssystem)**.** Udvikl en algorithme baseret på en associativ række før følgende problem: Givet et stor database af 3-tupler (transaktion, pris, kundenummer). For hver kunde skal den totale regningssumme udreg-

[2] For at være helt nøjagtige burde vi også tage hensyn til omkostningerne ved at evaluere hakkefunktionen og flytte indgange. For at holde notationen enkel antager vi i dette kapitel, at disse operationer samlet tager konstant tid.

[3] O. a.: Det engelske verbum »*to hash*« går ligesom det danske »at hakke« tilbage til det franske »*hacher*«. Den engelske madret »*hash*« – biskemad gjort på hakkede kød, gerne gårdagens levninger – findes i det danske køkken som »hachis«, og serveres her med brun sovs. Læg mærke til at vi på dansk benytter ordet »hash« for at betegne et udbredt rusmiddel, navngivet efter hampplantens arabiske navn »*hashish*«. Det danske »hash« har altså hverken en etymologisk eller kulinarisk sammenhæng med den primære betydning af det engelske »*hash*«.

nes. Algoritmen skal bruge et lineært antal operationer på den associative række.

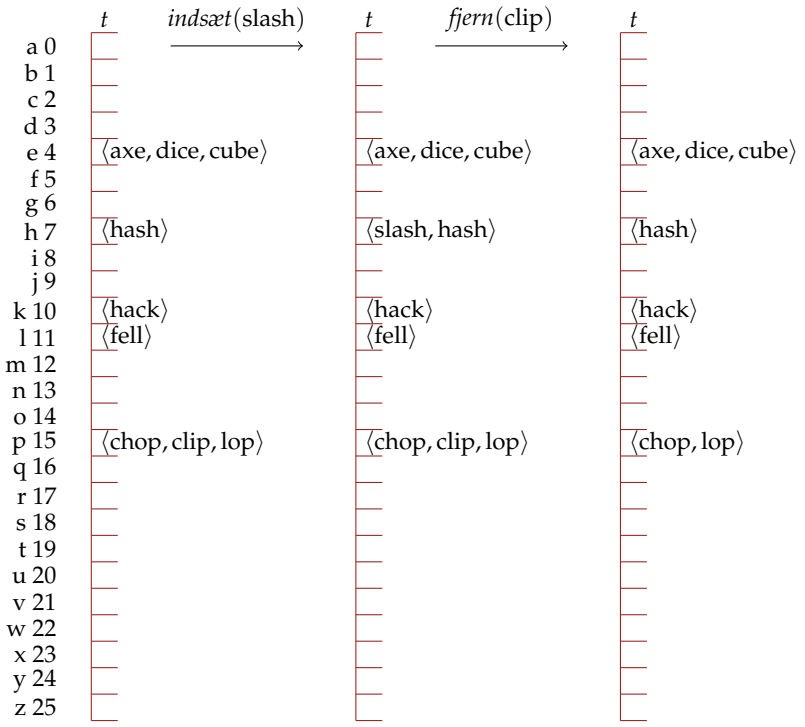
Opgave 4.4 (Gennemløb af hakketabel)**.** Beskriv en implementation af *foralle*-operationen, både for separat kobling og for hakning med åben adressering og lineær probering. Hvad er kørselstiden for din løsning?

4.1 Separat kobling

Hakning med separat kobling gør brug af en række (kaldt »tabellen«) $t[0..m-1]$ af følger, som kan organiseres som enkelthægtede lister (se figur. 4.1). Ideen er, at indgang e befinder sig i følgen $t[h(e)]$. Operationerne for den associative række lader sig implementere let: For at *finde* en indgang med nøgle x gennemgår vi følgen $t[h(x)]$; hvis vi herved støder på en indgang e med $\mathit{nøgle}(e) = x$, returnerer vi den, i modsat fald returnerer vi værdien $\perp$. For at *fjerne* en indgang med nøgle x , gennemgår vi ligeledes følgen $t[h(x)]$; hvis vi herved støder på en indgang e med $\mathit{nøgle}(e) = x$, fjerner vi den fra følgen. For at *indsætte* en indgang e gennemgår vi ligeledes følgen; hvis vi herved støder på en indgang e' med $\mathit{nøgle}(e') = \mathit{nøgle}(e) = x$, erstattes denne med e; ellers tilføjes e på en vilkårlig plads i listen. Operationen $\mathit{opbyg}(\{e_1, \ldots, e_n\})$ realiseres som n indsætninger, hvor vi kan bortse fra at gennemgå den pågældende følge. Derfor har hver operation kørselstid $O(n)$.

Pladsbehovet for hele datastrukturen er $O(n+m)$. For at søge efter, indsætte eller fjerne en indang med nøgle x , skal vi gennemsøge følgen $t[h(x)]$. I værste fald, fx når *find* søger efter en ikke-eksisterende nøgle, skal hele følgen gennemgås. Når det går rigtig galt og hakkefunktionen afbilder samtlige indgange til den samme tabelindgang, er kørselstiden $\Theta(n)$. I værste fald opfører separat kobling sig altså ikke bedre end lineære lister.

Findes der hakkefunktioner, som leder til, at alle følger er korte? Svaret er klart »nej«. En hakkefunktion afbilder mængden af nøgler til værdimængden $\{0, \ldots, m-1\}$, hvorfor der for hver hakkefunktion må findes N/m nøgler, der alle havner på samme plads i tabellen. I de fleste anvendelser gælder $n < N/m$, og i så fald kan separat kobling degenerere til lineær søgning. Vi betragter her tre muligheder for at håndtere denne opførsel. Den første tilgang er gennemsnitsanalyse, , som vi skal beskæftige os med i dette afnit. Den anden tilgang består i at benytte randomisering og vælge hakkefunktionen tilfældigt fra en mængde af hakkefunktioner. Herom nærmere i dette og næste afsnit. Den tredje tilgang består i at ændre algoritmen. For eksempel kan vi prøve at tilpasse hakkefunktionen til mængden af nøgler, som skal gemmes i øjeblikket. Dette skal vi undersøge i afsnit 4.5 og se, at det leder til god opførsel i værste fald.



Figur 4.1. Separat kobling. Vi benytter en række t af følger. I eksemplet indeholder datastrukturen en mængde af ord (korte engelske synonymer af »*hash*«), som er gemt med en hakkefunktion, som afbilder et ord til dets sidste bogstavs position i alfabetet, dvs. til en værdi i $\{0, \ldots, 25\}$. Åbenbart er dette ikke nogen særlig god hakkefunktion.

Lad H være mængden af alle funktioner fra *Nøgle* til $\{0, \ldots, m-1\}$. Vi antager, at hakkefuktionen h er valgt tilfældigt fra H. [4] Vi vil nu vise, at der for hver fast mængde S af n indgange gælder, at operationerne *indsæt*, *fjern* og *find* tager forventet tid $O(1 + n/m)$.

Sætning 4.1. *Hvis separat kobling bruges til at gemme n indgange i en hakketabel med m tabelpladser, og hakkefunktionen er valgt tilfældigt fra H, så har operationerne* indsæt, fjern *og* find *forventet kørselstid $O(1 + n/m)$.*

Bevis. I beviset har vi brug for begreberne »stokastisk variabel« og »forventningsværdi« fra sandsynlighedsregningen, samt at forventningsværdien er lineær, som beskrevet i afsnit A.3. Lad h være en tilfældigt valgt funktion fra H. Vi betragter kørselstiden for en operation *indsæt*, *fjern* eller *find* for en

[4] Denne antagelse er helt urealistisk. Der findes m^N mange funktioner in H, derfor skal der bruges $N \log m$ bits alene for at beskrive en funktion i H. Det strider mod målet at reducere datastrukturens pladsbehov fra N til n.

fast nøgle x. Hver af disse operationer behøver konstant tid samt tiden for at gennemsøge følgen $t[h(x)]$. De forventede kørselstid er derfor $O(1 + E[X])$, hvor den stokastiske variabel X angiver længden af følgen $t[h(x)]$. Lad S med $|S| = n$ være mængden af indgange, som er gemt i tabellen. Betragt for hver indgang $e \in S$ den stokastiske *indikatorvariabel* X_e, som angiver, om h afbilder e til samme tabelplads som x:

$$X_e = \begin{cases} 1\,, & \text{hvis } h(e) = h(x)\,; \\ 0\,, & \text{ellers}\,. \end{cases}$$

Der er to tilfælde. Hvis S ikke indeholder nogen indgang med nøglen x, så er $X = \sum_{e \in S} X_e$. Hvis derimod $\mathit{nøgle}(e_0) = x$ for $e_0 \in S$, så er $X = 1 + \sum_{e \in S \setminus \{e_0\}} X_e$. I det første tilfælde følger ved linearitet af forventningsværdien, at

$$E[X] = E\left[\sum_{e \in S} X_e\right] = \sum_{e \in S} E[X_e] = \sum_{e \in S} \Pr(X_e = 1)\,.$$

En tilfældig valgt funktion afbilder e med samme sandsynlighed til hver af de m tabelpladser, uafhængigt af $h(x)$. Derfor har vi $\Pr(X_e = 1) = 1/m$, hvilket medfører $E[X] = n/m$. I det andet tilfælde, hvor nøgle x findes ikke i S, får vi på samme måde $E[X] = 1 + (n-1)/m < 1 + n/m$. I begge tilfælde er sætningen bevist. □

Vi kan opnå lineært pladsbehov og forventet konstant kørselstid for alle tre operationer ved at sikre, at der gælder $m = \Theta(n)$. Til dette formål kan vi lade tabellens større tilpasse sig n, som beskrevet i afsnit 3.2 om ubegrænsede rækker.

Opgave 4.5 (Ubegrænsede hakketabeller)**.** Forklar, hvordan man med separat kobling kan sikre, at der altid gælder $m = \Theta(n)$. Du kan gå ud fra en hakkefunktion $h' \colon \mathit{Nøgle} \to \mathbf{N}$. Sæt $h(x) = h'(x) \bmod m$ og tilpas tabelstørrelsen til n.

Opgave 4.6 (Ubenyttet lagerplads)**.** Separat kobling spilder plads i den forstand, at nogle tabelpladser forbliver ubenyttede. Beregn det forventede antal tomme tabelpladser under antagelse af, at hakkefunktionen er tilfældig. *Vink*: Definer de stokatiske indikatorvariabler $Y_0, \ldots, Y_{m-1}$, med $Y_i = 1$ hvis $t[i]$ er tom.

Opgave 4.7 (Gennemsnitstilfælde)**.** Antag, at hakkefunktionen afbilder mængden af mulige nøgler uniformt over tabellen, dvs. $|\{\, x \in \mathit{Nøgle} \colon h(x) = i\}| \leq \lceil N/m \rceil$ for alle $i \in \{0, \ldots, m-1\}$. Antag desuden, at at vi har gemt en tilfældig mængde af nøgler i tabellen, dvs. at mængden af nøgler i S er en tilfældig n-delmængde af *Nøgle*. Vis, at der gælder for hver tabelplads i, at det forventede antal indgange, der afbildes til i, er højst $\lceil N/m \rceil \cdot n/N \approx n/m$.

4.2 Universalhakning

Udeladt

4.3 Hakning med lineær probering

Separat kobling kaldes også *åben hakning*, fordi der til kollisionshåndtering bruges lagerplads uden for den egentlige tabel, fx i form af lister. Derimod taler man om *lukket* hakning, hvis samtlige ingange gemmes på selve tabelpladserne. Ved at slække kravet om at gemme indgang e på plads $t[h(e)]$, opstår fleksibilitet – også kaldt »åben adressering« –, der gør det muligt helt at undgå at bruge en sekundær datastruktur. Der findes mange undersøgelser af metoder for *åben adressering* [41], jf. [19, kap. 3.3]. Vi vil nøjes med at se på den enkleste metode, som kaldes *lineær probering*. Ubenyttede tabelpladser frå en særlig indgang $\perp$. En indgang e findes enten i $t[h(e)]$ eller længere til højre i tabellen, som vi tænker os vandret. Vi vil dog kun holde os fra den foretrukne plads $t[h(e)]$, når det er absolut nødvendigt: når e er gemt i $t[i]$ med $i > h(e)$, så skal alle plaserne fra $h(e)$ til og med $i - 1$ være optaget af andre indgange. (Denne *invariant* opretholdes af operationerne.)

Implementationerne af *indsæt* og *find* er ligetil. For at udføre *indsæt*(e) gennemsøger vi tabellen sekventielt, begyndende i tabelplads $[h(e)]$, intil vi enten møder en indgang e' med $\mathit{nøgle}(e') = \mathit{nøgle}(e) = x$ eller en ledig plads i tabellen. I det første tilfælde erstattes e' med e, i det andet bliver e gemt på den ledige plads, se fx figur 4.2. Fremgangsmåden for at søge efter nøgle x med operationen *find*) er lignende: vi gennemsøger tabellen sekventielt fra plads $t[h(x)]$, indtil vi finder en indgang med nøgle x. Søgningen afbrydes som mislykket, så snart vi når en tabelplads indeholdende $\perp$. Det lyder jo meget enkelt, men der optræder en lille komplikation. Hvad sker der, hvis vi under en indsættelse eller søgning når til slutningen af tabel t? En enkel reparation består i at altid afsætte yderligere m' tabelpladser uden for de m pladser, der omfattes af hakkefunktionen h. For »velartede« hakkefunktioner burde det række med en værdi for m', som er langt mindre end m, for at tage hånd om overløb. En anden mulighed, som betragtes i opgave 4.8 (jf. afsnit 3.4), består i at behandle tabllen som en cyklisk række. Denne variant er mere robust, men også langsommere.

Implementationen af sletteoperationen *fjern* er ikke lig så enkel. Det er ikke nok at overskive den slettede ingang med $\perp$, fordi dette ville gøre invarianten ugyldig. (Antag fx, at $h(x) = h(z)$ og $h(y) = h(x) + 1$, og at nøglerne x, y og z insættes i den rækkefølge. Så står z på plads $h(x) + 2$. Når ny y bliver fjernet ved at overskrive plads $h(x) + 1$ med $\perp$, kan z ikke længere nås i en søgning.) Der er tre løsninger på dette problem. Den første mulighed er at forbyde fjernelse. Den anden er at markere fjernede indgange, men ikke overskrive dem med $\perp$. Søgninger afbrydes som før, når vi møder $\perp$, men ikke når vi møder en indgang, som er markeret som slettet. Problemet med

indsæt axe, chop, clip, cube, dice, fell, hack, hash, lop, slash

an	bo	cp	dq	er	fs	gt	hu	iv	jw	kx	ly	mx
0	1	2	3	4	5	6	7	8	9	10	11	12
⊥	⊥	⊥	⊥	*axe*	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
⊥	⊥	*chop*	⊥	*axe*	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
⊥	⊥	*chop*	*clip*	*axe*	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
⊥	⊥	*chop*	*clip*	*axe*	*cube*	⊥	⊥	⊥	⊥	⊥	⊥	⊥
⊥	⊥	*chop*	*clip*	*axe*	*cube*	*dice*	⊥	⊥	⊥	⊥	⊥	⊥
⊥	⊥	*chop*	*clip*	*axe*	*cube*	*dice*	⊥	⊥	⊥	⊥	*fell*	⊥
⊥	⊥	*chop*	*clip*	*axe*	*cube*	*dice*	⊥	⊥	⊥	*hack*	*fell*	⊥
⊥	⊥	*chop*	*clip*	*axe*	*cube*	*dice*	*hash*	⊥	⊥	*hack*	*fell*	⊥
⊥	⊥	*chop*	*clip*	*axe*	*cube*	*dice*	*hash*	*lop*	⊥	*hack*	*fell*	⊥
⊥	⊥	*chop*	*clip*	*axe*	*cube*	*dice*	*hash*	*lop*	*slash*	*hack*	*fell*	⊥

fjern(clip)

⊥	⊥	*chop*	~~*clip*~~	*axe*	*cube*	*dice*	*hash*	*lop*	*slash*	*hack*	*fell*	⊥
⊥	⊥	*chop*	*lop*	*axe*	*cube*	*dice*	*hash*	~~*lop*~~	*slash*	*hack*	*fell*	⊥
⊥	⊥	*chop*	*lop*	*axe*	*cube*	*dice*	*hash*	*slash*	~~*slash*~~	*hack*	*fell*	⊥
⊥	⊥	*chop*	*lop*	*axe*	*cube*	*dice*	*hash*	*slash*	⊥	*hack*	*fell*	⊥

Figur 4.2. Hakning med lineær probering. Tabellen med 13 pladser er tegnet vandret og indeholder engelske synomymer for »(at) hakke«. Hakkefunktion afbilder hvert ords sidste bogstav til tallene $\{0, \ldots, 12\}$ som antydet i figuren: »a« og »n« sendes til 0, »b« og »o« 1 osv. Ordene indsættes et at gangen i alfabetisk rækkefølge. Til sidst fjernes ordet »*clip*«. I figuren er vist, hvordan tilstanden i tabellen forandrer sig efter hver operation. De pladser, som inspiceres under udførelsen af den aktuelle operation, er fremhævede.

denne fremgangsmåde er, at antallet af ikke-tomme tabelpladser (optaget eller markeret) vokser, så søgninger bliver efterhånden langsommere end nødvendigt. Dette problem kan afhjælpes ved regelmæssigt at genopbygge tabellen og faktisk fjerne de markerede indgange. Den tredje mulighed er at opretholde invarianten. Hvordan skal det ske? Antag, at vi vil fjerne indgangen på tabelplads i. Vi overskriver pladsen med $\perp$; dette kan skabe et »hul«, som forstyrrer søgningen efter indgange med mindre hakkeværdier. Nu gennemsøger vi tabelpladserne til højre for i sekventielt og leder efter indgagne, som ødelægger invarianten: Sæt j til $i + 1$. Hvis $t[j] = \perp$, er vi færdige.

Ellers lad f være den indgang, som står på plads $t[j]$. Hvis $h(f) > i$, kan vi øge j med 1 og gentage. Hvis $h(f) \leq i$, så ville »hullet« på plads i ødeløgge invarianten for indgang f, og søgningen efter $nøgle(f)$ skulle slå fejl. Vi flytter derfor f til $t[i]$, flytter $\perp$ til $t[j]$, og sætter $i{:=}j$. (Mao. lader vi f og »hullet« bytte plads og gemmer »hullets« nye position på plads i.) Ny begynder vi en ny gennemgang fra $j{:=}j+1$. Figur 4.2 viser et eksempel.

Opgave 4.8 (Cyklisk lineær probering)**.** Implementer varianten af lineær probering, hvor tabelstørrelsen ikke er $m + m'$, men m. For forhindre overløb i tabellens højre ende, organiseres gennemsøgningen cyklisk. (1) Tilpas operationerne *indsæt* og *fjern* ved at erstatte inkrementationen $i{:=}i+1$ med $i{:=}(i+1) \bmod m$. (2) Specificer udsagnet $mellem(i, j, k)$ om tabelpositioner i $\{0, \ldots, m-1\}$, som er *sand* hvis og kun hvis i befinder sig skarpt mellem j og k ifølge den cykliske opfattelse. (3) Omformuler invarianten, så den bruger *mellem*. (4) Tilpas operationen *remove*.

Opgave 4.9 (Hakketabel med lineær probering)**.** Implementer en *ubegrænset* hakketabel, altså en hakketabel, hvis størrelse ikke er kendt i forvejen, med lineær probering og universelle hakkefunktioner. Vælg en ny hakkefunktion, hver gang tabellen bliver mindre eller større. Lad α, β og γ være konstanter med $1 < \gamma < \beta < \alpha$, som vi kan vælge. Opret en tæller til antallet n af gemte indgange. Udvid tabellen til en ny størrelse $m = \beta n$, så snart $n > m/\gamma$. Formindsk tabellen til en ny størrelse $m = \beta n$, så snart $n < m/\alpha$. Medmindre du vil bruge cyklisk lineær probering som beskrevet i opgave 4.8, sæt tabelstørrelsen til $m' = m/\delta$ for noget $\delta < 1$ og vælg en ny hakkefunktion (uden at ændre m eller m'), når tabellen er ved at løbe over i højre side. (Bemærkning: Der kan opstå problemer, når man forsøger at kombinere enkle universelle hakkeklasser med lineær probering, jf. afsnit 4.7.)

4.4 Sammenligning af separat kobling og lineær probering

Vi har nu set to forskellige fremgangsmåder for realisering af hakketabeller. Hvilken er bedre? Det spørgsmål kan vi ikke besvare på baggrund af en teoretisk analyse, fordi svaret afhænger af anvendelsessituationen og af mange tekniske parametre. Vi skal her nævne nogle kvalitative askpekter og skitsere nogle eksperimentelle resultater.

En af fordelene ved separat kobling er »referenceintegritet«, hvilket betyder, at en indangs lagerposition ikke ændrer sig, efter den er blevet indrettet. Senere søgninger kommer altid til at finde indgangen på præcis samme plads i lageret. Derfor kan man arbejde med pegere på listeknuder. I modsætning hertil vil fjernelser ved lineær probering flytte indgange; absolutte referencer eller pegere på flyttede indgange bliver herved ugyldige.

Fordelen ved lineær probering er, at hver adgang til tabellen arbejder med et sammenhængende lagersegment. Moderne processorers lagerarkitektur er optimeret for dette adgangsmønster, mens de er langsomme til at følge

pegere ved gennemløb af en lineær liste, som ikke får plads i gemmelageret. Et problem ved lineær probering er, at søgetiderne vokser hurtigt, når antal indgange nærmer sig tabelstørrelsen, og at tabellen kan flyde over. Ved separat kobling forekommer ingen sådanne abrupte overgange – de forventede søgetider forbliver små, selv når nøgleantallet overstiger tabelstørrelsen. På den anden side skal separat kobling bruge lagerplads til pegerne, som man ved lineær probering ville kunne have brugt til en større tabel. En retfærdig sammenligning må derfor tage hensyn til hele lagerforbruget og ikke bare tabelstørrelsen.

Vi har implementeret begge metoder og gennemført udførlige eksperimenter. Resultatet var, at begge metoder opfører sig nogenlunde lige godt, hvis de har samme lagerplads til rådighed. Forskellene er så små, at konklusionerne afhænger af implementationsdetaljer, specielle egenskaber ved oversætteren, operativsystemet og den benyttede regner. Derfor nævner vi her ingen eksakte resultater.

Vi fandt dog implementationen af separat kobling vanskeligere. For at kunne konkurrere med lineær probering, skal alle forbedringer i afsnit 4.6 gennemføres. Separat kobling bliver meget langsommere, hvis man sløser med implementationen eller organiserer lagerstyringen dårligt.

Med hensyn til matematiske argumenter skal man nævne, at udsagn i stil med sætning **??** *kun* gælder for kombinationen af separat kobling og c-universelle klasser og ikke for lineær probering med c-universelle klasser. For at opnå lineær søgetid har lineær probering brug for hakkefunktioner med stærkere sandsynlighedsegenskaber, eller antage fuld tilfældighed. (Se hertil afsnit 4.7.)

4.5 Perfekt hakning

Udeladt

4.6 Implementationsaspekter

Udeladt

4.7 Historiske anmærkninger og yderligere resultater

Udeladt

5

Sortering og udvalg

Sandsynligvis har læseren en intuitiv opfattelse af, hvad *sortering* drejer sig om. Alligevel vil vi definere opgaven formelt. Input er en følge $s = \langle e_1, \ldots, e_n \rangle$ af n indgange. Hver indgang e_i er forsynet med en *nøgle* $k_i = key(e_i)$. Nøglerne er taget fra et ordnet univers, dvs. at der defineret en *lineær* (også kaldt *total*) ordning $\leq$ (se appendix A). For at holde notationen overskuelig, overfører vi ordningsrelationen fra nøgler til indgange, så vi skriver $e \leq e'$ når vi mener $key(e) \leq key(e')$. (Idet forskellige indgange kan have samme nøgle, danner den lineære ordning $\leq$ altså en lineær *præ*ordning på indgangene, se appendix A). Opgaven består i at skabe en følge $s' = \langle e'_1, \ldots, e'_n \rangle$, som er en permutation af s, så der gælder $e'_1 \leq \cdots \leq e'_n$. Læg mærke til, at rækkefølgen af indgange med samme nøgle er vilkårlig.

Selvom en og samme datatype kan tillade mange forskellige sammenligningrelationer, har visse datatyper en naturlig ordningsrelation: For tal er dette den sædvanlige ordning. For tupler, strenge og følger bruges den leksikografiske ordning (se appendix A). Den leksikografiske ordning på strenge findes i forskellige udgaver. Man kan se store og små bogstaver som ens eller forskellige, og i mange sammenhænge behandles bogstaver med accenter eller ligaturer på forskellige måder.

Opgave 5.1. Givet lineære ordninger $\leq_A$ på A og $\leq_B$ på B, definer en lineær ordning på $A \times B$.

Sortering er et allestedsnærværende algoritmisk værktøj, som ofte optræder som forberedende skridt i mere komplekse algoritmer. Vi skal se nærmere på nogle eksempler.

- *Forberedelse til hurtig søgning*. I afsnit 2.5, som omhandlede binærsøgning, så vi at søgning i en ordnet telefonbog er lettere at gennemføre, såvel for mennesker som for datamaskiner. Derudover understøtter en ordnet telefonbog flere operationer som at finde samtlige indgange inden for et område . Søgning behandles indgående i kap. 7. For at søge i uordnede mængder kan man også bruge hakning.
- *Gruppering*. Ofte ønsker man at bringe identiske indgange på en og samme plads, måske for at tælle dem, eller for at identificere og fjerne eller håndtere dubletter. Også her tilbyder hakning en alternativ løsning. Men

sortering byder på flere fordele: vi skal møde hurtige, pladseffektive, deterministiske sorteringsmåder, som skalerer til kæmpestore datamængder.

- *Behandling i sorteret rækkefølge*. Mange algoritmer forenkles væsentligt af at behandle input i sorteret rækkefølge. I opgave 5.2 finder man et eksempel. Andre eksempler er Kruskals algoritme i afsnit 11.3 og nogle af algoritmerne for rygsæksproblemet i kap. 12. Det kan også være nyttigt at tænke på sortering, når man prøver at løse opgave 8.6 om intervalgrafer .

I afsnit 5.1 skal vi præsentere nogle enkle sorteringsalgoritmer. Deres tidsforbrug er kvadratisk, men de er alligevel brugbare for små problemstørrelser. Vi skal også se på nogle maskinnære optimeringer. I afsnit 5.2 præsenteres *flettesortering*, en enkel del-og-hersk-algoritme, som har kørselstid $O(n \log n)$. Som vi skal vise i afsnit 5.3, er denne tidsgrænse optimal for alle *sammenligningsbaserede* sorteringsalgoritmer, dvs. algoritmer, so betragter indgangene som »sorte kasser«, som kan tillader parvis sammenligning og flytning. *Kviksortering*, som behandles i afsnit 5.4, beror ligeledes på en del-og-hersk-tilgang. Måske er den den mest anvendte sorteringsalgoritme. Kviksortering er et fremragende eksempel på en randomiseret algoritme. Algoritmens grundlæggende idé fører også til en enkel algoritme for *udvalgsproblemet*: I afsnit 5.5 skal vi se, hvordan man kan finde den $k$te-mindste indgang i en mængde af størrelse n i forventet tid $O(n)$. Når nøgler har en kendt struktur, som fx tal eller talfølger, eller kan behandles som sådan, kan man sortere hurtigere end den nedre grænser fra afsnit **??**. Hertil skal nøgler udsættes for andre operationer end sammenligninger, fx ved at udnytte nøglens numeriske repræsentation. Endelig skal afsnit 5.7 præsentere meget gode algoritmer på basis af *kviksortering* og *flettesortering*, som tillader sortering af data, som ikke får plads i hovedlageret.

Opgave 5.2 (Planlægning: Hotelreservationer)**.** En hoteldirektør har n reservationer til næste sæson. Hotellet har k identiske værelser. En reservation består af en ankomstdato og en afrejsedato. Hoteldirektøren vil vide, om hotellets værelser er nok til at opfylde alle ønsker. Konstruer en algoritme, som efter sortering løser problemet i tid $O(n)$. *Vink*: Sorter først (multi-)mængden af alle ankomst- og afrejsetidspunkter. Behandl derefter den denne mængde i ét gennemløb i stigende rækkefølge.

Opgave 5.3 (Sortering med lille nølgemængde)**.** Konstruer en algoritme, som sorterer n indgange i forventet tid $O(k \log k + n)$, hvis der kun forekommer k forskellige nøgler i input. *Vink*: Anvend et senere resultat i dette kapitel, nemlig at man kan sortere k nøgler i tid $O(k \log k)$. Kombiner hakning og sortering.

Opgave 5.4 (Resultatkontroll)**.** Det er nemt at kontrollere, om en sorteringsmåde faktisk har skabt en voksende følge. Derimod er det mindre nemt at kontrollere, om den resulterende følge er en permutation af det oprindelige input. Hvis nøglerne er heltal, findes der dog en enkel og hurtig Monte Carlo-algoritme til formålet: (a) Vis, at $\langle e_1, \ldots, e_n \rangle$ er en permutation af $\langle e'_1, \ldots, e'_n \rangle$,

hvis og kun hvis polymomiet q givet ved

$$q(x) = \prod_{i=1}^{n}(x - e_i) - \prod_{i=1}^{n}(x - e'_i)$$

er nulpolynomiet. Her er x en variabel. (b) For givet $\epsilon > 0$ lad p være et primtal med $p > \max\{n/\epsilon, e_1, \ldots, e_n, e'_1, \ldots, e'_n\}$. Idéen er nu at evaluere polynomiet foroven modulo p i $x = r$ for en tilfældig værdi $r \in \{0, \ldots, p-1\}$. Vis at dersom $\langle e_1, \ldots, e_n\rangle$ *ikke* er en permutation af $\langle e'_1, \ldots, e'_n\rangle$, så er $q(r) = 0$ med sandsynlighed højst ϵ. *Vink*: Et polynomium af grad højst n, som ikke er nulpolynomiet modulo p, har højst n rødder, når man regner modulo p.

5.1 Enkle sorteringsteknikker

Vi præsenterer nu to enkle sorteringsteknikker, *udvalgssortering* og *indsættelsessortering*.

Ved udvalgssortering gentager man følgende skridt: Vælg den første indgang i inputfølgen, slet den der, og føj den til outputfølgen. Outputfølgen er tom til at begynde med; processen fortsætter, til inputfølgen er tom. For eksempel,

$$\langle\,\rangle, \langle 4,7,1,1\rangle \rightsquigarrow \langle 1\rangle, \langle 4,7,1\rangle \rightsquigarrow \langle 1,1\rangle, \langle 4,7\rangle \rightsquigarrow \langle 1,1,4\rangle, \langle 7\rangle \rightsquigarrow \langle 1,1,4,7\rangle, \langle\,\rangle\,.$$

Algoritmen kan implementeres sådan, at den kan nøjes med en enkelt række af n indgange og arbejder *på plads*, dvs. at den ud over inputrækken og et konstant antal lagerceller for iterationsvariable o. lign. ikke behøver mere plads. Kørselstiden er kvadratisk.

I afsnit **??** beskrives en mere raffineret implementation af udvalgssortering, som forvalter inputfølgen som en *prioritetskø*. Prioritetskøen gør det muligt at vælge og fjerne en mindste indgang effektivt. Det resulterende algoritme har kørselstid $O(n \log n)$ og er meget brugt. Den er effektiv, deterministisk, arbejder på plads, og inputfølgen kan udvides dynamisk med nye elementer, så længe disse er større end de allerede udtagne. Den sidste egenskab er af stor betydning for den hændelsesstyret diskret simulering, hvor hændelser behandles i tidslig voksende rækkefølge og en hændelse kan lede til nye hændelser, som skal udføres senere.

Ved udvalgssortering udtages den næste indgang fra inputfølgen med henblik på at bevare løkkeinvarianten, nemlig at outputfølgen er sorteret under hele forløbet. Ved indsættelsessortering gælder samme løkkeinvariant; her tager man dog en *vilkårlig* indgang fra inputfølgen og sikrer i stedet, at den anbringes på rette plads i outputfølgen. For eksempel,

$$\langle\,\rangle, \langle 4,7,1,1\rangle \rightsquigarrow \langle 4\rangle, \langle 7,1,1\rangle \rightsquigarrow \langle 4,7\rangle, \langle 1,1\rangle \rightsquigarrow \langle 1,4,7\rangle, \langle 1\rangle \rightsquigarrow \langle 1,1,4,7\rangle, \langle\,\rangle\,.$$

Fig. 5.1 viser en implementation af indsættelsessortering, som arbejder på plads i ind- og outputrækken $a[1..n]$. Implementationen realiserer den algoritmiske idé på en oplagt måde, bortset fra et enkelt kneb, som gør det muligt at

styre den indre løkke med kun én sammenligning. Når indgangen e, som er den næste at blive indføjet, er mindre end alle tidligere indføjede indgange, kan man den stilles forrest uden videre sammenligninger. Ellers passerer man hele den sorterede delfølge fra højre til venstre, så længe der ikke står en indgang, som er større end e. Denne proces må nødvendigvis standse, fordi $a[1] \leq e$. .

I værstefald er indsættelsessortering temmelig langsom. Hvis fx input er sorteret i aftagende orden, bliver hver eneste indgang flyttet hele vejen til plads $a[1]$, hvilket betyder, at den $i$te iteration af den ydre løkke skal flytte i indgange. Samlet giver dette

$$\sum_{i=2}^{n}(i-1) = -n + \sum_{i=1}^{n} i = \frac{n(n+1)}{2} - n = \frac{n(n-1)}{2} = \Omega(n^2)$$

mange flytninger af indgange (se (A.11)).

```
Procedure indsættelsessortering(a: Række[1..n] af Element)
  for i:=2 til n udfør
    invariant a[1] ≤ ··· ≤ a[i − 1]
    //sæt a[i] på rette plads
    e:=a[i]
    hvis e < a[1] så //nyt minimum
      for j:=i ned til 2 udfør a[j]:=a[j − 1]
      a[1]:=e
    ellers //anvend a[1] som vogterindgang
      for j:=i ned til −∞ så længe a[j − 1] > e udfør a[j]:=a[j − 1]
      a[j]:=e
```

Figur 5.1. Indsættelsessortering.

På trods af den dårlige værstefaldsopførsel er indsættelsessortering en nyttig metode. Den er hurtig for små input (fx $n \leq 10$) og kan derfor bruges i bunden af del-og-hersk-algoritmer for sorteringsproblemet. Desuden er input i mange anvendelsesområder i forvejen allerede »næsten sorteret« ; i disse siutationen er indsættelsessortering hurtig.

Opgave 5.5 (Næsten sorteret input)**.** Vis, at indsættelsessortering har kørselstid $O(n + D)$ med $D = \sum_i |r(e_i) - i|$, hvor $r(e_i)$ er *rangen* (dvs. positionen) af e_i i det sorterede output.

Opgave 5.6 (Gennemsnitsanalyse)**.** Betragt mængden af alle $n!$ mange input, som består af permutationer af tallene fra 1 til n. Vis, at den gennemsnitlige kørselstid for indsættelsessortering på disse input er $\Omega(n^2)$. *Vink*: Vis, at der i gennemsnit en tredjedel af alle de indgange, der står i den oprindelige rækkes højre tredjedel, skal flyttes til rækkens venstre tredjedel. Kan du finde et bedre

bevis, som desuden viser, at der i gennemsnit kræves $\frac{1}{4}n^2 - O(n)$ gennemløb af den indre løkke?

Opgave 5.7 (indsættelsessortering med få sammenligninger)**.** Modificer den indre løkke af den rækkebaserede indsættelsessorteringsalgoritme i fig. 5.1, så den kun behøver $O(n \log n)$ sammenligninger mellem indgange. *Vink*: Binærsøgning, se afsnit 2.5. Hvad er kørselstiden af denne variant?

Opgave 5.8 (Effektiv indsættelsessortering?)**.** Brug datastrukturen for ordnede følger fra kap. 7 for at skabe en variant af indsættelsessortering med kørselstid $O(n \log n)$.

Opgave 5.9 (Formel verifikation)**.** Brug din foretrukne verifikationsformalisme (fx hoarelogik) for at vise formelt, at indsættelsessortering producerer en permutation af inputfølgen.

5.2 Flettesortering

Flettesortering eller en enkel og direkte anvendelse af del-og-hersk-princippet Den usorterede følge deles op i to omtrent lige store dele. Begge dele sorteres rekursivt, hvorefter begge delfølger flettes sammen til én ordnet følge. Denne tilgang er effektiv, fordi fletningen af to ordnede følger a og b kan gøres enkelt og hurtigt. Den mindste indgang for begge følger er enten den første indgang i a eller den første indgang i b. Vi flytter denne mindste indgang til outputfølgen, fortsætter med at finde den næstmindste indgang, osv. Processen gentages, indtil alle indgange er vandret over i output. I fig. 5.2 vises algoritmen i pseudokode, og i fig. 5.3 vises et eksempel på en kørsel på konkret input. Hvis de indgående følger er (enkelt-)hægtede lister (se afs. **??**), behøves hverken reservation eller frigørelse af listeindgange. Hver iteration i den indre løkke af proceduren *flet* fører til én sammenligning af indgange og flytter én ingang til outputfølgen; dette kan gøres i konstant tid. Derfor er kørselstiden for fletning lineær.

Sætning 5.1. *Proceduren* flet *anvendt på to følger af samlet længde n udføres i tid $O(n)$ og bruger højst $n - 1$ sammenligninger af indgange.*

For kørselstiden af flettesortering får vi følgende:

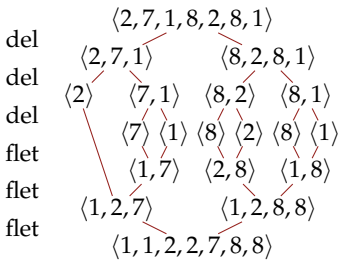
Sætning 5.2. *Flettesortering af en følge af længde n udføres i tid $O(n \log n)$ og bruger højst $\lfloor n \log n \rfloor$ sammenligninger af indgange.*

Bevis. Lad $C(n)$ være antallet af sammenligninger, som i værste fald gennemføres på en inputfølge af n indgange. Åbenbart er $C(1) = 0$, og fra sætning 5.1 får vi $C(n) \leq C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + n - 1$. Hovedsætningen for rekursionsligninger (sætning 2.3) giver da $C(n) = O(n \log n)$. Vi skal nu give to direkte beviser, som leverer eksplicitte konstanter. Det første bevis etablerer $C(n) \leq 2n\lceil \log n \rceil$, det andet sågar $C(n) \leq n \log n$.

```
Funktion flettesorter(⟨e_1, ..., e_n⟩): Følge af Element
  hvis n = 1 så returner ⟨e_1⟩
  ellers returner flet(flettesorter(⟨e_1, ..., e_⌊n/2⌋⟩),
                       flettesorter(⟨e_⌊n/2⌋+1, ..., e_n⟩))

// Flet to ordnede følger, repræsenteret som lister
Funktion flet(a, b: Følge af Element): Følge af Element
  c:=⟨⟩
  løkke
    invariant både a, b og c er i stigende ordning og ∀e ∈ c, e' ∈ a ∪ b: e ≤ e'
    hvis a.tom                 så c.sammenføj(b); returner c
    hvis b.tom                 så c.sammenføj(a); returner c
    hvis a.første ≤ b.første så c.flytTilBagest(a.første)
    ellers                        c.flytTilBagest(b.første)
```

Figur 5.2. Flettesortering.



a	*b*	*c*	Operation
⟨1,2,7⟩	⟨1,2,8,8⟩	⟨⟩	flyt fra *a*
⟨2,7⟩	⟨1,2,8,8⟩	⟨1⟩	flyt fra *b*
⟨2,7⟩	⟨2,8,8⟩	⟨1,1⟩	flyt fra *a*
⟨7⟩	⟨2,8,8⟩	⟨1,1,2⟩	flyt fra *b*
⟨7⟩	⟨8,8⟩	⟨1,1,2,2⟩	flyt fra *a*
⟨⟩	⟨8,8⟩	⟨1,1,2,2,7⟩	sammenføj *b*
⟨⟩	⟨⟩	⟨1,1,2,2,7,8,8⟩	

Figur 5.3. Kørsel af *flettesorter*$(\langle 2,7,1,8,2,8,1\rangle)$. *Til venstre* vises strukturen af de rekursive kald i i *flettesorter*. *Til højre* vises kørslen af *flet* i det yderste kald.

Hvis n er en 2-potens, definerer vi $D(1) = 0$ og $D(n) = 2D(n/2) + n$. Et enkelt induktionsbevis viser, under anvendelse af ligningen $\log(n/2) = \log n - 1$, at der gælder $D(n) = n \log n$ når n er en topotens. Lad nu n være vilkårlig. Vi betragter det entydigt bestemte k, som opfylder $2^{k-1} < n \leq 2^k$, og påstår, at der gælder $C(n) \leq D(2^k)$. Heraf følger direkte $C(n) \leq D(2^k) = 2^k \cdot k \leq 2n\lceil \log n \rceil$. Uligheden $C(n) \leq D(2^k)$ viser vi ved induktion efter k. For $k = 0$ gælder $n = 1$ og $C(1) = 0 = D(1) = D(2^0)$. For $k > 1$ indser vi først, at der gælder $\lfloor n/2 \rfloor \leq \lceil n/2 \rceil \leq 2^{k-1} < n$. Ved at anvende rekursionsligningen og induktionsantagelsen får vi nu uligheden

$$C(n) \leq C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + n - 1 \leq 2D(2^{k-1}) + 2^k - 1 \leq D(2^k)\,.$$

Det afslutter det første bevis.

Vi vender os nu mod den andet, mere nøjeregnende bevis, hvor vi vil etablere grænsen

$$C(n) \leq n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1$$

ved induktion efter n. For $n = 1$ gælder påstanden åbenbart. Antag altså $n > 1$. Betragt det k, som opfylder $2^{k-1} < \lceil n/2 \rceil \leq 2^k$, dvs. $k = \lceil \log(\lceil n/2 \rceil) \rceil$. Så gælder ifølge induktionshypotesen, at $C(\lceil n/2 \rceil) \leq \lceil n/2 \rceil k - 2^k + 1$. For $2^{k-1} < \lfloor n/2 \rfloor$ følger ligeledes af induktionshypotesen, at $C(\lfloor n/2 \rfloor) \leq \lfloor n/2 \rfloor k - 2^k + 1$. For $\lfloor n/2 \rfloor = 2^{k-1}$ giver induktionshypotesen, at $C(\lfloor n/2 \rfloor) \leq \lfloor n/2 \rfloor (k-1) - 2^{k-1} + 1$. Men højresiden er lig med $2^{k-1} \cdot (k-1) - 2^{k-1} + 1 = 2^{k-1} \cdot k - 2^k + 1 = \lfloor n/2 \rfloor k - 2^k + 1$. Vi får altså i hvert fald:

$$\begin{aligned} C(n) &\leq C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + n - 1 \\ &\leq \big(\lfloor n/2 \rfloor k - 2^k + 1\big) + \big(\lceil n/2 \rceil k - 2^k + 1\big) + n - 1 \\ &= nk + n - 2^{k+1} + 1 = n(k+1) - 2^{k+1} + 1 = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1\,. \end{aligned}$$

Nu skal vi vise, at $k = \lceil \log n \rceil$ opfylder uligheden $nk - 2^k + 1 \leq n \log n$. For $n = 2^k$ gælder $2^k \cdot k - 2^k + 1 \leq 2^k \cdot k$ åbenbart. For $n < 2^k$ indser vi $nk - 2^k + 1 \leq n(k-1) + (n - 2^k + 1) \leq n \log n$.

Grænsen for kørselstiden etableres med en lignende rekursionsligning (jf. opgave 2.8). □

Flettesortering er den foretrukne metode for at sortere hægtede lister. Derfor bruges denne fremgangsmåde ofte i funktions- og logikprogrammeringssprog, idet disse sprog bruger lister som den primære datastruktur. I afsnit 5.3 skal vi se, at flettesortering i grunden er optimal med hensyn til antallet af sammenligninger. Derfor er det også et godt valg, når sammenligninger er dyre. Ved implementationen baseret på rækker har flettesortering desuden den fordel, at den gennemgår lagerafsnit sekventielt. Det fører til effektivitetsforedele i situationer med et lagerhierarki; mere om dette emne i afsnit 5.7. Alligevel er flettesortering ikke det foretrukne metode til at sortere rækker, fordi *flet* ikke arbejder på plads, men behøver yderligere lager. (En mulig udvej betragtes i opgave 5.15.)

Opgave 5.10. Forklar, hvordan man føjer k nye indgange til en ordnet liste i tid $O(k \log k + n)$.

Opgave 5.11. Fig. 5.2 beskriver proceduren *flet* for lister, men *flettesortering* bare i abstrakt følgeterminologi. Beskriv *flettesortering* detaljeret for hægtede lister.

Opgave 5.12. Implementer flettesortering i et funktionsprogrammeringssprog.

Opgave 5.13. Implementer flettesortering for rækker i dit foretrukne imperative programmeringssprog. Ved siden af inputrækken bruger man en hjælperække af størrelse n; de to rækker skiftes til at indeholde mellemresultater og outputfølgen under opbygning. Afgør eksperimentelt, om den faktiske kørselstid bliver bedre ved at skifte fra rekursion til indsættelsessortering for små input. I så fald, ved hvilken længde er det bedst at skifte?

Opgave 5.14. I vores beskrivelse af *flet* indeholder hvert løkkegennemløb tre sammenligninger – en sammenligning af indgange og to tomhedstest for løkkeafslutning. Udvikl en variant, som i stedet benytter vogterindgange og derved klarer sig med kun én test for løkkeafslutning. Går det uden brug af attrapknuder i enden af listen?

Opgave 5.15. I opgave 3.19 beskrives en repræsentation af følger som en liste af rækker kaldt »blokke«. Implementer fletning og flettesortering for denne datastruktur. Under fletningen bør tomme inputblokke genbruges til output. Sammenlign flettesorterings tids- og pladsbehov for denne datastruktur, for enkelthægtede lister og for rækker, også under hensyntagen til konstante faktorer.

5.3 En nedre grænse

Algoritmer giver øvre grænser på problemers beregningskompleksitet. På baggrund af diskussionen i sidste afsnit ved vi, at n objekter kan sorteres i tid $O(n \log n)$. Kan det gøres hurtigere – måske helt ned til lineær tid? Hvis svaret skulle være »ja«, burde vi kunne fremvise en bedre algoritme og dens analyse. Men hvordan skulle vi kunne begrunde svaret »nej«? Vi skulle jo bevise at ingen algoritme, nok så genial, kan sortere i tid $o(n \log n)$. Sådan et bevis kaldes en *nedregrænse*. Og hvad er nu svaret? Svaret er såvel »ja« som »nej«. Det er »nej«, hvis vi holder os til sammenligningsbaserede algoritmer, og det er »ja«, hvis vi tillader andre slags algoritmer. Sorteringsmåder, som ikke er baserede på sammenligninger, behandles i afsnit 5.6.

Men hvad er egentlig en sammenligningsbaseret sorteringsalgoritme? Input er en mængde $\{e_1, \ldots, e_n\}$ af n indgange, og algoritmen må ikke gøre andet med disse indgange end parvise sammenligninger og flytninger. Den eneste måde, at få information om indgange, er at sammenligne dem med hinanden. Især har algoritmen ikke lov til at benytte nøglens repræsentation, fx som bitstreng. Til sidste leverer algoritmen en outputfølge $s = \langle e'_1, \ldots, e'_n \rangle$, som er en permutation af inputfølgen og opfylder $e'_1 \leq \cdots \leq e'_n$. Deterministiske sammenligningsbaserede algoritmer kan man opfatte som træer, kaldt sammenligningstræer. Dette indses på følgende måde. Vejene i træet repræsenterer de forskellige beregninger, hvor træets indre knuder svarer til sammenligninger, og en yderligere knude i slutningen af beregningen svarer til output. Idet beslutninger i beregningen kun kan være baserede på nøglesammenligninger, kan vejene kun forgrene sig i sammenligningsknuder. Det springende punkt er, at hvilken sammenligning der udføres hernæst *udelukkende* afhænger af resultatet af de tidligere sammenligninger. Figur 5.4 viser et sammenligningstræ for tre indgange.

Når algoritmen standser, må den have høstet så meget information om input, så den kan lægge sig fast på en permutation π så inputindgangene i rækkefølgen $(e_{\pi(1)}, \ldots, e_{\pi(n)})$ er sorterede. Hvornår kan algoritmen lægge sig

fast? For at forstå dette, hjælper følgende tankeeksperiment. Antag, at inputnøglerne alle er forskellige og betragt hver af de $n!$ mange permutationer π af mængden $\{1,\ldots,n\}$. Permutationen π svarer til situationen $e_{\pi(1)} < e_{\pi(2)} < \cdots < e_{\pi(n)}$. For vilkårligt π kan vi nu udføre algoritmen og besvare alle den sammenligninger i overensstemmelse med denne rækkefølge. Dette leder til et blad i sammenligningstræet, som vi kalder ℓ_π.

Lemma 5.1. *Givet et naturligt tal n, en sorteringsalgoritme og det tilhørende sammenligningstræ for input af længde n. For to forskellige permutationer π og σ af $\{1,\ldots,n\}$ er bladene ℓ_π og ℓ_σ forskellige.*

Bevis. Antag modsætningsvist, at π og σ er forskellige, men ℓ_π og ℓ_σ er ens. Betragt to inputfølger $(e_1,\ldots,e_n)$ og $(e'_1,\ldots,e'_n)$ med $e_{\pi(1)} < \ldots < e_{\pi(n)}$ og $e'_{\sigma(1)} < \ldots < e'_{\sigma(n)}$. Hver af dem leder til samme blad i sammenligningstræet; dvs. at algoritmen foretager samme omflytning for hver. Men så må mindst en af følgende blive fejlsorteret, hvilket er en modstrid med at algortimen er en sorteringsalgoritme. □

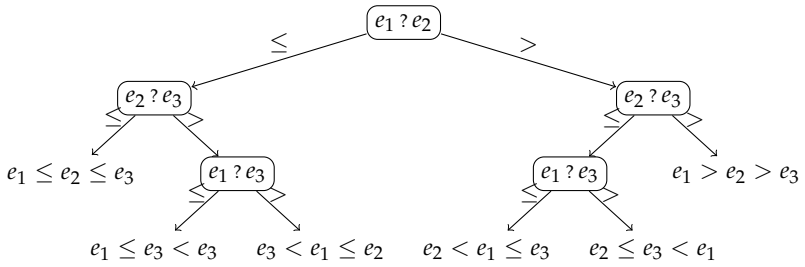
Lemmaet medører, at hvert sammenligningstræ, som svarer til en sorteringsalgoritme, skal have mindste $n!$ blade, ét for hvert permutation. Idet et binærtræ af dybde T har højst 2^T blade, må der gælde

$$2^T \geq n! \quad \text{og derfor} \quad T \geq \log n!\,.$$

Fra en enkel nedre grænse (A.8) på $n!$ fås da

$$T \geq \log n! \geq \log\left(\left(\frac{n}{e}\right)^n\right) = n\log n - n\log e\,.$$

Vi sammenfatter dette i følgende sætning:



Figur 5.4. Et træ til sortering af tre indgange. Først sammenligner vi e_1 med e_2. Hvis $e_1 \leq e_2$, sammenligner vi e_2 med e_3. Hvis $e_2 \leq e_3$, så gælder $e_1 \leq e_2 \leq e_3$, og vi er færdige. I modsat fald sammenligner vi e_1 med e_3. For hvert af de to mulige resultater er vi færdige. Hvis $e_1 > e_2$, så sammenligner vi e_2 med e_3. Hvis $e_2 > e_3$, så har vi $e_1 > e_2 > e_3$ og er færdige. Ellers sammenligner vi e_1 med e_3. Begge mulige resultater afslutter sorteringen. I værste fald sker der tre sammenligninger. Det gennemsnitlige antal sammenligninger over alle seks mulige opstillinger er $\frac{1}{6}(2+3+3+2+3+3) = \frac{8}{3}$.

Sætning 5.3. *Hver sammenligningsbaserede sorteringsalgoritme bruger $n \log n - O(n)$ sammenligninger i værste fald.*

Vi nævner uden bevis, grænsen også gælder for randomiserede sorteringsalgoritmer samt for det gennemsnitlige tilfælde, hvor alle permutationer af inputobjekterne optræder med samme sandsynlighed. Grænsen gælder til og med for problemet at afgøre, om inputfølgen indeholder en dublet, hvilket jo umiddelbart virker som en nemmere problem.

For næste sætning går vi ud fra, at inputfølgen består af n forskellige nøgler, og at hver permutation har samme sandsynlighed $1/n!$ for at sortere inputfølgen.

Sætning 5.4. *Hver sammenligningsbaserede sorteringsalgoritme bruger i gennemsnit $n \log n - O(n)$ sammenligninger, dvs.*

$$\frac{\sum_\pi d_\pi}{n!} = n \log n - O(n)\,,$$

hvor summen er over alle $n!$ permutationer π af mængden $\{1, \ldots, n\}$, og d_π er dybden af bladet ℓ_π.

Opgave 5.16 (Sammenligningsbaseret minimum)**.** [TODO]

Opgave 5.17. *Unikat*-problemet er at afgøre, om elementerne i en givet følge af n indgange med nøgler fra en mængde med total ordning alle er forskellige. Vis, at en sammenligningsbaseret algoritme for dette problem behøver $\Omega(n \log n)$ sammenligninger. Hvorfor står dette ikke i modstrid med det faktum, at unikatproblemet kan løses i forventet lineær tid med hakning?

Opgave 5.18 (Nedre grænse for det gennemsnitlige tilfælde)**.** [TODO]

Opgave 5.19 (Optimal sortering af få indgange)**.** [TODO]

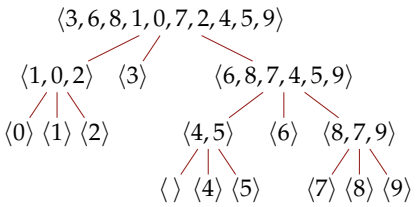
5.4 Kviksortering

Kviksortering er en del-og-hersk-algoritme , som på en vis måde er komplementær til flettesortering fra afsnit 5.2. Kviksortering udfører broderparten af sit arbejde *inden* de rekursive kald. Ideen er at placere de givne indgange i en eller flere følger på en sådan måde, at de tilsvarende nøgleområder ikke overlapper hinanden. I så fald er det nok, at sortere delfølgerne rekursivt og sammenføje resultaterne. For at opnå en fuldkommen dualitet til flettesortering, også i analysen, skulle vi opdele inputfølgen i to følger af eksakt samme længde. Desværre er dette ikke ligetil. Vi kan dog komme i nærheden af dette ideal ved at vælge en tilfældig indgang, som vi bruger til at dele følgen i to. Denne indgang kalder vi *spaltenøgle*. . Lad p betegne den valgte spaltenøgle. Indgangende inddeles nu efter, om de er større, lig med eller mindre end p. Disse tre dele af input organiseres som hver sin følge. Figur 5.5

viser denne ide skematisk, mens figur 5.6 viser forløbet på et eksempelinput. Kviksortering har forventet kørselstid $O(n \log n)$, hvilket vi skal vise i afsnit 5.4.1. I afsnit 5.4.2 diskuterer vi forbedringer af implementationen, som gør kviksortering til den i praksis mestbenyttede sorteringsalgoritme.

Funktion *kviksorter*(s: Følge **af** Element): Følge **af** Element
  hvis $|s| \leq 1$ **så returner** s *// Basis*
  vælg $p \in s$ tilfældigt *// Spaltenøgle*
  $a := \langle e \in s : e < p \rangle$
  $b := \langle e \in s : e = p \rangle$
  $c := \langle e \in s : e > p \rangle$
  returner sammenføjningen af *kviksorter*(a), b og *kviksorter*(c)

Figur 5.5. Skematisk fremstilling af kviksortering af lister.



Figur 5.6. Kørsel af *kviksorter* (fig. 5.5) på $\langle 3,6,8,1,0,7,2,4,5,9 \rangle$, hvor spaltenøglen altid er valgt som første element i delfølgen. Det første kald af *kviksorter* benytter spaltenøglen 3 og danner delfølgerne $\langle 1,0,2 \rangle$, $\langle 3 \rangle$ og $\langle 6,8,7,4,5,9 \rangle$. Det rekursive kald for den tredje delfølge benytter spaltenøglen 6 og danner delfølgerne $\langle 4,5 \rangle$, $\langle 6 \rangle$ og $\langle 8,7,9 \rangle$.

5.4.1 Analyse

Ved analysen af kørselstiden af kviksortering på inputfølge $s = \langle e_1, \ldots, e_n \rangle$ koncentrerer vi os på antal gennemførte sammenligninger. Herved tæller vi »trevejs-sammenligningen« af indgange, som resulterer i »mindre end«, »lig med« eller »større end«, som én operation. Med udgangspunkt i antal sammenligninger er de andre andre operationers bidrag til den samled kørselstid kun konstante faktorer og additive termer.

Lad $C(n)$ betegne antallet af sammenligninger, som kviksortering gennemfører i væreste fald på nogen inputfølge af længde n og noget valg af spaltenøgle.

Opførelsen i værste fald er nemt at beskrive. Delfølgerne a, b og c i figur. 5.5 dannes ved at spaltenøglen sammenlignes med alle andre indgange. Dette

kræver $n-1$ sammenligninger. Hvis vi lader k hhv. k' betegne antallet af indgange, som er mindre hhv. større end spaltenøglen, får vi rekursionligningen givet ved $C(0) = C(1) = 0$ og

$$C(n) \leq n - 1 + \max\left\{ C(k) + C(k') \colon 0 \leq k \leq n-1, 0 \leq k' < n-k \right\} .$$

Ved induktion indses, at

$$C(n) \leq \frac{n(n-1)}{2} = \Theta(n^2) .$$

Værste fald intræffer, når alle indgange er forskellige, og algoritmen altid vælger den største eller den mindste indgang som spaltenøgle. Da gælder $C(n) = n(n-1)/2$.

Den forventede opførelse er betydligt bedre. Vi begynder med at gøre det nogenlunde plausibelt, at det forventede antal sammenligninger er $O(n \log n)$, og giver derefter et nøjagtig bevis for den præcise grænse $2n \ln n$. Vi koncentrerer os på det tilfælde, hvor alle indgange har forskellige nøgler. I de andre tilfælde er kørselstiden endnu lavere, fordi en spaltenøgle, som optræder mere end én gang, fører til en længere delfølge b i midten, som ikke skal behandles videre. For hver indgang e_i vil vi lade X_i betegne antal sammenligninger mellem e_i og en spaltenøgle. Da er $\sum_i X_i$ det totale antal sammenligninger. Hver gang, e_i sammenlignes med en spaltenøgle, havner det i en strengt kortere delfølge. Derfor er $X_i \leq n-1$ (hvilket vi ville kunne bruge til et alternativt bevis for den kvadratiske grænse). Vi vi kalde en sammenligning »god for e_i«, hvis e_is nye delfølge højst har længde $\frac{3}{4}$ af den gamle delfølge. Hver indgang e_i kan højst have $\log_{4/3} n$ gode sammenligninger. Sandsynligheden, at den valgte spaltenøgle ikke er god for e_i, er højst $\frac{1}{2}$, fordi i så fald spaltenøglen må høre til enten den mindste eller den største fjerdedel af følgens indgange. Derfor er $E[X_i] \leq 2 \log_{4/3} n$. Ved addition af denne ulighed for $i \in \{1, \ldots, n\}$ fås $E\left[\sum_i X_i\right] = O(n \log n)$.

Vi skal nu give en anden og bedre grænse.

Sætning 5.5. *Det forventede antal $\overline{C}(n)$ af sammenligninger, som kviksortering bruger på følger af længde n, er*

$$\overline{C}(n) \leq 2n \ln n < 1{,}39 n \log n .$$

Bevis. Lad os skrive $s' = \langle e'_1, \ldots, e'_n \rangle$ for inputfølgens indgange i voksende rækkefølge. Indgangene e'_i og e'_j sammenlignes med hinanden højst én gang, nemlig når den ene af dem vælges som spaltenøgle, mens den anden (stadig) tilhører samme delfølge. Derfor kan vi tælle sammenligninger ved at betragte indikatorvariable X_{ij} for $i < j$ defineret ved

$$X_{ij} = \begin{cases} 1, & \text{hvis } e'_i \text{ og } e'_j \text{ bliver sammenlignet;} \\ 0, & \text{ellers.} \end{cases}$$

Vi får

$$\overline{C}(n) = E\left[\sum_{i=1}^{n}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n}\sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n}\sum_{j=i+1}^{n} \Pr(X_{ij}=1)\,.$$

Den midterste omskrivning følger af linearitet af middelværdien (se afsnit A.2). Den sidste ulighed bruger $E[X_{ij}] = \Pr(X_{ij} = 1)$, hvilket gælder, idet vi har at gøre med indikatorvariable (se afsnit A.3). Inden vi kan fortsætte med at forenkle udtrykket for $\overline{C}(n)$, skal vi bestemme $\Pr(X_{ij} = 1)$.

Lemma 5.2. *For vilkårlige $i < j$ gælder* $\Pr(X_{ij} = 1) = \dfrac{2}{j-i+1}$.

Bevis. Betragt mængden $M = \{e'_i, \ldots, e'_j\}$ med $(j - i + 1)$ elementer. Så længe ingen af indgangene i M vælges til spaltenøgle, bliver e'_i og e'_j ikke sammenlignet, og alle indgange i M forbliver i samme delfølge i de rekursive kald. Før eller siden må nogen indgang $p \in M$ blive valgt til spaltenøgle. Hver indgang i M kan vælges med samme sandsynlighed, nemlig $1/|M|$. Hvis $p = e'_i$ eller $p = e'_j$, bliver e'_i og e'_j sammenlignet, dvs. $X_{ij} = 1$. Ellers havner e'_i og e'_j i hver sin delfølge og sendes videre til forskellige rekursive kald, så de aldrig kan blive sammenlignet og der derfor gælder $X_{ij} = 0$. Vi konkluderer $\Pr(X_{ij} = 1) = \frac{2}{|M|} = \frac{2}{j-i+1}$. □

Vi kan nu færdiggøre beviset for sætning 5.5:

$$\begin{aligned}\overline{C}(n) &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} \Pr(X_{ij}=1) = \sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n}\sum_{k=2}^{n-i+1} \frac{2}{k} \\ &\le \sum_{i=1}^{n}\sum_{k=2}^{n} \frac{2}{k} = 2n\sum_{k=2}^{n}\frac{1}{k} = 2n(H_n - 1) \le 2n(1 + \ln n - 1) = 2n\ln n\,.\end{aligned}$$

Til de sidste tre skridt bruges egenskaberne ved det $n$te harmoniske tal $H_n := \sum_{k=1}^{n} 1/k \le 1 + \ln n$ se (A.12). Afslutningsvist kan man vurdere $2n\ln n = (2\ln 2)n\log n$ med $2\ln 2 = 1{,}386\,29\ldots$. □

Læg mærke til, at beregningerne ligner dem i afsnit 2.7 for problemet at finde mellemmaksima, selvom det drejer sig om et helt andet problem.

5.4.2 Forbedringer

Udeladt

5.5 Udvalg

Udeladt

5.6 Grundtalssortering

Overskriften for dette afsnit kunne være »At sprænge den nedre grænse«, men det er åbenbart meningsløst. Den nedre grænse for sortering er et absolut umulighedsresultat, som udelukker løsningen til et bestemt problem i en bestemt beregningsmodel hurtigere end den givne grænse. Grænsen kan also ikke sprænges. Men den gælder selvfølgeligt kun for algoritmer, som falder ind under beregningsmodellen. Grænsen udelukker ikke, at der findes hurtigere løsninger i en beregningsmodel med et mere righoldigt repertoire af operationer. I en vis forstand kan man sågar benytte den nedre grænse som rettesnor for at finde en hurtigere algoritme.

Hvad indebærer disse overvejelser for sorteringsproblemet? Indtil nu har vi holdt os til sammenligningsbaserede sorteringsmedoder. Algoritmens eneste mulighed for at lære noget om indgangene størrelse var ved at sammenligne dem parvist. Men hvis nøglerne har struktur, findes der virkningsfulde måder at gå mere information, som gør det muligt for os at gennembryde ned nedre grænse på $\Omega(n \log n)$, som gælder for sammenligningsbaserede sorteringsmetoder. Både tal og strenge har struktur – de repræsenterer følger af cifre hhv. tegn.

Vi begynder med en meget enkel algoritme, *nøglesortering*, som er hurtig, når nøglerne er små, naturlige tal , fx i området $\{0, \ldots, K-1\}$. Algoritmen behøver kun tid $O(n+K)$. Vi forestiller os en række $b[0..K-1]$ af *bøtter* , som er tomme til at begynde med. Vi gennemgår den givne følge og lægger hver indgang med nøgle k i bøtte $b[k]$. Det kan gøres i konstant tid per indgang, fx ved at realisere hver bøtte som hægtet liste. Afslutningsvist sammenføjer vi alle disse lister for at opnå en sorteret udgiftsfølge. I figur 5.7 er algoritmen angivet i pseudokode. Når indgangene fx består af par med førstekomponent i området $\{0, \ldots, 3\}$, får man for indgiftsgølgen

$$s = \langle (3,a), (1,b), (2,c), (3,d), (0,e), (0,f), (3,g), (2,h), (1,i) \rangle$$

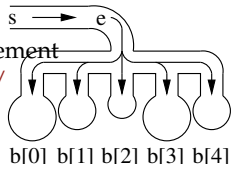
bøtterne

$$b = [\langle (0,e), (0,f) \rangle,\ \langle (1,b), (1,i) \rangle,\ \langle (2,c), (2,h) \rangle,\ \langle (3,a), (3,d), (3,g) \rangle]$$

og udgiften $\langle (0,e), (0,f), (1,b), (1,i), (2,c), (2,h), (3,a), (3,d), (3,g) \rangle$. Dette eksempel tydliggør en væsentlig egenskab ved nøglesortering. Det drejer sig nemlig om en *stabil* sorteringsmetode, hvilket betyder, at indgange med samme nøgle har samme rækkefølge i indgiften som i udgiften. Hertil er det væsentligt, at vi hægter indgangene *sidst* i den pågældende bøtteliste.

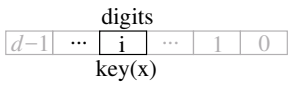
Bøttesortering danner grundlaget for konstruktionen af en sorteringsmetode for større nøgler. Idéen bag *grundtalssortering* er at udtrykke naturlige tal med cifre i området $\{0, \ldots, K-1\}$. Så bruges *nøglesorter* én gang per cifferposition. Figur 5.8 viser udgaven af grundtalssortering for nøgler i området $\{0, \ldots, K^d - 1\}$, med kørselstid $O(d(n+K))$. Indgangene sorteres først med hensyn til de lavest betydende ciffer (heraf navnet lbc), hernæst efter den

Procedure nøglesorter(s: Følge **af** Element)
  $b = \langle\langle\,\rangle, \ldots, \langle\,\rangle\rangle$: *Række*$[0..K-1]$ **af** Følge **af** Element
  for hvert $e \in s$ **udfør** $b[key(e)].indsætBagest(e)$ //
  s:=Sammenføjningen af $b[0], \ldots, b[K-1]$



Figur 5.7. Sortering af nøgler i området $\{0, \ldots, K-1\}$.

Procedure LbcGrundtalssorter(s: Følge **af** Element)
  for i:=0 **til** $d-1$ **udfør**
    Definer *nøgle*(x) som $(x \text{ div } K^i) \bmod K$ // x
    nøglesorter(s)
    invariant s er sorteret mht. det $j$te ciffer for $j \in \{i, \ldots, 0\}$



Figur 5.8. Sortering af nøgler i området $\{0, \ldots, K^d - 1\}$ med lbc-grundtalssortering.

næstlaveste position osv., indtil der til sidst sorteres efter det højest betydende ciffer. Metoden kaldes *lbc-grundtalssortering*. Det er ikke umiddelbart klart, at metoden virker. Korrektheden baserer sig på stabiliteten af bøttesortering. På grund af stabiliteten er indgangene med samme ciffer på position i allerede sorterede med hensyn til positionerne $\{i-1, \ldots, 0\}$, når *nøglesorter* sorterer dem med hensyn til cifret på position i. Hvis vi fx har $K = 10$, $d = 3$ og

$$s = \langle 017, 042, 666, 007, 111, 911, 999 \rangle\,,$$

så sker sorteringen sådan:

$$\begin{aligned} s &= \langle 11\underline{1}, 91\underline{1}, 0\underline{4}2, 6\underline{6}6, 0\underline{1}7, 0\underline{0}7, 9\underline{9}9 \rangle\,, \\ s &= \langle 0\underline{0}7, 1\underline{1}1, 9\underline{1}1, 0\underline{1}7, 0\underline{4}2, 6\underline{6}6, 9\underline{9}9 \rangle\,, \\ s &= \langle \underline{0}07, \underline{0}17, \underline{0}42, \underline{1}11, \underline{6}66, \underline{9}11, \underline{9}99 \rangle\,. \end{aligned}$$

Procedure uniformSorter(s: Følge **af** Element)
  n:=$|s|$
  $b = \langle\langle\,\rangle, \ldots, \langle\,\rangle\rangle$: *Række*$[0..n-1]$ **af** Følge **af** Element
  for hvert $e \in s$ **udfør** $b[\lfloor key(e) \cdot n \rfloor].tilføjBagest(e)$
  for i:=0 **til** $n-1$ **udfør** sorter $b[i]$ i tid $O(|b[i]| \log |b[i]|)$
  s:=sammenføjningen af $b[0], \ldots, b[n-1]$

Figur 5.9. Sortering af tilfældige nøgler i intervallet $[0, 1)$.

Grundtalssortering bruges også i en anden variant, hvor man begynder sorteringen på den højest betydende cifferposition, kaldt *hbc-grundtalssortering*. Vi anvender først *bøttesorter* på cifret på den mest betydende position, hvorefter vi fortsætter med at rekursivt sortere hver bøtte for sig. Det eneste problem

er, at de enkelte bøtter kan blive meget mindre end K, så det kan blive for dyrt at bruge *bøttesorter*. I den situation må vi ty til en anden algoritme. Metoden virker særligt godt, når vi kan gå ud fra, at nøglerne er uniformt fordelte. Mere nøjagigt vil vi antage, at nøglerne er reelle tal med $0 \leq key(e) < 1$. Algoritmen *uniformSorter* i figur 5.9 skalerer nøglerne op til tal mellem 0 und $n-1 = |s|-1$ og fordelen dem i n bøtter, således at nøglerne i intervallet $[i/n, (i+1)n)$ havner i bøtte $b[i]$. Hvis vi fx betragter $s = \langle \frac{8}{10}, \frac{4}{10}, \frac{7}{10}, \frac{6}{10}, \frac{3}{10} \rangle$, får vi fem bøtter, som hver har ansvar for et interval af længde $\frac{2}{10}$:

$$b = [\langle\,\rangle, \quad \langle \tfrac{3}{10} \rangle, \quad \langle \tfrac{4}{10} \rangle, \quad \langle \tfrac{7}{10}, \tfrac{6}{10} \rangle, \quad \langle \tfrac{8}{10} \rangle] ;$$

kun bøtte $b[3] = \langle \frac{7}{10}, \frac{6}{10} \rangle$ udgør et ikketrivielt delproblem. Proceduren *uniformSorter* er meget effektiv på *tilfældige* nøgler.

Sætning 5.6. *Når n nøgler er valgt uafhængigt og uniformt i intervallet $[0,1)$, så sorteres de af* uniformSorter *i tid $O(n)$ i det gennemsnitlige tilfælde og i tid $O(n \log n)$ i det værste fald.*

Bevis. Vi betragter kun gennemsnitstilfældet, analysen af værste fald er henlagt til en øvelse. Den samlede kørselstid T udgøres af $O(n)$ operationer for forelingen af nølger i bøtter og sammenføjningen af sorterede lister samt tiden for sortering af de enkelte bøtter. Sidstnævnte behøver tid $\sum_{i<n} T_i$, hvor vi skriver T_i for sorteringstiden af bøtte $b[i]$.

Ved linearitet af middelværdien (A.2)

$$E\left[\sum_{i<n} T_i\right] = \sum_{i<n} E[T_i] = n \cdot E[T_0]\,,$$

hvor den anden lighed forklares af, at fordelingen af antal elementer er den samme i hver bøtte, idet nøglerne var uniformt fordelte. Vi har altså

$$E[T] = O(n) + n \cdot E[T_0]\,.$$

Det står tilbage at vise $E[T_0] = O(1)$. Vi skal vise noget stærkere, nemlig at $E[T_0] = O(1)$ gælder, selv hvis sorteringsalgoritmen bruger kvadratisk tid, som fx ved indsættelsessortering. Beviset hertil forløber på lignende måde som analysen af hakning i kapitel 4.

Sæt $B_0 = |b[0]|$. Da er $E[T_0] = O(E[B_0^2])$. Den stokastiske variabel B_0 er binomialfordelt (A.7) med n forsøg og sandsynlighed $1/n$; derfor gælder

$$\Pr(B_0 = k) = \binom{n}{k} \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \leq \frac{n^k}{k!}\frac{1}{n^k} = \frac{1}{k!} \leq \left(\frac{e}{k}\right)^k ,$$

hvor den sidste ulighed følger af den enkle vurdering (A.8) af $k!$. Vi kan nu udregne

$$E[B_0^2] = \sum_{k \le n} k^2 \cdot \Pr(B_0 = k) \le \sum_{k \le n} k^2 \left(\frac{e}{k}\right)^k \le \sum_{k \le 5} k^2 \left(\frac{e}{k}\right)^k + e^2 \sum_{k \ge 6} \left(\frac{e}{k}\right)^{k-2}.$$

Den første sum i det sidste udtryk er åbenbart en konstant. For den anden sum indser vi, at der for $k \ge 6$ gælder $e/k \le \frac{1}{2}$. Derfor er summen begrænset af den geometriske række $\sum_{k \ge 0} (\frac{1}{2})^k = 2$, se (A.13). Vi konkluderer, at begge termer er konstante, så $E[T] = O(n)$. □

Opgave 5.20. Implementer en effektiv sorteringsalgoritme for indgange med nøgler i området $\{0, \ldots, K-1\}$, som til ind- og udgift benytter datastrukturen fra opgave 3.19. For n indgange og blokstørrelse B bør pladsbehovet være $n + O(n/B + KB)$.

5.7 Sortering i det ydre lager

Udeladt

5.8 Implementationsaspekter

Udeladt

6

Prioritetskøer

Virksomheden KnM *sælger skræddersyet tøj af højeste kvalitet. Virksomheden er ansvarlig for markedsføring, opmåling, forsendelse, osv., men selve fremstillingen er overladt til uafhængige skrædderier. Virksomheden beholder 20% af overskuddet. Da den blev grundlagt i det nittende århundrede, fandtes der fem kontraktskrædderier. I dag behersker den 15% af verdensmarkedet, på verdensplan arbejder tusindvis af skrædderier for* KnM*.*

Din opgave er at tildele ordrer til kontraktskrædderierne. Reglen er, at næste ordre går til det skrædderi, hvis samlede ordreværdi i indeværende år er mindst. Grundlæggerne af KnM *benyttede en kridttavle for at holde rede på de løbende ordresummer – i datalogisk terminologi havde de en lineær liste af værdier og brugte lineær tid til at finde det rigtig skrædderi. Imidlertid er virksomheden blevet for stor til denne løsning. Kan du give en »skalerbar« løsning, som kun kræver, at man kigger på et lille antal indgange for at finde ud af, hvem der skal få næste ordre?*

Næste år ændres reglen. For at gøre det attraktivt at levere til tiden, modtager et skrædderi nu den næste ordre, hvis dens udestående *bestillinger har den mindste totale værdi. Det betyder, at så snart en ny bestilling leveres, skal dens værdi trækkes fra det pågældende skrædderis udestående ordreværdi. Er din strategi for fordeling af bestillinger tilstrækkelig fleksibel til også at tage hensyn til den nye regel uden tab af effektivitet?*

Datatypen, som løser problemet foroven, kaldes en *prioritetskø*. Vi begynder med at specificere datatypens funktionalitet. Prioritetskøer forvalter en mængde Q af *indgange* med *nøgler* under følgende operationer:

$Q.\mathit{opbyg}(\{e_1,\ldots,e_n\})$	$Q{:=}\{e_1,\ldots,e_n\}$
$Q.\mathit{tilføj}(e)$	$Q{:=}Q\cup\{e\}$
$Q.\mathit{mindste}$	**returner** min Q (en indgang med mindste nøgle)
$Q.\mathit{fjernMindste}$	$e{:=}\min Q$; $Q{:=}Q\setminus\{e\}$; **returner** e

Disse operationer er tilstrækkelige til den første del af vores eksempel. Hvert år bygger vi en ny prioritetskø, hvori hvert skrædderi er indang med en nøgle, som sættes til 0 i begyndelsen. For at uddele en ny ordre, udtager vi en indgang e med minimal nøgle, adderer ordrens værdi til denne nøgle,

[1] Billedet viser en kø foran Mao-mausoleet (V. Berger, se `http://commons.wikimedia.org/wiki/Image:Zhengyangmen01.jpg`).

og stiller den modificere indgang tilbage i køen. I afsnit 6.1 præsenterer vi en enkel og effektiv implementation af denne grundfunktionalitet.

[TODO udeladt afsnit om adresserbare prioritetskøer]

[TODO udeladt afsnit om anvendelsesområder]

Opgave 6.1 (Naiv prioritetskø)**.** Vis, hvordan man implementerer en (størrelsesbegrænset) prioritetskø som række. Hvis køens maksimale størrelse er w, bruges en række af størrelse w; de faktisk eksisterende indgange i prioritetskøen indtager rækkens første n pladser. Sammenlign kørselstiderne for operationerne af to varianter af denne implementation: den ene holder rækkens elementer i sorteret orden, den anden holder elementerne usorteret.

Opgave 6.2 (Addresserbar prioritetskø som dobbelthægtet liste)**.** Vis, hvordan man implementerer en adresserbar prioritetskø som dobbelthægtet liste. Hver listeknude repræsenterer en indgang i prioritetskøen. Sammenlign kørselstiderne for operationerne af to varianter af denne implementation: den ene holder listens elementer i sorteret orden, den anden holder elementerne usorteret.

6.1 Binærhobe

En hob (eller binærhob) er en enkel og effektiv implementation af en (ikke-adresserbar) prioritetskø [60]. På samme måde som en række kan den udvides til en ubegrænset udgave som beskrevet i afsnit **??**. Man kan sågar bruge hobe til at implementere adresserbare prioritetskøer, men vi skal lære endnu bedre muligheder at kende senere.

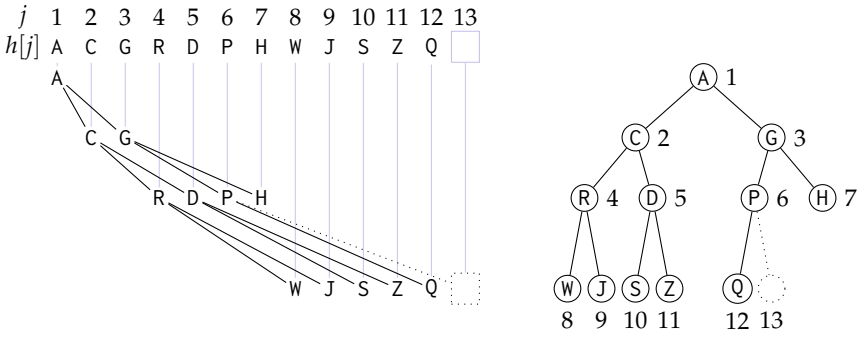
For at gemme prioritetskøens n indgange benytter vi de første n positioner i rækken $h[1..w]$. Vi siger at en række en *hobordnet* eller bare »udgør en hob«, dersom den opfylder *hobreglen*

$$h[\lfloor j/2 \rfloor] \leq h[j] \quad \text{for alle } j \in \{2, \ldots, n\}\,.$$

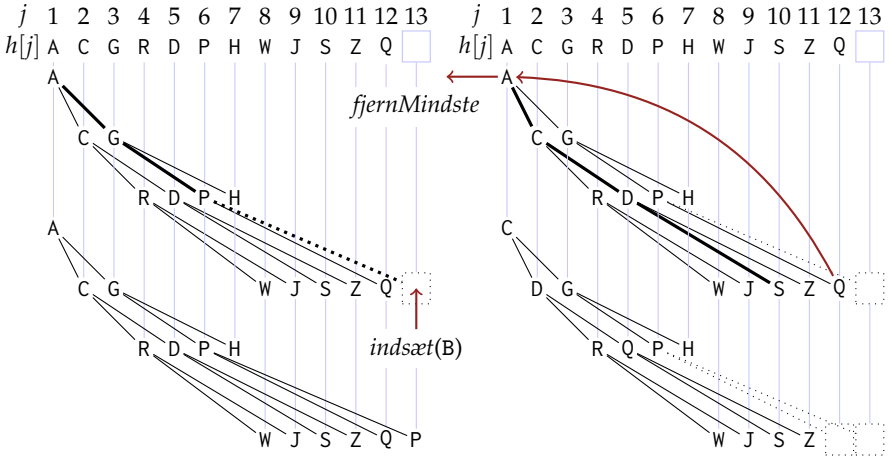
Hvad skal hobreglen betyde? For at forstå denne definition betragter vi en bijektion mellen tallene $\{1, \ldots, n\}$ og knuderne i et *venstrefuldstændigt* binærtræ med n knuder, som er vist i fig. 6.1. Hobreglen medfører, at nøglerne langs hver vej fra en knude til roden er monotont svagt aftagende; især gælder der, at indgangen ved roden, dvs. rækkeposition 1, indeholder en mindste nøgle. Operationen *mindste*, som skal finde sådan en indgang, behøver altså kun tid $O(1)$. Konstruktionen af en tom hob med plads til n indgange behøver ligeledes konstant tid, fordi der blot skal stilles en række af størrelse w til rådighed. [2] Fig. 6.3 viser pseudokoden for disse enkle operationer.

At celle $h[1]$ indeholder et mindste element svarer til situationen i en ordnet række, som er indiceret fra 1. Hobreglen er dog et meget svagere krav

[2] O.a.: Vi går her ud fra vores antagelser om maskinmodellen, ifølge hvilken en ikke-initialiseret række kan skabes i konstant tid. Ellers tager det tid $O(w)$.



Figur 6.1. En hob med $n = 12$ indgange, som er gemt i en række med $w = 13$ pladser. Figuren viser hoben både som række og som det implicitte binærtræ, her tegnet på to måder. Knuderne svarer til indeksmængden $\{1, \ldots, n\}$, hvor 1 står for roden. Rodens børn har indeks 2 og 3. Generelt har børnene til knude j indeks $2j$ og $2j + 1$ (forudsat at disse tal ligger i $\{1, \ldots, n\}$). Forgængeren til knude j for $j \in \{2, \ldots, h\}$ har indeks $\lfloor j/2 \rfloor$. Der gælder invariant i dette implicitte træ, at nøglen i en forgængerknude aldrig er større end nøglerne i nogen af dens børneknuder, dvs. at hele træet er hobordnet.



Figur 6.2. Operationer på hoben fra fig. 6.1. *Til venstre* vises effekten af at tilføje den nye indgang B. Vi skriver B i en ny bladknude med index 13. De tykke kanter markerer vejen fra den nye bladknude til roden. Indgang B vandrer ad denne vej op til den første knude, hvis forgænger ikke er større end B. For at skabe plads til B, bliver de andre indgange på vejen forskudt et niveau længere ned. *Til højre* vises effekten af at fjerne rodindgangen. (Denne indgang indeholder altid en mindste nøgle.) De tykke kanter markerer vejen p ned fra roden, som i hver knude følger den mindre nøgle. Indgangen Q, fundet i bladet med størst indeks, flyttes først til roden, for derefter at vandre nedad langs p, til den finder en knude, hvis efterfølger (på p) ikke er mindre end Q eller ikke eksisterer. De andre indgange på vejen forskydes et niveau opad for at gøre plads til Q.

Klasse *BinærHobPK*(w: **N**) **af** Element
  h: *Række*[1..w] **af** *Element* //*Hoben h er*
  $n = 0$: **N** //*tom i begyndelsen.*
  invariant $\forall j \in 2..n$: $h[\lfloor j/2 \rfloor] \leq h[j]$ //*hobreglen*
  Funktion *mindste* **antag** $n > 0$; **returner** $h[1]$ //$\Rightarrow$ *roden indeholder den mindste nøgle*

Figur 6.3. En klasse for prioritetskøer på basis af binærhobe med fast størrelsesbegrænsning w.

på rækkestrukturen end at være sorteret. For eksempel findes der kun én sorteret udgave af mængden $\{1,2,3\}$, men både $\langle 1,2,3\rangle$ og $\langle 1,3,2\rangle$ er lovlige hob-repræsentationer af denne mængde.

Opgave 6.3. Angiv samtlige hobrepræsentationer af mængden $\{1,2,3,4\}$.

Vi skal nu se, at denne større fleksibilitet tillader meget effektive implementationer af *tilføj* og *fjernMindste*. Vi skal beskrive disse procedurer, så de er lette at forstå og deres korrekthed er let at vise. I afsnittet **??** skal vi skitsere, hvordan en effektiv implementation kunne se ud. Instættelsesoperationen *tilføj*(e) placerer en ny indgang e i første omgang sidst i hoben, dvs. den øger n med 1 og sætter derefter e på position $h[n]$. Herefter flyttes e til en passende position på vejen fra bladet $h[n]$ til roden:

Procedure *tilføj*(e: *Element*)
  antag $n < w$
  n++; $h[n]$:=e
  sigtOp(n)

Herved skubber operationen *sigtOp*(j) indgangen i knude j i retning af roden, indtil roden er nået eller indgangen møder en forgænger, hvis nøgle ikke er større (se figur 6.2). Vi skal vise, at h dermed igen er en hob. Lad os sige at »kun $h[j]$ er måske for lille«, når h enten selv er en hob eller kan gøres til en hob ved at erstatte nøglen x i $h[j]$ med en passende nøgle x' med $x' > x$.

Procedure *sigtOp*(j: **N**)
  antag kun $h[j]$ er måske for lille
  hvis $j = 1 \vee h[\lfloor j/2 \rfloor] \leq h[j]$ **så returner**
  ombyt($h[j], h[\lfloor j/2 \rfloor]$)
  antag kun $h[\lfloor j/2 \rfloor]$ er måske for lille
  sigtOp($\lfloor j/2 \rfloor$)
  antag h er en hob

Opgave 6.4. Vis: (a) Antagelserne i algoritmen etablerer dens korrekthed. Hertil skal man vise, at hvis kun $h[j]$ måske er for lille, så er enten h en hob eller ombytningen *ombyt*($h[j], h[\lfloor j/2 \rfloor]$) medfører, at kun $h[\lfloor j/2 \rfloor]$ måske er for lille. (b) *sigtOp*(n) har kørselstid $O(\log n)$. (Dette medfører samme grænse for *tilføj*.)

Et kald af *fjernMindste* returnerer indholdet i roden og erstatter det med inholdet i knude n. Fordi $h[n]$ måske er større end $h[2]$ eller $h[3]$, kan hobreglen nu være brudt. Den genoprettes ved et kald til proceduren $sigtNed(1)$.

```
Funktion fjernMindste: Element
  antag n > 0
  resultat = h[1] : Element
  h[1]:=h[n];   n--
  sigtNed(1)
  returner resultat
```

Proceduren $sigtNed(1)$ skubber den nye indgang i roden nedad, til hobreglen er retableret. For at være mere nøjagtig, betragt vejen p, som begynder ved roden og ved hver knude fortsætter ved det barn som har den mindre nøgle (s. fig. 6.2); i tilfælde af nøglelighed er beslutningen ligegyldig. Man følger denne vej, til man når en knude, hvis børn (hvis de findes) indeholder nøgler, som er mindst lige så store som $h[1]$s nøgle. Indgangen $h[1]$ sættes på denne plads; alle indgange på vejen p vandrer et niveau opad. På denne måde gælder hobreglen igen. Strategien formuleres enklest rekursivt. Vi benytter en lignende invariant som ved *sigtOp*, og siger at »kun $h[j]$ er måske for stor«, hvis enten h selv er en hob eller kan gøres til en hob ved at erstatte nøglen x i $h[j]$ med en passende nøgle x' med $x' < x$.

```
Procedure sigtNed(j: N)
  antag kun h[j] er muligvis for stor
  hvis 2j ≤ n så // j er ikke et blad
    hvis 2j + 1 > n ∨ h[2j] ≤ h[2j + 1] så m:=2j ellers m:=2j + 1
    antag nøglen i søskendeknuden til m,
          såvidt den findes, er ikke større end nøglen i m
    hvis h[j] > h[m] så //h[j] er for stor
      ombyt(h[j], h[m])
      antag kun h[m] er muligvis for stor
      sigtNed(m)
  antag h er en hob
```

Opgave 6.5. Vis: (a) Angtagelserne i algoritmen sikrer dens korrekthed. Hertil skal man vise, at hvis kun $h[j]$ måske er for stor, så er enten h en hob, eller operationen $ombyt(h[j], h[m])$ medfører, at kun $h[m]$ måske er for stor. (b) Prodeduren *sigtNed* behøver ikke mere end $2 \log n$ nøglesammenligninger. (c) Vis, at antallet af nøglesammenligninger kan reduceres til $\log n + O(\log \log n)$. *Vink*: Bestem først »vejen af mindre børn« p fra roden til et blad. Brug nu binærsøgning for at finde den rigtige plads til $h[1]$. (Se afsnit **??** for andre varianter af *sigtNed*.)

Vi betragter nu operationen *opbyg* for hobe. Det er åbenlyst, at man kan opbygge en hob af n givne indgange i tid $O(n \log n)$ ved at tilføje dem en efter en. Interessant nok kan dette gøres bedre ved at opbygge hoben nedefra og op.

Med proceduren *sigtNed* kan man etablere hobreglen i et undertræ af højde $k+1$, hvis den allerede gjaldt i begge undertræer af højde k. I den følgende opgave behandles denne ide i detaljer.

Opgave 6.6 (*Opbyg* for binærhob)**.** Betragt en vilkårlig række $h[1..n]$ af indgange. Disse skal permuteres således, at hobreglen gælder. Hertil bruges følgende to procedurer:

Procedure *opbygHobBaglæns*
  for $j{:=}\lfloor n/2 \rfloor$ **ned til** 1 **udfør** *sigtNed*(j)

Procedure *opbygHobRekursivt*$(j\colon \mathbf{N})$
  hvis $4j \leq n$ **så**
    opbygHobRekursivt$(2j)$
    opbygHobRekursivt$(2j+1)$
  sigtNed(j)

(a) Vis, at både *opbygHobBaglæns* og *opbygHobRekursivt*(1) etablerer hobreglen.

(b) Implementer begge algoritmer effektivt og sammenlign deres kørselstider for tilfældige tal som nøgler og $n \in \{\, 10^i \colon 2 \leq i \leq 8\}$. Meget afhænger af, hvor effektiv implementationen af *opbygHobRekursivt* bliver. For små undertræer kan det være meningsfuld at »udrede« rekursionen for små undertræer, dvs. at erstatte den med direkte opdeling i forskellige tilfælde.

(c) (*) For store n udgøres forskellen mellem de to algoritmer hovedsageligt af de effekter, som lagerhierarkiet medfører. Analysér begge algoritmer med henblik på antallet af nødvendige i/u-operationer i fjernlagermodellen, som beskrevet sidst i afsnit 2.2. Vis især, at *opbygHobRekursivt* kun behøver $O(n/B)$ i/u-operationer, når blokstørrelsen er b og det hurtige lager har størrelse $M = \Omega(B \log B)$ hat.

Følgende sætning sammenfatter vores resultater om hobe:

Sætning 6.1. *Betragt en ikke-adresserbar prioritetskø implementeret som hob. Konstruktionen af en tom hob og funktionen* mindste *tager konstant tid, operationerne* fjernMindste *og* tilføj *kan udføres i logaritmisk tid* $O(\log n)$*, og* opbyg *tager lineær tid.*

Bevis. Det implicitte binærtræ, som svarer til en hob med n indgange, har dybde $k = \lfloor \log n \rfloor$. Operationerne *tilføj* og *fjernMindste* gennemløber en del af vejen fra roden til et blad og har derfor logaritmisk kørselstid. Operationenen *mindste* rapporterer indholdet af roden, hvilket tager konstant tid. Konstruktionen af en tom hob kræver blot allokering af en række, hvilket tager konstant tid. Uanset hvilken af de to procedurer for *opbyg* man bruger: Operationen *sigtNed* kaldes højst en gang for de hver af de højst 2^l knuder på dybde l. Et sådant kald tager tid $O(k-l)$. Den totale kørselstid for *opbyg* er derfor asymptotisk begrænset af

$$\sum_{l=0}^{k-1} 2^l(k-l) = 2^k \sum_{l=0}^{k-1} \frac{k-l}{2^{k-l}} \leq 2^k \sum_{j\geq 1} \frac{j}{2^j} = O(n)\,.$$

Den sidste ulighed benytter formel (A.14). □

Hobe danner grundlaget for sorteringsmetoden *hobsortering*. Ud fra n indgange, som skal sorteres, dannes i første omgang en hob h med proceduren *opbyg*. Derefter følger $(n-1)$ kald af *fjernMindste*. Herved forkortes hoben med en indgang ad gangen. Inden det $i$te kald af *fjernMindste* står den $i$temindste indgang i roden $h[1]$. Vi ombytter $h[1]$ og $h[n-i+1]$ og lader den nye rodindgang vandre nedad til sin rette plads i den resterende hob $h[1..n-i]$ med *sigtNed*. Til sidst står indgangene i rækken i faldene orden. Selvfølgeligt kan man også sortere i stigende orden ved at bruge en »størsteprioritetskø«, dvs. en datastruktur, som tillader tilføjelse af en indgang samt fjernelse af den *største* indgang. Hertil skal man blot erstatte »$\leq$« med »$\geq$« hhv. »$<$« med »$>$« i samtlige procedurer.

Binærhobe implementerer ikke umiddelbart datatypen »adresserbar prioritetskø«, fordi indgange flyttes omkring i rækken ved tilføjelse og fjernelse, og man derfor ikke kan benytte indgangens position i rækken som greb.

Opgave 6.7 (Adresserbar binærhob)**.** Udvid hoben så den implementer en adresserbar prioritetskø (men uden operationen (*flet*). Hvor mange yderligere peger per indgang er nødvendige? Der findes en løsning, som klarer sig med to ekstra pegere per indgang.

Opgave 6.8 (Massetilføjelse)**.** Konstruer en algoritme, som tillader tilføjelsen af k nye indgange til en binærhob med n indgange, og som har kørselstid $O(k + \log n)$. *Vink*: Gennemgå hoben nedefra og op, som ved hobopbygning.

[TODO resten af kapitlet udeladt]

7

Sorterede følger

Vi tilbringer alle en betydelig del af vores tid med at lede efter ting. Sådan er det også med datamater: De slår op efter telefonnumre, saldoer, flyreservationer, regninger og betalinger, osv. I mange anvendelser skal der søges i dynamiske, dvs. foranderlige, datamængder. Nye reservationer føjer til reservationssystemet, reservationer ændres eller slettes, og reservationer bliver til faktisk gennemførte flyafgange. Vi har allerede mødt én løsning på dette problem, nemlig hachning. hadsning. hasning. hashning, hashing. Men ofte ønsker vi at holde den foranderlige datamængde i sorteret orden. For at sikre denne tilstand manuelt benytter man et kartotek. Nye kartotekskort kan indsættes, fjernes, kortene kan gennemses i sorteret rækkefølge og man kan bruge en slags binærsøgning for at finde et bestemt kort. I gamle date bestod kartotekssystemet af store biblioteker af hundretusindvis af kort.[1]

I dette kapitel drejer det sig om at forvalte en mængde S af indgange. Hver indgang e har en nøgle $nøgle(e)$ taget fra en mængde *Nøgle*, som er forsynet med en lineær ordning. Dette definerer en rækkefølge på S. Ved siden af søgning vil vi også støtte operationer til tilføjelse og fjernelse. Dette fører til følgende grundoperationer på en *sorteret følge* S:

$S.lokaliser(k : Nøgle)$: Returner indgangen med den mindste nøgle i $\{ e \in S : nøgle(e) \geq k \}$.

$S.tilføj(e : Element)$: Hvis der findes $e' \in S$ med $nøgle(e') = nøgle(e)$, så $S{:=}(S \setminus \{e'\}) \cup \{e\}$. Ellers $S{:=}S \cup \{e\}$.

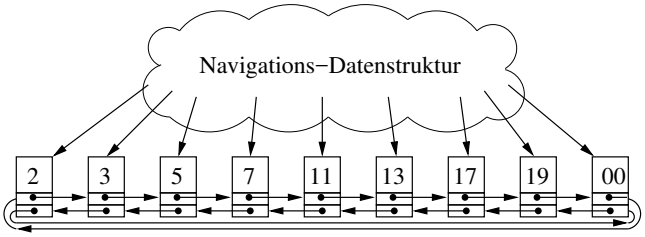
$S.fjern(k : Nøgle)$: Hvis der findes $e \in S$ med $nøgle(e) = k$, så $S{:=}S \setminus \{e\}$.

Operationen $lokaliser(k)$ *lokaliserer* nøglen k in S, dvs. at den finder en indgang med nøgle k, hvis den findes i S, og ellers den indgang, som har den minst mulige nøgle større end k. Operationen $tilføj(e)$ tjener både til at tilføje en ny indgang e med en ny nøgle, og til at overskive en eksisterene indgang med samme nøgle; denne form at tilføjelse sikrer, at nølgerne af alle indgange i S altid er parvist forskellige. (En alternativ konvention betragtes i opgave **??**

[1] Fotografiet viser kartotekskort i universitetsbiblioteket i Graz. (Dr. M. Gossler).

zurück.) Operationen *fjern*(k) fjerner indgangen med nøgle k, såfremt den findes. Vi skal vise, at man kan implementer sorterede følger på en sådan måde, at alle operationer kan gøres i tid $O(\log n)$, hvor n er længden af følgen.

Hvad er forholdet mellem sorterede følger og datastrukturerne i de foregående kapitler? Sorterede følger er mere fleskible end sorterede rækker, fordi de stillere mere effektive former af operationerne *tilføj* og *fjern* til rådighed. De er langsommere, men også kraftigere end hakketabeller, fordi *lokaliser*(k) også virker, når søgenøglen k slet ikke forekommer i S. Prioritetskøer er et specialtilfælde af sorterede følger; de kan kun finde eller fjerne indgangen med den mindste nøgle.



Figur 7.1. En sorteret følge som dobbelthægtet liste med en ekstra datastruktur til navigation.

Vores hovedsagelige realisering af sorteret følge består af en dobbelthægtet liste og en ekstra datastruktur, som tjener til navigering ved *lokaliser*-operationen. I fig. 7.1 vises denne idé skematisk. Man skal huske, at en doppelthægtet liste repræsenterer n indgange ved hjælp af $n + 1$ knuder, en for hver indgang samt en ekstra attrapknude. Attrapknuden indeholder den særlige nøgleværdi ∞, som er større end alle faktisk forekommende nøgler. Vi vedtager, at operationen *lokaliser*(k) skal da levere et greb på den første knude i listem, hvis nøgle er mindst lige så stor som k. Når k er større end samtlige knuder i S, skal *lokaliser* returnere et greb på attrapknuden. Vi har set i afsnit 3.1.1, at dobbelhægtede lister stiller mange forskellige operationer til rådighed. De fleste af disse operationer kan også implementeres effektivt for sorterede følger. For eksempel kan vi overtage konstanttidsoperationerne *første*, *sidste*, *efterfølge* og *forgænger* For operationerne *fjern*$(h\colon \mathit{Greb})$, *indsætFør* og *indsætEfter* skal vi se implementationer med konstant amortiseret omkostning; operationerne »sammenføje« og »spaltning« kan implementeres i logaritmisk tid. Indeksoperatoren $[\cdot]$ og bestemmelsen af positionen for en givet indgang i følgen kræver ligeledes logaritmisk tid. Før vi betragter navigationsdatastrukturen nærmere, vil vi dog kaste et blik på konkrete anvendelser for sorterede følger.

Bedste-først-heuristik. Antag, at vi skal pakke ting i spande. Tingene ankommer enkeltvis og skal umiddelbart ved ankomst pakkes i en af spandene. Hver ting

i har en vægt $v(i)$, og hver spand har en maksimal kapacitet. Målet er at bruge så få spande som muligt. (Dette er en trinvis variant af det såkaldte »*spandpakning*-problem«.) En populær heuristisk strategi for dette problem hedder »*bedste pasform*«; den består i at putte den aktuelle ting i ind i den spand, blandt spandende med restkapacitet mindst $v(i)$, som har mindst kapacitet [8]. For at implementere denne strategi, holder vi spandende i en følge S, sorteret efter restkapacitet. For at anbring en ny ting i, kalder vi $S.\mathit{lokaliser}(v(i))$, fjerner den fundne spand fra følgen, reducerer dens restkapacitet med $v(i)$ og tilføjer den atter til S. (Jf. opgave **??**.)

Strygelinjealgoritmer. Givet en mængde af vandrette og lodrette linjestykker i planen ønsker vi at finde alle deres skæringspunkter. En såkaldt »*strygelinje*-algoritme« for dette problem består konceptuelt af at lade en lodret linje stryge over planen fra venstre til højre. Strygelinjen medfører informationen om hvilke vandrette linjestykker, den skærer, i en sorteret følge S. Sorteringsnøglen er det vandrette linjestykkes y-koordinat. Når strygelinjen møder venstre endepunkt af et vandret linjestykke, tilføjes dette til S; når strygelinjen møder højre endepunkt, fjernes det fra S. Når linjen møder et lodret linjestykke s med x-koordinat x_s, som i lodret retning dækker $[y, y']$, kaldes $S.\mathit{lokaliser}(y)$ og S gennemløbes til højre fra den lokaliserede position, indtil nøglen y' er overskredet. [2] De lodrette linjestykker, der bliver mødt under gennemløbningen, er netop dem, der danner et skæringspunkt med s. Strygelinjealgoritmen kan generaliseres til linjestykker med vilkårlig hældning [5], bøjede linjer og mange andre geometriske problemer [12].

Databaseindeks. Et grundlæggende databaseproblem er at organisere store mængder af poster, så man kan gennemsøges effektivt. Såkaldet B-træer, en variant af datastrukturen »(a, b)-træ«, som vi skal undersøge i afsnit 7.2, er en af de vigtigste konstruktioner til implementation af store databaser.

Den mest brugte datastruktur til navigation er *søgetræer*. Her gemmes indgangene i træets blade. [3] Ofte bruger man navnet for navigationsdatastrukturen for at betegne den samlede datastruktur for den sorterede følge. Vi introducerer søgetræsalgoritmer i tre skridt. Til opvarmning ser vi på ubalancerede binære søgetræer i afsnit 7.1, hvilke under visse gunstige forhold kan gennemføre operationen *lokaliser* i tid $O(\log n)$. Søgetræer er træer, som er rettet væk fra roden, og hvis indre knuder hver har to børn. Fordi det ikke her helt nemt at holde binære søgetræer under indsættelser og fjernelser inden for området for »gunstige forhold«, går vi videre til en mere generel struktur, (a, b)-træerne, som tillader knuder med større udgrad end 2. I afsnit 7.2 skal vi se, hvordan man kan implementere (a, b)-træer ved hjælp af tre grundlæggende operationer på en sådan måde, at de i værste fald kræver logaritmisk tid. I afsnit **??** og **??** skal vi udvise søgetræer med mekanismer,

[2] Denne slags »områdeopslag« nævnes i afsnit **??**.

[3] Der findes også en variant af søgetræer, hvor de indre knuder også indeholder indgange.

som tillader os at understøtte endnu flere operationer effektivt. Afsnit **??** ser nærmere på de (amortiserede) omkostninger af tilføjelse og fjernelse.

7.1 Binære søgetræer

Navigation i det binært søgetræ foregår omtrent som når man spørger om vej i en fremmed by: Man stiller et spørgsmål, følger det svar, man har fået, stiller et nyt spørgsmål, følger det nye svar, osv. indtil man har nået sit mål.

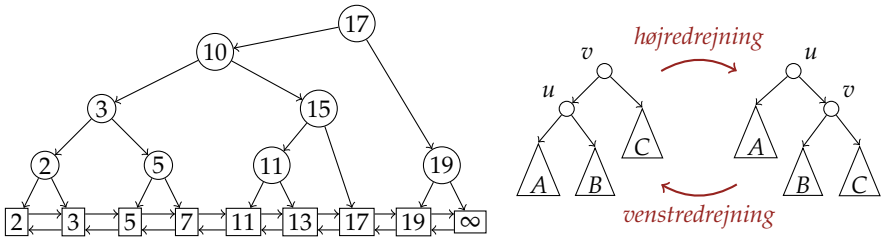
Et *binært søgetræ* er et binært træ, hvis blade indeholder indangene i en sorteret følge, inklusive attrapindgangen, fra venstre til højre i stigende rækkefølge. FOr at *lokalisere* en nøgle k, vil vi begynde ved roden og finde den entydigt bestemte vej til bladet med en mindste nøgle, som er mindst lige så stor som k. Hvordan finder man den rigtige vej? Hertil indeholder de indre knuder i træet ligeledes nøgler (ikke indgange!), som styrer søgningen; disse nøgler kaldes *spaltenøgler*. I et binært søgetræ med mindst to blade besidder hver indre knude præcis to børn, det *venstre* og det *højre* barn. For en spaltenøgle s i en indre knude v gælder nu følgende: hver nøgle k i vs venstre undertræ opfylder $k \leq e$; hver nøgle k i vs højre undertræ opfylder $k > s$.

Lad nu T være et træ, der opfylder denne definition. Betragt nøglen k. Det er ikke vanskeligt skridt for skridt at finde den rigtige vej gennem T til den efterlyste indgang, dvs. til indgangen med den mindste nøgle k', som opfylder $k' \leq k$. Begynd ved roden, og gentag følgende skridt indtil du når et blad: Hvis der gælder $k \leq s$ for spaltenøglen s i den aktuelle knude, gå til højre barn, ellers gå til venstre barn. Hvis T ingeholder en indgang med nøgle k, er det nemt at se, at vi havner i den indangs bladknude. Vi skal dog også undersøge, hvad der sker, når k ikke forekommer. Ved tilfældeanalyse kan man overbevise sig om, at følgende invariant gælder: Enten indeholder den aktuelle knudes undertræ indgangen k' eller dens højreste blad indeholder den umiddelbare forgænger til k' i den sorterede følge. Processen ender med et blad med nøglen k''. Denne sammenlignes med k. På grund af invarianten findes der bare to muligheder: Hvis $k \leq k''$, så er k''-bladet selv den søgte knude; hvis $k > k''$, så listeefterfølgeren til k''-bladet den søgte knude.

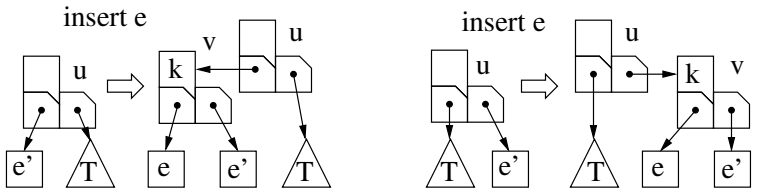
I figur 7.2 vises et eksempel på et binært søgetræ. Læseren bør teste søgemetoden på forskellige nøgler som fx 1, 9, 13 og 25. Husk at dybden af et træ er længden af den længste vej fra roden til noget blad. Dybden er altså det maksimale antal sammenligninger, der skal bruges for at finde vej til det blad, der svarer til søgenøglen k.

Opgave 7.1. Vis, at der for hvert mængde af $n \geq 1$ indgange findes et binært søgetræ med $n+1$ blade og med dybde $\lceil \log(n+1) \rceil$.

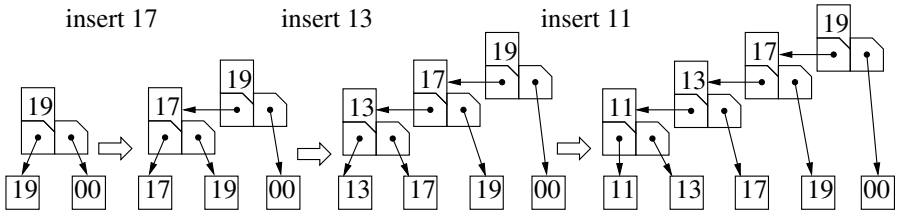
Et søgetræ med $n+1$ blade og med dybde $\lceil \log(n+1) \rceil$ kaldes *perfekt balanceret*. Det heraf resulterende logaritmiske søgetid udgør en dramatisk forbedring i forhold til den lineære søgetid $\Omega(n)$, som behøves for gennemløbning af en liste. Den dårlige nyhed er, at det er ganske dyrt at opretholde



Figur 7.2. *Venstre*: Den sorterede følge $\langle 2, 3, 5, 7, 11, 13, 17, 19\rangle$, repræsenteret som et binært søgetræ. *Højre*: Drejning i et binært søgetræ. Trekanterne repræsenterer undertræer. Læg mærke til, at knuderne u og v bytter forælder/barn-roller.



Figur 7.3. Naiv indsættelse af indgang e i et binært søgetræ. En trekant repræsenterer et helt undertræ.



Figur 7.4. Naiv indsættelse af indgange i sorteret rækkefølge leder til, at træet degenererer til en liste.

den perfekt balance under indsættelse og fjernelse af indgange. For at forstå problemet bedre, betragter vi den »naive« indsættelsesprocedure, som er vist i figur 7.3. Vi lokaliserer den nye indgang es nøgle k i en liste, dvs. at vi finden den knude e', som indeholder den på k følgende nøgle. Så tilføjes e til listen, og en ny knude v skabes med venstre barn e, højre barn e' og spaltenøgle k. Den tidligere forgænger til e' peger nu på v. I værste fald vælger hver indsættelse et blad på størst mulig dybde, sådan at træets dybde vokser hver gang. Figur 7.4 viser et eksempel på denne effekt: : Træet kan degeneree til en liste, og vi er tilbage i en situation, som kræver lineær søgning.

En enkel løsning til dette problem er en sund portion optimisme: Det værste fald behøver jo ikke at indtræffe. Det viser sig faktisk, at indsættelsen

af n indgange i *tilfældig* orden fører til et træ, hvis gennemsnitlige dybde er omtrent $\approx 2.99 \log n$ [13]. . Vi skal her udelade beviset for denne påstand, men i stedet kort skitsere sammenhængen med kviksortering for i det mindste at sandsynliggøre resultatet. Hvordan kan man havne i et træ som i figur. 7.2 ved naiv indsættelse? Man begynder med at indsætte 17, dette opdeler mængden af indgange i delmængderne $\{2, 3, 5, 7, 11, 13\}$ og $\{19\}$. Fra de resterende indgange i den første delmængde indsætter vi nu 7; hvilket opdeler den i delmængderne $\{2, 3, 5\}$ og $\{11, 13\}$. I terminologien fra kviksortering skulle vi sige, at 17 blev valgt som spalteførste rekursive kald, og 7 som næste kald for den venstre delmængde. Opbygningen af et binært søgetræ og udførslen af kviksortering viser sig altså at være helt analoge processer, som udfører de samme sammenligninger, selvom det sker på forskellige tidspunkter. Hver indgang sammenlignes med rodnøglen 17. Ved kviksortering finder disse sammenligninger sted, når mængen opdeles i første kald i forhold til spaltenøglen 17. Ved opbygningen af det binære søgetræ finder disse sammenligninger sted i forbindelse med indsættelsen af de enkelte indgange. For eksempel sker sammenligningen mellem 17 og 11 under kviksortering i det første rekursive kald, mens det i søgetræet sker, når 11 bliver sat ind. Vi har set i sætning 5.5, at kviksortering af n indgange bruger forventet $O(n \log n)$ mange sammenligninger. Ifølge den lige skitserede analogi er det forventede antal sammenligninger ved obbygningen af et binært søgetræ, hvor indgangene sættes ind i tilfældig rækkefølge, ligeledes $(n \log n)$. Derfor koster hver tilføjelse forventet $O(\log n)$ mange sammenligninger. Man kan vise, at hver indsættelse behøver $O(\log n)$ mange sammenligninger med høj sandsynlighed, og at den gennemsnitlige dybde af træet er omtrent $2.99 \log n$.

Hvordan kan vi holde dybden logaritmisk i værste fald? Der findes forskellige metoder, som sammenfattes i afsnit **??**, og hvoraf 2 betragtes nærmere i afsnit 7.2. Først skal vi betragte in konstruktion, som tillader træknuder at have forskellig udgrad; derefter skal vi se, hvordan man kan holde binærtræer balancerede ved hjælp af rotationer.

Opgave 7.2. Figur 7.2 viser, hvordan man kan ændre udseendet at et binærtræ med såkaldte rotationer. Anvend gentagne rotationer på undetræerne i træet i figur 7.2 for at gøre knuden med nøgle 11 til rodknude.

Opgave 7.3. Forklar, hvordan man kan implementere et *implicit* binært søgetræ. Her er træet venstrefuldstændigt og gemt som en række på samme måde som en binærhob (jf. afsnit **??**). Hvad er fordele og ulemper sammenlignet med en pegerbaseret implementation? Sammenlign en søgeproces i det implicitte binærtræ med binærsøgning i en sorteret række.

7.2 Ab-træer og rød-sorte træer

8

Repræsentation af grafer

Videnskabelige resultater er først og fremmest tilgængelige som artikler i tidsskrifter og konferencerapporter, men også i forskellige kilder på WWW. [1] *Disse artikler er ikke selv-indeholdte, men bygger videre på og citerer tidligere publiceret arbejde med beslægtet indhold. Hvis man i dag læser en artikel fra 1975, som indeholder et interessant delresultat, spørger man sig umiddelbart, hvilke fremskridt der er gjort siden dengang. Især er man interesseret i nyere artikler, der citerer 1975-resultatet. Projekter som »Google Scholar« stiller denne funktionalitet til rådighed ved at analysere literaturhenvisningerne i en artikel og bygge en database, der gør det muligt at finde de artikler, som citerer en given artikel. .*

Situationen foroven kan enkelt modelleres som en rettet graf. Grafen indeholder en knude for hver artikel og en kant for hver citation. Kanten (u, v) fra artikel u til artikel v betyder, at v citeres af u. I denne repræsentation gemmer hver knude (artikel) u alle udgående kanter (til de i u citerede artikler), men ikke de indgående kanter (fra de artikler, som citerer u). Hvis vi også skulle gemme de indgående kanter sammen med hver knude, ville det være let at finde de citerende artikler. En af de centrale opgaver ved realiseringen af Google Scholar er netop konstruktionen af de modsatrettede kanter. Dette eksempel illustrerer, at omkostningen ved en elementær grafoperation, nemlig at finde de indgående kanter, er stærkt afhængig af grafens repræsentation. Hvis de indgående kanter er gemte eksplicit, er operationen enkel, ellers er den ikke-triviel.

I dette kapitel gennemgår vi forskellige muligheder for digital repræsentation af grafer. Vi vil fokusere på rettede grafer og går ud fra, at en urettet graf $G = (V, E)$ er repræsenteret som det tilsvarende dobbeltrettede graf $G' = (V, \bigcup_{\{u,v\} \in E} \{(u, v), (v, u)\})$ (jf. afsnit 2.9). I figur 8.1 vises foroven en urettet graf den dens dobbeltrettede modpart. de fleste datastrukturer, som behandles i dette kapitel, tillader også repræsentationen af (parallelle) multikanter og løkker (v, v). Vi vil begynde med at skabe os et overblik over de operationer, som muligvis skal kunne understøttes.

- *Adgang til informationskomponenter.* Ofte vil man have adgang til information, som er knyttet til et given knude eller kant, fx vægten af den kant eller afstanden til en særlig knude. I mange grafrepræsentationer er knuder og

[1] Billedet forover viser et edderkoppespind.

kanter objekter, og vi kan derfor direkte gemme informationen i objektets instansvariable. Medmindre andet er sagt, sætter vi $V = \{1, \ldots, n\}$, så vi kan gemme den til en knude knyttede information i række. Hvis der ikke er en anden mulighed, kan man gemme knude- og kantinformation i en hakketabel. I hvert fald kan adgangen til informationskomponenter implementeres på en sådan måde, at de tager konstant tid. I resten af bogen vil vi abstrahere bort fra de forskellige muligheder for denne adgang og bruger datatyperne *knuderække* og *kantrække* for at vise, at det drejer sig om en rækkelignende datastruktur, hvis indeksområde er mængden af kanter hhv. knuder.

- *Navigation.* For en givet knude vil man ofte have adgang til de udgående kanter. Det viser sig, at denne operation er af central betydning for de fleste grafalgoritmer. Som vi har set i det indledende eksempel, er det sommetide også nødvendigt at have adgang til de indgående kanter til en knude.
- *Kantforespørgsler.* For et givet par af knuder (u, v) vil vi måske vide, om parret udgør en kant i grafen. Dette kan man altid implmentere som en hakketabel, men sommetider ønsker vi en endnu hurtigere proces. En mere speciel, men vigtig forespørgsel er givet kanten (u, v) i G at finde (repræsentationen af) den *modsatrettede kant* $(v, u) \in E$, såfremt den findes. Denne operation kan implementeres ved at lade en peger forbinde hver kant med sin modsatrettede kant.
- *Konstruktion, omvandling og udgift.* Sommetider er en graf givet i en anden repræsentation end den, der er bedst egnet til løsningen af det aktuelle algoritmiske problem. Det er ikke noget større problem, fordi de fleste grafrepræsentationer kan omvandles til hinanden i lineær tid.
- *Aktualisiering.* Sommetider vil man tilføje eller fjerne knuder og kanter i en graf. Fx er beskrivelsen af nogle algoritmer enklere, hvis man tilføjer en knude, som kan nå alle andre knuder (se fx figur **??**).

8.1 Uordnede kantfølger

Den måske enkleste fremtilling af en gra fer som uordnet følge af kanter. Hver kant indeholder et par af knudeindeks og muligvis yderligere information som fx kantvægten. Om disse knudepar skal repræsentere en rettet eller urettet graf er et fortolkningsspørgsmål. Repræsentation som kantliste bliver ofte brugt til input og output. I denne repræsentation er det let at tilføje knuder og kanter i konstant tid. Mange andre operationer, især navigation i grafen, er derimod så langsomme, at repræsentationen stort set er ubrugelig. Derfor arbejder kun få grafalgoritmer godt med kantlister; de fleste har nemlig brug for hurtig adgang til en knudes nabokanter. I disse tilfælde foretrækker man de ordende repræsentationer, som diskuteres i de følgende afsnit. I kapitel 11 skal vi møde to algoritmer for minimalt spændetræ; den ene virker

fint selv med kantlisterepræsentationen, den anden har brug for en mere raffineret datastruktur.

8.2 Naborækker – statiske grafer

For at tillade hurtigere adgang til de udgående kanter for given knude kan man for hver knude gemme dens udgånde kanter i en række. Når der ikke er knyttet yderlige information til kanterne, indeholder denne række blot målknudernes indeks. Når grafen er *statisk*, dvs. ikke forandres i løbet af tiden, kan man føje de små rækker sammen til én stor kantrække E. En yderligere række V med $n+1$ indgange gemmer begyndelsespositionerne for de små rækker, dvs. for hver knude v indeholder $V[v]$ det første index i E, som indeholder en kant med startknude v. I $V[n+1]$ står attrapværdien $m+1$. For hver knude v finder man de udgående kanter fra v nemt som $E[V[v]], \ldots, E[V[v+1]-1]$; attrapværdien sikrer, at denne regel også gælder for $v = n$. Et eksempel på sådan en naborække finder man i figur 8.1 (i midten til venstre).

Lagerbehovet for en graf med n knuder og m kanter er $n + m + \Theta(1)$ maskinord. Det er endnu mere kompakt end de $2m$ maskinord, som bruges i kantlisterepræsentationen.

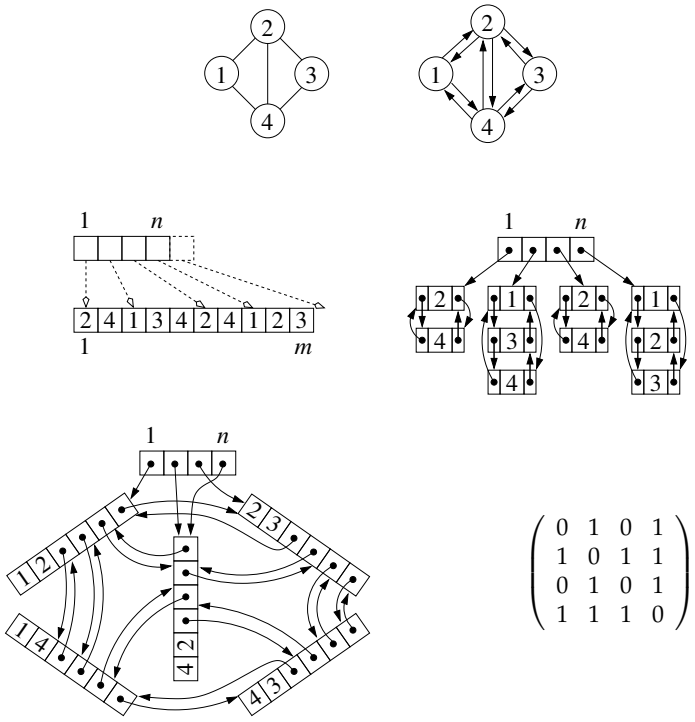
Repræsentationen som naborække kan udvides til at gemme yderligere information: Kantinformation kan man gemme enten i en yderligere række med m indgange eller i selve kantrækken. Hvis de indgående kanter også skal repræsenteres eksplicit, kan man gemme den omvendte graf i yderligere rækker V' og E'.

Opgave 8.1. Beskriv en algoritme som konverterer kantlisterepresentationen til naborækker i lineær tid. Brug kun $O(1)$ ekstra plads. *Vink*: Tænk på problemet som opgaven at sortere kanter efter startknude og ændr algoritmen i figur **??** til heltalssortering, så den kan bruges her.

8.3 Naborækker – dynamiske grafer

Kantrækker repræsenterer grafer på en kompakt og effektiv måde. Deres hovedsaglige ulempe er, at tilføjelse og fjernelse af kanter er dyr. Antag fx, at vi skal tilføje en ny kant (u, v). Selv hvis der stadig er plads i kantrækken E, skal en kant for hver af knuderne $u+1$ til n flyttes til højre, hvilket kræver tid $O(n)$.

I kapitel 3 har vi set, hvordan man implementerer dynamiske følger. Alle disse løsninger kan bruges til at konstruere en datastruktur for dynamiske grafer. For hver knude v fremtiller vi følgen E_v af dens udgående kanter (eller dens indgående kanter, eller både og) som ubegrænset række eller (dobbelt eller enkelt) hægtet liste. Fordele og ulemper ved de forskellige følgerepræsentationer overføres selvfølgeligt. Ubegrænsede rækker bruger lagerhierarkiet



Figur 8.1. *Foroven*: En urettet graf og dens tilsvarende dobbeltrettede graf. *Midte*: Repræsentation af samme graf som naborække og som nabolister. *Forneden*: Repræsentation som sammenhægtede kantobjekter og som nabomatrix.

bedre. Hægtede lister muliggør tilføjelse og fjernelse på vilkårlig plads i konstant tid. De fleste grafer er »tynde« i den forstand, at hver knude i gennemsnit kun har få ind- og udgående kanter. Hægtede lister til repræsentation af tynde grafer bør implementeres uden attrapknuden fra afsnit 3.1, fordi de nødvendige ekstra n knuder udgør et betydligt pladsspilde. I eksemplet i figur 8.1 (midten) bruges derfor cirkulært dobbelthægtede lister.

Opgave 8.2. Antag, at hver knude us udgående kanter er gemt i en ubegrænset række E_u. En kant $e = (u, v)$ er givet ved sin position i E_u. Forklar, hvordan e kan fjernes i amortiseret konstant tid. *Vink*: De andre kanter relative orden behøver ikke at blive bevaret.

Opgave 8.3. Betragt igen algoritmen fra afsnit 2.9, som afgør, om en givet rettet graf er kredsfri. Forklar, hvordan den kan implementeres til at arbejde i lineær tid. Det kræves en passende repræsentation af dynamiske grafer samt en algoritme, som kan bruge denne repræsentation effektivt. *Vink*: Brug en kø til at gemme knuder med udgrad 0.

Doppeltrettede grafer forekommer i mange sammenhænge. Urettede grafer kan repræsenteres som dobbeltrettede på en naturlig måde, og visse algortimer for rettede grafer har også brug for hurtig adgang til en knudes indgående kanter. I denne situation ønsker man ofte at gemme den information, som hører til en urettet kant (eller til en rettet kant og dens omvending), kun én gang. Ligeledes ønsker man nem adgang fra en kant til dens omvending.

Vi vil beskrive to løsninger, som opfylder disse krav. Den første forsyner enkelt hver rettede kant med to yderligere pegere. Den ene peger på den omvendte kant, den anden på den information, som er knyttet til kanten.

Den anden løsning nøjes med en enkelt listeknude for hver kant og henviser til denne knude i to nabolister. For urettede grafer skulle altså listeknuden for den urettede kant $\{u, v\}$ forekomme både i listen E_u og i listen E_v. Hver knude u gemmer en peger på en af sine hosliggende kanter og giver derigennem adgang til hele nabolisten. Hvis nabolisten skal være dobbelthægtet (selvom en cirkulært hægtet liste ofte er tilstrækkelig), behøver sådan en listeknude fire pegere: to pegere på us naboliste E_u og to pegere på vs naboliste E_v. Fra den pågældende listeknude kan man gennemløbe alle dens hosliggende kanter. Et eksempel er vist forneden i figur 8.1. Det er en smule omstændigt at finde frem til knuden i den anden ende af en kant. Listeknuden for kanten bruger ikke nogen særlig ordning til at gemme endeknuderne. Når vi altså i kantlisten støder på en hosliggende kant til u, skal vi inspicere begge knuder og vælge den, der er forskellig fra u. Alt afhængig af, om dette er den første eller anden knude i objektet, skal næste skridt i gennemløbet tages ud fra det første eller det andet par af pegere. Et elegant alternativ er at gemme værdien $u \oplus v$ i listeknuden for kanten $\{u, v\}$. [40] Bitvis ekslusivt-eller af denne indgang med et endepunkt leverer så det andet. Denne repræsentation sparer pladsen for et knudenavn. Diese Darstellung spart den Platz für einen Knotennamen ein. Dog skal man nu træffe en beslutning of, i hvilken ordning listepegerne svarer til kantens endepunkter. For eksempel kan man vedtage, at listeknuden for kanten $\{u, v\}$ med $u < v$ bruger sig første par af pegere til us naboliste og sit andet par til vs.

For rettede grafer, hvor hver kant (u, v) skal gemmes som udkant til u og som indkant til v, forekommer listeknuden for den rettede kant (u, v) i listen af us udkanter og i listen av vs indkanter. Også her skal man for hver kant bruge fire pegere i dens listeknude, to for dens rolle som udkant og to for dens rolle som indkant. Hver grafknude har en peger på en af sine udkanter, og en peger på en af sine indkanter.

8.4 Nabomatriser

En graf med n knuder kan repræsenteres som *nabomatrix* A af dimension $n \times n$. Indgangen A_{ij} er 1, hvis $(i, j) \in E$, og 0 ellers. Indsættelse og fjernelse af en kant såvel som kantforespørgsler kan gennemføres i konstant tid. For finde en hosliggende kant til en knude, kræves tid $\Theta(n)$. For meget tætte

grafer med $m = \Omega(n^2)$ bør det anses som effektivt. Lagerbehovet er n^2 bit. For meget tætte grafer kan dette være bedre end de $n + m + O(1)$ maskinord, som man bruger på naborækker. Men selv for disse grafer er besparelsen ringe, når kanterne indeholder yderligere information.

Opgave 8.4. Forklar, hvordan man kan repræsentere en urettet graf på n knuder og uden løkker med kun $n(n-1)/2$ bit. Kantforespørgsler skal gøres i tid $O(1)$.

Måske vigtigere end lageraspekterne ved brug af nabomatriser er den strukturelle sammenhæng mellem grafer og lineær algebra, , som skabes af nabomatrisen A. På den ene side kan nogle grafteoretiske problemer løses ved hjælp af metoder fra lineær algebra. Hvis vi fx sætter $C = A^k$, så er C_{ij} antallet af (ikke nødvendigvist simple) veje fra knude i til knude j med præcis k kanter.

Opgave 8.5. Forklar, hvordan man kan gemme en $n \times n$-matrix A med m indgange forskellig fra 0 i plads $O(m+n)$, således at matrix-vektor-produktet Ax kan udregnes i tid $O(m+n)$. Beskriv multiplikationsalgoritmen. Udvid repræsentationen, så den også omfatter produkter på formen $x^{\mathsf{T}}A$ i tid $O(m+n)$.

På den anden side kan grafteoretiske koncepter være nyttige til at løse problemer fra lineær algebra. Antag fx, at vi vil løse matrixligningen $Bx = c$, hvor B er en symmetrisk matrix. Begragt den tilsvarende nabomatrix A, for hvilken $A_{ij} = 1$ hvis og kun hvis $B_{ij} \neq 0$. Når en algoritme for sammenhængskomponenter finder ud af, at grafen, som beskrives af A, har to sammenhængskomponenter, kan vi bruge denne opdeling til at omorganisere rækker og søjler i B til en ensbetydende ligning:

$$\begin{pmatrix} B_1 & 0 \\ 0 & B_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}.$$

Denne ligning kan løses ved at løse hver af de to ligninger $B_1 x_1 = c_1$ og $B_2 x_2 = c_2$ for sig. I praksis er situationen mere speget, fordi matriser sjældent har den egenskab, at deres tilsvarende urettede graf er usammenhængende. Alligevel kan man bruge mere præcise grafteoretiske koncepter som fx snit til at opdage strukturer i matrisen, som muliggør mere effektiv løsning af ligningssystemet.

8.5 Implicitte repræsentationer

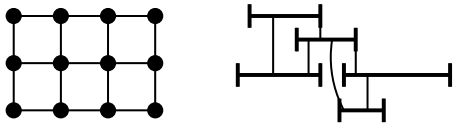
Mange anvendelser arbejder på grafer med speciel struktur. Ofte kan man udnytte denne situation til at opnå enklere og mere effektive repræsentationer. Vi giver to eksempler.

Gittergrafen $G_{k\ell}$ med knudemængden $V = \{1, \ldots, k\} \times \{1, \ldots, \ell\}$ og kantmængden

$$E = \{\, ((i,j),(i,j')) \in V^2 \colon \left|j - j'\right| = 1 \,\} \cup \{\, ((i,j),(i',j)) \in V^2 \colon \left|i - i'\right| = 1 \,\}$$

er en graf, som er helt bestemt af de to parametre k og ℓ. Figur 8.2 viser $G_{3,4}$. Kantvægte bør man gemme i to todimensionelle rækker, en for de vertikale og en for de horisontale kanter.

En *intervalgraf* er givet ved en endelig mængde af intervaller i $\mathbf{R}$. Til hvert interval findes en knude i grafen, og to knuder er naboer, hvis de tilsvarende intervaller overlapper.



Figur 8.2. Gittergrafen G_{34} (*venstre*) og en intervalgraf med fem knuder og seks kanter (*højre*).

Opgave 8.6 (Repræsentation af intervalgrafer)**.** (a) Vis, at der for hver mængde af n intervaller i $\mathbf{R}$ findes en mængde af intervaller, hvis endepunkter er heltal i $\{1,\ldots,2n\}$ og som definerer samme graf.

(b) Beskriv en algoritme, som afgør, om en intervalgraf givet som n intervaller er sammenhængende. *Vink*: Algoritmen bør sortere intervallernes endepunkter og derefter gennemgå dem i stigende rækkefølge. Oprethold herunder mængden af af påbegyndte, men ej afsluttede, intervalle.

(c*) Beskriv en repræsentation af intervalgrafer, som nøjes med $O(n)$ plads og understøtter effektiv navigation i følgende forstand: Givet et interval I skal alle intervaller I' rapporteres, som overlapper I; ved overlap forstås her at I indeholder et af I's endpunkter eller der gælder $I \subseteq I'$. Hvordan kan man givet I finde samtlige intervaller af begge slags?

8.6 Implementationsaspekter

Udeladt

8.7 Historiske anmærkninger og yderligere resultater

Udeladt

9

Grafgennemløb

Antag, at du arbejder i trafikstyrelsen i en lille by med et smukt centrum fra middelalderen. [1] *En uhellige alliance af detajlhandlende, som ønsker flere parkeringspladser ved gaden, og det grønne parti, som helst vil fjerne al biltrafik fra byen, er kommet frem til, at næsten alle gader skal være ensrettede. Du vil forhindre det værste og sikre, at den foreslåede plan i det mindste tillader, at man stadig kan komme med bil fra hvert sted i byen til hvert andet.*

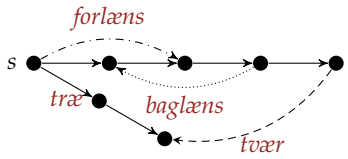
Utrykt i graftermonologi (se afsnit 2.9) lyder spørgsmålet, om det rettede graf, som dannes af gaderne, er *stærkt sammenhængende*. Samme problem dukker op i andre sammenhænge. I et kommunikationsnetværk med unidirektionale kanaler (fx radiosendere) vil man vide, hvem der kan kommunikere med hvem. Bidirektional kommunikation er mulig inden for de *stærke sammenhængskomponenter* af den rettede graf, som er givet ved kommunikationskanalerne.

I afsnit 9.2.2 skal vi møde en enkel og effektiv algoritme for stærke sammenhængskomponenter (ssk). Beregningen af stærke sammenhængskomponenter og mange andre grundlæggende problemer kan reduceres til en systematisk gennemløb af grafen, som besøger hver kan præcis én gang. Vi betragter de to vigtigste gennemløgsstrategier: *bredde først-søgning* (bfs) i afsnit 9.1 og *dybde først-søgning* (dfs) i afsnit 9.2. Begge strategier konstruerer rettede skove og inddeler kanterne i fire klasser: *træ*-kanter, som udgør skoven, *forlæns*-kanter, som forløber parallelt med veje av trækanter, *baglæns*-kanter, som forløber antiparallelt med veje av trækanter, og endeligt *tvær*-kanter. Den sidste kaller udgøres af alle øvrige kanter. De forbinder forskellige undertræer i skoven. Figur 9.1 viser eksempler på kantklasserne.

9.1 Bredde først-søgning

En enkel metode til at besøge alle knuder (og kanter), som kan nås fra en startknude s, er *bredde først-søgning* (bfs). Bfs besøger grafen *lag for lag*. Start-

[1] Kobberstikket viser en del af Frankfurt am Main omkring 1628 (M. Merian).



Figur 9.1. Kanterne i en rettet graf, klassificeret som træ-, forlæns-, baglæns- eller tvær-kanter. Knuden s danner træets rod.

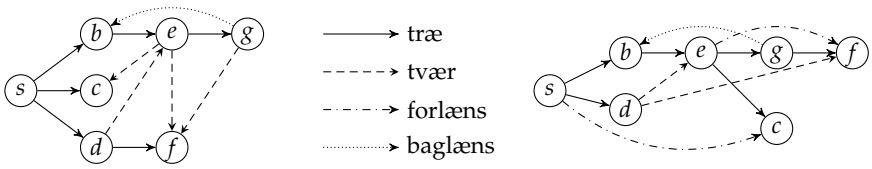
knuden s udgør lag 0. Naboerne (eller efterfølgerne i en rettet graf) til s udgør lag 1. Generelt udgøres lag $i+1$ af de knuder, som har en nabo (i en rettet graf: forgænger) i lag i men ingen nabo (eller forgænger) i lagene 0 op til $i-1$. Når v tilhører lag i, siger vi også, at v har *dybde* i, og at v har *afstand* i fra s.

```
Funktion bfs(s: KnudeId): (Knuderække af 0..n) × (Knuderække af KnudeId)
    d = ⟨∞,...,∞⟩ : Knuderække af 0..n        // Afstand fra roden
    forælder = ⟨⊥,...,⊥⟩ : Knuderække af KnudeId
    d[s]:=0
    forælder[s]:=s        // Løkke: signalerer »rod«
    Q = {s} : Mængde af KnudeId        // aktuelt lag i bfs-træet
    Q' = ∅ : Mængde af KnudeId        // næste lag i bfs-træet
    for ℓ:=0 til ∞ så længe Q ≠ ⟨ ⟩ udfør        // Besøg lag for lag
        invariant Q indeholder alle knuder med afstand ℓ fra s
        for hvert u ∈ Q udfør
            for hvert (u,v) ∈ E udfør        // Besøg udgående kanter fra u
                hvis forælder[v] = ⊥ så        // Fundet hidtil ubesøgt knude
                    Q':=Q' ∪ {v}        // Gem den til næste lag
                    d[v]:=ℓ+1
                    forælder[v]:=u        // Aktualiser bfs-træet
        (Q,Q'):=(Q',∅)        // Skift til næste lag
    returner (d,forælder)        // Bfs-træet er nu { (v,w): w ∈ V, v = forælder[w] }
```

Figur 9.2. Bredde først-søgning fra knude s.

Algoritmen i figur 9.2 får en startknude s, bestemmer alle de knuder, der kan nås fra s, og konstruerer et bfs-træ med rod s på disse knuder. For hver knude v i træet noterer algoritmens dens afstand $d(v)$ fra s samt forgængerknuden $\mathit{forælder}(v)$, fra hvilken v blev besøgt for første gang. Hertil bruges to knuderækker $\mathit{forælder}$ og d. Algoritmen leverer et par $(d, \mathit{forælder})$. I begyndelsen gælder kun s som besøgt; alle andre knuder gemmer som forgænger en særlig værdi $\perp$ for at vise, at ikke blev besøgt endnu. Dybden af s er 0. I algoritmens hovedløkke bygges bfs-træet lag for lag. Vi arbejder med to mængder Q og Q'; mængden Q indeholder altid knuderne i det aktuelle lag, mens næste lag opbygges i Q'. De indre løkker inspicerer alle kanter (u,v), som fører fra en knude u i det aktuelle lag Q. Når vi møder en knude v, som endnu ikke har en forgængerreference, føjer vi den til mængden Q' og sætter $d[v]$ samt forgængeren $\mathit{forælder}[v]$ til de pågældende værdier. Kanten

(u, v) bliver trækant. De øvrige kanter er forlæns- og tværkanter; algoritmen er ligeglad med forskellen. I figur 9.3 vises et eksempel på et bfs-træ med tilsvarende forlæns- og tværkanter.



Figur 9.3. Eksempel på klassifikationen af kanter ved bredde først-søgning (*venstre*, med lag) og dybde først-søgning (*højre*) i træ-, forlæns-, baglæns- og tværkanter. (Ved bfs opstår aldrig forlænskanter; selve algoritmen skelner ikke mellem baglæns- og tværkanter.) Bfs besøger knuderne i rækkefølge s, b, c, d, e, f, g; dfs besøger dem som s, b, e, g, f, c, d.

Bredde først-søgning har den nyttige egenskab, at dens trækanter danner de korteste veje (med hensyn til antal kanter) fra roden s til knuderne i træet. Den slags veje kan man fx bruge til at finde togforbindelser, hvor den rejsende skal skifte tog mindst ofte, og veje i kommunikationsnetværk som minimerer antal mellemstationer (også kaldet »hop«). En sådan korteste vej fra s til v kan man finde ved at følge forgængerreferencer fra v gentagne gange.

Opgave 9.1. Vis, at der efter bredde først-søgningen i G gælder $d[v] \leq d[u] + 1$ for hver kant (u, v) Især kan der ikke findes forlænskanter. Konkluder herfra, det minimale antal kanter på en vej fra s til v er $d[v] = d(v)$.

Opgave 9.2. Hvad går der galt i en bfs-implementation, hvis vi initaliserer $\mathit{forælder}[s]$ til $\perp$ i stedet for s? Giv et eksempel på en fejlslagen beregning.

Opgave 9.3. Bfs-træer er sædvandligvis ikke entydigt bestemte af grafens struktur. Især har vi ikke lagt os fast på, i hvilken rækkefølge vi vil behandle knuderne i det aktuelle lag. Tegn bfs-træet, som opstår i grafen i figur 9.3 ved bredde først-søgning fra s, når knude d behandles før knude b.

Opgave 9.4 (Kø-baseret bredde først-søgning)**.** Forklar, hvordan bredde først-søgning kan implementeres med en enkelt kø bestående af de knuder, hvis udkanter endnu er blevet undersøgt. Vis, at den resulterende algoritme og den figur 9.2 viste algoritme leverer samme bfs-træ, når den senere algoritme gemmenløber mængden Q og opbygger mængden Q' i en passende rækkefølge. Sammenlign den kø-baserede udgave af bredde først-søgning med Dijkstras algoritme fra afsnit 10.3 og Jarník-Prims algoritme fra afsnit 11.2. Hvad har de til fælles? Hvad er de væsentlige forskelle?

Opgave 9.5 (Bredde først-søgning med konkret grafrepræsentation)**.** Giv en præcis beskrivelse af bfs-algoritmen under antagelse af, at grafen er repræsenteret som naborække fra afsnit 8.2. Algoritmen skal bruge tid $O(n + m)$.

Opgave 9.6 (Urettet bfs og sammenhængskomponenter)**.** Betragt en urettet graf.

(a) Giv en detaljeret forklaring af bredde først-søgning for urettede grafer i tid $O(n+m)$. Hertil er det bekvemt at bruge grafens dobbelrettede udgave. Når kanten (u,v) er blevet betragtet fra knude u, skal derefter dens modsatrettede kant (v,u) ignoreres fra v. Vis, at der ikke kan opstå baglænskanter.

(b) Gør rede for, hvordan man kan modificere bredde først-søgning til at beregne en spændeskov (se kapitel 11) i tid $O(n+m)$. Desuden skal algoritmen i hver sammenhængskomponent udpege en knude r til repræsentant og for hver knude i sammenhængskomponenten tildele værdien r til rækkenindgangen $komponent[v]$. *Vink*: Brug en ydre løkke til at gennemløbe alle knuder $s \in V$ og udfør $bfs(s)$ fra hver hidtil ubesøgte knude s. Undlad at ændre forgænderrækken *forælder* mellem de enkelte udførsler af bfs. Bemærk, at en isoleret knude udgør sin egen sammenhængskomponent.

Opgave 9.7 (Transitiv lukning)**.** Den *transitive lukning* $G^+ = (V, E^+)$ af en graf $G = (V, E)$ indeholder kanten $(u,v) \in E^+$, hvis og kun hvis der findes en vej fra u til v i E af længde mindst 1. Konstruer en algoritme, som beregner den transitive lukning af en givet graf G. *Vink*: Udfør $bfs(v)$ for hver knude v for at identificere de knuder, som kan nås fra v. Helst bør rækkerne d og *forælder* ikke initaliseres påny ved hvert af disse kald. Hvad er algoritmens kørselstid?

9.2 Dybde først-søgning

Man kan opfatte bredde først-søgning som en forsigtig, konservativ strategi for grafgennemløb, som grundigt undersøger kendte ting inden den vover sig ud i ukendt territorium. I så henseende er *dybde først-søgning* (dfs) lige modsat: Når dfs støder på en ny knude, leder den umiddelbart videre i resten af grafen med denne knude som udgangspunkt. Den vender ikke tilbage, inden alle andre muligheder er udtømte. Sammenlignet med de pænt lagdelte træer, som skabes af bredde først-søgning, skaber dybde først-søgning træer, som er ubalancerede og underligt formede. Alligevel gør kombinationen af stadigt fremadrettet søgning og datamatens perfekte hukommelse dybde først-søgning til en overordentlig nyttig strategi. I figur 9.4 er vist et generelt skema til algoritmer baseret på dybde først-søgning. Ud fra skemaet kan vi breskrive specifikke algoritmer ved at erstatte pladsholderne *initialiser*, *rodfæst*, *passerTrækant*, *passerIkketrækant*, og *vendTilbage* med passende kodestumper.

Dybde først-søgning gør brug af knudemarkeringer. I begyndelsen er alle knuder umarkerede. En knude v markeres som *aktiv*, når først opdages og søgningen fra v begynder. Senere, når denne søgning er afsluttet, markeres v som

færdig, hvorefter markeringen aldrig ændres mere. Hovedløkken i dybde først-søgning leder efter en umarkeret knude s og kalder $dfs(s, s)$, for at gennemsøge grafen fra s og opbygge et træ med s som rod. Det rekursive kald $dfs(u, v)$ mønstrer alle vs udgående kanter (v, w) og afgør, om der rekursivt skal foretages en søgning fra w. Første komponent u i argumentet (u, v) angiver, at v blev opdaget via dens indgående kant (u, v). For rodknuden s misbruger vi første argumentposition ved at sætte s selv som »attrap-argument«. Vi skriver $dfs(*, v)$, når det for sammenhængen er ligegyldig, på hvilken måde v blev opdaget.

Kaldet $dfs(*, v)$, som realiserer søgningen fra v, begynder med at markere v som *aktiv*. Herefter mønstres de udgående kanter (v, w) en efter en. Betragt kanten (v, w). Hvis målknuden w allerede er markeret, dvs. at den blev opdaget tidligere, udgør w allerede en knude i dfs-skoven, og (v, w) kan ikke være en trækant. Derfor kaldes $passerIkketækant(v, w)$, men der forekommer ikke noget rekursivt kald af dfs. Hvis målknuden w derimod ikke allerede er markeret, dvs. at den er uopdaget, bliver (v, w) en trækant. Da kaldes $passerTrækant(v, w)$, efterfulgt af det rekursivt kald $dfs(v, w)$ for at påbegynde udforskningen fra w. Først når alle aktioner og (rekursive) kald i forbindelse med med trækanten (v, w) er afsluttede, betragtes den næste udgående kant fra v. Denne proces gentages. indtil alle udgående kanter fra v er blevet behandlet. Til sidst i dfs-kaldet for knude v udføres proceduren *vendTilbage* med argumentet (u, v) for at foretage oprydningsarbejde eller sammenføre data. Disse aktioner kan være forskellige, afhængigt af, om v udgør en rodknude (i hvilket fald $u = v$) eller ej ($u \neq v$). Endeligt markeres v som *færdig*, og kaldet returnerer.

På hvert tidspunkt under udførelsen af dybde først-søgningen i figur figur 9.4 er et eller flere aktive kald af proceduren dfs i gang. Nærmere betegnet findes der knuder $v_1, \ldots, v_k$ sådan at den næste kant, der skal behandles, udgår fra v_k, og at de aktive kald netop er $dfs(v_1, v_1)$, $dfs(v_1, v_2)$, $\ldots$, $dfs(v_{k-1}, v_k)$. I denne situation er netop knuderne $v_1, \ldots, v_k$ markerede som *aktiv*, og rekursionsstakken indeholder følgen $\langle (v_1, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \rangle$. Vi siger, at knude v er aktiv (hhv. færdig), når den er markeret som *aktiv* (hhv. *færdig*). Vi siger også kortfattet, at $v_1, \ldots, v_k$ udgør rekursionsstakken. Knude v_k kaldes for den *aktuelle knude*. Vi siger, at en knude er *nået*, når $dfs(*, v)$ allerede er hændt. De nåede knuder er altså lig med de aktive og de færdige knuder.

Opgave 9.8. Giv en ikke-rekursive formulering af dybde først-søgning. Der findes to mulige fremgangsmåder: Man kan opretholde en af aktive knuder og gemmer for hver aktive knude v mængden af ikke-undersøgte kanter. (Hertil er det nok med en peger ind i listen eller rækken af udgående kanter fra v.) Altenativt og porgrammeringsmæsssigt mere elegant lader man stakken bestå af de ikke hidtil undersøgte udgående kanter fra samtlige aktive knuder.

```
Dybde først-søgning i en rettet graf G = (V, E)
Fjern samlige knudemarkeringer
initialiser
for hvert s ∈ V udfør
    hvis s er umarkeret så
        rodfæst(s)        // Gør s til rod
        dfs(s, s)         // Opbyg et nyt dfs-træ med rod s

Procedure dfs(u, v: KnudeId)      // udforsk v, kommende fra u.
    marker v som aktiv
    for hvert (v, w) ∈ E udfør
        hvis w er markeret så passerIkketrækant (v, w)      // w er besøgt før
        ellers passerTrækant(v, w)      // w er hidtil ubesøgt
            dfs(v, w)
    vendTilbage(u, v)      // Ryd op; sammenfat og returner data
    marker v som færdig
    returner
```

Figur 9.4. Et algoritmeskema for dybde først-søgning i en rettet graf $G = (V, E)$. Vi siger, at kaldet $dfs(*, v)$ *udforsker* knuden v. Denne udforskning er gennemført med kaldets afslutning.

9.2.1 Dfs-numre, sluttider og topologisk sortering

Dybde først-søgning har mange former og anvendelser. I dette afsnit vil vi bruge dybde først-søgning for at nummerere knuderne på to forskellige måder. Som sidegevist får vi en metode til at afgøre, om en graf er kredsfri. Vi nummererer knuderne i den rækkefølge, vi når dem (rækken *dfsNr*), og i den rækkefølge, vi afslutter dem (rækken *slutnummer*). Hertil benytter vi to globale tællere *dfsPos* og *slutPos*, som begge begynder med 1. Når algoritmen opdager en ny rod eller følger en trækant til en ny knuder v, sættes $dfsNr[v]$ til *dfsPos*, og *dfsPos* øges med 1. Når vi fra knude v vender tilbage ad en trækant (u, v) til u, fordi v er færdig, sættes $slutNr[v]$ til *slutPos*, og *slutPos* øges med 1. Dette realiseres ved følgende valg af underprogrammer:

initialiser:	$dfsPos = 1 : 1..n; \quad slutPos = 1 : 1..n$
$rodfæst(s)$:	$dfsNr[s] := dfsPos$ ++
$følgTrækant(v, w)$:	$dfsNr[w] := dfsPos$ ++
$vendTilbage(u, v)$:	$slutNr[v] := slutPos$ ++

Dfs-nummereringens ordning er så nyttig, at vi ophøjer den til en ordningsrelation med sit eget symbol »$\prec$«. Vi definerer altså for vilkårlige knuder u og v:

$$u \prec v \quad \Leftrightarrow \quad dfsNr[u] < dfsNr[v]\,.$$

Vi skal nu se, at nummereringerne *dfsNr* og *slutNr* indkoder vigtig information over forløbet af dybde først-søgningen. Først skal vi vise, at dfs-numrene langs hver vej i dfs-træet er voksende, og derefter, at man kan klassificere kanterne

ud fra disse numre. Under forløbet kan numrene også bruges til at inkode knudemarerkingerne. Vi udvider hertil *initialiser* således, at indgangene i rækkerne *dfsNr* og *slutNr* alle initialiseres til 0.. Da er knuden v umarkeret netop når hvis $dfsNr[v] = 0$, knuden v er aktiv netop når både $dfsNr[v] > 0$ og $slutNr[v] = 0$, og knude v er færdig netop når både $dfsNr[v] > 0$ og $slutNr[v] > 0$.

Lemma 9.1. *Knuderne på dfs-rekursionsstakken er altid ordnede med hensyn til* $\prec$.

Bevis. Tælleren *dfsPos* øges efter hver tildeling til en indgang $dfsNr[v]$. Når en knude v bliver aktiv under et kald $dfs(u, v)$ og lægges på rekursionsstakken, har $dfsNr[v]$ umiddelbart inden fået det hidtil største dfs-nummer. □

Typen af en kant (v, w) kan aflæses fra dfs-nummeret og endenummeret af v og w samt markeringen af w på det tidspunkt, hvor (v, w) betragtes. Kombinationerne er sammenfattede i tabel 9.1. Indgangene kan motiveres på følgende måde. Vi lægger først mærke til, at der for to kald til proceduren *dfs* for forskellige knuder bare findes to muligheder: Enten *indlejrer* de hinanden i den forstand, at det ene kald stadig er aktiv, når det andet kald begynder, eller så er de *disjunkte*, dvs. at det andet kald først påbegyndes, når det første er afsluttet. Hvis kaldet $dfs(*, w)$ er indlejret i kaldet $dfs(*, v)$, begynder w-kaldet efter v-kaldet og asluttes inden det. Men det betyder, at $dfsNr[v] < dfsNr[w]$ og $endeNr[w] < endeNr[v]$. Hvis $dfs(*, w)$ og $dfs(*, v)$ er diskunkte og w-kaldet begynder før v-kaldet, så ender det også før v-kaldet. Det betyder, at $dfsNr[v] > dfsNr[w]$ og $slutNr[w] < slutNr[v]$.

Hernæst bemærker vi, at den *umiddelbare* struktur af indlejrede rekursive kald er gengivet i trækanterne. Mere nøjagtigt er en kant (v, w) en trækant, netop når inspektionen af kanten (v, w) i løbet af kaldet $dfs(*, v)$ udløser kaldet $dfs(v, w)$. Dette sker netop når w stadig er umarkeret i dette øjeblik. Denne observation medfører, at et kaldet $dfs(*, w)$ er indlejret i kaldet $dfs(*, v)$, hvis der findes en vej af trækanter fra v til w. Hvis nu (v, w) er en træ- eller forlænskant, så findes der en sådan vej af trækanter fra v til w, og derfor er $dfs(*, w)$ indlejret i $dfs(*, v)$. Heraf følger, at w har et større dfs-nummer og et mindre slutnummer end v. Man kan skelne disse to typer ved at kigge på den markering, som w har ved inspektionen af kant (v, w). Hvis w er umarkeret, så er (v, w) en trækant. Hvis w er markeret, må w være færdig, og (v, w) er en forlænskant. (Antag modstætningsvist at w er aktiv. Så skulle w stadig ligge på rekursionsstakken. Men da v selv ligger øverst på rekursionsstakken, medfører lemma 9.1, at $dfsNr[w] < dfsNr[v]$, hilket er en modstrid.)

En baglænskant (v, w) løber antiparallel til en vej af trækanter; derfor har w et mindre dfs-nummer og et største slutnummer end v. Når kanten (v, w) behandles, er kaldet $dfs(*, v)$ aktivt, og på grund af indlejringen er kaldet $dfs(*, w)$ også aktivt. Derfør fører behandlingen af kanten (v, w) til, at w markeres som *aktiv*.

Til sidst betragter vi tværkanterne. Disse forløber hverken parallelt eller antiparallelt til nogen vej af trækanter. Det betyder ifølge betragtningerne

Kanttype	Dfs-nummer	Slutnummer	Markering af w
Træ-	$v \prec w$	$v > w$	umarkeret
Forlæns-	$v \prec w$	$v > w$	*færdig*
Baglæns-	$v \succ w$	$v < w$	*aktiv*
Tvær-	$v \succ w$	$v > w$	*færdig*

Tabel 9.1. Klassifikation af en kant (v, w). Under »slutnummer« betyder »$v > w$« at $slutNr(v) > slutNr(w)$.

foroven, at de to kald $dfs(*, v)$ og $dfs(*, w)$ hverken er indlejrede på den ene eller den anden måde. Derfor er kaldene disjunkte. Det vil sig, at w enten markeres som *færdig*, før $dfs(*, v)$ begynder, eller w først markeres, efter $dfs(*, v)$ er afsluttet. Det sidste kan ikke forekomme, fordi w da skulle være umarkeret, når kanten (v, w) bliver behandlet, hvilket skulle gøre (v, w) til en trækant. Derfor bliver w markeret som først *aktiv* og så *færdig*, inden $dfs(*, v)$ begynder. Der gælder altså $dfsNr[v] > dfsNr[w]$ og $slutNr[w] < slutNr[v]$, samt at knuden w er færdig, når kanten (v, w) behandles. Resultaterne af ovenstående analyse sammenfattes i følgende lemma.

Lemma 9.2. *Tabel 9.1 viser, hvordan typen af en kant (v, w) kan bestemmes ud fra værdierne i* dfsNr *og* slutNr *samt ws markering på tidspunktet af behandlingen af (v, w).*

Opgave 9.9. Modificer dybde først-søgning, så den markerer hver kant med dens type, når den behandles.

Slutnumrene kan bruges til at afgøre, om en rettet graf er acyklisk.

Lemma 9.3. *Følgende er ækvivalente:*

(i) G er acyklisk.
(ii) Dybde først-søgning på G skaber ingen baglænskanter.
(iii) For hver kant (v, w) i G gælder $\text{slutNr}[v] > \text{slutNr}[w]$.

Bevis. Hver baglænskant forløber antiparallelt til en vej af trækanter, så de tilsammen danner en kreds. Derfor kan dybde først-søgning på acykliske grafer aldrig skabe en baglænskant. Fra tabel 9.1 aflæser vi, at alle kanter bortset fra baglænskanter er rettede fra en knude med større slutnummer til en knude med mindre slutnummer. Antag nu, at iii gælder. Da er slutnumrene langs hver vej skarpt faldende, så ingen vej kan danne en kreds, og G er acyklisk. □ □

En lineær ordning af knuderne i en rettet acyklisk graf, hvori hver kant går fra en »mindre« til en »større« knude, kaldes en *topologisk ordning*. Ifølge lemma 9.3 er den ordingen af knuderne efter aftagende slutnummer, en topologisk ordning. Mange problemer på rettede acykliske grafer kan løses

effektivt ved at behandle knuderne i rækkefølge givet af en topolisk ordning. Fx skal vi i afsnit 10.2 møde en hurtig og enkel algoritme til at finde korteste veje i en acyklisk graf, som benytter denne fremgangsmåde.

Opgave 9.10 (Topologische Sortierung)**.** Konstruer er dfs-baseret algoritme, som for en givet rettet graf G beregner en topologisk ordning, hvis G er kredsfri. I modsat fald skal algoritmen returnere en kreds.

Opgave 9.11. Findes der en bfs-baseret algoritme, som beregner en topologisk ordning for en givet rettet acyklisk graf?

Opgave 9.12. Ved dybde først-søgning i urettede grafer er det oplagt at undersøge en urettet kant $\{v, w\}$ bare i én retning. Dette gør man ved at »slukke for« kanten (w, v) i nabolisten til w, når (v, w) besøges fra v. Vis, at der hverken forekommer forlæns- eller tværkanter ved dybde først-søgning i urettede grafer.

9.2.2 Stærke sammenhængskomponenter

Udeladt

9.3 Implementierungsaspekte

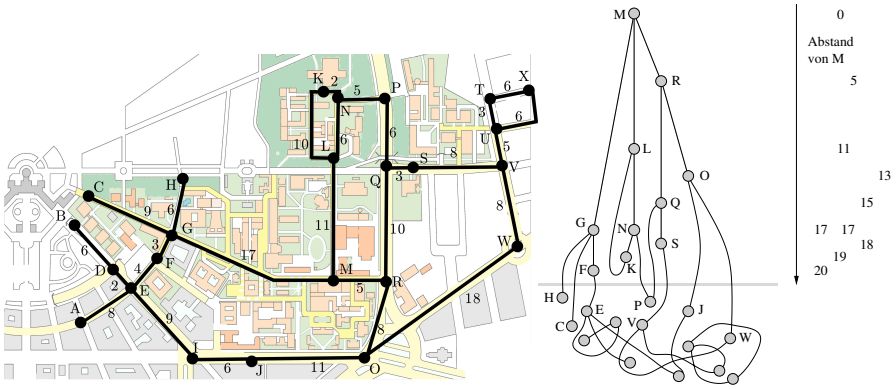
Udeladt

9.4 Historische Anmerkungen und weitere Ergebnisse

Udeladt

10

Korteste veje



Problemet at finde en korteste, hurtigste eller billigste vej i en netværk er allestedsnærværende. Hver af os løser det dagligt. Er man på plads s og ønsker at være på plads t, spørger man om vej for at komme fra s til t på den bedste måde. Brandvæsnet ønsker at kende den hurtigste vej fra brandstationen s til samtlige pladser t i hele byen – varianten af korteste-veje-problemet »med samme udgangspunkt« eller »med en enkelt kilde«. Sommetider ønsker man til og med en fuldstændig tabel over alle parvise afstande – denne variant hedder »korteste veje for alle par«. Et eksempel på den slags tabeller med afstande mellem større byer kan man finde i de fleste vejatlas.

Der findes en algoritme til ruteplanlægning, som kun bruger et vejkort, en rulle sytråd og en masse fingerfærdighed, men ingen computer. Man lægger tynde tråde langs gaderne på vejkortet. Ved vejkryds knytter man trådene sammen til knuder; udgangspunktet (»kilden«) får ligeledes en knude. Nu lyfter man forsigtigt kildeknuden lodret opad, til hele netværket dingler under den. Lykkes man med dette uden at forkludre trådene, og knuderne er så små, at kun stramme tråde kan hindre dem i at falde ned, så definerer de spændte trådstykker netop de korteste veje fra udgangsknuden. Billedet foroven[1] *viser Instituttet for Teknologi i Karlsruhe (KIT) og den trådbaserede ruteplanlægningsalgoritme med M som kilde.*

Ruteplanlægning i vejnet er en af de mange anvendelser for korteste-vejeberegninger. Men der er mange andre problemer, som løses bedste af korteste-veje-beregninger, når man har fundet en passende modellering i termer af grafer. Ahuja *et al.* [2] nævner anvendelser som netværksstrømme, boligbyggeri, forvaltning af lagerbeholdning i produktion og handel, dna-sekvensering, rygsæksproblemet (se også kapitel 12), produktionsplanlægning, skemalæg-

[1] © KIT, Institut für Photogrammetrie und Fernerkundung.

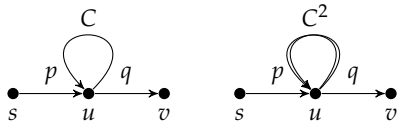
ning af arbejdstider fx for callcentre, parkering, approksimation af stykvist linjære funktioner og fordeling af inspektionstid af et samlebånd.

I den mest almene formulering af korteste-veje-problemet betrager man en rettet graf $G = (V, E)$ og en omkostningsfunktion c, som afbilder hver kant til en reelt tal, som modellerer omkostningen ved at passere denne kant. Det viser sig, at problemet i denne formulering er overordentligt tidskrævende at løse. Defor interesserer vi os for forskellige begrænsninger, som tillader enkle og effektive algoritmer: ikkenegative kantomkostninger, , heltallige kantomkostninger og acykliske grafer. Læg mærke til at det meget særlige tilfælde, hvor alle kanter har omkostning 1 allerede blev løst i kapitel 9.1 – bredde først-træet med rod s er en kompakt repræsentation af de korteste veje fra s til samtlige andre knuder. I afsnit 10.1 betragter vi indledningsvist nogle grundlæggende begreber, som leder til et fælles grundlag for alle korteste-veje-algoritmer. En systematisk fremgangsmåde skal hjælpe os at holde styr på de mange forskellige korteste-veje-algoritmer. Som første eksempel på en hurtig og enkel algoritme for en begrænset problemklasse betragter vi i afsnit 10.2 en fremgangsmåde for acykliske grafer. I afsnit 10.3 kommer vi til den mest-benyttede algoritme for kortest veje, nemlig Dijkstras algoritme for rettede grafer med ikkenegative kantomkostninger. Hvor effektiv Dijkstras algoritme i virkeligheden er, afhænger af, hvor effektive de benyttede prioritetskøer er. I et indledende kursus eller ved første gennemlæsning kan man sagtens afslutte temaet »korteste veje« med Dijkstras algoritme. Men resten af kapitlet indeholder mange andre interessante aspekter af temaet. Vi begynder men en analyse af gennemstnitsopførelsen af Dijkstra i afsnit 10.4, som antyder, at operationerne på prioritetskøen i normaltilfældet er mindre dominerende end hvad værstefaldsanalysen fortæller. I afsnit 10.5 diskuterer vi *monotone prioritetskøen for heltallige nøgler*, som udnytter en særlig egenskab i Dijkstras algoritme. Når man kombinerer disse køer med gennemsnitsanalysen, opnår man lineær forventet udførelsetid! Afsnit 10.6 behandler situationen med vilkårlige kantomkostninger, og i afsnit 10.7 behandler vi problemet at finde korteste veje mellem alle par af knuder. Vi viser, at problemet kan reducere til først at løse en instans med én kildeknude og vilkårlige kantomkostninger, og derefter n instanser med én kildeknude og ikkenegative kantomkostninger. I løbet af reduktionen møder læseren også det generelt nyttige koncept af knudepotentialer. Kapitlet afrundes med afsnit 10.8, fokusseres på korteste-veje-*forespørgsler*.

10.1 Fra grundbegreberne til en almen metode

Omkostningsfunktionen for kanterne udvides til veje på den oplagte måde. Omkostningen af en vej er summen af omkostningerne af dens kanter, dvs. at omkostningen for vejen $p = \langle e_1, \ldots, e_k \rangle$ er givet af $c(p) = \sum_{1 \leq i \leq k} c(e_i)$. Den tomme vej har omkostning 0. Vi bruger begreberne »omkostning«, »længde« og »vægt« i flæng, så »kortest« og »billigst« betyder det samme.

For knuder s og v interesserer vi os for en *korteste vej* fra s til v, dvs. en vej fra s til v, som koster mindst muligt. Vi undgår her bevidst den bestemte artikel som i »den korteste vej«, idet der kan findes flere forskellige veje, som alle i denne forstand er kortest mulige. Findes der overhovedet altid en korteste vej? Læg mærke til, at der kan findes uendeligt mange veje fra s til v. Hvis fx vejen $r = pCq$ fra s til v indeholder en kreds C, kan vi gennemløbe kredsen vilkårligt ofte og danner hver gang en ny vej fra s til v, se figur 10.1. Lad os beskrive denne sitation mere nøjagtigt: Lad p være en vej fra s til u, lad C være en kreds fra u til u, og lad q være en vej fra u til v. Betragt nu vejen $r^{(i)} = pC^iq$, som begynder med at gennemløbe p, derefter gennemløber C præcis i gange, og endeligt gennemløber q. Omkostningen for $r^{(i)}$ er $c(p) + i \cdot c(C) + c(q)$. Hvis C er en *negativ kreds*, dvs. hvis $c(C) < 0$, så findes der ingen kortest vej fra s til v. (Det er fordi mængden $\{ r^{(i)} : i \geq 0 \} = \{ c(p) + c(q) - i \cdot |c(C)| : i \geq 0 \}$ indeholder negative tal af vilkårlig stor absolutværdi.) Tilsvarende skal vi nu vise, at der findes kortest veje, hvis bare grafen G ikke indeholder negative kredse.



Figur 10.1. En ikke-simpel vej pCq fra s til v

Lemma 10.1. *Hvis G ikke indeholder negative kreds og v kan nås fra s, så findes der en korteste vej p_0 fra s til v. Desuden kan p_0 antages af være simpel.*

Bevis. Lad p_0 være en *simpel* vej fra s til v med minimale omkostninger blandt de simple veje fra s til v. Læg mærke til, at denne mængde af simple veje er endelig, og p_0 derfor eksisterer. Antag modsætningsvist, at p_0 ikke er en korteste vej fra s til v. Da findes en kortere vej r fra s til t, som ikke er simpel. Vi kan skrive r som pCq som i figur 10.1, hvor C mer en kreds og pq er en simpel vej. Vi har $c(p_0) \leq c(pq)$, og derfor $c(pq) + c(C) = c(r) < c(p_0) \leq c(pq)$. Heraf følger $c(C) < 0$, så C er en negativ kreds, i modstrid med antagelsen. Derfor er p_0 en korteste vej. □

Opgave 10.1. Giv en stærkere udgave af ovenstående lemma ved at vise følgende: For hver knude v, som kan nås fra s, findes en korteste vej fra s til v hvis og kun hvis der ikke findes en negativ kreds, som kan nås fra s, og fra hvilken man kan nå v.

For hvert par af knuder s og v definerer vi »korteste-veje-afstanden« eller bare *afstanden $\mu(s, v)$ fra s til v* som

$$\mu(s,v) := \begin{cases} +\infty\,, & \text{hvis der ikke findes en vej fra } s \text{ til } v\,; \\ -\infty\,, & \text{hvis der findes en vej fra } s \text{ til } v\text{, men ingen korteste} \\ & \text{vej fra } s \text{ til } v\,; \\ c(p)\,, & \text{hvis } p \text{ er en korteste vej fra } s \text{ til } v\,. \end{cases}$$

Når startknuden s er underforstået, skriver vi kort $\mu(v) := \mu(s,v)$. Læg mærke til, at når v kan nås fra s, uden at der findes en korteste vej, så må der findes veje fra s til v med vilkårlig store negative omkostninger. Derfor giver det mening at sætte $\mu(v) = -\infty$. Korteste veje har mange kønne egenskaber, som vi formulerer som øvelsesopgaver.

Opgave 10.2 (Delveje af korteste veje)**.** Vis, at en delvej af en korteste vej selv er en korteste vej, dvs. at der hvis p, q or r er veje som sammen danner en korteste vej pqr, så er q også en korteste vej.

Opgave 10.3 (Korteste-veje-træer)**.** Antag, at alle n knuder i en graf kan nås fra s og at der ikke forekommer negative kredse. Vis, at der findes et træ T med n knuder, som er rodfæstet i s og i hvilket alle veje er korteste veje i grafen. *Vink*: Begynd med at antage, at vejen fra s til v er entydigt bestemt. Betragt delgrafen T, som består af alle korteste veje fra s. Anvend opgave **??** for at vise, at T er et træ. Udvid argumentet til det tilfælde, hvor de korteste veje ikke er entydigt bestemte.

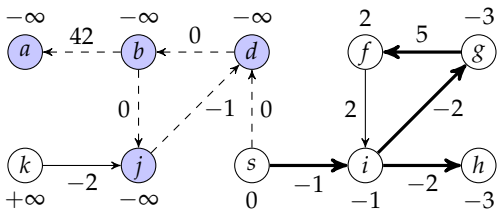
Vores strategi for at finde korteste veje fra startknuden s er en generalisering af bredde først-søgning fra figur 9.3. Vi arbejder med to rækker af typen *Knuderække*, som hedder d og *forælder*. Her indeholder indgangen $d[v]$ vores nuværende bedste bud på afstanden fra s til v, og $\mathit{forælder}[v]$ gemmer vs forgænger på en vej fra s til v, som er den korteste, vi har fundet hidtil. Værdien $d[v]$ kan man opfatte som *skønnet afstand* til v, et »pessimistisk« skøn som aldrig undervurderer den sande afstand. I begyndelsen har vi $d[s] = 0$ og $\mathit{forælder}[s] = s$. Alle andre knuders afstand skønner vi til at ligge »uendelig langt borte«, og de har ingen forgænger.

Den oplagte måde for at forbedre afstandsskønnene er at føre afstansinformation videre gennem enkelte kanter. Når der findes en vej fra s til u med omkostning højst $d[u]$, og $e = (u,v)$ er en udgående kant fra u, så findes der også en vej fra s til v med omkostning højst $d[u] + c(e)$. Hvis dette omkostning er mindre end den bedste hidtidlige afstandsværdi $d[v]$, opdaterer vi $d[v]$ og $\mathit{forælder}[v]$ tilsvarende. Denne proces kaldes at *slække* kanten e.

Procedure *slæk*$(e = (u,v) : \mathit{Edge})$
  hvis $d[u] + c(e) < d[v]$ **så** $d[v] := d[u] + c(e)$; $\mathit{forælder}[v] := u$

(Regnereglerne for værdien ∞ er de oplagte: Der gælder $a < \infty$ og $\infty + a = \infty$ for alle tal a, samt $\infty \not< \infty$.)

Lemma 10.2. *Antag at der efter en vilkårlig følge af kantslækninger gælder $d[v] < \infty$. Da findes en vej fra s til v af længde $d[v]$.*



Figur 10.2. En graf med startknude s og korteste-veje-afstande $\mu(v)$ angivet over eller under hver knude. Kantomkostningerne er angivet ved siden af hver kant. De fede kanter hører til korteste veje fra s. Stiplede linjer og schatterede knuder hører til en uendelig familje af veje med startknude s og vilkårligt stærkt negative omkostninger. Knude k kan ikke nås fra s.

Bevis. Induktion efter antal slækninger. Påstanden gælder klart inden første slækning: Den tomme vej fører fra s til s og har længde $0 = d[s]$; ingen andre knuder er berørte af udsagnet, da de skønnes at være uendeligt langt borte. Betragt nu slækningen af kanten $e = (u, v)$. Ifølge induktionsantagelsen findes der en vej p af længde $d[u]$ fra s til u og en vej af længde $d[v]$ fra s til v. Hvis $d[u] + c(e) \geq d[v]$, er der intet at vise. Ellers er pe en vej fra s til v og har længde $d[u] + c(e)$. □

Algoritmerne i dette kapitel følger alle strategien at gentagne gange slække kanter, indtil enten der for hver knude v er fundet en korteste vej fra s til v, eller en negativ kreds er blevet opdaget. For eksempel gengiver (omvendingen af) de fede kanter i figur 10.2 informationen i *forælder*-rækken, som er indsamlet efter tilstrækkelig mange kantslækninger: knuderne f, g, i og h kan nås fra s via disse kanter og har de korrekte $\mu(\cdot)$-værdier 2, -3, -1 og -3. Knuderne b, j og d danner en negativ kreds, som kan nås fra s, hvilket betyder, at deres korteste-veje-omkostning er $-\infty$. Knude a kan nås fra denne kreds og har derfor ligeledes afstand $\mu(a) = -\infty$. Derimod kan knude k ikke nås fra s; dens afstand er $+\infty$.

Men hvad udgør en god følge af kantslækninger? Lad $p = \langle e_1, \ldots, e_k \rangle$ være en vej fra s til v. Hvis vi slækker vejens kanter i rækkefølge fra e_1 til e_k, gælder derefter $d[v] \leq c(p)$. Dersom p er en korteste vej fra s til v, kan ifølge lemmaet foroven værdien $d[v]$ aldrig blive mindre end $c(p)$, og derfor gælder der efter denne følge af kantslækninger, at $d[v] = c(p)$.

Lemma 10.3 (Korrekthedskriterium). *Betragt udførelsen af en følge R af slækninger, som indeholder en korteste vej $p = \langle e_1, e_2, \ldots, e_k \rangle$ fra s til v som delfølge, dvs. at der findes indeks $t_1 < t_2 < \cdots < t_k$ med $R[t_1] = e_1$, $R[t_2] = e_2$, …, $R[t_k] = e_k$. Da gælder $d[v] = \mu(v)$. Desuden beskriver* forælder*-informationen for knuderne en vej fra s til v af længde μ.*

Bevis. Vi fremstiller R og p skematisk. Første linje angiver tidspunktet. Til tidspunkt t_1 slækkes e_1, til tidspunkt t_2 slækkes e_2, osv.:

$$\begin{array}{ll} & 1, 2, \ldots, t_1, \ldots, t_2, \ldots\ldots, t_k, \ldots \\ R{:=}\langle & \ldots \quad, e_1, \ldots, e_2, \ldots\ldots, e_k, \ldots\rangle \\ p{:=} & \langle e_1, \quad e_2, \quad \ldots \quad, e_k\rangle \end{array}$$

Idet p er en korteste vej, har vi $\mu(v) = c(e_1) + \cdots + c(e_k)$. For $i \in \{1, \ldots, k\}$ lad v_i betegne målknuden af e_i; desuden definerer vi $t_0 = 0$ og $v_0 = s$. Lad os først vise, at efter tidspunkt t_i gælder uligheden

$$d[v_i] \leq c(e_1) + \cdots + c(e_i)\,.$$

Dette indses ved induktion efter i. Efter tidspunkt t_0, altså helt i starten, gælder uligheden, fordi vi initielt sætter $d[s] = 0$. Betragt nu $i > 0$. Ifølge induktionsantagelsen gælder uligheden

$$d[v_{i-1}] \leq c(e_1) + \cdots + c(e_{i-1})$$

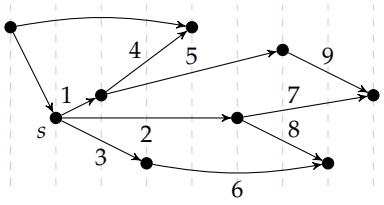
efter slækningen til tidspunkt t_{i-1}. Yderligere slæknigner i tidsrummet mellem t_{i-1} og t_i ændrer ikke uligheden, idet $d[v_{i-1}]$ ikke kan vokse, når der slækkes en kant. Slækningen til tidspunkt t_i sikrer, at $d[v_i] \leq d[v_{i-1}] + c(v_{i-1}, v_i)$. Ved addition med den sidste ulighed etablerer man induktionsantatelsen for tidspunktet t_i, hvilket beviser uligheden for alle $i \geq 0$. Uligheden for $i = k$ medfører da $d[v] \leq \mu(p)$. Yderligere slækninger efter tidspunkt t_k kan kun gørd $d[v]$ endnu mindre. På den anden side medfører 10.2, at der gælder $d[v] \geq \mu(v)$ uanset tidspunkt. Vi konkluderer $d[v] = \mu(v)$.

Vi mangler at vise, at informationen i *forælder*-rækken beskriver de korteste veje. Vi viser dette under den ekstra antagelse, at korteste veje er entydige; generelle tilfælde er en øvelse, jf.opgave **??**. Overvejelserne i første del af beviset medfører, at der efter slækningerne i R gælder $d[v_i] = \mu(v_i)$ for alle $i \in \{1, \ldots, k\}$. På noget tidspunkt udfører altså en operation på formen $\mathit{slæk}(u, v_i)$ i R tildelingerne $d[v_i] = \mu(v_i)$ og $\mathit{forælder}[v_i] = u$. (Læg mærke til, at dette sagtens kan ske inden tidspunkt t_i, men ikke bagefter.) Ifølge beviset for lemma 10.2 findes der en vej af længde $\mu(v_i)$ fra s til v_i, dvs. en korteste vej, der ender med kanten (u, v_i). Ifølge antagelsen er den korteste vej unik, derfor må der gælde $u = v_{i-1}$ og dermed $\mathit{forælder}[v_i] = v_{i-1}$. Efter at $d[v_i]$ er blevet sat til den korrekte værdi $\mu(v_i)$, beholder $\mathit{forælder}[v_i]$ værdien v_{i-1} i resten af forløbet. □ □

Opgave 10.4. Gentag ovenstående bevis, men uden antagelsen om entydighed af korteste veje.

Opgave 10.5. Lad S være en følge af kanterne i G i vilkårlig rækkefølge, og betragt følgen $S^{(n-1)}$ af $(n-1)$ gentagelser af S. Vis at der efter slækningerne $S^{(n-1)}$ gælder $\mu(v) = d[v]$ for hver knude v med $\mu(v) \neq -\infty$.

Som vi skal se i de følgende afsnit, kan man finde langt mere effektive slækningsrækkefølger, når grafen en acyklisk eller kantvægtene ikke-negative. Vi vender tilbage til generelle grafer i afsnit 10.6.



Figur 10.3. En mulig rækkefølge for kantslækningerne til beregning af korteste veje i en rettet acyklisk graf fra knude s. Den topologiske sortering af knuderne er vist ved deres placering efter x-koordinat; knuderne er sorterede fra venstre til højre. Udgående kanter fra knuder »til venstre« for s behøver slet ikke slækkes.

10.2 Rettede acykliske grafer

I en rettet acyklisk graf forekommer ingen rettede kredse og derfor heller ingen negative kredse. Desuden har vi konstateret i afsnit 9.2.1k, at knuderne i en sådan graf tillader en topologisk ordning, dvs. at de kan bringes i rækkefølge $\langle v_1, \ldots, v_n \rangle$, sommetider kaldet en *topologisk sortering*, sådan at $(v_i, v_j) \in E$ altid medfører $i < j$. En topologisk sortering kan findes i tid $O(m+n)$ med dybde først-søgning. Læg mærke til, at knuderne langs hver vej i en rettet acyklisk graf står i voksende rækkefølge i den topologiske ordning. Ifølge lemma 10.3 kan vi altså beregne korteste-veje-afstande ved at slække først alle udgående kanter fra v_1, derefter alle udgående kanter fra v_2, og så videre. I figur 10.3 er vist et eksempel. På denne måde bliver hver kant slækket højst enkelt gang. Fordi slækning tager konstant tid, er udførelsestiden totalt $O(m+n)$.

Sætning 10.1. *I acykliske grafer kan korteste veje fra en enkelt knude beregnes i tid* $O(m+n)$.

Opgave 10.6 (Ruteplanlægning ved offentlig transport)**.** Problemet at komme hurtigt frem med offentlig transport kan modelleleres som korteste-veje-problem i en acyklist graf. Betragt en bus eller et tog, som forlader station p ved tidspunkt t og ankommer til næste station p' ved tidspunkt t'. Denne forbindelse opfattes som kant fra knude (p,t) til knude (p',t'). Desuden tilføjer vi såkaldte *ventekanter* mellem på hinanden følgende afgangs- og ankomsttider ved hver station p; fx for tidspunkter t og t' med $t < t'$ som kant fra (p,t) til (p,t').

(a) Vis, at den resulterende graf er rettet og acyklisk.
(b) En ekstra knude skal bruge til at fastlægge den rejsendes udgangspunkt som sted og tid. Der skal også bruges en kant, som forbinder denne startknude med transportknetværket. Hvad er målknuden for denne kant?
(c) Antag, at vi har fundet et træ af korteste veje fra startknuden til alle knuder i transportnetværket. Hvilken er nu den rute, som den rejsende er interesseret i?

10.3 Ikke-negative kantomkostninger (Dijkstras algoritme)

I dette afsnit går vi ud fra, at der ikke forekommer kanter med negativ omkostning. Det medfører direkte, at der ikke forekommer negative kredse, og at der derfor findes en korteste vej til hver knude, som kan nås fra s. Vi skal vise, at man kan vælge slækningsrækkefølgen på en snedig måde, så hver kant bare slækkes en gang.

Men hvad er den rigtige rækkefølge? Langs en korteste vej vokser korteste-veje-afstandene – eller rettere, de bliver ikke mindre. Det leder til idéen at bearbejde knuderne i rækkefølgen af voksende afstande. (Vi siger, at vi *bearbejder* en knude, når vi slækker alle dens udgående kanter.) Lemma 10.3 siger os, at denne slækningsrækkefølge medfører, at de korteste veje bliver beregnede, i det mindste, hvis alle kanter har positiv længde. I algoritmen kender vi selvfølgeligt ikke korteste-veje-afstandene (endnu), men kun de *skønnede* afstande $d[v]$. Heldigvis stemmer for en hidtil ubearbejdede knuder med minimalt afstandsskøn den sande afstand og den skønnede afstand overens. Det skal vi vise i sætning 10.2. Disse overvejelser munder ud i algoritmen i figur. 10.4, som er kendt som »Dijkstras algoritme for korteste veje«. I figur 10.5 vises et eksempel på en kørsel.

I grunden er Dijkstras algoritme ikke andet end tråd-og-knude-algoritmen fra kapitlets indledning, i det mindste for urettede grafer. Hvis vi lægger alle tråde på et bord og langsom lyfter startknuden i vejret, slipper de andre knuder bordpladen i samme rækkefølge som deres korteste-veje-afstand.

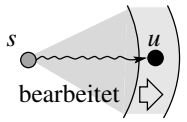
Sætning 10.2. *Dijkstras algoritme løser korteste-veje-problemet med samme udgangspunkt for grafer med ikke-negative kantomkostninger.*

Bevis. Beviset består af to dele. I første del skal vi vise, at alle knuder, der kan nås fra s, bearbejdes før eller senere. I anden del skal vi vise, at en knudes afstandsskøn er lig med dens korteste-veje-avstand på det tidspunkt, hvor den bearbejdes.

En knude v kaldes *opdaget*, når dens afstandsskøn tildeles en endelig værdi, dvs. $d[v] < +\infty$. Det ses direkte fra den ydre løkke, at algoritmen før eller siden behandler hver knude v, der er blevet opdaget.

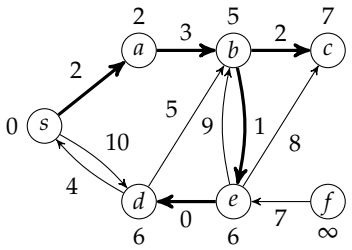
Dijkstras algoritme
Alle knuder er »ubearbejdede«; initialiser d und *forælder*
så længe der findes en ubearbejdet knude med afstandsskøn $< +\infty$ **udfør**

u:= en ubearbejdet knude minimalt afstandsskøn
slæk alle us udgående kanter (u, v);
hermed er u »bearbejdet«



Figur 10.4. Dijkstras algoritme: Korteste veje fra samme udgangspunkt i grafer med ikke-negative kantvægte

Operation	Kø
tilføj(s)	$\langle (s,0) \rangle$
fjernMindste$\rightsquigarrow (s,0)$	$\langle\,\rangle$
slæk $s \xrightarrow{2} a$	$\langle (a,2) \rangle$
slæk $s \xrightarrow{10} d$	$\langle (a,2), (d,10) \rangle$
fjernMindste$\rightsquigarrow (a,2)$	$\langle (d,10) \rangle$
slæk $a \xrightarrow{3} b$	$\langle (b,5), (d,10) \rangle$
fjernMindste$\rightsquigarrow (b,5)$	$\langle (d,10) \rangle$
slæk $b \xrightarrow{2} c$	$\langle (c,7), (d,10) \rangle$
slæk $b \xrightarrow{1} e$	$\langle (e,6), (c,7), (d,10) \rangle$
fjernMindste$\rightsquigarrow (e,6)$	$\langle (c,7), (d,10) \rangle$
slæk $e \xrightarrow{9} b$	$\langle (c,7), (d,10) \rangle$
slæk $e \xrightarrow{8} c$	$\langle (c,7), (d,10) \rangle$
slæk $e \xrightarrow{0} d$	$\langle (d,6), (c,7) \rangle$
fjernMindste$\rightsquigarrow (d,6)$	$\langle (c,7) \rangle$
slæk $d \xrightarrow{4} s$	$\langle (c,7) \rangle$
slæk $d \xrightarrow{5} b$	$\langle (c,7) \rangle$
fjernMindste$\rightsquigarrow (c,7)$	$\langle\,\rangle$



Figur 10.5. Kørsel af Dijkstras algoritme på grafen *til højre*. De fede kanter danner et korteste-veje-træ, tallene ved hver knude angiver deres korteste-veje-afstand. Tabellen *til venstre* viser de udførte skridt. Køen indeholder en tupel $(v, d[v])$ for hver opdagede knude v, som endnu ikke er bearbejdet. Vi siger at en knude er *opdaget*, hvis dens afstandsskøn are mindre end $+\infty$. I begyndelsen er s den eneste opdagede knude; den er ikke bearbejdet. Algoritmens skridt står i første spalte, køens indhold efter det pågældende skridt står i anden spalte.

Første del: Vi har netop indset, at det er nok at vise, at hver knude v, der kan nås fra s bliver opdaget før eller siden. Betragt hertil en vej $p = \langle v_1, \ldots, v_k \rangle$ med $s = v_1$ og $v = v_k$. Vi skal vise ved induktion efter i, at v_i bliver opdaget. Knude v_1 bliver opdaget, idet intialiseringen sætter $d[s] = 0 < +\infty$. Betragt nu $i > 1$. Ifølge induktionshypotesen bliver v_{i-1} opdatet, hvilket betyder $d[v_{i-1}] < +\infty$. Før eller siden bliver v_{i-1} bearbejdet, hvilket fører til kaldet *slæk*(v_{i-1}, v_i). Herefter (hvis ikke før) gælder $d[v_i] \leq d[v_{i-1}] + c(v_{i-1}, v_i) < +\infty$, og v_i er opdaget.

Anden del: Vi viser følgende påstand ved induktion efter antal løkkegennemløb (»runder«): Hvis knude u behandles i runde t, så gælder $d[u] \leq \mu[u]$. (Læg mærke til, at den modsatte ulighed $d[u] \geq \mu[u]$ altid gælder ifølge lemma 10.2.) Påstanden gælder for $t = 1$, fordi runde 1 bearbejder startknuden s, og vi har $d[s] = 0 = \mu(s)$. Betragt nu runde $t > 1$ og lad u med $u \neq s$ være den knude, som behandles i denne runde. Vælg en vilkårlig vej $p = \langle v_1, \ldots, v_k \rangle$ med $s = v_1$ og $u = v_k$. Lad v_i for $i \in \{2, \ldots, k\}$ være den første knuden på vejen, som *ikke* bearbejdes før runde t. Dette er veldefinieret, idet v_k har denne egenskab. Da bliver v_{i-1} bearbejdet før runde t. I den forbindelse udføres kaldet *slæk*(v_{i-1}, v_i). Derfor gælder, at

$$d[v_i] \leq d[v_{i-1}] + c(v_{i-1}, v_i) \leq \mu(v_{i-1}) + c(v_{i-1}, v_i)\,.$$

(Første ulighed etableres af operationen *slæk*(v_{i-1}, v_i); anden ulighed er netop induktionsantagelsen.) Vejen $\langle v_1, \ldots, v_{i-1}\rangle$ fra s til v_{i-1} har omkostning mindst $\mu(v_{i-1})$, så der gælder

$$\mu(v_{i-1}) + c(v_{i-1}, v_i) \leq c(\langle v_1, \ldots, v_{i-1}\rangle) + c(v_{i-1}, v_i) = c(\langle v_1, \ldots, v_i\rangle)\,.$$

Idet $\langle v_1, \ldots, v_i\rangle$ udgør en delvej af p, og kantomkostningerne ikke er negative, gælder desuden $c(\langle v_1, \ldots, v_i\rangle) \leq c(p)$. Ved at sammenholde de sidste tre uligheder får vi, at der efter operationen *slæk*(v_{i-1}, v_i) gælder $d[v_i] \leq c(p)$. Selv hvis $d[v_i]$ herefter skulle aftage, gælder uligheden også i begyndelsen af runde t. Nu vælger algoritmen i runde t knuden u som knuden med minimalt afstandsskøn til bearbejdning, så der må gælde $d[u] \leq d[v_i]$, hvilket medfører $d[u] \leq c(p)$.

Da p var valgt vilkårlig, gælder $d[u] \leq \mu(u)$, og vi har etableret induktionshypotesen for t. □

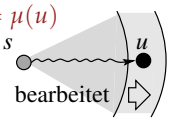
Opgave 10.7. Lad $v_1, v_2, \ldots$ være rækkefølgen, som knuderne bearbejdes i. Vis, at der gælder $\mu(v_1) \leq \mu(v_2) \leq \ldots$. Med andre ord bearbejdes knuderne i rækkefølge af (svagt) voksende korteste-veje-afstand.

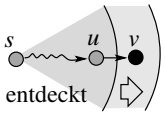
Opgave 10.8 (Kontrol af korteste-veje-afstand)**.** Antag, at alle kantomkostninger er positive, og at alle knuder kan nås fra s. Lad d være en knuderække med ikke-negative reelle indgange, som opfylder $d[s] = 0$ og $d[v] = \min_{(u,v)\in E} d[u] + c(u, v)$ for alle $v \neq s$. Vis, at der for alle v gælder $d[v] = \mu(v)$. Er denne påstand stadig sand, når der der er kanter med omkostning 0?

Vi skal nu se nærmere på, hvordan Dijkstras algoritme kan implementeres. Alle ubearbejdede knuder stilles i en addresserbar prioritetskø (se afsnit **??**), med afstandsskønnet som nøgle. Operationen *fjernMindste* leverer så altid den knude, som skal bearbejdes hernest. Til adressering af indgange for *sænkNøgle*-operation bruger vi her bare knudenummeret i stedet for greb. Med udgangspunkt i den sædvanlig prioritetskø kan man implementere sådan en datastruktur, so vi kalder *KnudePK*, med en yderligere knuderække, som oversætter knudenumre til greb. Hvis prioritetskøen er implementeret ved hjælp af en træsstruktur, som fx ved fibonaccihobe eller parhobe, kan man gemme indgange af *KnudePK*, dvs. træknuderne, direkte i en knuderække. Kaldet *sænkNøgle*(v) opdaterer *KnudePK*-køen under antagelse af, at nøglen $d[v]$ blev mindsket umiddelbart inden.

Figur 10.6 viser en detaljeret fremstilling af algoritmen. Vi vil nu analysere algoritmens udførelsestid med udgangspunkt i tidsforbruget i det enkelte prioritetskøoperationer. Initialiseringen af rækkerne d og *forælder* samt prioritetskøen $Q = \{s\}$ koster tid $O(n)$. Ved hver iteration af den ydre løkke kræver betingelsen $Q = \emptyset$ konstant tid, dvs. total tid $O(n)$. Hver knude s, som kan nås fra s, bliver fjernet fra prioritetskøen præcis én gang. Bortset fra s selv, bliver hver af disse køer også stillet i prioritetskøen præcis én gang. Derfor forekommer der totalt højst n kald af *fjernMindste* og *tilføj* til Q. Idet hver

```
Funktion Dijkstra(s: KnudeId): Knuderække×Knuderække // returnerer (d, forælder)
  d = ⟨∞,...,∞⟩ : Knuderække af R ∪ {∞} // afstandsskøn
  forælder = ⟨⊥,...,⊥⟩ : Knuderække af KnudeId
  forælder[s]:=s // løkke signaliserer roden
  Q: KnudePK // kø af ubearbejdede opdagede knuder
  d[s]:=0;  Q.tilføj(s)
  så længe Q ≠ ∅ udfør
    u:=Q.fjernMindste // der gælder d[u] = μ(u)
    for hvert kant e = (u,v) ∈ E udfør
      hvis d[u] + c(e) < d[v] så // slæk
        d[v]:=d[u] + c(e)
        forælder[v]:=u // opdater træ
        hvis v ∈ Q så Q.sænkNøgle(v)
        ellers Q.tilføj(v)
  returner (d, forælder)
```





Figur 10.6. Pseudokode for Dijkstras algoritme

knude bearbejdes højst én gang, slækkes hver kant højst én gang, og derfor kan der forekomme højst m kald til *sænkMindste*. Den samlede udførelsestid for Dijkstras algoritme er altså

$$T_{Dijkstra} = O(m \cdot T_{sænkNøgle}(n) + n \cdot (T_{fjernMindste}(n) + T_{tilføj}(n)))\,,$$

hvor $T_{fjernMindste}$, $T_{tilføj}$ og $T_{sænkNøgle}$ angiver udførelsestiderne for *fjernMindste*, *tilføj* hhv. *sænkMindste*. Læg mærke til, at disse udførelsestider afhænger af størrelsen $|Q| = O(n)$ af køen.

Opgave 10.9. Givet n, m med $n \leq m \leq n(n-1)/2$. Konstruer en graf G med n knuder og m kanter og velvalgte kantvægte, sådan at $m - (n-1)$ af kantslækningerne i løbet af Dijkstras algoritme udløser et kald af *sænkNøgle*.

I originalpublikationen fra 1959 foreslog Dijkstra følgende implementation af prioritetskøen: Hold rede på antallet af opdagede, ubearbejdede knuder samt to rækker, , hver indiceret med knudemængden som indeks – en række d til afstandsskønnene og en række, som for hver knude angiver, om den er enten uopdaget, opdaget men ubearbejdet eller bearbejdet. Så koster *tilføj* og *formindskNøgle* tid $O(1)$. Operationen *fjernMindste* koster $O(n)$, fordi hele rækken skal gennemgås for at finde en opdaget ubearbejdet knude med minimalt afstandsskøn. Den samlede tid bliver altså

$$T_{Dijkstra59} = O(m + n^2)\,.$$

Siden Dijkstras artikel har man fundet på meget bedre prioritetskøer. Hvis vi bruger en binær hob eller en fibononaccihob som beskrevet i afsnit **??**, får vi

$$T_{DijkstraBHeap} = O((m+n)\log n)$$

hhv.

$$T_{DijkstraFibonacci} = O(m + n\log n)\,.$$

Ifølge den asympototiske analyse er implementationen med fibonaccihob bedre, medmindre grafen er tynd, med $m = O(n)$. I praksis giver fibonaccihobe ikke nødvendigvis den hurtigste implementation, både fordi analysen skuler ganske store konstante faktorer og der normalt forekommer langt færre *sænkNøgle*-operationer, end værstefaldsanalysen forudser. Vi skal underbygge disse eksperimentelle observationer med en teoretisk analyse i næste afsnit.

10.4 Gennemsnitsanalyse af Dijkstras algoritme

Udeladt

10.5 Monotone prioritetskøer over heltal

Udeladt

10.6 Vilkårlige kantomkostninger (Bellman og Fords algoritme)

For acykliske grafer med ikke-negative kantomkostninger kunne vi nøjes med m kantslækninger. For vilkårlige grafer, dvs. grafer som kan indeholde kredse, findes der ikke noget tilsvarende resultat. Men man kan hurtigt overbevise sig om, at $O(mn)$ kantslækninger er tilstrækkelige for at etablere korrekthedskriteriet i lemma 10.3: Algoritmen af Bellman og Ford [4, 16], som er vist i figur 10.7, kører i $n-1$ runder. I hver runde slækkes hver kant én gang. Idet en simpel vej består af højst $n-1$ kanter, er hver korteste vej en delfølge af disse slækninger. Ifølge lemma 10.1 og lemma 10.3 gælder der efter den sidste runde derfor $d[v] = \mu(v)$ for hver knude v med $-\infty < \mu(v) < \infty$. Desuden beskriver rækken *forælder* korteste veje fra s til disse knuder. For en knude v, som ikke kan nås fra s, gælder $d[v] = \infty$, som ønsket.

På den anden side er det mindre klart, hvordan man identificerer de knuder v, som opfylder $\mu(v) = -\infty$. Ifølge øvelsesopgave 10.1 er det netop de knuder, som kan nås fra s med en vej, som indholder en kreds med negativ længde. Følgende to egenskaber ved negative kredse er centrale:

Lemma 10.4. *Lad $C = \langle v_0, \ldots, v_k\rangle$ med $v_0 = v_k$ være en kreds med negativ længde i grafen G, som kan nås fra s. Da eksisterer $i \in \{1,\ldots,k\}$, således at $d[v_{i-1}] + c((v_{i-1}, v_i)) < d[v_i]$.*

```
Funktion BellmanFord(s: KnudeId): Knuderække × Knuderække
    d = ⟨∞,...,∞⟩ : Knuderække af R ∪ {−∞,∞} // Afstand fra startknuden
    forælder = ⟨⊥,...,⊥⟩ : Knuderække af KnudeId
    d[s]:=0;  forælder[s]:=s // Løkke angiver roden
    for i:=1 til n − 1 udfør
        for hvert e ∈ E udfør slæk(e) // Runde i
    for hvert e = (u,v) ∈ E udfør // Efterbehandling
        hvis d[u] + c(e) < d[v] så infect(v)
    returner (d,forælder)

Procedure smit(v)
    hvis d[v] > −∞ så
        d[v]:= − ∞
        for hvert (v,w) ∈ E udfør smit(w)
```

Figur 10.7. Bellman og Fords algoritme for korteste veje i vilkårlige grafer

Bevis. Idet C kan nås fra s, gælder $d[v_i] < +\infty$ for alle $i \in \{1,\ldots,k\}$. Antag modsætningsvist, at der for alle disse i gælder $d[v_{i-1}] + c((v_{i-1},v_i)) \geq d[v_i]$. Når man lægger disse k uligheder sammen, går $d[v_i]$-termerne ud mod hinanden, og vi får uligheden $\sum_{i=1}^{k} c((v_{i-1},v_i)) \geq 0$. Men så er længden af C ikke negativ, i modstrid med antagelsen. □

Lemma 10.5. *Lad v være en knude, som kan nås fra s. Da gælder $mu(v) = -\infty$ hvis og kun hvis der eksisterer en kant $e = (u,w)$, således at $d[u] + c(e) < d[w] < +\infty$, og v kan nås fra w.*

Bevis. Antag først $\mu(v) = -\infty$. Da kan v nås fra s via en kreds $C = \langle v_0,\ldots,v_k\rangle$ med negativ længde. Ifølge lemmaet foroven indeholder C en kant $e = (v_{i-1},v_i)$, som opfylder $d[v_{i-1}] + c((v_{i-1},v_i)) < d[v_i]$. Desuden kan v nås fra v_i. Antag omvendt, at $e = (u,w)$ er en kant med $d[u] + c(e) < d[w] < +\infty$. Vi skal først indse, at

$$\mu(w) = -\infty\,. \tag{10.1}$$

Antag modsætningsvis, at $\mu(w) > -\infty$. Idet w klart kan nås fra u, må der også gælde $\mu(u) > -\infty$. Vi har altså $-\infty < \mu(u) < +\infty$, og kan bruge lemma 10.3 til at indse, at $\mu(u) = d[u]$ og $\mu(w) = d[w]$. Men uligheden $\mu(u) + c(e) < \mu(w)$ absurd, så (10.1) gælder. Med andre ord kann w nås fra s via en kreds af negativ længde. Via w kan også v nås på samme måde, så $\mu(v) = -\infty$. □

Disse overvejelser fører til følgende måde at identificere knuder v med $\mu(v) = \infty$ ved at sætte $d[v] = -\infty$. I efterbehandlingen prøver for hver kant $e = (u,v)$, om $d[u] + c(e) < d[v]$. I så fald kalder vi v for »smittet«, sætter $d[v]:= -\infty$ og kalder den rekursive procedure $smit(v)$, som gennemfører en slags bredde først-søgning for at sikre, at $d[w] = \infty$ for alle knuder w, som kan nås fra v.

Opgave 10.10. Vis, at de knnuder, for hvilke kaldes $smit(v)$ udføres en eller flere gange præcis er de knnuder, for hvilke $\mu(v) = -\infty$, og at hele efterbehandlingen tager tid $O(m)$. *Vink*: Genbrug argumenterne fra dybde førstsøgning.

Opgave 10.11. Nogen foreslår en alternativ efterbehandling: Sæt $d[v]$ til $-\infty$ for alle knuder v med $d[v] < +\infty$, for hvilke der ikke findes en vej af forgængerpegere *forælder* tilbage til startknuden s. Vis et eksempel, hvor denne metode kan overse en knude v med $\mu(v) = -\infty$.

Opgave 10.12 (Arbitrage)**.** Betrage en mængde C af valutaer med vekselkurs r_{ij} mellem valuta i og j, dvs. at man får r_{ij} enheder af valuta j for en enhed af valuta i. En *valutaarbitrage* opstår, når der findes en følge af elementære vekslinger (*transaktioner*), som begynder med en enhed i en valuta og ender med mere end en enhed i samme valuta.

(a) Forklar, hvordan man kan afgøre, om en tabel af valutakurser tillader valutaarbitrage. *Vink*: $\log(xy) = \log x + \log y$.
(b) Udvid algoritmen til at beregne en følge af transaktioner, som maksimerer den gennemsnitlige gevinst *per transaktion*!

I afsnit 10.10 beskriver vi yderligere forbedringer af Bellman og Fords algoritme, som gør den mere anvendlig i praksis.

10.7 Korteste veje mellem alle par af knuder

Udeladt

10.8 Korteste-veje-opslag

Udeladt

10.9 Implementationsaspekter

Udeladt

10.10 Historiske bemærknigner og videre resultater

Udeladt

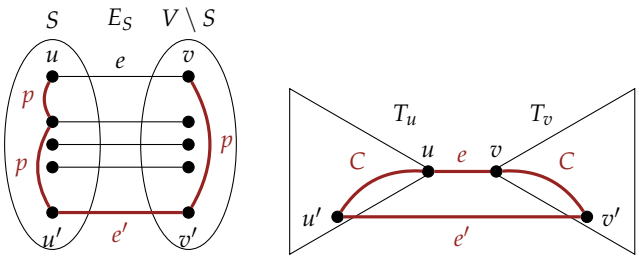
11

Minimale spændetræer

Det sydfynske øhav skal omsider gøres landfast. Den totale udgift for alle de nødvendige broer skal gerne holdes så lavt som muligt. Direkte forbindelser mellem øerne er ikke påkrævede, bare hver ø kan nås fra hver af de andre. Du har en liste over mulige broforbindelser mellem par af øer og deres forventede omkostning. Hvilke projekter sætter du i værk?

Mere generelt ønsker vi at løse følgende problem. Givet en sammenhængende urettet graf $G = (V, E)$ med positive reelle kantvægte $c\colon E \to \mathbf{R}_+$. Et *minimalt spændetræ* (*mst*) af G er en mængde $T \subseteq E$ af kanter med egenskaben, at grafen (V, T) er et træ med mindst mulig total vægt $c(T) := \sum_{e \in T} c(e)$. Som i sidste kapitel bruger vi begreberne »vægt« og »omkostning« i flæng. I vores eksempel repræsenterer hver knude en ø, kanterne er de mulige broprojekter og deres vægt er omkostningen ved at bygge den tilsvarende bro. I hele kapitlet er G både sammenhængende og urettet.

Minimale spændetræer er måske den enkleste udgave af en vigtig familie af grafproblemer, som opstår i forbindelse med *netværksstruktur*. Minimale spændetræer er et så grundlæggende koncept, at de dukker op i mange andre sammenhænge, fx klyngedannelse, undvigelse af flaskehalse ved ruteplanlægning og approksimationsalgoritmer for beregningsmæssigt svære problemer. Det ser vi nærmere på i afsnit 11.5 og 11.7. En anden god grund til at betragte minimale spændetræer er, at at dette problem tillader algoritmer, som er enkle, elegante og hurtige. I afsnit 11.1 skal vi udlede to enkle men grundlæggende egenskaber ved mst'er, som danner fundamentet for de fleste mst-algoritmer. Jarník og Prims algoritme går ud fra en enkelt startknude og bygger et mst ved gentagen udvidelse af et enkelt træ, den beskrives i afsnit 11.2. Kruskals algoritme bygger samtidigt flere træer i dele af grafen, som ikke umiddelbart har noget med hinanden at gøre, og smelter dem sammen til stadigt større træer. Denne algoritme ser vi nærmere på i afsnit 11.3. For en effektiv implementation af Kruskals algoritme har man brug for en datastruktur, som opretholder en klassedeling af en mængde af elementer



Figur 11.1. Snit- og kredsegenskaberne. Billedet *til venstre* illustrerer beviset for snitegenskaben. Kanten e har minimal vægt blandt kanterne i E_S, dvs. kanterne, som har et endepunkt S og et endepunkt i $V \setminus S$. Vejen p er den entydigt bestemte vej i mst'et, som forbinder es endpunkter u og v. Vejen p må indeholde en kant $e' \in E_S$. Billedet *til højre* illustrerer beviset for kredsegenskaben. Her er C en kreds i G, kanten $e = \{u, v\}$ er en kant på C med maksimal vægt og T er et mst, som indeholder e. Komponenterne af $T \setminus \{e\}$ er T_u og T_v, kanten e' ligger på kredsen C og forbinder T_u og T_v.

(dvs. en opdeling i disjunkte delmængder) under to operationer: »afgør, om to elementer tilhører samme delmængde« og »forén to delmængder«. Forén og find-datastrukturen beskrives i afsnit 11.4. Den har mange andre anvendelser ved siden af minimale spændetræer.

Opgave 11.1. Hvis den givne graf G ikke er sammenhængende, kan man spørge efter en *minimal spændeskov* – en kantmængde, som beskriver et mst for hver sammenhængskomponent af G. Beskriv en metode til at finde minimale spændeskove med en vilkårlig mst-algoritme, men *uden* at begynde med at finde sammenhængskomponenterne i G. *Vink*: Tilføj $n - 1$ ekstra kanter.

Opgave 11.2 (Opspændende kantmængder)**.** En kantmængde $T \subseteq E$ i en sammenhængende graf $G = (V, E)$ er *opspændende*, hvis (V, T) er sammenhængende. Er en opspændende kantmængde med minimal vægt altid et træ? Er den altid et træ, dersom kantvægtene er positive?

Opgave 11.3. Reducér problemet at finde et spændetræ med *maksimal* total vægt til mst-problemet.

11.1 Snit- og kredsegenskaberne

Vi skal etablere to enkle lemmaer. Det ene tillader os at medtage kanter til et voksende mst, det andet tillader os at udelukke kanter fra at komme i betragtning. Vi har brug for begrebet grafsnit. Et *snit* i en sammenhængende graf G er en klassedeling $(S, V \setminus S)$ af knudemængden V i to ikke-tomme dele S og $V \setminus S$. Snittet definerer en mængde $E_S = \{\, \{u, v\} \in E \colon u \in S, v \in V \setminus S \,\}$ af *snitkanter*, som går mellem S og $V \setminus S$. Beviserne for de næste to lemmaer er illustrerede i figur 11.1.

Lemma 11.1 (Snitegenskab). *Lad $(S, V \setminus S)$ være et snit med snitkantsmængde E_S og lad $e \in E_S$ være en kant med minimal vægt i E_S. Antag, at T' er en kantmængde, som er indeholdt i et mst og ikke indeholder nogen kant fra E_S. Da er også $T' \cup \{e\}$ indeholdt i et mst.*

Bevis. Betragt et mst T i G med $T' \subseteq T$. Lad es endepunkter være $u \in S$ og $v \in V \setminus S$. Idet T er et spændetræ, eksisterer der i T en (entydigt bestemt) vej p fra u til v. Denne vej må indeholde en kant $e' = \{u', v'\}$ med $u' \in S$ og $v' \in V \setminus S$. (Det er ikke udelukket, at $e = e'$.) Ifølge antagelsen gælder $e' \notin T'$. Betragt nu kantmængden T'' givet ved $T'' = (T \setminus \{e'\}) \cup \{e\}$. Denne danner ligeledes et spændetræ, idet fjernelsen af e' fra T danner to deltræer, som af e igen samles til ét træ. Idet $c(e)$ er minimal i E_S, har vi $c(e) \leq c(e')$, og derfor $c(T'') \leq c(T)$. Spændetræet T'' er altså minimalt og derfor ligeledes et mst. Det er klart, at $T' \cup \{e\} \subseteq T''$. □

Lemma 11.2 (Kredsegenskab). *Lad C være en simpel kreds i G, og lad e være en kant på C med maksimal vægt. Betragt grafen $G' = (V, E \setminus \{e\})$. Hvert mst i G' er også et mst i G.*

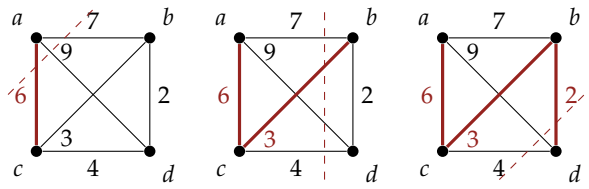
Bevis. Betragt et mst T i G. Hvis e ikke tilhører T, så har alle mst'er i G' samme vægt som T. Antag altså $e = \{u, v\} \in T$. Hvis vi fjerner e fra T, opstår to deltræer T_u og T_v med u i T_u og v i T_v. Idet C er en kreds, eksisterer en kant $e' = \{u', v'\}$ med $e' \neq e$ på C med u' i T_u og v' i T_v. Mængden T' givet ved $T' = (T \setminus \{e\}) \cup \{e'\}$ er et spændetræ i G', hvis vægt $c(T') = c(T) - c(e) + c(e')$ ikke er større end $c(T)$. Derfor har alle mst'er i G' samme vægt som T. □

Snitegenskaben leder til en enkel og generel grådig algoritme for mst-konstruktion. Begynd med en tom kantmængde $T' = \emptyset$. Så længe T' ikke er sammenhængende, gentager man følgende skridt: Vælg et snit $(S, V \setminus S)$ under den forudsætning, at ingen kant fra T' forbinder begge sider af snittet, og udvid T' med en snitkant af minimal vægt. Når (V, T') er blevet sammenhængende, er T' et mst.

Forskellige metoder til valg af snittet i denne generelle tilgang leder til forskellige veldefinerede algoritmer. I de følgende afsnit diskuterer vi to tilgange i detaljer; i afsnit 11.7 skitseres en tredje. Desuden skal vi gøre os klart, hvordan man finder en kant af minimal vægt på snittet.

Kredsegenskaben fører ligeledes til en enkel strategi for at finde et mst. Begynd med $E' = E$. Gentag følgende skridt så længe E' ikke er kredsfri: Find en kreds C i E' og fjern fra E' en kant fra C med maksimal vægt. Når E' er blevet kredsfri, er E' et mst. I modsætning til snitegenskaben er det dog ikke lykkedes nogen at konkretisere denne tilgang til en effektiv mst-algoritme, så vi skal ikke forfølge denne idé videre.

Opgave 11.4. Vis, at en graf har præcis ét mst, hvis alle kantvægte er forskellige. Vis endvidere, at mst'et i så fald er det samme som for grafen, hvor hver kantvægt er erstattet med kantens rangordning blandt kantvægtene.



Figur 11.2. Tre snit (vist som stiplede linjer) svarende til skridtene i Jarník og Prims algoritme med a som den første knude. Kanterne, som føjes til mst'et, er $\{a, c\}$, $\{c, b\}$ og $\{b, d\}$.

11.2 Jarník og Prims algoritme

Jarník og Prims algoritme (kort »JP-algoritmen«) [28, 44, 14] for minimale spændetræer ligner meget Dijkstras algoritme for korteste veje.[1] JP-algoritmen begynder med en (vilkårlig) startknude s, opfattet som træ bestående af en enkelt knude og tom kantmængde. Nu opbygges et mst ved at tilføje en knude ad gangen. Lad $S \neq V$ være mængden af knuder, som hidtil er føjet til træet. Da danner $(S, V \setminus S)$ et snit. Vælg en kant $e = \{v, u\}$ med $v \in S$ og $u \in V \setminus S$ og minimal vægt blandt disse kanter og føj u til S og e til træets kantmængde. Udfordringen er at bestemme kanten e på en effektiv måde. Algoritmen gemmer hertil i en prioritetskø for hver knude $w \in V \setminus S$ en hosliggende kant $\{v, w\}$ med minimal vægt, som forbinder S med w. Kantens vægt skrives i rækkeindgangen $d[w]$; knuden v skrives i rækkeindgangen $\mathit{forælder}[w]$. En indgang u i Q med minimal d-værdi leverer nu den ønskede kant som $\{\mathit{forælder}[u], u\}$. Når knuden u føjes til S, undersøges de hos v liggende kanter for at afgøre, om de danner bedre forbindelser til knuderne i $V \setminus S$. I figur 11.2 vises JP-algoritmens fremgangsmåde ved hjælp af et eksempel; i figur 11.3 vises pseudokoden. Når knuden u føjes til S, og en hos u liggende kant $\{u, w\}$ bliver undersøgt, skal man kunne afgøre, om w tilhører S eller ej. Man kunne gemme denne information i en knudeindiceret række af booleske værdier. Hvis alle kantvægte er positive, kan man i stedet bruge d-rækken: For knude v gælder $v \in S$ hvis $d[v] = 0$, og $v \notin S$ hvis $d[v] > 0$.

Dette trick sparer lagerplads og en sammenligning i den indre løkke. Læg mærke til, at $c(e) < d[w]$ kun kan gælde, hvis $d[w] > 0$ og dermed $w \notin S$ gælder, og e udgør en bedre forbindelse fra w til S. Betingelsen »**hvis** $w \in Q$« prøver man nemmest ved at undersøge, om den gamle værdi af $d[w]$ (som man skal huske at gemme, inden man overskriver den) er lig med ∞. Den eneste væsentlige forskel til Dijkstras algoritme ligger i, at nøglerne i prioritetskøen er kantvægte i stedet for vejlængder. Analysen for Dijkstras

[1] Dijkstra publicerede algoritmen i sin grundlæggende 1959-artikel [14] om korteste veje. Idet Prim havde publiceret samme algoritme to år tidligere, kaldes den ofte for »Prims algoritme«. Algoritmen optræder dog meget tidligere i en artikel af Jarník fra 1930 [28].

algoritme overføres direkte til JP-algoritmen, så vi kan opnå tid $O(n \log n + m)$ ved brug af fibonaccihobe.

Opgave 11.5. Dijkstras Algoritmen algoritme for korteste veje kan nøjes med en monoton prioritetskø. Vis, at det *ikke* er nok for JP-algoritmen.

Opgave 11.6 (Gennemsnitsanalyse af JP-algoritmen)**.** Antag, at Gs kanter er valgt tilfældigt fra $\{1, \ldots, m\}$. Vis, at det forventede antal *sænkNøgle*-operationer, som udføres i løbet af JP-algoritmen, er begrænset af $O(n \log(m/n))$. *Vink*: Den efterspurgte analyse minder meget om analysen af Dijkstras algoritme i sætning **??**.

11.3 Kruskals algoritme

Jarník og Prims algoritme er nok den bedste generelle fremgangsmåde for mst-problemet. Alligevel vil vi præsentere et alternativ, nemlig Kruskals algoritme [33], som også har visse fordele. Især kan den undvære en raffineret repræsentation af grafen – Kruskals algoritme kan bruges, selv når grafen er givet som kantfølge. Udførelsestiden for tynde grafer med $m = O(n)$ tåler sammenligning med JP-algoritmen.

Pseudokoden i figur 11.4 er yderst kompakt. Algoritmen betragter Gs kanter i rækkefølge af voksende vægt og vedligeholder en kredsfri kantdelmængde T, dvs. en skov, som er tom fra starten. Algoritmen opretholder invarianten, at T kan udvides til et mst. Når en kant e bliver betragtet, bliver den enten medtaget i mængden T eller forkastet. Beslutningen træffes på grundlag af invarianten og snitegenskaben. Ifølge invarianten ved vi, at T

```
Funktion JPmst : Mængde af Kant
  d = ⟨∞,...,∞⟩ : Knuderække[1..n] af R ∪ {∞}  // d[u] indeholder mindste vægt
  // af kant mellem S og u
  forælder: Knuderække af KnudeId // {forælder[u], u} er letteste kant mellem S og u
  Q: KnudePK // bruger d[·] som prioritet
  Q.tilføj(s) for vilkårligt s ∈ V
  så længe Q ≠ ∅ udfør
    u:=Q.fjernMindste
    d[u]:=0 // d[u] = 0 betyder u ∈ S
    for hvert kant e = {u, w} ∈ E udfør
      hvis c(e) < d[w] så // c(e) < d[w] medfører d[w] > 0 og dermed w ∉ S
        d[w]:=c(e)
        forælder[w]:=u
        hvis w ∈ Q så Q.sænkNøgle(w) ellers Q.tilføj(w)
    invariant ∀w ∈ Q: d[w] = min { c({v, w}) : {v, w} ∈ E ∧ v ∈ S }
  returner { {forælder[v], v} : v ∈ V \ {s} }
```

Figur 11.3. Jarník og Prims mst-algoritme. Vi antager, at kantvægtene er positive.

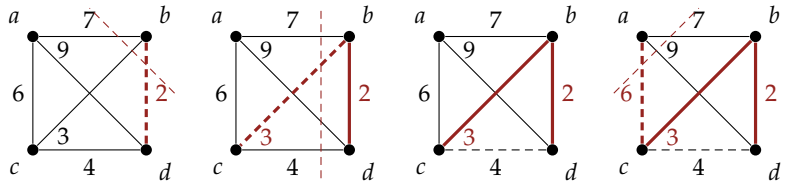
kan udvides til et mst. Der er to tilfælde. Hvis endepunkterne af e tilhører samme sammenhængskomponent af (V, T), forbliver T uforandret, så invarianten gælder stadig. Hvis endepunkterne u og v af e ligger i forskellige sammenhængskomponenter af (V, T), betragter vi snittet $(S_u, V \setminus S_u)$, hvor S_u er sammenhængskomponenten af u i (V, T). Ingen af Ts kanter krydser dette snit; kanten e forbinder S_u med $V \setminus S_u$ og er den letteste snitkant. (Det er fordi kanterne blev betragtet i rækkefølge af stigende vægt. Hvis der skulle være en snitkant e' med $c(e') < c(e)$ så havde den allerede været betragtet tidligere og medtaget i T.) Ifølge snitegenskaben er også $T \cup \{e\}$ indeholdt i et mst. Vi kan altså føje e til T og bevare invarianten. I figur 11.5 er forløbet af Kruskals algoritme vist ved eksempel.

Vi mangler at overbevise os om, at T til sidst selv er et mst. Ifølge invarianten er T indeholdt i et mst. Antag modsætningsvist, at (V, T) er usammenhængende. Så findes en kant $e \in E$, som forbinder to komponenter af (V, T). Men algoritmen må have betragtet denne kant undervejs i løbet, og må da have føjet den til T. Men så er de to komponenter sammenhængende, hvilket er absurd. Derfor er (V, T) sammenhængende til sidst og derfor et mst.

Implementationer af Kruskals algoritme har brug for at hurtigt kunne afgøre, om en givet kant forbinder to komponenter af (V, T). I næste afsnit skal vi se en så effektiv datastruktur for dette problem, af vi kan bortse fra dens udførelsestid i analysen. Den dominerende omkostning for Kruskals algoritme bliver den indeledende sortering af kantvægtene, som kan gøres i

Funktion *KruskalMst*(V, E, c): *Mængde* **af** *Kant*
  $T{:=}\emptyset$
  forhvert $\{u, v\} \in E$ i rækkefølge af stigende vægt **udfør**
    hvis u og v ligger i forskellige træer i T **så**
      $T{:=}T \cup \{\{u, v\}\}$ *//forén to træer*
    invariant T er en skov, som er indeholdt i et mst
  returner T

Figur 11.4. Kruskals mst-algoritme



Figur 11.5. Kruskals algoritme bruger i dette eksempel snitegenskaben to gange for at konstatere, at det findes et mst indeholdende kanterne $\{b, d\}$ og $\{b, c\}$. Hernæst forkastes kanten $\{c, d\}$, fordi dens to endepunkter ligger i samme træ, nemlig træet med knudemængde $\{b, c, d\}$. Til sidst medtages kanten $\{a, c\}$ i mst'et, baseret på snitegenskaben ved snittet $(\{a\}, \{b, c, d\})$.

tid $O(m \log m)$ med en effektiv sammenligningsbaseret sorteringsalgoritme. Den konstante faktor, som skjuler sig bag store-O-notationen er ganske lille, så for tynde grafer med $m = O(n)$ kan Kruskals algoritme slå JP-algoritmen og dens udførelsestid på $O(m + n \log n)$.

Opgave 11.7 (Mst for datastrømme)**.** Antag, at grafens kanter kun bliver vist en gang (fx over en netværksforbindelse), og at lageret er for lille til at gemme dem allesammen. Kanterne bliver *ikke* nødvendigvis vist i rækkefølge af voksende vægt.

(a) Skitser en algoritme, som under disse antagelser beregner et mst i plads $O(n)$.
(b) Forbedr algoritmen således, at den kører i tid $O(m \log n)$. *Vink*: Behandl kanterne i stakke af størrelse $O(n)$. (Et alternativ er at bruge datastrukturen *dynamisk træ* af Sleator og Tarjan [51].)

11.4 Forén og find

En *klassedeling* af en mængde M er en familie $M_1, \ldots, M_k$ af disjunkte delmængder af M, som sammen *dækker* M, dvs. at der gælder $M_i \cap M_j = \emptyset$ for alle $i \neq j$ og $M = M_1 \cup \cdots \cup M_k$. En delmængde M_i kaldes sommetider en *klasse* i klassedelingen. For eksempel leder skoven T i Kruskals algoritme til en klassedeling af knudemængden V. Klassedelingens klasser er sammenhængskomponenterne i (V, T). Der kan forekomme komponenter, som kun består af en enkelt knude. Især består klassedelingen i begyndelsen udelukkende af den slags etpunktsmængder. Kruskals algoritme udfører nu to operationer på denne klassedeling: Den afgør, om to elementer hører til samme delmængde (dvs. at knuderne hører til samme deltræ), og forener to delmængder til én (nemlig når der tilføjes en kant til T).

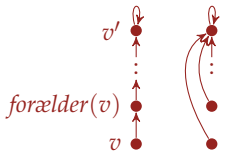
Forén og find-datastrukturen repræsenterer en klassedeling af mængden $\{1, \ldots, n\}$ på en måde, så begge operationer kan gennemføres effektivt. I hver klasse udnævnes et af elementerne til klassens *repræsentant*, som også bruges som klassens navn. Valget af repræsentant gøres af algoritmen, ikke af brugeren. I begyndelsen udgør hvert element $v \in \{1, \ldots, n\}$ en klasse for sig; repræsentanten for klassen $\{v\}$ er dermed v selv. Funktionen *find*(v) returnerer repræsentanten for den klasse, som indeholder v. For at afgøre, om to elementer tilhører samme klasse, bestemmer man deres repræsentanter og sammenligner dem. Operationen *forén*(r, s), anvendt på repræsentanter af forskellige[2] klasser, forener disse klasser til en ny og bestemmer en ny repræsentant.

For at implementere Kruskals algoritme ved hjælp af forén og find-datastrukturen, behøver man kun små ændringer til proceduren i fig. 11.4. I

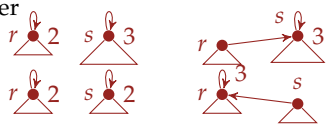
[2] I nogle fremstillinger kan *forén* bruges med vilkårlige elementer som argument. Så bliver operationen på repræsentanter typisk kaldt »*hægt*«.

Klasse *Forén og find*$(n\colon \mathbf{N})$
  forælder $= \langle 1,\ldots,n\rangle\colon$ *Række*$[1..n]$ **af** $[1..n]$
  rang $= \langle 0,\ldots,0\rangle\colon$ *Række*$[1..n]$ **af** $[0..\log n]$

Funktion *find*$(v\colon 1..n)\colon 1..n$
  hvis *forælder*$[v] = v$ **så returner** v
  ellers $v' :=$ *find*(*forælder*$[v]$)
    forælder$[v] := v'$
    returner v'



Procedure *forén*$(r,s\colon 1..n)$
  antag r,s repræsentanter af forskellige klasser
  hvis *rang*$[r] <$ *rang*$[s]$ **så** *forælder*$[r] := s$
  ellers *forælder*$[s] := r$
    hvis *rang*$[r] =$ *rang*$[s]$ **så** *rang*$[r]{+}{+}$



Figur 11.6. En effektiv forén og find-datastruktur til forvaltning af en klassedeling af mængden $\{1,\ldots,n\}$.

begyndelsen kaldes klassens konstruktør, som gør hver knude v til sin egen klasse. Vi behøver bare at erstatte **hvis** -kommandoen med følgende:

r := *find*(*u*); *s* := *find*(*v*);
hvis $r \neq s$ **så** $T := T \cup \{\{u,v\}\}$; *forén*(r,s);

Forén og find-datastrukturen kan realiseres på følgende måde. Hver klasse repræsenteres som et rodfæstet træ, hvor roden tjener som repræsentant. Forgængeren til elementet v i træet gemmes i indgangen *forælder*$[v]$ i rækken *forælder*$[1..n]$. For rodknuder r gælder *forælder*$[r] = r$, hvilket svarer til en løkke ved roden.

For at implementere operationen *find*(v) følger vi forælderpegere fra v, til vi når en løkke. Denne løkke sidder ved vs repræsentant. Implementationen af *forén*(u,v) er lige så nem. Vi gør den ene repræsentant til den andens forælder. Den sidstnævnte mister herved sin rolle som repræsentant; førstnævnte tjener som repræsentant for den nyskabte klasse. Den proces, vi har beskrevet indtil nu, leder til en realisering af forén og find-datastrukturen, som er korrekt, men ineffektiv. Ineffektiviten skyldes, at *forælder*-pegerne kan danne lange kæder, som skal gennemløbes gentagne gange for hver *find*-operation. I værste fald skal vi bruge lineær tid for hver *find*-operation.

På grund af dette problem er der foretaget to forbedringer i algoritmen i fig. 11.6. Den første forbedring forhindrer, at træerne bliver for dybe. Hertil gemmes for hver repræsentant et naturligt tal, kaldt repræsentantens *rang*. I begyndelsen, hvor hvert element er sin egen repræsentant, er rangen 0. Når *forén*-operationen udføres på to repræsentanter af forskellig rang, gøres repræsentanten med den lavere rang til den andens barn. Når rangene er ens, kan

den nye repræsentant vælges vilkårligt, men vi øger den nye repræsentants rang med 1. Denne første forbedring hedder *forening efter rang*.

Opgave 11.8. Antag, at *find*-operationen ikke forandrer træstrukturen (eller at operationen ikke forekommer.) Vis, at rangen af en repræsentant er lig med dybden af dets undertræ.

Den anden forbedring hedder *vejforkortning*. Den sikrer, at en kæde af forælderpegere aldrig gennemløbes to gange. Hertil ændrer vi ved hver udførelse af $\mathit{find}(v)$ alle de besøgte knuders forælderpegere til at pege på repræsentanten for vs klasse. I fig. 11.6 er denne regel formuleret som en rekursiv procedure. Proceduren gennemløber først vejen fra v til dens repræsentant og benytter så rekursionsstakken for pånyt at gennemløbe samme vej til v i modsat retning. Mens rekursionsstakken afstakkes, ændres forælderpegerne. Alternativt kan vejen også gennemløbes to gange i samme retning. Ved første gennemløb bestemmes repræsentanten; ved andet gennemløb ændres forælderpegerne.

Opgave 11.9. Beskriv en ikke-rekursiv implementation af *find*.

Sætning 11.1. *Med forening efter rang har alle træer dybde højst* $\log n$.

Bevis. Vi viser først sætningen uden brug af vejforkortning. Det betyder, at *find*-operationen ikke ændrer datastrukturen. I så fald er rangen af en repræsentant altid lig med dybden af dens undertræ (opg. 11.8). Vi kan altså nøjes med at begrænse rangen af hver rod med $\log n$. Hertil vil vi vise (ved induktion), at et træ, hvis rod har rang k, indeholder mindst 2^k knuder. For $k = 0$ gælder udsagnet klart. Det eneste tidspunkt, hvor rangen af en rod forandres, er når en *forén*-operation øger den fra $k-1$ til k. I så fald har den fået et nyt barn, som selv var repræsentant med rang $k-1$. Ifølge induktionshypotesen indeholdt rodens og det nye barns træer inden operationen hver mindst 2^{k-1} knuder. Efter operationen har det nye træ derfor mindst $2^{k-1} + 2^{k-1} = 2^k$ knuder, og vi har bevist påstanden. Vi konkluderer, at den største rang er højst $\log n$, idet der kun er n knuder i det hele.

Beviset gælder stadig, når vi tilføjer forbedringen »vejforkortning«, idet vejforkortning aldrig gør et træ dybere. □

Kombinationen af strategierne vejforkortning og forening efter rang gør forén og find-datastrukturen overordentligt effektiv. Hver operations amortiserede omkostning er nærmest konstant.

Sætning 11.2. *Forén og find-datastrukturen i fig. 11.6 udfører m find- og $n-1$ forénoperationer i tid $O(m\alpha(m,n))$, hvor*

$$\alpha(m,n) = \min\{\, i \geq 1 \colon A(i, \lceil m/n \rceil) \geq \log n \,\},$$

med

$$\begin{aligned} A(1,j) &= 2^j, && \text{for } j \geq 1\,, \\ A(i,1) &= A(i-1,2), && \text{for } i \geq 2\,, \\ A(i,j) &= A(i-1, A(i,j-1)) && \text{for } i \geq 2, j \geq 2\,. \end{aligned}$$

Det er ikke umiddelbart let at se, hvad disse formler skulle betyde. Funktionen[3] A vokser ekstremt hurtigt. Der gælder $A(1,j) = 2^j$ for $j \geq 1$, endvidere $A(2,1) = A(1,2) = 2^2 = 4$, $A(2,2) = A(1,A(2,1)) = 2^4 = 16$, $A(2,3) = A(1,A(2,2)) = 2^{16}$, $A(2,4) = 2^{2^{16}}$, $A(2,5) = 2^{2^{2^{16}}}$, $A(3,1) = A(2,2) = 16$, $A(3,2) = A(2,A(3,1)) = A(2,16)$, og så videre.

Opgave 11.10. Anslå værdien af $A(5,1)$.

For alle praktisk tænkelige [4] værdier n gælder $\alpha(m,n) \leq 5$, og forening efter rang og vejforkortning garanterer tilsammen i alt væsentligt konstant amortiseret tid for hver operation. Beviset for sætning 11.2 går ud over denne lærebogs rammer. Vi henviser læseren til [53, 48]. Her skal vi i stedet vise en svagere udgave, hvis udsagn for praktiske formål er lige så stærkt. Hertil definerer vi talfølgen $T_k, k \geq 0$ ved induktion: $T_0 = 1$ og $T_k = 2^{T_{k-1}}$ for $k \geq 1$. Tallet T_k kan altså fremstilles som tårn af 2-potenser $2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ af højde k. Man ser, at der for $k \geq 2$ gælder ligningen $T_k = A(2,k-1)$. De første værdier for denne meget hurtig voksende talfølge ser sådan ud:

k	0	1	2	3	4	5	6	$\cdots$
T_k	1	2	4	16	65536	2^{65536}	$2^{2^{65536}}$	$\cdots$

For $x > 0$ definerer vi nu $\log^* x$ som $\min\{\, x \colon T_k \geq x \,\}$. Det er samtidig det mindste tal $k \geq 0$, hvis k-foldige logaritme $\log(\log(\cdots \log x) \cdots))$ er højst 1. Funktionen $\log^* x$ vokser ekstremt langsomt, fx gælder $\log^* x \leq 5$ for alle $x \leq 10^{19729} (< 2^{65536})$.

Sætning 11.3. *Forén og find-datastrukturen med forening efter rang og vejforkortning udfører m find- og $(n-1)$ forén-operationer i tid $O((m+n)\log^* n)$.*

Bevis. (Baseret på [26].) Vi betragter en vilkårlig følge af m *find*- og $n-1$ *forén*-operationer, med udgangspunkt i initialiseringen for mængden $\{1,\ldots,n\}$. Idet *forén* tager konstant tid, kan vi indskrænke os til analysen af *find*.

Hidtil har kun rodknuderne haft tildelt en rang. Disse range kan vokse i løbet af operationsfølgen. Nu definerer vi for hver knude v dens *sidste rang* $sr(v)$ som rangen af v i det øjeblik, da den for første gang gøres til barn af en anden knude. Skulle v stadig være en rod til sidst, sætter vi $sr(v)$ til vs endelige rang. Vi gør følgende observationer:

[3] Bogstavet A henviser til logikeren W. Ackermann, som sent i 1920rne som den første angav en lignende funktion.

[4] Ovs. anm.: Man skal holde sig for øje, at antallet af elementarpartikler i universet anslås til at være mindre end 10^{100}.

(i) Langs hver vej i træerne er *sr*-værdierne skarpt voksende nedefra og op.
(ii) Når en knude v får sin slutrang h, indeholder vs undertræ mindst 2^h knuder.
(iii) Der er højst $n/2^h$ knuder med slutrang h.

Bevis for observationerne: (i) Ved induktion i operationerne vil vi vise, at der altid gælder $sr(v) < sr(u)$, når v er barn til u. I begyndelsen er alle range 0, og ingen knude er barn til en anden. I det øjeblik, hvor v med gøres til barn af rodknuden u ved en forening, gælder efter operationen at $rang(v) < rang(u)$ ifølge algoritmen og $sr(v) = rang(v)$ ifølge definitionen af *sr*. Idet der for alle rodknuder gælder $rang(u) \leq sr(u)$, har vi $sr(u) < sr(v)$. En *find*-operation ændrer ikke ved denne egenskab, da den kun kan gøre v til barn af en af sine aner, som ved induktion har endnu højere slutrang.

(ii) I beviset for sætning 11.1 så vi, at en rod af rang h indeholder mindst 2^h knuder i sit undertræ. Det vil sige, at en knude v med slutrang $sr(v) = h$ i det øjeblik, hvor den bliver barn til en anden knude, har mindst 2^h knuder i sit undertræ. (Bemærk, at v senere kan miste knuder fra sit undertræ på grund af vejforkortning.)

(iii) Betragt for fast h og hver knude v med slutrang h mængden M_v af knuder i vs undertræ i det øjeblik, hvor v bliver gjort til barn af en anden knude u (hhv. til sidst, hvis v da stadig er en rod). Vi påstår at mængderne M_v er disjunkte. (Heraf følger påstanden, fordi (ii) medfører, at hver mængde M_v indeholder mindst 2^h elementer, og der er n elementer i det hele.) Ifølge (i) gælder $sr(u) > h$. Betragt nu $w \in M_v$. På grund af foreninger kan w få nye aner i det videre forløb. Men i det øjeblik, hvor v' bliver sådan en ny ane, må v' være rod i et undertræ som også indeholder u; ifølge (i) har vi altså $sr(v') > sr(u) > h$. Derfor kan w aldrig tilhøre undertræet til nogen anden knude med slutrang h.

Knuderne med *sr*-værdi større end 0 inddeles nu i *ranggrupper* $G_0, G_1, \ldots$. Ranggruppe G_k indeholder alle knuder med slutrang i $\{T_{k-1} + 1, \ldots, T_k\}$, hvor vi tolker T_{-1} som 0. For eksempel indeholder G_4 alle knuder med slutrang i intervallet $\{17, \ldots, 65536\}$. Da ifølge sætning 11.1 ingen rang kan blive større end $\log n$, kan vi allerede nu ane, at der for problemstørrelser af n, som måtte forekomme i praksis, aldrig vil være en knude i ranggruppe G_5 eller højere. For at være nøjagtig betragter man følgende: Når en knude v ligger i ranggruppe G_k for $k \geq 1$, så gælder $T_{k-1} < sr(v) \leq \log n$. Heraf følger, at $T_k = 2^{k-1} < 2^{sr(v)} \leq n$, hvoraf $k < \log^* n$. Antallet af ikketomme ranggrupper er altså begrænset af $\log^* n$.

Vi analyserer nu omkostningen af *find*-operationen. Vi anslår omkostningen for $find(v)$ som antallet r af knuder på vejen $v = v_1, \ldots, v_{r-1}, v_r = s$ fra v til træets rod s. (Denne værdi er proportional med operationens tidsforbrug, inklusive vejforkortningens pegerændringer.) I bund og grund fordeler vi omkostningerne på knuderne v_i langs vejen, med omkostning 1 per knude. Vi undtager dog følgende knuder, hvis omkostning vi direkte tilskriver $find(v)$-operationen.

(a) omkostningen for knuder v med $sr(v) = 0$ (da må gælde, at v er et blad).
(b) omkostningen for knuder v_i med $sr(v_i) > 0$ og hvis forælder v_{i+1} tilhører en højere ranggruppe.
(c) omkostningen for rodknuden v_r og dens umiddelbare efterfølger v_{r-1}.

Fordi vi fra (i) har, at rangen langs vejen er voksende, og der ikke er mere end $\log^* n$ ikketomme ranggrupper, kan der ikke findes mere end $(\log^* n) - 1$ knuder, der er omfattet af (b). Sammen med de højst 3 knuder, der er omfattet af (a) og (c), er altså totalt højst $2 + \log^* n$ knuders omkostning tilskrevet *find*(v). Sammenlagt over alle m *find*-operationer giver det en omkostning på $m(2 + \log^* n)$.

Det står tilbage at tage hånd om de omkostninger, det er blevet tildelt knuderne. Betragt en knude v i ranggruppe G_k. Når v tildeles omkostningen 1 ved en *find*-operation, har v en forælder u (som tilhører G_k pga. (b)), som ikke er roden s pga. (c). Derfor får v ved vejforkortningen s som ny forælder. Ved betragtning (i) gælder $sr(u) < sr(s)$. Det betyder, at når v tildeles omkostningen 1, får den en ny forælder af højere slutrang. Efter højst T_k af disse hændelser må v have fået en forælder i en højere ranggruppe, og herefter tildeles ikke flere omkostninger til v. Knuden v tildeles altså højst omkostning T_k. Summeret over alle knuder i ranggruppen G_k bliver den totale omkostning ikke mere end $|G_k| \cdot T_k$.

For at vurdere $|G_k| \cdot T_k$ betragter vi knuderne i G_k. Deres mulige slutrange er $T_{k-1} + 1, \ldots, T_k$. Ifølge betragtning (iii) findes der højst $n/2^h$ knuder med slutrang h. Derfor har vi

$$|G_k| \leq \sum_{h=T_{k-1}+1} T_k \frac{n}{2^h} < \frac{n}{2^{T_{k-1}}} = \frac{n}{T_k},$$

ifølge definitionen af T_k. Dette giver $|G_k| \cdot T_k < n$. Det betyder at den totale omkostning af alle knuder i ranggruppe k er højst n. Vi har allerede indset, at der ikke er flere end $\log^* n$ ikketomme ranggrupper. Derfor er omkostningen for samtlige knuder begrænset af $n \log^* n$.

Til sidst adderer vi omkostningerne, som er tildelt operationerne og knuderne, og får $m(2 + \log^* n) + n \log^* n = O((m + n) \log^* n)$, som ønsket. □

11.5 Anvendelser

Udeladt

11.6 Implementationsaspekter

Dele udeladt

Forén og find-datastrukturen kan implementeres på en mere pladsbesparende måde ved at observere, at kun repræsentanter har en rang og kun ikke-repræsentanter har en forgænger. Vi kan altså undgå rækken *rang* i figur 11.6; i stedet sætter vi for rodknude r med rang g indgangen $\mathit{forælder}[r] = n + 1 + g$. I stedet for to rækker kan vi derfor nøjes med én række, hvis værdier ligger i $\{1, \ldots, n + 1 + \lceil \log n \rceil\}$.

11.7 Historiske anmærkninger og videre resultater

Udeladt

12

Generelle tilgange til optimeringsproblemer

12.1 Lineær programlægning

12.2 Grådige algoritmer

En algoritme er *grådig*, hvis den behandler sine objekter i en specifik (ofte nøje udvalgt) rækkefølge og træffer beslutninger om hvert objekt (fx om det skal medtages i løsningen eller ej) i samme øjeblik, som objektet betragtes. Når en beslutning er taget, bliver den ikke ændret senere. Algoritmen for rygsæksproblemet fra sidste afsnit følger det grådige princip: Vi betrager objekterne i rækkefølgen givet ved deres profittæthed. Algoritmerne for korteste veje i acykliske grafer og for ikke-negative kantomkostninger (afsnit 10.2 og 10.3) og for minimale spændetræer (kapitel 11) er ligeledes grådige. For eksempel betragter Dijkstras algoritme knuderne i voksende afstand af TODO. For disse problemer finder den grådige strategi en optimal løsning.

12.3 Dynamisk programlægning

12.4 Udtømmende søgning

12.5 Lokal søgning

12.6 Evolutionære algoritmer

12.7 Implementationsaspekter

12.8 Historiske anmærkninger og videre resultater

Udeladte

A

Appendix

A.1 Matematiske symboler

$\{e_1, \ldots, e_n\}$ mængden af elementer $e_1, \ldots, e_n$.

$\{\, e \colon P(e) \,\}$ mængden af elementer, for hvilke P gælder.

$\langle e_1, \ldots, e_n \rangle$ følgen af indgange $e_1, \ldots, e_n$.

$\{\, e \in S \colon P(e) \,\}$ delfølgen af indgange fra S, for hvilke P gælder.

$|x|$ absolutværdien af x for et reelt tal x.

$\lfloor x \rfloor$ det største heltal n med $n \leq x$ for et reelt tal x (gulvfunktionen, nedrunding).

$\lceil x \rceil$ det minste heltal n med $n \geq x$ for et reelt tal x (loftsfunktionen, oprunding).

$[a, b]$ $\{\, r \in \mathbf{R} \colon a \leq x \leq b \}$.

$i..j$ forkortelse for $\{i, \ldots, j\}$

A^B mængden af funtioner fra A til B

$A \times B$ mængden af ordnede par (a, b) med $a \in A$ og $b \in B$.

$\perp$ en udefineret værdi

∞ uendeligt

$-\infty$ minus uendeligt

$\forall x \colon P(x)$ alle x opfylder udsagnet P

$\exists x \colon P(x)$ der ekisterer et x, som opfylder udsagnet P

$\mathbf{N}$ mængden af ikke-negative heltal, $\mathbf{N} = \{0, 1, 2, \ldots\}$.

$\mathbf{N}_+$ mængden af positive heltal, $\mathbf{N} = \{1, 2, \ldots\}$.

$\mathbf{Z}$ mængden af hele tal, $\mathbf{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$.

$\mathbf{R}$ mængden af reelle tal.

$\mathbf{R}_{>0}$ mængden af positive reelle tal.

$\mathbf{Q}$ mængden af rationelle tal.

$|$, &, «, », $\oplus$ bitvis eller, og, venstreskift, højreskift og eksklusivt eller.

$\sum_{i=1}^{n} a_i = \sum_{1 \le i \le n} a_i = \sum_{i \in \{1,\ldots,n\} a_i} = a_1 + \cdots + a_n$.

$\sum_{s \in S} a_x$ hhv. $\sum_{P(x)} a_x$ eller $\sum_{x \text{ med } P(x)} a_x$ Summen over elementerne i en endelig mængde S, hhv. over alle elementer som upfylder prædikatet P.

$\prod_{i=1}^{n} a_i = \prod_{1 \le i \le n} a_i = \prod_{i \in \{1,\ldots,n\} a_i} = a_1 \cdot \cdots \cdot a_n$.

$n!$ $\prod_{i=1}^{n} i$, kaldt »n fakultet«. Der gælder $0! = 1$.

H_n $\sum_{i=1}^{n} 1/i$, kaldt »det $n$te harmoniske tal« (se ulighed (A.12)).

$\log x$ logaritmen til x med grundtal 2, $\log_2 x$, for $x > 0$.

$\log^* x$ for $x > 0$, det mindste tal $k \ge 0$ for hvilket $\log(\log(\cdots(\log x)\cdots)) \le 1$ (k-foldig gentagelse af logaritmen).

$\ln x$ den naturlige logaritme til x med grundtal 2,718 281 828 459 0 …, for $x > 0$.

$\mu(s, t)$ længden af den korteste vej fra s til t; $\mu(t) := \mu(s, t)$.

div kvotienten ved heltalsdivision; $m \operatorname{div} n := \lceil m/n \rceil$ for $n > 0$.

mod resten ved heltalsdivision; $m \bmod n := m - n(m \operatorname{div} n)$ for $n > 0$.

$a \equiv b \pmod{m}$ for $m > 0$: tallene a og b er kongruente modulo m, dvs. $a + im = b$ for et heltal i.

$\prec$ en vilkårlig ordensrelation.

1, 0 de booleske værdier »sand« og »falsk«.

Σ^* mængden $\{ (a_1, \ldots, a_n) : n \in \mathbf{N}, a_1, \ldots, a_n \in \Sigma \}$ af tegnfølger (eller »ord«) $a_1 \cdots a_n = (a_1, \ldots, a_n)$ over det endelige alfabet Σ.

$|x|$ antallet n af bogstaver i $x = (a_1, \ldots, a_n) \in \Sigma^*$ for ord x.

A.2 Matematiske begreber

antisymmetrisk: En relation R kaldes *antisymmetrisk*, hvis der for alle a og b gælder, at aRb og bRa medfører $a = b$.

ækvivalensrelation: en transitiv, refleksiv og symmetrisk relation.

asymptotisk notation:

$$\begin{aligned}
\mathrm{O}(f(n)) &= \{\, g(n) \colon \exists c > 0 \colon \exists n_0 \in \mathbf{N}_+ \colon \forall n \geq n_0 \colon g(n) \leq c \cdot f(n) \,\}. \\
\Omega(f(n)) &= \{\, g(n) \colon \exists c > 0 \colon \exists n_0 \in \mathbf{N}_+ \colon \forall n \geq n_0 \colon g(n) \geq c \cdot f(n) \,\}. \\
\Theta(f(n)) &= \mathrm{O}(f(n)) \cap \Omega(f(n)). \\
o(f(n)) &= \{\, g(n) \colon \forall c > 0 \colon \exists n_0 \in \mathbf{N}_+ \colon \forall n \geq n_0 \colon g(n) \leq c \cdot f(n) \,\}. \\
\omega(f(n)) &= \{\, g(n) \colon \forall c > 0 \colon \exists n_0 \in \mathbf{N}_+ \colon \forall n \geq n_0 \colon g(n) \geq c \cdot f(n) \,\}.
\end{aligned}$$

Se også afsnit 2.1.

konkav: En funktion f er konkav på intervallet $[a, b]$, hvis

$$\forall x, y \in [a, b] \colon \forall t \in [0, 1] \colon f(tx + (1 - t)y) \geq tf(x) + (1 - t)f(y).$$

konveks: En funktion f er konveks på internvallet $[a, b]$, hvis

$$\forall x, y \in [a, b] \colon \forall t \in [0, 1] \colon f(tx + (1 - t)y) \leq tf(x) + (1 - t)f(y).$$

legeme: en mængde af tal (indeholdende nul og et), for hvilke der er defineret følgende operationer: addition og (en hertil invers operation) subtraktion, multiplikation og (en hertil invers operation) division med elementer, som ikke er nul. Addition og multiplikation er kommutativ og associativ og har neutralelementer, som opfører sig som de reelle tal nul og et. De væsentligste eksempler på legemer er: $\mathbf{R}$, de reelle tal; $\mathbf{Q}$, de rationelle tal; $\mathbf{Z}_p$, heltallene modulo et primtal p.

leksikografisk ordning: den kanoniske måde at udvide en lineær ordning af en mængde til tupler, strenge og følger over samme mængde. Der gælder $\langle a_1, \ldots, a_k \rangle < \langle b_1, \ldots, b_\ell \rangle$ hvis og kun hvis findes et $i \leq \min\{k, \ldots, \ell\}$ med $\langle a_1, \ldots, a_{i-1} \rangle = \langle b_1, \ldots, b_{i-1} \rangle$ og $a_i < b_i$ eller eller når $k < \ell$ og $\langle a_1, \ldots, a_k \rangle = \langle b_1, \ldots, b_k \rangle$. En alternativ, rekursiv definition er: $\langle\,\rangle < \langle b_1, \ldots, b_\ell \rangle$ for alle $\ell > 0$; for $k > 0$ og $\ell > 0$ gælder $\langle a_1, a_2, \ldots, a_k \rangle < \langle b_1, b_2, \ldots, b_\ell \rangle$ hvis og kun hvis $a_1 < b_1$ eller $a_1 = b_1$ og $\langle a_2, \ldots, a_k \rangle < \langle b_2, \ldots, b_\ell \rangle$.

lineær ordning: (også: total ordning) en refleksiv, transitiv, antisymmetrisk og total relation. Ofte benyttes symbolet $\leq$ for en lineær ordning. For alle $a \leq b$ skrives da også $g \geq a$.. Forkortelse (den *strenge* version af en total ordning): Vi skriver $a < b$ når $a \leq b$ og $a \neq b$; tilsvarende står $a > b$ for $a \geq b$ og $a \neq b$. I så fald er relationene $<$ transitiv, *irrefleksiv* (idet $a < b$ medfører $a \neq b$) og *total* i den forstand at der for hvert par a, b gælder $a < b$ eller $a = b$ eller $a > b$.

lineær præordning: (også: total præordning eller lineær/total kvasiordning) en refleksiv, transitiv og total relation. Også hertil benyttes ofte symbolerne $\leq$ og $\geq$.
Den strenge variant defineres i dette tilfælde på følgende måde: $a < b$ hvis der gælder $a \leq b$ men ikke $a \geq b$.

median: en indgang med rang $\lceil \frac{1}{2}n \rceil$ i en mængde med n indgange.

multiplikativ invers: Når man multiplicerer et objekt x med dets *multiplikative invers* x^{-1}, opnås $x \cdot x^{-1} = 1$, dvs. multiplikationens neutralelement. Især gælder der i et *legeme*, at hvert element bortset fra nul (additionens neutralelement) har en multiplikativ invers.

primtal: et heltal n med $n \geq 2$ er et primtal, dersom der ikke findes heltal $a, b > 1$ så $n = a \cdot b$.

rang: Lad den lineære præordning $\leq$ være defineret på en endelig mængde $S = \{e_1, \ldots, e_n\}$. En injektiv afbildning $r \colon S \to \{1, \ldots, n\}$ er en *rangfunktion* for S, hvis der gælder $r(e_i) < r(e_j)$ for alle e_i, e_j med $e_i < e_j$. Når $\leq$ er en lineær ordning på S, eksisterer der præcis en rangfunktion.

refleksiv: En relation $R \subseteq A \times A$ kaldes *refleksiv*, hvis $\forall a \in A \colon (a, a) \in R$.

relation: En (binær) relation R er en mængde af ordnede par. Vi skriver ofte relationen med infiksnotation: udsagnet aRb betyder bare $(a, b) \in R$. (Alle relationer, som forekommer i denne bog, er binære.)

symmetrisk relation: Relationen R er *symmetrisk*, hvis der for alle a og b gælder, at aRb medfører bRa.

total ordning: dss. lineær ordning.

total relation: en relation $R \subseteq A \times A$ er total på A, dersom for alle $a, b \in A$ gælder mindst et af udsagnene aRb og bRa. Når en relation R er total og transitiv, danner følgende definition en passende ækvivalensrelation $\sim_R$ på A: $a \sim_R b$ hvis og kun hvis aRb og bRa.

transitiv: relationen R kaldes *transitiv*, hvis der vor alle a, b, c gælder, at aRb og bRc medfører aRc.

A.3 Grundlæggende sandsynlighedsregning

Sansynlighedsregningen er baseret på begrebet *udfaldsrum*. $\mathcal{S}$ For at beskrive kast med to terninger, skulle vi anvende udfaldsrummet $\{1, \ldots, 6\} \times \{1, \ldots, 6\}$ med 36 elementer, dvs. at elementerne i udfaldsrummet (som også kaldes elementarhændelser) er tuplerne (x, y) med $1 \leq x, y \leq 6$ og $x, y \in \mathbf{N}$. Generelt er udfaldsrummet en vilkårlig ikke-tom mængde. I denne

bog er alle udfaldsrum endelige.[1] For hvert tilfældige eksperiment tildeles hvert element $s \in \mathcal{S}$ en *sandsynlighed* eller *punktsandsynlighed* $p_s \geq 0$, så der gælder $\sum_{s \in \mathcal{S}} p_s = 1$. Den funktion, som afbilder hver hændelse s til sin sandsynlighed p_s, hedder *sandsynlighedsfunktionen* eller *fordeling*. Udfaldsrummet sammen med en fordeling kaldes *sandsynlighedsfelt*. I denne bog bruges næsten udelukkende *uniforme fordelinger*; sådanne fordelinger opfylder $p_s = p = 1/|\mathcal{S}|$. En delmængde $\mathcal{E}$ af udfaldsrummet kaldes en *hændelse*. Sandsynligheden $\Pr(\mathcal{E})$ af en hændelse $\mathcal{E} \subseteq \mathcal{S}$ er summen af punktsandsynlighederne for elementarhændelserne. I det uniforme tilfælde indebærer det, at $\Pr(\mathcal{E}) = |\mathcal{E}|/|\mathcal{S}|$. For eksempel indeholder udfaldsrummet for terningekast hændelsen $\{\,(x,y) \colon x + y = 7\,\} = \{(1,6),(2,5),\ldots,(6,1)\}$ med sandsynlighed $\frac{6}{36} = \frac{1}{6}$, hvorimod hændelsen $\{\,(x,y) \colon x + y \geq 8\,\}$ har sandsynlighed $\frac{15}{36} = \frac{5}{12}$.

En *stokastisk variabel* er en afbildning fra udfaldsrummet til de reelle tal. Stokastiske variable betegnes sædvanligvis med store bogstaver, for at adskille dem fra andre værdier. I terningeeksemplet kan vi lade stokastiske variable X angive antallet af øjne på den første terning, Y angive antallet af øjne på den anden terning og S summen af de to terninger. Formelt har vi $(x,y) \in \mathcal{S}$, at $X((x,y)) = x$, $Y((x,y)) = y$ og $S((x,y)) = x + y = X((x,y)) + Y((x,y))$.

Givet nogle stokastiske variable kan man definere nye ved at danne udtryk ud fra variablerne og sædvanlige værdier. Fx kan man ud fra de stokastiske variable V og W skabe de nye stokastiske variable $(V + W)(s) = V(s) + W(s)$, $(V \cdot W)(s) = V(s) \cdot W(s)$ og $(V + 3)(s) = V(s) + 3$.

Hændelser angives ofte som prædikater, i hvilke der indgår stokastiske variable. For eksempel betegner $X \leq 2$ hændelsen $\{\,(1,y),(2,y) \colon 1 \leq y \leq 6\,\}$; hermed kan man udtrykke $\Pr(X \leq 2) = \frac{1}{3}$. På samme måde kan man udtrykke $\Pr(X + Y = 11) = \Pr(\{(5,6),(6,5)\}) = \frac{1}{18}$.

En *stokastisk indikatorvariabel* (kort: indikatorvariabel) er en stokastisk variabel, som kun antager værdierne 0 og 1. Indikatorvariabler spiller en fremtrædende rolle i sandsynlighedsteoretiske analyser af algoritmer, fordi de gør det muligt at udtrykke komplekse algoritmers opførsel i termer af enkel matematiske objekter. Indikatorvariabler betegnes ofte med bogstaverne I og J. Hændelser og indikatorvariabler svarer til hinanden en-til-en: Til hændelsen $\mathcal{E}$ hører indikatorvariablen $I_{\mathcal{E}}$, som har værdien 1 netop for $s \in \mathcal{E}$. Hvis en hændelse er beskrevet af prædikatet P, skriver man ofte kort $[P]$ for den tilsvarende indikatorvariable, dvs. $[P](s) = 1$ hvis $P(s)$ gælder, ellers $[P](s) = 0$.

Middelværdien for en stokastisk variabel $Z : \mathcal{S} \to \mathbf{R}$, også kaldt forventingsværdien, er

$$E[Z] = \sum_{s \in \mathcal{S}} p_s \cdot Z(s) = \sum_{z \in Z[\mathcal{S}]} z \cdot \Pr(Z = z)\,. \tag{A.0}$$

[1] Udsagnene i dette afsnit gælder i alt væsentligt med de samme beviser også for tælleligt uendelige udfaldsrum. Den slags rum behøver man fx til at modellere eksperimentetet »slå en terning, indtil du får en 6er.«

Mao. bidrager hver hændelse s med værdien af Z på s, ganget med sandsynligheden af s. Alternativt kan vi for hvert z i værdiområdet $Z[\mathcal{S}]$ for Z samle alle hændelser s, som opfylder $Z(s) = z$, til en enkelt hændelse $Z = z$ og så summere over alle $z \in Z[\mathcal{S}]$.

For den stokastiske variable X fra vores eksempel foroven gælder $E[X] = \frac{1}{6}(1+2+3+4+5+6) = \frac{21}{6} = 3{,}5$, dvs. at middelværdien for antallet af øjne på den første terning er 3,5. Det samme gælder naturligvis for den anden terning. For indikatorvariable I sælder

$$E[I] = 0 \cdot \Pr(I = 0) + 1 \cdot \Pr(I = 1) = \Pr(I = 1)\,.$$

Ofte er det ikke udfaldsrummet, som er så interessant, men den stokastiske variable Z og den opførsel i $\mathbf{R}$. I så fald er det nok at kende mængden $Z[\mathcal{S}]$ sammen med sandsynlighederne $\Pr(Z = z)$, $z \in Z[\mathcal{S}]$. Man kan sågar opfatte $Z[\mathcal{S}]$ som (afledt) udfaldsrum, hvis fordeling er givet ved $\Pr(Z = z)$ for $z \in Z[\mathcal{S}]$. Derfor kaldes funktionen $z \mapsto \Pr(Z = z)$, som knytter den pågældende sandsynlighed til hvert $z \in Z[\mathcal{S}]$, ofte for *fordelingen af* Z. . To stokastiske variable Y og Z med samme fordeling kaldes *identisk fordelte*, begrebet udvides til mere end to variable.

For stokastiske variable Z, som kun antager værdier i de naturlige tal, gælder følgende meget nyttige formel for middelværdien:

$$E[Z] = \sum_{k \geq 1} \Pr(Z \geq k), \text{ hvis } Z[\mathcal{S}] \subseteq \mathbf{N}\,. \tag{A.1}$$

Beviset er meget let. Sæt $p_k = \Pr(Z \geq k)$ og $q_i = \Pr(Z = i)$ for $k, i \in \mathbf{N}$. Da gælder $p_k = \sum_{i \geq k} q_i$ og derfor, efter en enkel ombytning af summationsrækkefølgen,

$$E[Z] = \sum_{z \in Z[\mathcal{S}]} z \cdot \Pr(Z = z) = \sum_{i \in \mathbf{N}} i \cdot \Pr(Z = i) = \sum_{i \in \mathbf{N}} \sum_{1 \leq k \leq i} q_i = \sum_{k \geq 1} \sum_{i \geq k} q_i = \sum_{k \geq 1} p_k.$$

Ofte er man interesseret i middelværdien for en stokastisk variabel, som er defineret i termer af en anden stokastisk variabel. For summen af stokastiske variabler er dette særlig enkelt. Grunden er *linearitet af middelværdien* for stokastiske variabler. For vilkårlige stokastiske variabler V og W gælder

$$E[V + W] = E[V] + E[W]\,. \tag{A.2}$$

Denne ligning er yderst anvendelig. Beviset er nemt, det koger i al væsentlighed ned til at bruge den distributive lov. Der gælder

$$\begin{aligned} E[V + W] &= \sum_{s \in \mathcal{S}} p_s \cdot (V(s) + W(s)) \\ &= \sum_{s \in \mathcal{S}} p_s \cdot V(s) + \sum_{s \in \mathcal{S}} p_s \cdot W(s) \\ &= E[V] + E[W]\,. \end{aligned}$$

Som første anvendelse vil vi beregne det forventede antal øjne af to terninger. Der gælder

$$E[S] = E[X + Y] = E[X] + E[Y] = 3{,}5 + 3{,}5 = 7\,.$$

Læg mærke til, at resultatet fremkommer stor set uden beregninger. Hvis man ville etablere samme resultat uden at bruge middelværdiens linearitet, skulle man gennemføre nogle omstændige udregninger:

$$\begin{aligned} E[S] &= 2 \cdot \tfrac{1}{36} + 3 \cdot \tfrac{2}{36} + 4 \cdot \tfrac{3}{36} + 5 \cdot \tfrac{4}{36} + 6 \cdot \tfrac{5}{36} + 7 \cdot \tfrac{6}{36} + 8 \cdot \tfrac{5}{36} + \cdots + 12 \cdot \tfrac{1}{36} \\ &= \frac{2 \cdot 1 + 3 \cdot 2 + 4 \cdot 3 + 5 \cdot 4 + 6 \cdot 5 + 7 \cdot 6 + 8 \cdot 5 + \ldots + 12 \cdot 1}{36} = 7\,. \end{aligned}$$

Lige så enkelt som (A.2) viser man, at der gælder $E[aV] = aE[V]$ for $a \in \mathbf{R}$. Man kan let udvide (A.2) til flere end to termer, vilket bevises ved fuldstændig indkuktion.

Opgave A.1. Hvad er det forventede antal øjne ved kast med tre terninger?

Som eksempel på et noget mere komplekst udfaldsrum betragter vi nu et tilfældigt eksperiment, hvor n bolde kastes i m kurve. Hver bold lander i en (uniformt) tilfældig kurv, og forskellige kast er uafhængige. Formelt er udfaldsrummet mængden af alle funktioner f fra $\{1, \ldots, n\}$ til $\{1, \ldots, m\}$. Udfaldsrummet har størrelse m^n; for hvert $i \in \{1, \ldots, n\}$ angiver $f(i)$ den kurv, som den $i$te bold havner i. Alle udfald i udfaldsrummet har samme sandsynlighed. Hvor mange bolde kan vi forvente i den første kurv? Antallet af bolde i første kurv er en stokastisk variabel, som vi kalder W. For at bestemme dens middelværdi, benytter vi indikatorvariablen I_i for $i \in \{1, \ldots, n\}$. Variablen I_i er 1, når den $i$te bold havner i første kurv, og 0 ellers. Formelt: $I_i = [f(i) = 1]$. Da gælder $W = \sum_i I_i$ og derfor

$$E[W] = E\left[\sum_i I_i\right] = \sum_i E[I_i] = \sum_i \Pr(I_i = 1)\,.$$

Her har vi brugt linearitet af middelværdien for den anden lighed, mens den trejde følger af, at $I_i$erne er indikatorvariabler. Nu skal vi bare bestemme sandsynligheden for, at $I_i = 1$. Idet boldene kastes tilfældigt i kurvene, er der samme sandsynlighed for hver kurv for at blive ramt af den $i$te bold. [2] Heraf følger $\Pr(I_i = 1) = 1/m$, og derfor

$$E[W] = \sum_i \Pr(I_i = 1) = \sum_i \frac{1}{m} = \frac{n}{m}\,.$$

Produktet af stokastiske variabler opfører sig anderledes en summen. Generelt gælder $E[X \cdot Y] \neq E[X] \cdot E[Y]$. Der findes en vigtig undtagelse: for

[2] Formelt: der findes nøjagtigt m^{n-1} funktioner f med $f(i) = 1$, og $m^{n-1}/m^n = 1/m$.

uafhængige stokastiske variabler X og Y gælder lighed. Her kaldes stokastiske variabler $X_1, \ldots, X_k$ for uafhængige, hvis der gælder

$$\Pr(X_1 = x_1 \wedge \ldots \wedge X_k = x_k) = \prod_{i=1}^{k} \Pr(X_i = x_i) \text{ for alle } x_1, \ldots, x_k \in \mathbf{R}. \quad \text{(A.3)}$$

Hvis vi følger eksemplet med de to terninger, er antal øjne på den første hhv. den anden terning uafhængige stokastiske variabler. Antallet på den første terning og summen af de to terninger derimod ej uafhængige. (Ikke-uafhængige stokastiske variabler kaldes *afhængige*.)

Opgave A.2. Lad I og J være uafhængige indikatorvariabler og definer X som $X = (I + J) \bmod 2$, dvs. at X er 1 hvis og kun hvis I og j er forskellige. Vis, at I og J er uafhængige, men at I, J og X er afhængige.

Betragt nu to uafhængige stokastiske variable X og Y. For at vise, at middelværdien er multiplikativ, regner vi:

$$\begin{aligned}
E[X] \cdot E[Y] &= \left(\sum_x x \cdot \Pr(X = x) \right) \cdot \left(\sum_y y \cdot \Pr(X = y) \right) \\
&= \sum_{x,y} x \cdot y \cdot \Pr(X = x) \cdot \Pr(X = y) \\
&= \sum_{x,y} x \cdot y \cdot \Pr(X = x \wedge Y = y) \\
&= \sum_z \sum_{x,y\colon z = x \cdot y} z \cdot \Pr(X = x \wedge Y = y) \\
&= \sum_z z \cdot \sum_{x,y\colon z = x \cdot y} \Pr(X = x \wedge Y = y) \\
&= \sum_z z \cdot \Pr(X \cdot Y = z) \\
&= E[X \cdot Y] \,.
\end{aligned}$$

Hvor (u)sandsynligt er det, at et stokastisk variable afviger fra sin middelværdi? *Markovs ulighed* giver en enkel, men nyttig grænse. Lad X være en ikke-negativ stokastisk variabel og vælg $c > 0$ vilkårlig. Da gælder

$$\Pr(X \geq c \cdot E[X]) \leq \frac{1}{c} \,. \quad \text{(A.4)}$$

Beviset er nemt. Vi har:

$$\begin{aligned}
E[X] &= \sum_{z \in X[\mathcal{S}]} z \cdot \Pr(X = z) \\
&\geq \sum_{z \in X[\mathcal{S}],\, z \geq c \cdot E[X]} z \cdot \Pr(X = z) \\
&\geq c \cdot E[X] \cdot \Pr(X \geq c \cdot E[X]) \,,
\end{aligned}$$

hvor den første lighed udnytter, at vi summerer over en delmængde af mulige værdier og X er ikke-negativ. Den anden ulighed udnytter, at faktoren z i hver term i linje 2 er mindst $cE[X]$.

For stokastiske variabel, som opfylder stærkere krav, kan man etablere meget stærkere grænser. Følgende situation opstår flere steder i denne bog. Vi har en sum $X = X_1 + \ldots + X_n$ af n uafhængige indikatorvariabler $X_1, \ldots, X_n$ og vil vurdere sandsynligheden for, at X afviger meget fra sin middelværdi. I denne situation er følgende variant af *chernoffgrænsen* nyttig. For hvert $\epsilon > 0$ gælder:

$$\Pr(X < (1-\epsilon)E[X]) \leq \exp(-\epsilon^2 \tfrac{1}{2} E[X])\,, \tag{A.5}$$

$$\Pr(X > (1+\epsilon)E[X]) \leq \left(\frac{e^{\epsilon}}{(1+\epsilon)^{(1+\epsilon)}}\right)^{E[X]}\,. \tag{A.6}$$

Den slags grænser kaldes *halegrænser*, fordi de begrænser »halen« af sandsynlighedsfordelingen, dvs. den del, for hvilken X er langt fra sin middelværdi. Vi betragter et eksempel. Når vi slår plat og krone n gange og X_i er indikatorvariablen, som er 1 hvis og kun hvis det $i$te kast er »krone«, så angiver den stokastiske variabel $X = X_1 + \ldots + X_n$ antallet af kroneudfald under hele processen. Åbenbart gælder $E[X] = \frac{1}{2}n$. Chernoffgrænsen fortæller os, at $\Pr(X \leq (1-\epsilon)\frac{1}{2}n) \leq \exp(-\epsilon^2 \frac{1}{4}n)$. Hvis vi fx sætter $\epsilon = \frac{1}{10}$, får vi $\Pr(X \leq \frac{9}{10} \cdot \frac{1}{2}n) \leq \exp -\frac{1}{100} \cdot \frac{1}{4}n$. For $n = 10\,000$ er middelværdien 5000, og sandsynligheden for, at X ligger under 4500, er altså mindre end det forsvindende lille e^{-25}.

Opgave A.3. Vurder sandsynligheden for at X er større end 5050 i eksemplet foroven.

Når indikatorvariablerne er uafhængige og har samme fordeling, fx $\Pr(X_i = 1) = p$ for noget tal p, så er X *binomialfordelt* (mit parametre p og n):

$$\Pr(X = k) = \binom{n}{k} p^k (1-p)^{n-k}, \text{ for } 0 \leq k \leq n\,. \tag{A.7}$$

Opgave A.4 (Bolde i kurve, fortsætning)**.** Lad W som foroven være antallet af bolde i første kurv. Vis:

$$\Pr(W = k) = \binom{n}{k} \left(\frac{1}{m}\right)^k \left(1 - \frac{1}{m}\right)^{n-k}\,,$$

og forsøg at beregne $E[W]$ som $\sum_k \Pr(W = k)k$.

A.4 Nogle nyttige formler og uligheder

Vi begynder med at opremse nogle nyttige formler og grænser. Beviser for nogle af dem finder man forneden.

- Enkle grænser for fakultetsfunktionen:

$$\left(\frac{n}{e}\right)^n \leq n! \leq n^n, \text{ eller skarpere: } e\left(\frac{n}{e}\right)^n \leq n! \leq (en)\left(\frac{n}{e}\right)^n. \quad \text{(A.8)}$$

- Stirlings approksimationsformel for fakultetsfunktionen:

$$n! = \left(1+\Theta(\frac{1}{n})\right)\sqrt{2\pi n}\left(\frac{n}{e}\right)^n, \quad \text{(A.9)}$$

eller mere nøjagtig:

$$\sqrt{2\pi n}\cdot\left(\frac{n}{e}\right)^n \cdot e^{\frac{1}{12n+1}} \;<\; n! \;<\; \sqrt{2\pi n}\cdot\left(\frac{n}{e}\right)^n \cdot e^{\frac{1}{12n}}, \text{ für } n \geq 1. \quad \text{(A.9')}$$

- En øvre grænse for binomialkoefficienten:

$$\binom{n}{k} \leq \left(\frac{n\cdot e}{k}\right)^k. \quad \text{(A.10)}$$

- Summen af de første n positive heltal:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}. \quad \text{(A.11)}$$

- De harmoniske tal:

$$\ln n \leq H_n = \sum_{i=1}^{n} \frac{1}{i} \leq \ln n + 1. \quad \text{(A.12)}$$

- Geometrisk sum og geometrisk række:

$$\sum_{i=0}^{n-1} q^i = \frac{1-q^n}{1-q} \quad \text{for } q \neq 1 \text{ og} \quad \sum_{i\geq 0} q^i = \frac{1}{1-q} \quad \text{for } |q| < 1. \quad \text{(A.13)}$$

$$\sum_{i\geq 0} 2^{-i} = 2 \quad \text{og} \quad \sum_{i\geq 0} i\cdot 2^{-i} = \sum_{i\geq 1} i\cdot 2^{-i} = 2. \quad \text{(A.14)}$$

- Jensens ulighed: For hver konkave funktion f og hver følge $(x_1,\ldots,x_n)$ af reelle tal i definitionsområdet for f gælder:

$$\sum_{i=1}^{n} f(x_i) \leq n\cdot f\left(\frac{\sum_{i=1}^{n} x_i}{n}\right), \quad \text{(A.15)}$$

for hver konvekse funktion f og hver følge $x_1,\ldots,x_n$ af reelle tal i definitionsområdet af f gælder:

$$\sum_{i=1}^{n} f(x_i) \;\geq n\cdot f\left(\frac{\sum_{i=1}^{n} x_i}{n}\right). \quad \text{(A.16)}$$

A.4.1 Beviser

For (A.8) observerer vi først, at der gælder $n! = n(n-1)\cdots 1 \leq n^n$. Fra analysen vides det, at eksponentialfunktionen kan skrives som $\exp(x) = \sum_{i\geq 0} x^i/i!$. Derfor gælder $e^n \geq n^n/n!$, hvilket medføre den nedre grænse for $n!$.

For de skarpere grænser bemærker vi, at der for alle $i \geq 2$ gælder uligheden $\ln i \geq \int_{i-1}^{i} \ln x\, dx$, hvilket medfører

$$\ln n! = \sum_{2\leq i\leq n} \ln i \geq \int_1^n \ln x\, dx = \Big[x(\ln x - 1)\Big]_{x=1}^{x=n} = n(\ln n - 1) + 1\,.$$

Heraf følger

$$n! \geq e^{n(\ln n - 1)+1} = e(e^{\ln n - 1})^n = e\left(\frac{n}{e}\right)^n.$$

På samme måde følger af uligheden $\ln(i-1) \leq \int_{i-1}^{i} \ln x\, dx$, at der gælder $(n-1)! \leq \int_1^n \ln x\, dx = e\left(\frac{n}{e}\right)^n$, hvilket medfører $n! \leq (en)\left(\frac{n}{e}\right)^n$.

Uligheden (A.10) følger næsten direkte af (A.8). Vi har

$$\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k!} \leq \frac{n^k}{(k/e)^k} = \Big(\frac{n\cdot e}{k}\Big)^k.$$

Ligning (A.11) fås ved udregning med et enkelt kneb:

$$\begin{aligned} 1 + \cdots + n &= \tfrac{1}{2}\big((1 + \cdots + n) + (n + \cdots + 1)\big) \\ &= \tfrac{1}{2}\big((1+n) + \cdots + (n+1)\big) \\ &= \frac{n(n+1)}{2}\,. \end{aligned}$$

Det er ikke vanskelig at begrænse de tilsvarende summer af højere potenser. For eksempel gælder $\int_{i-1}^{i} x^2\, dx \leq i^2 \leq \int_i^{i+1} x^2\, dx$; heraf følger

$$\sum_{1\leq i\leq n} i^2 \leq \int_1^{n+1} x^2\, dx = \Big[\frac{x^3}{3}\Big]_{x=1}^{x=n+1} = \frac{(n+1)^3 - 1}{3}$$

og

$$\sum_{1\leq i\leq n} i^2 \geq \int_0^n x^2\, dx = \Big[\frac{x^3}{3}\Big]_{x=0}^{x=n} = \frac{n^3}{3}\,.$$

For (A.12) benytter vi ligeledes begrænsningen baseret på integralet. Der gælder $\int_i^{i+1}(1/x)\, dx \leq 1/i \leq \int_{i-1}^{i}(1/x)\, dx$, og derfor

$$\ln n = \int_1^n \frac{1}{x}\, dx \leq \sum_{1\leq i\leq n} \frac{1}{i} \leq 1 + \int_1^{n-1} \frac{1}{x}\, dx \leq 1 + \ln n\,.$$

Ligning (A.13) følger af

$$(1-q)\cdot\sum_{0\le i\le n-1} q^i = \sum_{0\le i\le n-1} q^i - \sum_{1\le i\le n} q^i = 1-q^n\,.$$

For $|q| < 1$ kan vi lade n gå mod uendelig, får at nå frem til $\sum_{i\ge 0} q^i = 1/(1-q)$. For $q = 1/2$ får vi $\sum_{i\ge 0} 2^{-i} = 2$. Endvidere gælder

$$\begin{aligned}\sum_{i\ge 1} i\cdot 2^{-i} &= \sum_{j\ge 1} 2^{-j} + \sum_{j\ge 2} 2^{-j} + \sum_{j\ge 3} 2^{-j} + \ldots \\ &= \left(1+\tfrac{1}{2}+\tfrac{1}{4}+\tfrac{1}{8}+\cdots\right)\cdot\sum_{i\ge 1} 2^{-i} \\ &= 2\cdot 1 = 2\,.\end{aligned}$$

For den første ligning skal man blot observere, at for hvert i optræder termen 2^{-i} netop i de første i af højresidens j-summer.

Ligning (A.15) kan man vise ved induktion efter n. For $n = 1$ er der intet at vise. Betragt altså $n \ge 2$. Sæt $x^* = \sum_{1\le i\le n} x_i/n$ og $\bar{x} = \sum_{1\le i\le n-1} x_i/(n-1)$. Da gælder $x^* = ((n-1)\bar{x} + x_n)/n$, og derfor

$$\begin{aligned}\sum_{1\le i\le n} f(x_i) &= f(x_n) + \sum_{1\le i\le n-1} f(x_i) \\ &\le f(x_n) + (n-1)\cdot f(\bar{x}) = n\cdot\left(\frac{1}{n}\cdot f(x_n) + \frac{n-1}{n}\cdot f(\bar{x})\right) \\ &\le n\cdot f(x^*)\,,\end{aligned}$$

hvor den første ulighed bruger induktionshypotesen og den anden ulighed bruger definitionen af begrebet »konkav«, med $x = x_n$, $y = \bar{x}$ og $t = 1/n$. Udvidelsen til konvekse funktioner følger umiddelbart, idet $-f$ er konkav, når f er konveks.

Litteratur

1. A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.
2. R. K. Ahuja, R. L. Magnanti, and J. B. Orlin. *Network Flows*. Prentice Hall, 1993.
3. G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer, 1999.
4. R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.
5. J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, pages 643–647, 1979.
6. Boost.org. Boost C++ Libraries. www.boost.org.
7. M. R. Brown and R. E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal of Computing*, 9:594–614, 1980.
8. E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: A survey. In D. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pages 46–93. PWS, 1997.
9. S. A. Cook. *On the Minimum Computation Time of Functions*. PhD thesis, Harvard University, 1966.
10. S. A. Cook. The complexity of theorem proving procedures. In *3rd ACM Symposium on Theory of Computing*, pages 151–158, 1971.
11. A. De, P. P. Kurur, C. Saha, and R. Saptharishi. Fast integer multiplication using modular arithmetic. *SIAM Journal on Computing*, 42(2):685–699, 2013.
12. M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.
13. L. Devroye. A note on the height of binary search trees. *Journal of the ACM*, 33:289–498, 1986.
14. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
15. R. Floyd. Assigning meaning to programs. In J. Schwarz, editor, *Mathematical Aspects of Computer Science*, pages 19–32. AMS, 1967.
16. L. R. Ford. Network flow theory. Technical Report P-923, Rand Corporation, Santa Monica, California, 1956.
17. M. Fürer. Faster integer multiplication. *SIAM Journal on Computing*, 39(3):979–1005, 2009.

18. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
19. G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures: In Pascal and C*. Addison-Wesley, 2nd edition, 1991.
20. M. T. Goodrich and R. Tamassia. JDSL – the data structures library in Java. www.jdsl.org.
21. R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 2nd edition, 1994.
22. J. Hartmanis and J. Simon. On the power of multiplication in random access machines. In *5th IEEE Symposium on Foundations of Computer Science*, pages 13–23, 1974.
23. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–585, 1969.
24. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
25. R. D. Hofstadter. Metamagical themas. *Scientific American*, pages 16–22, January 1983.
26. J. E. Hopcroft and J. D. Ullman. Set merging algorithms. *SIAM J. Comput.*, 2(4):294–303, 1973.
27. S. Huddlestone and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.
28. V. Jarník. O jistém problému minimálním. *Práca Moravské Přírodovědecké Společnosti*, 6:57–63, 1930.
29. K. Jensen and N. Wirth. *Pascal User Manual and Report: ISO Pascal Standard*. Springer, 1991.
30. A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7(7):595–596, 1963.
31. J. Katajainen and B. B. Mortensen. Experiences with the design and implementation of space-efficient deques. In *Workshop on Algorithm Engineering*, volume 2141 of Lecture Notes in Computer Science, pages 39–50. Springer, 2001.
32. D. E. Knuth. *MMIXware: A RISC Computer for the Third Millennium*, volume 1750 of Lecture Notes in Computer Science. Springer, 1999.
33. J. Kruskal. On the shortest spanning subtree of a graph and the travelling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.
34. LEDA (Library of Efficient Data Types and Algorithms). `www.algorithmic-solutions.com`.
35. L. Levin. Universal search problems. *Problemy Peredachi Informatsii*, 9(3):265–266, 1973.
36. K. Mehlhorn. Amortisierte Analyse. In T. Ottmann, editor, *Prinzipien des Algorithmenentwurfs*, pages 91–102. Spektrum Lehrbuch, 1998.
37. K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
38. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.
39. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
40. S. Näher and O. Zlotowski. Design and implementation of efficient data types for static graphs. In *10th European Symposium on Algorithms*, volume 2461 of Lecture Notes in Computer Science, pages 748–759. Springer, 2002.
41. W. W. Peterson. Addressing for random access storage. *IBM Journal of Research and Development*, 1(2), Apr. 1957.

42. J. Pinkerton. *Voyages and Travels*, volume 2. 1808.
43. P. J. Plauger, A. A. Stepanov, M. Lee, and D. R. Musser. *The C++ Standard Template Library*. Prentice Hall, 2000.
44. R. C. Prim. Shortest connection networks and some generalizations. *Bell Systems Technical Journal*, pages 1389–1401, Nov. 1957.
45. A. Schönhage. Storage modification machines. *SIAM Journal on Computing*, 9:490–508, 1980.
46. A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.
47. R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, 1996.
48. R. Seidel and M. Sharir. Top-down analysis of path compression. *SIAM Journal of Computing*, 34(3):515–525, 2005.
49. J. C. Shepherdson and H. E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10(2):217–255, 1963.
50. M. Sipser. *Introduction to the Theory of Computation*. MIT Press, 1998.
51. D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.
52. D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.
53. R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.
54. R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.
55. A. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics Doklady*, 150(3):496–498, 1963.
56. J. von Neumann. First draft of a report on the EDVAC. Technical report, University of Pennsylvania, 1945.
57. L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly, 3rd edition, 2000.
58. I. Wegener. *Complexity Theory: Exploring the Limits of Efficient Algorithms*. Springer, 2005.
59. R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley, 1995.
60. J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.

Indeks