KAPITTEL 3

Litt logikk og noen andre småting

Logikk er sentralt både i matematikk og programmering, og en innføring i de enkleste delene av logikken er hovedtema i dette kapitlet. I tillegg ser vi litt på sammenhengen mellom induksjonsbevis og det som kalles rekursiv programmering.

3.1 Logikk

I dagligtale kommer vi med mange slags utsagn, og en del er forholdsvis presise slik at vi kan vurdere om de er sanne eller ikke. For eksempel er det nokså universell enighet om at utsagnene 'Oslo er hovedstaden i Norge' og 'Jorda er en del av universet' er sanne, mens det er noe større uenighet om utsagnet 'Pannekaker er det beste som fins' er sant.

Matematikken er bygd opp av presise logiske utsagn som er satt sammen i logiske resonnementer. I matematikk er et logisk utsagn presist, og som regel bare interessant dersom vi er i stand til å avgjøre om det er riktig eller ikke.[1] Hvis vi begrenser oss til utsagn som opplagt er sanne eller gale kan vi angi sannhetsgehalten med to verdier, for eksempel 0 (gal) og 1 (sann). Dessuten kan logiske utsagn kombineres med logiske operatorer som 'og' og 'eller'. For eksempel kan de to sanne utsagnene 'π er større enn 3' og 'π er mindre enn 4' kombineres med 'og' til det sanne utsagnet 'π er større enn 3 og π er mindre enn 4' som forenklet kan skrives 'π ligger mellom 3 og 4'.

Litt kunnskap om logikk er også nyttig i forbindelse med programmering. I programmeringsspråk har vi tilgang til if-tester og ulike slags løkker for å styre programutviklingen. En if-test er basert på et logisk utsagn som det kan avgjøres om er sant eller galt, og en løkke vil bli utført inntil et logisk utsagn ikke lenger er sant.

3.1.1 Logiske variable og sammenligninger

En grunnleggende funksjonalitet i datamaskinens elektronikk består i å kunne avgjøre om enkle utsagn om heltall og flyttall er sanne, og støtte for dette er bygd inn i så og si

[1] En viktig milepæl innen matematisk logikk var oppdagelsen til den tyske matematikeren Kurt Gödel (1906–1978) i 1931 at det fins utsagn innen aritmetikken (læren om tallene) som det er umulig å avgjøre om er sanne eller ikke. Slike utsagn kan sammenlignes med at en nordmann uttaler at 'Alle nordmenn lyver'.

alle programmeringsspråk. Hvis a og b er to tall kan maskinen kjapt svare ja eller nei på om relasjoner som

$$a < b, \quad a \leq b, \quad a = b, \quad a \geq b, \quad a > b,$$

holder, og ved hjelp av logiske variable kan vi ta vare på resultatet av en slik test. I Java deklareres logiske variable som `boolean`[2], mens 'sann' og 'gal' angis med de to logiske konstantene `true` og `false`. Hvis vi lar `p` være en variabel av type `boolean` kan vi gjøre tilordningen `p = 2<3`. Siden det er sant at 2 er mindre enn 3 vil `p` få verdien `true`, mens tilordningen `p = 2==3` vil føre til at `p` får verdien `false` (i Java er det operatoren `==` som tester likhet, mens operatoren `!=` tester ulikhet mellom tall.)

Direkte sammenligning av to tall slik som i `p = 2<3` er sjelden aktuelt. Siden vi vet at 2 er mindre enn 3 er dette det samme som å sette `p = true`. Det vanlige er å teste på relasjoner mellom variable. Hvis `a` og `b` er to variable som inneholder tall vil tilordningen `p = a<b` føre til at `p` får verdien `true` hvis tallet som er lagret i `a` er mindre enn tallet som er lagret i `b`, mens `p` vil få verdien `false` i alle andre tilfeller.

Ofte lagres ikke resultatet av et evaluert logisk uttrykk i en variabel, men brukes i stedet til å bestemme valg i programmet i forbindelse med `if`-tester og `while`-løkker. I Java vil for eksempel kodebiten

```
maxab = if (a<b) b
            else a;
```

føre til at det største av de to tallene lagret i `a` og `b` lagres i `maxab`, mens kodebiten

```
for (n=0; n<10; n++) {
        .
        .
        .
  }
```

gir en løkke som vil bli utført inntil det logiske uttrykket `n<10` ikke lenger er sant, altså inntil `n` blir 10. Dette betyr at løkka vil bli utført med `n=0, 1, 2, ..., 9`.

I diskusjonene som kommer under er det viktig å være klar over at et uttrykk som $a > 0$ i logisk sammenheng bare gir mening når a erstattes med et tall. Siden a kan anta alle mulige reelle verdier betegner derfor ulikheten $a > 0$ uendelig mange logiske uttrykk. Noen av disse uttrykkene er sanne, for eksempel de vi får når a erstattes med 1 eller π, mens noen ikke er sanne, for eksempel de vi får når a erstattes med 0 eller -1. Lar vi derimot bare a være et symbol kan vi ikke avgjøre om $a > 0$ er sant eller ikke. Det er også verdt å legge merke til at uansett hva a er vil alltid utsagnet $a \leq 0$ være det motsatte av $a > 0$ i den forstand at hvis det ene er sant vil det andre ikke være sant og omvendt.

3.1.2 Grunnleggende logiske operatorer

Som vi nevnte i innledningen setter vi ofte sammen logiske utsagn ved hjelp av 'og ' og 'eller', både i dagligtale og matematikk. Dette kan vi også gjøre når vi programmerer. De

[2] George Boole (1815–1864) var en engelsk matematiker som i 1854 publiserte en bok om logikk og hvordan man kan gjøre beregninger med logiske utsagn.

tre grunnleggende operasjonene som vi kan anvende på logiske uttrykk er 'ikke' (NOT), 'og' (AND) samt 'eller' (OR), de er *logiske operatorer*. Disse operatorene uttrykkes på forksjellige måter i ulike programmeringsspråk. En del språk bruker de engelske termene i parentes mens Java bruker `!` (NOT), `&` (AND) og `|` (OR). Når vi ikke eksplisitt beskriver programkode vil vi bruke de vanlige matematiske betgnelsene som er $\neg$ (NOT), $\wedge$ (AND) og $\vee$ (OR).

Operatoren $\wedge$ kombinerer to logiske utsagn p og q,

$$p \wedge q,$$

og kombinasjonen er sann hvis både p og q er sanne, i alle andre tilfeller er kombinasjonen ikke sann. Dette tilsvarer bruken av 'og' som logisk sammenbindingsord i dagligtale. Hvis a er et reelt tall er for eksempel uttrykket

$$(a > 0) \wedge (a < 1) \tag{3.1}$$

sant hvis a er et tall som ligger mellom 0 og 1, men ikke sant ellers.

Som ellers i matematikk har vi brukt parenteser i (3.1) for å gjøre betydningen klar. Når vi programmerer kan vi med fordel bruke parenteser på samme måte, både for å gjøre betydningen klar for oss selv og andre som skal lese programmet, og for å være sikre på at maskinen tolker programmet på riktig måte.

Den andre grunnoperatoren er $\vee$. Kombinasjonen

$$p \vee q$$

er ikke sann hvis hverken p eller q er sanne, i alle andre tilfelle er kombinasjonen sann. Med andre ord er det nok at en av p eller q er sanne for at $p \vee q$ skal være sann. Et eksempel er

$$(a > 2) \vee (a < 1),$$

der a er et reelt tall. Dette uttrykket vil ikke være sant hvis a ligger mellom 1 og 2, altså hvis $1 \leq a \leq 2$; i alle andre tilfeller vil minst en av de to ulikhetene være sanne og derfor gjøre hele uttrykket sant. Et annet eksempel er

$$(a > 0) \vee (a^2 > 0). \tag{3.2}$$

Vi ser at minst en av de to ulikhetene alltid vil være sanne, bortsett fra når a er null. Kombinasjonen er derfor sann for alle verdier av a, unntatt $a = 0$.

Denne tolkningen av 'eller' er litt annerledes enn den vi bruker i dagligtale. Hvis noen sier at 'huset var gult eller hvitt' vil det vanligvis ikke kunne bli tolket som at huset hadde begge farver. I vår matematiske språkbruk derimot, er $p \vee q$ sann hvis både p og q er sanne. I uttrykket i (3.2) er for eksempel begge ulikhetene tilfredstilt for $a > 0$ slik at kombinasjonen er sann. Med tolkningen fra dagligtale vil noen muligens synes at uttrykket burde være usant når $a > 0$.

Det fins en annen logisk operator med akkurat denne tolkningen, nemlig 'eksklusiv eller' (exclusive or, XOR). Denne operatoren har ingen standard betegnelse i matematikk,

men vi skal bruke notasjonen $\hat{\vee}$, mens Java bruker notasjonen `^`. Uttrykket $p \mathbin{\hat{\vee}} q$ (som altså skrives `p^q` i Java) vil dermed være sant når en av p og q er sanne, men ikke sant hvis p og q er like (enten begge sanne eller begge ikke sanne).

Som et eksempel på 'eksklusiv eller' kan vi se på uttrykket i (3.2) med $\vee$ erstattet med $\hat{\vee}$,

$$(a > 0) \mathbin{\hat{\vee}} (a^2 > 0).$$

Dette vil være sant hvis $a < 0$ for da er høyresiden av $\hat{\vee}$ sann mens venstresiden ikke er sann. For $a < 0$ er begge sidene sanne så uttrykket er ikke sant, mens for $a = 0$ er ingen av sidene sanne så uttrykket er ikke sant.

Den fullstendige oppførselen til de tre logiske operatorene AND, OR og XOR er oppsummert i tabell 3.1 som kalles en *sannhetstabell*. Merk at verdien 'sann' her er angitt med 1 mens 'usann' er angitt med 0.

p	q	$p \wedge q$	$p \vee q$	$p \mathbin{\hat{\vee}} q$
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	1	0

Tabell 3.1. Fullstendig beskrivelse av de tre logiske operatorene AND, OR og XOR.

Den siste logiske operatoren vi skal ta for oss er negasjon. Når denne operatoren settes foran et utsagn reverseres betydningen til det motsatte. For eksempel er negasjonen av 'bilen er rød' utsagnet 'bilen er ikke rød', men det er verd å merke seg at hvis vi vet at vi bare har med røde og blå biler å gjøre så vil det siste utsagnet kunne skrives om til 'bilen er blå'.

I matematikk er negasjon ofte angitt med $\neg$, mens negasjon i Java angis med `!`. Uannsett hva a er så er utsagnet $\neg(a = 0)$ det samme som $a \neq 0$. Hvis a kan være et vilkårlig reelt tall er $\neg(a > 0)$ det samme som $a \leq 0$, men legg merke til at hvis vi begrenser oss til bare å se på positive heltall er $\neg(a > 0)$ det samme som å si at 'a eksisterer ikke'.

Så langt har vi bare sett på logiske uttrykk som kombinerer to utsagn slik som i $p \wedge q$. Vi kan kombinere flere utsagn slik som i

$$(p \wedge q) \wedge r, \tag{3.3}$$

$$(p \vee q) \vee (r \vee s), \tag{3.4}$$

der p, q, r og s alle er logiske utsagn (vi skal se på blandede uttrykk senere). Her har vi satt inn parenteser siden vi så langt bare har klare regler for hvordan vi kan kombinere to logiske utsagn. Men det er ikke vanskelig å se at verdien av uttrykkene i (3.3) og (3.4) er uavhengig av hvordan parentesene er plassert. Det første uttrykket (3.3) er sant bare når p, q og r alle er sanne, uansett hvordan uttrykket utstyres med parenteser, mens det andre uttrykket (3.4) alltid er sant bortsett fra i det ene tilfellet der ingen av utsagnene

p, q, r og s er sanne. Vi kan derfor fjerne parentesene og skrive

$$p \land q \land r,$$
$$p \lor q \lor r \lor s,$$

uten noen fare for misforståelse. Det er mange matematiske operasjoner som har denne egenskapen, for eksempel addisjon og multiplikasjon, og den har derfor fått et eget navn; vi sier at operatorene $\land$ og $\lor$ er *assosiative*. Assosiativiteten gjelder også når vi har lengre lenker med logiske uttrykk: hvis alle operatorene er enten $\land$ eller $\lor$ blir resultatet uavhengig av rekkefølgen vi bruker operatorene.

Hva med den tredje logiske operatoren $\hat{\lor}$, er den assosiativ? Hvis vi ser på de to uttrykkene

$$(p \mathbin{\hat{\lor}} q) \mathbin{\hat{\lor}} r, \qquad p \mathbin{\hat{\lor}} (q \mathbin{\hat{\lor}} r),$$

så er spørsmålet om de alltid er like. Ved å sjekke alle mulige kombinasjoner og sette opp en sannhetstabell vil vi se at de to uttrykkene er like og derfor at 'eksklusiv eller' også er en assosiativ logisk operator. Assosiativiteten gjelder også for lengre lenker med $\hat{\lor}$, men en generell beskrivelse av $\hat{\lor}$ er litt mer komplisert enn for $\land$ og $\lor$. Det viser seg at hvis vi har en lang rekke med logiske utsagn lenket sammen med $\hat{\lor}$ så er verdien sann hvis antall sanne utsagn er et odde tall og ikke sann hvis antall sanne utsagn er et partall.

La oss til slutt i denne seksjonen nevne en opplagt egenskap ved alle de tre logiske operatorene $\land$, $\lor$ og $\hat{\lor}$, de er alle *kommutative*. Dette betyr at rekkefølgen av de logiske utsagnene som bindes sammen med operatorene er likegyldig. For eksempel har vi helt opplagt $p \land q = q \land p$ og $p \lor q \lor r = q \lor r \lor p$. Denne egenskapen er det også mange andre matematiske operatorer som har, for eksempel multiplikasjon og addisjon, mens divisjon ikke er kommutativ.

3.1.3 Logisk aritmetikk

Det er ofte behov for å kombinere logiske operatorer. Anta for eksempel at vi skal sjekke om heltallet n er et primtall. En enkel måte å gjøre dette på er å teste om n er delelig med noe tall mindre enn seg selv. I mange programmeringsspråk fins det en egen operator som gir resten ved divisjon av to heltall, og i Java er denne operatoren gitt ved `%` slik at resten i divisjonen n/i kan finnes ved operasjonen `n % i`. I matematikk finnes det ingen standard notasjon for resten ved heltallsdivisjon så vi stjeler fra Java og skriver $n \,\%\, i$. For å løse problemet lar vi en variabel i løpe gjennom alle tall fra 2 opp til n og sjekke om resten ved divisjon av n med i blir 0, noe vi kan gjøre med testen $n \,\%\, i = 0$. Hvis resten blir 0 for en i kan vi stoppe, siden n da er delelig med i og følgelig ikke kan være noe primtall. Dessuten trenger vi ikke sjekke resten for de i som tilfredstiller $i^2 > n$ (prøv med $n = 100$ så ser du hvorfor). Vi skal altså teste stadig økende verdier av i inntil en eller begge de to testene $n \,\%\, i = 0$ og $i^2 > n$ slår til. Men dette svarer til at vi skal forsøke nye verdier av i inntil det logiske uttrykket

$$(n \,\%\, i = 0) \lor (i^2 > n) \tag{3.5}$$

slår til. Når dette skjer kan vi så ved en enkel if-test finne ut hvilken av de to testene i (3.5) det var som slo til og dermed om n er et primtall eller ikke.

Den naturlige måten å implementere det ovenstående er ved en løkke, for eksempel en while-løkke,

```
i=2;
while (?)  i += 1;
if (n%i != 0) then println(n + "er et primtall");
```

(Husk at i 'ekte' Javakode må du fortelle hvilken klasse `println` hører til slik at den fullstendige utskriftssetningen vil være

```
system.out.println(n + "er et primtall")
```

men slike detaljer tar vi ikke med i vår kode.) Men hva skal spørsmålstegnet erstattes med? Det logiske uttrykket i (3.5) gir beskjed om når vi skal stoppe, mens spørsmålstegnet skal erstattes med et logisk uttrykk som forteller oss når vi skal fortsette, altså negasjonen av uttrykket i (3.5). Litt ettertanke viser at denne negasjonen, $\neg\big((n \,\%\, i = 0) \lor (i^2 > n)\big)$, er det samme som

$$(n \,\%\, i \neq 0) \land (i^2 \leq n).$$

Kodebiten som sjekker om n er et primtall kan da uttrykkes som

```
i=2;
while ((n%i != 0) & (i*i <= n))
  i += 1;
if (n%i != 0) then
  println(n + " er et primtall")
else
  println("Den minste faktoren i " + n + " er " +i);
```

Som vi skal se etterhvert er det mange matematiske problemer som kan løses med løkker på denne formen, og ofte er det mer naturlig å formulere et stoppkriterium enn en betingelse for når løkka skal fortsette. Men siden løkka skal fortsette akkurat når den ikke skal stoppe er den ene betingelsen negasjonen av den andre. Og siden det fins presise regler for hvordan negasjonen av et logisk uttrykk kan bestemmes er det som regel enkelt å komme fra stoppbetingelsen til løkkebetingelsen.

Det logiske uttrykket (3.5) er på formen $p \lor q$, mens det vi trengte i while-løkka var negasjonen $\neg(p \lor q)$. Hvis vi ser på kodebiten over så brukte uttrykket $\neg p \land \neg q$ for å styre for-løkka, så hvis dette skal være riktig må

$$\neg(p \lor q) = (\neg p) \land (\neg q). \tag{3.6}$$

Den enkleste måten å sjekke dette er ved en sannhetstabell som i tabell 3.1, men det er heller ikke så vanskelig å se direkte.

I andre sammenhenger kan det det være aktuelt å ta negasjonen av et uttrykk som $p \land q$. Vi finner på samme måte at

$$\neg(p \land q) = (\neg p) \lor (\neg q). \tag{3.7}$$

De to reglene (3.6) og (3.7) kalles de Morgans[3] lover.

Det fins en masse andre regneregler for logiske operatorer som vi ikke kan komme inn på her. Men hvis du i en eller annen sammenheng blir sittende å manipulere større logiske uttrykk kan det være lurt å konsultere en ‘logisk formelsamling’.

3.1.4 Logikk og mengdelære

Det er en nær sammenheng mellom logikk og mengdelære som vi bare kort skal minne om her. Anta at vi har to mengder A og B, for å være konkrete kan vi tenke oss to delmengder av planet. Unionen av A og B som skrives $A \cup B$, er mengden vi får ved å ta med alle elementer som ligger i enten A eller B eller begge steder. En vanlig måte å beskrive denne unionen på er ved notasjonen

$$A \cup B = \{x \mid x \in A \vee x \in B\}.$$

Vi ser at det er en klar sammenheng mellom den logiske operatoren ‘or’ og mengdeoperasjonen ‘union’.

Mengdeoperasjonen ‘snitt’ og den logiske operatoren ‘og’ hører sammen på samme måte. Snittet mellom A og B er mengden som inneholder de elementene som ligger i både A og B,

$$A \cap B = \{x \mid x \in A \wedge x \in B\}.$$

Mengdeoperasjonen ‘komplement’ er relatert til den logiske operatoren ‘negasjon’ slik at komplementet til A kan skrives

$$\overline{A} = \{x \mid \neg(x \in A)\}.$$

Uttrykket $\neg(x \in A)$ skrives som regel som $x \notin A$. Legg merke til at denne operasjonen ikke er helt opplagt. Siden mengden A er gitt er det klart hva den inneholder, men det er ikke opplagt hva som ikke ligger i A. Vi har vært konkrete og sagt at A er en del av planet, så da må komplementet til A være den delen av planet som ikke ligger i A. Hvis A betegner alle de reelle tallene i intervallet $[0, 1]$ vil vanligvis $\overline{A}$ betegne de reelle tallene som ikke ligger i dette intervallet. Men hvis vi ikke er interessert i negative tall vil komplementet til A betegne alle reelle tall som er større enn 1. Moralen er at i mengdelære er det viktig å spesifisere hva ‘universet’ er, en ‘supermengde’ som inneholder alle objektene som er interessante i den sammenhengen vi opererer i.

Den logiske operatoren $\hat{\vee}$ gir også opphav til en mengdeoperasjon, nemlig mengden av de elementene som ligger i en av mengdene A eller B, men ikke i begge,

$$\{x \mid x \in A \mathbin{\hat{\vee}} x \in B\}.$$

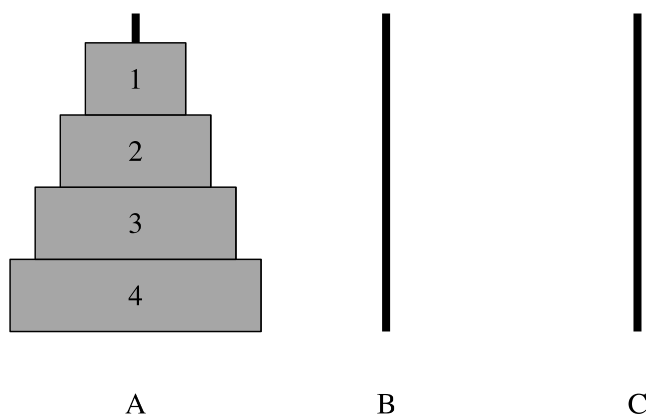
Denne operatoren har ingen standard notasjon i matematikk.

[3] Augustus De Morgan var (1806–1871) var en englesk matematiker. De Morgan var også den som først brukte begrepet *matematisk induksjon* og la en rigorøs basis for denne bevismetoden.

3.2 Induksjonsbevis og rekursjon

Til å begynne med kan matematiske bevis synes ugjennomtrengelige og mystiske, og for mange er det mest mystiske av alt induksjonsbevis. På den annen side er det kravet om at alt skal bevises som gjør matematikken uangripelig, og induksjonsbevis er i mange tilfeller både elegante og kortfattede. Her skal vi se litt på en nær slektning av induksjonsbeviset, nemlig rekursiv programmering.[4] Dette kan være en meget slagkraftig og elegant måte å løse et programmeringsproblem på, men til å begynne med kan rekursjon, som induksjon, virke svært mystisk. Men et rekursivt program er, i motsetning til et induksjonsbevis, noe konkret som det går an å se hvordan oppfører seg. Forhåpentligvis er det derfor ikke så vanskelig å se hvordan rekursjon fungerer, og denne forståelsen bør også kunne kaste lys over induksjonsprinsippet.

Vi skal se på ett eksempel på rekursiv programmering. Problemet vi skal løse er en gammel nøtt kalt *Hanois tårn*. Vi har gitt tre tårn A, B, og C, og på tårn A ligger det n ringer, alle av ulik størrelse, med den største nederst og så i avtagende størrelse oppover med den minste øverst. Problemet består i å flytte ringene fra tårn A til tårn B ved å bruke tårn C til mellomlagring, men uten noen gang å legge en større ring oppå en mindre. Utgangsposisjonen er vist i figur 3.1 for $n = 4$.



Figur 3.1. Utgangsposisjonen i Hanois tårn med 4 ringer.

Vi skal skrive en prosedyre `hanoi(n,X,Y,Z)` for å vise hvordan problemet kan løses. Her har vi kalt tårnene X, Y og Z siden problemet er det samme uansett hvilke tre tårn vi flytter mellom. Vi tenker oss altså at `hanoi(n,X,Y,Z)` løser problemet med å flytte n ringer fra tårn X til tårn Y med Z som hjelpetårn. Under løsningsprosessen vil X, Y og Z være forskjellige ordninger av tårnene A, B og C, og under skal vi se at for å løse problemet med n ringer må vi flytte $n - 1$ ringer fra A til C med B som hjelpetårn og fra C til B med A som hjelpetårn.

La oss forsøke å løse problemet ved å tenke induktivt. Problemet er lett å løse hvis vi

[4] Merk at rekursiv programmering *ikke* gjennomgås i grunnkurset i programmering ved Institutt for informatikk, men er tema i et videregående informatikkurs.

ikke har noen ringer, da gjør vi ingen ting. La oss nå tenke oss at vi har løst problemet med $n-1$ ringer med en prosedyre `hanoi(n-1,X,Y,Z)` som flytter ringene fra tårn X til tårn Y ved å bruke tårn Z til mellomlagring. Utfordringen er nå å utnytte at vi kan løse problemet med $n-1$ ringer til å løse problemet med n ringer. Dette er helt i tråd med induksjonsbevis: Skal vi vise påstanden $P(n)$ sjekker vi først at $P(1)$ er riktig. Vi antar så at $P(k)$ gjelder for $k \leq n-1$ og viser ved hjelp av dette at $P(n)$ også er riktig.

For å være konkrete setter vi $n = 4$ slik at utgangspunktet er som i figur 3.1 med (X,Y,Z)=(A,B,C). Induksjonshypotesen vår er at vi vet hvordan vi skal løse problemet med 3 ringer. Dette utnytter vi på følgende måte: For å løse problemet med 4 ringer kan vi først flytte de 3 øverste ringene fra A til C med B som hjelpetårn (vi har jo antatt at vi vet hvordan dette skal gjøres). Den siste ringen på tårn A flytter vi så til tårn B, og så flytter vi til slutt de 3 ringene på tårn C til tårn B med tårn A som hjelpetårn. Denne framgangsmåten virker fornuftig også for generell n, og algoritmen ser da ut som følger, med X, Y og Z som navn på tårnene.

Algoritme 3.1. *Koden under gir løsningen på problemet med Hanois tårn og flytter n ringer fra tårn X til tårn Y med tårn Z som hjelpetårn:*

```
hanoi(n,X,Y,Z)
  int n;
  char X, Y, Z;
  if (n>0) {
    hanoi(n-1,X,Z,Y);
    println("Ring " + n + " fra " + X + " til " + Y);
    hanoi(n-1,Z,Y,X);
  }
```

For å løse problemet med 4 ringer kan vi nå i hovedprogrammet gjøre metodekallet `hanoi(4,´A´,´B´,´C´)`. Dette vil produsere følgende løsning:

```
Ring 1 fra A til C                           Ring 1 fra C til B
Ring 2 fra A til B                           Ring 2 fra C til A
Ring 1 fra C til B                           Ring 1 fra B til A
Ring 3 fra A til C     Ring 4 fra A til B    Ring 3 fra C til B
Ring 1 fra B til A                           Ring 1 fra A til C
Ring 2 fra B til C                           Ring 2 fra A til B
Ring 1 fra A til C                           Ring 1 fra C til B
```

Den første kolonnen med trekk er resultatet av `hanoi(3,A,C,B)` som svarer til kallet `hanoi(n-1,X,Z,Y)` i algoritme 3.1 i dette tilfellet, mens trekkene i den siste kolonnen er resultatet av `hanoi(3,C,B,A)` som svarer til kallet `hanoi(n-1,Z,Y,X)` i algoritme 3.1. Trekket i midten er det eksplisitte trekket som er gitt i algoritme 3.1.

Algoritme 3.1 løser problemet med Hanois tårn, men ser noe underlig ut siden prosedyren kaller seg selv; vi har jo ikke skrevet `hanoi(n-1,X,Y,Z)`. Men det er nettopp det vi har gjort! Ved å skrive `hanoi(n,X,Y,Z)` har vi også skrevet `hanoi(n-1,X,Y,Z)` siden

`n`, `X`, `Y` og `Z` er variable som kan anta vilkårlige verdier. Starten på induksjonen er implementert ved at vi ikke gjør noen ting om ikke n er positiv, mens vi for positive verdier av n utnytter induksjonshypotesen til å løse problemet. Vi kan derfor faktisk bevise at kodebiten over gir oss løsningen på problemet med Hanois tårn: Prosedyren gir opplagt løsningen på problemet når $n = 0$, og hvis vi antar at den gir løsningen med $n - 1$ ringer (induksjonshypotesen) gir den også løsningen med n ringer.

En prosedyre som kaller seg selv på denne måten sies å være *rekursiv*, og rekursiv programmering er tillatt i de fleste språk, inklusiv Java (bortsett fra at vi selvsagt må legge til noen deklarasjoner og annen kosmetikk).

Rekursjon kan være en slagkraftig og elegant måte å løse problemer på, men det er på langt nær alle problemer som lar seg løse på denne måten. For at rekursjon skal fungere må vi ha en hel familie av problemer slik at hver heltallig verdi av en parameter k gir opphav til et problem $P(k)$ som vi tenker oss har en løsning $L(k)$. I problemet med Hanois tårn er da $P(k)$ problemet med k ringer, mens $L(k)$ er løsningen i form av prosedyren vi skrev. Typisk vil $P(0)$ være et enkelt problem slik at løsningen $L(0)$ også er enkel, mens kompleksiteten i problemene så øker raskt med k. Det 'magiske trikset' består i å utnytte de tidligere løsningene $L(0)$, $L(1)$, …, $L(k-1)$ til å løse $P(k)$ med $L(k)$. I problemet med Hanois tårn bruker vi for eksempel $L(k-1)$ to ganger for å løse $L(k)$. Hvis vi skulle fylt inn den eksplisitte løsningen med $n - 1$ ringer på de to stedene i løsningen av Hanois tårn ville `hanoi(n,X,Y,Z)` blitt kompleks og uoversiktlig, mens det blir svært enkelt når vi kan bruke 'forkortelsene' `hanoi(n-1,X,Z,Y)` og `hanoi(n-1,Z,Y,X)`. Det flotte er at de fleste programmeringsspråk tillater slike konstruksjoner, som altså er kjent som rekursiv programmering.

Oppgaver

3.1 Bruk de Morgans lover og ekspander ut uttrykkene under.

a) $\neg((i > 0) \vee (i < 10))$ når i er heltallig.

b) $\neg((x > 0) \wedge (x < 1) \wedge (i = 10))$ når x er reell og i er heltallig.

c) $\neg((n = 1) \vee (n = 2) \vee (n = 3))$ når n er et naturlig tall som er mindre enn 4.

d) $\neg((x > 0) \wedge (x < 2) \wedge (x = 1))$ når x er et reelt tall.

3.2 Etabler de Morgans lover (3.6) og (3.7) ved å sette opp sannhetstabeller tilsvarende tabell 3.1.

3.3 Skriv en rekursiv prosedyre for å beregne $n! = 1 \cdot 2 \cdot 3 \cdots n$. Merk at i dette tilfellet vil en rekursiv prosedyre bruke mye lenger tid enn den naturlige løsningen med en enkel løkke.

3.4 I denne oppgaven skal vi se litt nærmere på den rekursive prosedyren for å løse problemet med Hanois tårn.

a) Finn løsningen på problemet med Hanois tårn når $n = 3$. Forsøk først å finne

løsningen ved prøving og feiling og sammenlign etterpå ved å gå i gjennom stegene som kallet `hanoi(3,A,B,C)` vil generere.

b) Programmer prosedyren `hanoi` og generer løsningen på problemet for $n = 4$ og $n = 5$.

3.5 I denne oppgaven skal vi se på noen egenskaper ved $\hat{\vee}$ operatoren.

a) Først skal vi verifisere at $\hat{\vee}$ er assosiativ når den binder sammen tre logiske utsagn. Gjør dette ved å sette opp en sannhetstabell for de to utrykkene $(p \hat{\vee} q) \hat{\vee} r$ og $p \hat{\vee} (q \hat{\vee} r)$ tabell 3.1,

p	q	r	$(p \hat{\vee} q) \hat{\vee} r$	$p \hat{\vee} (q \hat{\vee} r)$
0	0	0	0	0
1	0	0	1	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Fyll inn de linjene som mangler.

b) Med fire logiske utsagn må vi sjekke om de fire uttrykkene

$$(p \hat{\vee} q) \hat{\vee} r \hat{\vee} s, \qquad p \hat{\vee} (q \hat{\vee} r) \hat{\vee} s, \qquad p \hat{\vee} q \hat{\vee} (r \hat{\vee} s) \tag{3.8}$$

er like. Dette kan vi gjøre ved å utnytte assosiativiteten når vi har to eller tre utsagn. For eksempel har vi

$$(p \hat{\vee} q) \hat{\vee} r \hat{\vee} s = (p \hat{\vee} q) \hat{\vee} (r \hat{\vee} s) = p \hat{\vee} q \hat{\vee} (r \hat{\vee} s).$$

Bruk et lignende resonnement til å vise at alle de tre utrykkene i (3.8) er like.

c) Forklar hvordan argumentet over kan utnyttes til å etablere assosiativiteten i det generelle tilfellet.

d) Som nevnt i teksten er verdien av et logisk uttrykk med en rekke av utsagn bundet sammen av $\hat{\vee}$ avhengig av antall utsagn som er sanne: Hvis antall sanne utsagn er et partal er det totale logiske utsagnet ikke sant mens hvis antall sanne utsagn er et odde tall er det totale utsagnet sant. For å vise dette er det lurt å utnytte at $\hat{\vee}$ er kommutativ og sette alle utsagnene som ikke er sanne først. Gjør dette og sjekk at den generelle regelen gjelder.