Kompendium til TOD065 - Diskret matematisk programmering

Jon Eivind Vatne
Institutt for data- og realfag, HiB,
Tlf: 55587112,
Mob: 90203117,
jev@hib.no

27. oktober 2011

Introduksjon

Emnet vårt tar for seg programmering i matematikk. Vi skal lære mye om å representere tallverdier og andre matematiske objekter på datamaskinen, og vi skal programmere mange matematiske algoritmer. Dette kompendiet vil utvikles utover høsten, og de forskjellige kapitlene vil komme i den rekkefølgen vi kommer til stoffet i forelesningene.

Vi skal også bruke pensumlitteraturen i programmeringsfaget og matematikkfaget; det du leser nå er tillegg og forelesningsnotater.

Innhold

Kapittel 1

Heltall 1

Vi starter med å forsøke å forstå hele tall, hva vi vil gjøre med hele tall, og hvordan datamaskinen oppfatter hele tall.

1.1 Hva er et helt tall?

Det er vanskeligere enn man tror å gi en presis matematisk definisjon av hva et heltall er, men for våre behov er en intuitiv oppfatning nok.

Vi opererer med forskjellige matematiske begreper om hele tall:

Et positivt heltall er et av tallene

$$\{1, 2, 3, 4, 5, \cdots\}.$$

Et ikkenegativt heltall er et av tallene

$$\{0, 1, 2, 3, 4, 5, \cdots\}.$$

Et negativt heltall er et av tallene

$$\{-1, -2, -3, -4, -5, \cdots\}.$$

Et ikkepositivt heltall er et av tallene

$$\{0, -1, -2, -3, -4, -5, \cdots\}.$$

Et heltall er et av tallene

$$\{\cdots, -4, -3, -2, -1, 0, 1, 2, 3, 4, \cdots\}$$

Tegnet $\cdots$ markerer at vi fortsetter (i det uendelige) videre fra det som er skrevet helt ut.

Når vi skriver heltall, bruker vi (vanligvis) titallsystemet. Da bruker vi sifrene

$$0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \ ,$$

og posisjonen til hvert siffer, for å angi heltall. For eksempel er

$$240036$$

en forkortet skrivemåte for

$$2 \cdot 10^5 + 4 \cdot 10^4 + 0 \cdot 10^3 + 0 \cdot 10^2 + 3 \cdot 10^1 + 6.$$

Eksponentnotasjoneneksponentnotasjon—textbf, for de som trenger en oppfriskning, betyr at vi ganger sammen grunntallet så mange ganger som angitt av eksponenten:

$$10^4 = \underbrace{10 \cdot 10 \cdot 10 \cdot 10}_{4 \text{ ganger}} = 10.000$$

I skrivemåten 240036 kan vi tenke at det er 6 enere, 3 tiere, ingen hundrere og så videre.

Binære og hexadesimale tall

I dataverdenen brukes ofte binære tall og hexadesimale tall, fordi det gir metoder for å representere heltall på datamaskinen som optimerer plassbruken (i en viss forstand). Binære tall representerer heltall i totallsystemet. Da bruker vi sifrene 0 og 1, og et posisjonssystem som for titallsystemet. For eksempel er

$$101001_2$$

en forkortet skrivemåte for

$$1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 = 41 \text{ i titallsystemet.}$$

Hexadesimale tall representerer på samme måte heltall i sekstentallsystemet. Da brukes sifrene

$$0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad A \quad B \quad C \quad D \quad E \quad F \ .$$

De nye sifrene A, B, C, D, E, F representerer verdiene $10, 11, 12, 13, 14, 15$ henholdsvis. For eksempel betyr

$$2A0F_{16}$$

at vi skal ta

$2 \cdot 16^3 + A \cdot 16^2 + 0 \cdot 16^1 + F = 2 \cdot 16^3 + 10 \cdot 16^2 + 0 \cdot 16^1 + 15 = 10767$ i titallsystemet.

Når vi skriver et tall som $sifre_{tall}$ bruker vi tallsystemet som angitt av subskriptet, så 1000_{16} betyr $16^3 = 4096$, ikke $10^3 = 1000$. Vi pleier ikke presisere titallsystemet når det brukes.

Addisjon

Vi vender nå tilbake til titallsystemet, og tar addisjon av heltall helt presist. En datamaskin er stokk dum, og om vi skal kunne forklare datamaskinen hvordan vi vil ha en oppgave utført, må vi formulere skrittene datamaskinen skal ta helt nøyaktig. La oss anta at datamaskinen har fått vite hvordan man legger sammen sifrene vi bruker, altså at den i utgangspunktet kan regne ut $2+3$ og $7+9$, men ennå ikke har lært å legge sammen tall med flere sifre. Se på utregningen av $2369 + 7152$ under:

$$\begin{array}{rcccc} & {\scriptstyle 1} & {\scriptstyle 1} & & \\ & 2 & 3 & 6 & 9 \\ + & 7 & 1 & 5 & 2 \\ \hline & 9 & 5 & 2 & 1 \end{array}$$

Når vi skal legge sammen disse to tallene, begynner vi med enerne. Siden $9+2 = 11 = 10+1$ får vi 1 in mente (eneren på tierplassen representerer disse 10). Det betyr at når vi skal legge sammen tierne, som er 6 og 5, må vi også huske ettallet, og får $6 + 5 + 1 = 12 = 10 + 2$. Da blir det 2 på tierplassen, og 1 in mente. Siden det for hundrerne nå er $3 + 1 + 1 = 5$, som er mindre enn ti, blir det ingenting in mente nå. Endelig er $2 + 7 = 9$, så det blir 9 på tusenplassen. Nå er det ikke flere siffer igjen, og utregningen er ferdig.

Fremgangsmåten er klar: Vi legger sammen verdier på samme posisjon, men om svaret overstiger 10 får vi antall tiere in mente. Verdiene for tierne må så legges sammen, eventuelt også det som er in mente, og vi gjentar prosessen inntil vi har kommet til enden av tallene.

Vil en datamaskin forstå den beskrivelsen vi nå har gitt?

Det er ikke umulig at datamaskinen ville fått problemer dersom tallene har forskjellig antall sifre! For å regne ut $8 + 299$ forstår vi at vi utregningen skal være som

følger:

$$\begin{array}{rccc} & {\scriptstyle 1} & {\scriptstyle 1} & \\ & & & 8 \\ + & 2 & 9 & 9 \\ \hline & 3 & 0 & 7 \end{array}$$

Vi må enten fortelle datamaskinen at vi bare skal legge sammen sifre fra de posisjonene der det faktisk står noe, eller representere 8 som 008:

$$\begin{array}{rccc} & {\scriptstyle 1} & {\scriptstyle 1} & \\ & 0 & 0 & 8 \\ + & 2 & 9 & 9 \\ \hline & 3 & 0 & 7 \end{array}$$

Datamaskinen trenger ofte at vi presiserer regler som for oss mennesker er unødvendige! En annen feilkilde kan være at svaret har flere sifre enn summandene, som i $45 + 73$:

$$\begin{array}{rccc} & {\scriptstyle 1} & & \\ & & 4 & 5 \\ + & 7 & 3 & \\ \hline 1 & 1 & 8 & \end{array}$$

Her må datamaskinen forstå at svaret skal ha et siffer ekstra, enten ved å erstatte både 45 med 045 og 73 med 073, ved bare å bruke de posisjonene der det faktisk står noe (i dette tilfellet bare in mente), eller ved å lage en egen regel for det siste steget.

Når flere tall skal adderes, kan man for eksempel be datamaskinen legge sammen to først, så legge til et tredje, så et fjerde og så videre. Da kan man bruke fremgangsmåten for å legge sammen to tall flere ganger (en gang mindre enn antallet tall som skal legges sammen). Eller man kan legge sammen alle enerne først, finne ut hva som kommer in mente, legge sammen alle tierne deretter, finne ut hva som skal in mente, og så videre. Hvilken fremgangsmåte tror du er best? Hvis vi sammenlikner de to måtene for $48 + 76 + 97$ får vi utregningen

$$\begin{array}{rccc} & {\scriptstyle 1} & {\scriptstyle 1} & \\ & & 4 & 8 \\ + & 7 & 6 & \\ \hline 1 & 2 & 4 & \end{array} \qquad\qquad + \begin{array}{rccc} & {\scriptstyle 1} & {\scriptstyle 1} & \\ & 1 & 2 & 4 \\ & & 9 & 7 \\ \hline 2 & 2 & 1 & \end{array}$$

fra den første metoden, og

$$\begin{array}{rccc} & {\scriptstyle 2} & {\scriptstyle 2} & \\ & & 4 & 8 \\ + & 7 & 6 & \\ + & 9 & 7 & \\ \hline 2 & 2 & 1 & \end{array}$$

fra den andre. Om vi bruker den andre metoden kan vi få høyere tall enn 1 in mente, om vi har mange tall å legge sammen kan vi få mer enn 10 in mente også (som alle som har lagt sammen yatzy-resultater vet).

Subtraksjon

For subtraksjon går vi frem på en liknende måte som for addisjon, men heller enn å ha ting in mente låner vi fra den neste plassen. Det er forskjellige måter å skrive det på, men la oss for nå skrive det som å ha noe negativt in mente; det gjør prosessen litt klarere. For eksempel regner vi ut $103 - 88$ som

$$\begin{array}{rccc} & {\scriptstyle -1} & {\scriptstyle -1} & \\ & 1 & 0 & 3 \\ - & & 8 & 8 \\ \hline & & 1 & 5 \end{array}$$

For å kunne trekke fra på enerplassen må vi låne 10 fra tierplassen, som nå markeres ved å skrive -1 in mente. Da får vi $13-8=5$ på enerplassen. På tierplassen må vi nå ta $0-8-1$, siden -1 var in mente, og derfor må vi låne fra hundrerplassen. Dermed får vi $10-8-1=1$ på tierplassen. Det gjenstår $1-0-1=0$ på hundrerplassen.

De samme problemene som for addisjon finnes fortsatt, for eksempel har vi tenkt på 88 som 088 for utregningen på hundrerplassen. I tillegg har vi latt være å skrive svaret som 015, så nullen på hundrerplassen er fjernet. Det ekstra problemet vi kan ha er selvsagt at vi kan ta et lite tall minus et større tall, slik at svaret blir negativt. Hvis vi prøver å ta $88 - 103$ med samme fremgangsmåte som over, får vi et problem når vi kommer til hundrerplassen:

$$\begin{array}{rccc} & & 8 & 8 \\ - & 1 & 0 & 3 \\ \hline & ? & 8 & 5 \end{array}$$

På hundrerplassen har vi ingenting, og kan heller ikke låne noe, men skal trekke fra 1. For å komme unna dette kan vi enten ha en spesialregel for slike situasjoner (den kan være vanskelig å formulere, prøv selv!), eller vi kan kontrollere hvilket av tallene som er størst før utregningen starter, og når vi ser at $88 < 103$ regner vi ut $103 - 88 = 15$ og konkluderer at

$$88 - 103 = -(103 - 88) = -15.$$

Som et tredje alternativ kan vi utføre utregningen, men innse at det ikke fører frem når vi trenger å låne, men ikke kan låne, og så utføre subtraksjonen i motsatt rekkefølge.

Java løser dette problemet ved å representere negative tall ved hjelp av toerkomplement, som vi kommer til litt senere.

Med subtraksjonen på plass kan vi selvsagt også arbeide med negative tall, slik at for eksempel $(-75) + 83 = 83 - 75$. Vanlige regneregler for negative tall brukes, som at minus ganger minus blir pluss. Dermed er $345 - (-67) = 345 + 67$.

Multiplikasjon

For multiplikasjon er det flere måter å tenke på. La oss nøye oss med positive tall, siden fortegnet lett kan ordnes i tillegg som vi så i forbindelse med subtraksjon.

Vi antar at datamaskinen allerede kjenner den lille gangetabellen (som sier hvordan tallene $0 \cdots 9$ ganges med hverandre, ikke $1 \cdots 10$). Da kan vi gange sammen et tall med bare ett siffer og et annet tall ved å bruke omtrent samme fremgangsmåte som for addisjon:

$$\begin{array}{cccccc} & {\scriptstyle 3} & {\scriptstyle 5} & {\scriptstyle 4} & & \\ 6 & \cdot & 5 & 9 & 8 & \\ \hline & 3 & 5 & 8 & 8 & \end{array}$$

Siden $6 \cdot 8 = 48$ får vi 8 på enerplassen, og 4 in mente. På tierplassen får vi da $6 \cdot 9 + 4 = 58$, som gir 8 med 5 in mente. På hundrerplassen får vi $6 \cdot 5 + 5 = 35$, som altså gir 5 på hundrerplassen og 3 in mente. På tusenplassen er det nå bare noe in mente, så vi får 3 der.

Hva om begge tallene har flere sifre? Vi kan bruke det vi har gjort allerede flere ganger, sammen med posisjonssystemet. Om vi skal regne ut $73 \cdot 278$ kan vi tenke på $73 = 7 \cdot 10 + 3$ og bruke

$$73 \cdot 278 = (7 \cdot 10 + 3)278 = 7 \cdot 10 \cdot 278 + 3 \cdot 278 = 7 \cdot 2780 + 3 \cdot 278.$$

Om vi hadde flere sifre ville vi fått delmultiplikasjoner med flere sifre. Det er vanlig å skrive den faktiske utregningen i en tabell på denne måten (der det som kommer in mente ikke er tatt med):

$$\begin{array}{cccccc} 7 & 3 & \cdot & 2 & 7 & 8 \\ \hline & & & 8 & 3 & 4 \\ & 1 & 9 & 4 & 6 & 0^* \\ \hline & 2 & 0 & 2 & 9 & 4 \end{array} \tag{1.1}$$

Mange dropper å skrive den siste nullen som er merket med $*$. Om vi bruker denne fremgangsmåten må vi utføre multiplikasjon av et flersifret tall med et enkelt siffer gjentatte ganger, og så legge sammen svarene (antall ganger er antall sifre i det første tallet).

Oppgaver

Oppgave 1.1.1 *Kontrollregning:*

Kontroller alle utregningene i teksten ved å bruke Java (skriv programmer som utfører utregningene og skriver ut svaret).

Oppgave 1.1.2 *Antall sifre:*

Om vi har et tall med n sifre og et annet tall med m sifre, hva er det minimale og maksimale antallet sifre i summen, differansen og produktet?

1.2 Heltall på datamaskinen

Trenger vi å programmere addisjon og multiplikasjon selv? Datamaskinen har jo innebygde metoder for å gjøre det, men det ligger ofte skjulte begrensninger i hvordan dette virker. Vi skal se på hvordan dette virker i Java først, og så se litt på hvordan vi kan gå frem for å komme over problemene som oppstår.

Heltallstyper i Java

Java har flere typer som bestemmer heltall. En av dem er integer (engelsk for heltall), som er forkortet

```
int
```

Om vi for eksempel skriver

```
System.out.printf("%d+ %d = %d", 23, 45, 23+45);
```

vil resultatet av kjøringen bli at det skrives 23+45=68. Bokstaven 'd' står her for desimalheltall, altså et heltall skrevet i titallsystemet.

Hvis vi heller bruker litt større tall kan vi få problemer. Her er tallene eksplisitt definert som å være av typen `int`.

```
int sum1=2146469892;
int sum2=426543645;
System.out.printf("%14d%n + %11d%n = %11d%n", sum1, sum2,sum1+sum2);
```

gir resultatet

```
    2146469892
 +   426543645
 = -1721953759
```

Dette resultatet er opplagt feil!

Problemet er at Java bare setter av et bestemt antall bits til hvert heltall, og om svaret av en utregning blir et tall som er for stort til å skrives med det antallet bits, så vil svaret bli galt. Heltallstypen `int` virker for heltall fra og med $-2.147.483.648$ til og med 2.147.483.647. Vi kan teste dette ved å bruke koden

```
System.out.println(2147483647);
System.out.println(2147483647+1);
```

Da blir resultatet

```
2147483647
-2147483648
```

Det første tallet blir rett, men om vi legger til 1 kommer vi over grensen for typen `int`, og resultatet blir det laveste tallet som typen aksepterer (stort og negativt).

Hvorfor akkurat 2.147.483.647? Antallet tall mellom de to grensene er

$$2^{32} = 4294967296$$

Det er avsatt 32 bits til hver `int`. På samme måte er det avsatt 16 bits til heltallstypen `short`, og 64 bits til heltallstypen `long`. Det største tallet du kan skrive som `long` er

$$9.223.372.036.854.775.807 = 2^{63} - 1.$$

Kanskje virker det som 19 sifre er tilstrekkelig for det vi kan trenge å gjøre med hele tall, men for eksempel i kryptografi trenger man mye større tall. Moderne RSA-koding bruker typisk primtall med omtrent 300 sifre.

Antall bits

For å finne ut hvilken heltallstype som trengs, må vi forstå litt mer presist hvordan datamaskinen lagrer informasjon. Grunnenheten i datainformasjon er en *bit*, som vi tenker på som å representere enten 0 eller 1. For hver bit har vi altså to muligheter, så om vi har to bits har vi to ganger to muligheter, tre bits gir to ganger to ganger to muligheter, og så videre. Her er de 16 mulighetene vi har for fire bits:

0000	0100	1000	1100
0001	0101	1001	1101
0010	0110	1010	1110
0011	0111	1011	1111

Siden vi har 16 muligheter er det mulig å lagre 16 forskjellige verdier på 4 bits, $2^4 = 16$.

Generelt kan vi lagre 2^n ulike verdier om det er avsatt n bits. Her er en tabell over de første toerpotensene:

n	2^n	n	2^n
0	1	11	2048
1	2	12	4096
2	4	13	8192
3	8	14	16384
4	16	15	32768
5	32	16	65536
6	64	17	131072
7	128	18	262144
8	256	19	524288
9	512	20	1048576
10	1024	⋮	⋮

Det betyr for eksempel at 20 bits er nok til å skille tallene fra og med 1 til og med 1048576. Men det er ikke den mest hensiktsmessige måten å sette av plass til tallene på; vi er jo også interesserte i negative tall, og i 0. Standarden i Java er at heltall med n bits representerer verdiene fra og med -2^{n-1} til og med $2^{n-1}-1$. For eksempel bruker short 16 bits, og representerer tallverdiene fra og med $-2^{15} = -32768$ til og med $2^{15} - 1 = 32767$.

Datatypen `byte` representerer 8 bits, og kan også tenkes på som en heltallsverdi (med verdier fra og med -128 til og med 127). Datatypen `char` representerer tegn ved deres ascii-verdi, og kan tenkes på som en ikke-negativ heltallsverdi (med verdier fra og med 0 til og med 65535). Den bruker altså 16 bits.

Tabell over heltallstyper og deres verdier

navn	verdi	bits
byte	-128 til 127	8
short	-32.768 til 32.767	16
int	-2.147.483.648 til 2.147.483.647	32
long	-9.223.372.036.854.775.808 til 9.223.372.036.854.775.807	64
char	0 til 65535	16

Dersom man legger sammen to tall som er representert ved samme datatype, får svaret den samme datatypen. Dersom man legger sammen to tall som er representert ved forskjellige datayper, vil Java endre datatypen til den ene summanden, slik at de får samme type, og svaret får denne typen. Hvordan velger Java hvilken type som skal brukes? Svaret får den lengste typen.

Se på dette lille programmet, som legger sammen like og ulike typer.

```
int i1 = 1147538874;
int i2 = 2004213452;
int i3 = i1+i2;
long i4 = i1+i2;                    /*(1)*/
long i5 = i1;
i5 = i5+i2;                         /*(2)*/
int i6= (int) i5;
short i7 = (short) i6;
short i8 = (short) i5;              /*(3)*/
long i9 = (long) i1 +(long) i2;     /*(4)*/
System.out.printf("i1 er %d\n", i1);
System.out.printf("i2 er %d\n", i2);
System.out.printf("i3 er %d\n", i3);
System.out.printf("i4 er %d\n", i4);
System.out.printf("i5 er %d\n", i5);
System.out.printf("i6 er %d\n", i6);
System.out.printf("i7 er %d\n", i7);
System.out.printf("i8 er %d\n", i8);
System.out.printf("i9 er %d\n", i9);
```

Vi starter med to `int`, og alle de andre tallene representerer summen av de to første, men tallene er valgt slik at svaret blir for stort. Svaret som `int` blir derfor galt.

I punkt (1) legges de to tallene sammen som `int`, og så blir (det gale) svaret konvertert til `long`. I punkt (2) settes først en ny `long` lik den ene `int`. Så legges den andre til, men siden det nå er en `long` og en `int` som legges sammen, blir svaret en `long`, og summen blir det den egentlig skal være. Om vi skal gå fra en lang til en kort type, kan vi miste informasjon, og det skjer tre ganger i dette programmet. Merk spesielt at i punkt (3) får vi samme `short` om vi starter med en `long` eller den delvis konverterte `int`. I punkt (4) ser vi at vi også kan konvertere en kort type til en lang type eksplisitt, og det kan sikre at svaret blir korrekt.

Resultatet av å kjøre programmet er

```
i1 er 1147538874
i2 er 2004213452
i3 er -1143214970
i4 er -1143214970
i5 er 3151752326
i6 er -1143214970
i7 er -4986
```

```
i8 er -4986
i9 er 3151752326
```

NB! Reglene for typekonvertering er at en liten type kan konverteres til en lengre type implisitt (datamaskinen gjør det for oss), men som vi ser i punkt (1) risikerer vi at regneoperasjoner utføres før konverteringen. Derfor kan vi konverterere eksplisitt fra en kort til en lang type, som i (4). Om en lang type konverteres til en kort, er det fare for at informasjon går tapt. Derfor må vi i dette tilfellet konvertere eksplisitt, som i (3). Om vi ikke gjør det, får vi kompileringsfeil.

Toerlogaritme

Vi husker at a^b representerer tallet a ganget med seg selv b ganger, slik at for eksempel

$$2^4 = 2 \cdot 2 \cdot 2 \cdot 2 = 16.$$

Om b ikke er et heltall, men en brøk, bruker vi røtter. Om b er et reelt tall, ikke brøk, bruker vi en grense over brøk.

Den motsatte operasjonen er å ta logaritmen med grunntall a, som skrives $\log_a$. Logaritmen er formelt definert ved at

$$\log_a(a^b) = b.$$

Det kan også formuleres som at $\log_a(c)$ = det tallet a må opphøyes i for å få c. For eksempel er

$$\log_2(8) = 3$$

siden

$$2^3 = 8.$$

For å regne ut logaritmer presist bruker vi vanligvis kalkulator eller datamaskin.

Siden n bits kan skille mellom 2^n ulike verdier, trenger vi å avsette minst $\log_2(x)$ bits for å representere x ulike verdier. Hvor mange bits trenger vi for å skille tall fra og med -1000 til og med 1000? Det er total 2001 tall som skal representeres, og

$$\log_2(2001) \approx 10{,}97.$$

Vi må altså sette av minst 11 bits for å representere disse tallene. Merk at vi alltid må runde opp. Om vi skal ha nok plass til å representere tallene fra -35 til 35, altså 71 tall, er

$$\log_2 71 \approx 6{,}15,$$

så vi må sette av 7 bits.

Dersom vi skal regne ut toerlogaritmer for å finne ut hvor mange bits vi trenger, er det nok å regne ut den riktige heltallsverdien som toerlogaritmen rundes opp til. Det kan vi gjøre ved å starte med tallet (for eksempel 71), og telle hvor mange ganger vi må dele på to før vi kommer under 1:

$$71 \quad 35{,}5 \quad 18 \quad 9 \quad 4{,}5 \quad 2{,}25 \quad 1{,}2 \quad 0{,}6 < 1.$$

Vi måtte dele på to sju ganger for å komme under 1, så toerlogaritmen av 71 rundes opp til 7. Alternativt kan vi starte med 1 og telle hvor mange ganger vi må gange med 2 før vi kommer over tallet. Prøv selv med 71!

Representasjon av negative tall

Heltallstypene `byte`, `short`, `int` og `long`, representerer negative tall på samme måte. For å forstå hva som skjer, bruker vi en enda kortere type, med bare 4 bits. Da skal vi få representert tallene fra og med -8 til og med 7. De positive tallene (og null) er representert ved at den første bit-en er 0, og resten representerer tallet på vanlig måte:

Positive tall	
binært	titall
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

Negative tall er representert ved at den første bit-en er 1. Siden tallene med 0 først representerer 0 til 7, og tallene med 1 først skal representere -8 til -1, er det et lite triks som kalles toerkomplement. Ta det positive tallet, bytt ut 0 med 1 og 1 med 0, og legg så til 1 ($= 0001_2$). I tillegg lar vi tallet -8 være representert ved 1000_2. Det har fordelen at addisjon oppfører seg korrekt.

Negative tall	
binært	titall
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

Om vi skal legge sammen 4 og -3 blir utregningen som følger:

$$\begin{array}{c|cccc} {\scriptstyle 1} & {\scriptstyle 1} & & & \\ & 0 & 1 & 0 & 0 \\ & 1 & 1 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 1 \end{array}$$

Siden vi bare har avsatt 4 bits, blir den første bit-en, 1, i svaret glemt, og vi står igjen med 0001_2 som altså representerer tallet 1.

Om vi skal regne ut $2 - 5$ kan vi bruke subtraksjonsprinsippet, der vi må huske at binært er $10_2 - 1_2 = 1_2$:

$$\begin{array}{cccccc} {\scriptstyle -1} & {\scriptstyle -1} & & {\scriptstyle -1} & & \\ & 0 & 0 & 1 & 0 \\ - & 0 & 1 & 0 & 1 \\ \hline ? & 1 & 1 & 0 & 1 \end{array}$$

Vi har samme problemer som før, med at spørsmålstegnet ikke gir oss noe. Men siden det bare er avsatt fire bits, vil vi bare huske 1101_2 som er representasjonen for -3. En annen måte å regne det ut på er å skrive $2 - 5 = 2 + (-5)$, og dermed addere tallene. Kontroller selv at dette gir samme svar.

Dersom vi prøver å regne ut $7 + 1$ får vi

$$\begin{array}{c|cccc} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & \\ & 0 & 1 & 1 & 1 \\ + & 0 & 0 & 0 & 1 \\ \hline & 1 & 0 & 0 & 0 \end{array}$$

Dette representerer -8, og viser altså hvorfor det problemet vi tidligere har observert inntreffer.

Konklusjonen er at denne måten å representere negative tall på, sikrer at den vanlige måten å addere og subtrahere tall på, gir korrekt svar så lenge svaret ikke kommer under -2^{n-1} eller over $2^{n-1} - 1$.

Oppgaver

Oppgave 1.2.1 *Typen* `char`*:*

Undersøk (det vil si: Skriv programmet som sjekker) hva som skjer om du konverterer mellom `int` og `char`, både implisitt og eksplisitt. Hva gir kompileringsfeil, og hva gir resultatene du venter? Hva skjer med negative tall når de konverteres til `char`, som bare tar positive verdier?

Oppgave 1.2.2 *Multipliser negative tall:*

Bruk representasjonen ved toerkomplement (og fire bits), og regn ut

a) $(-3) \cdot (-2)$.

b) $(-3) \cdot 2$.

c) $2 \cdot (-4)$.

d) $(-2) \cdot (-4)$.

e) $2 \cdot 4$.

f) $(-1) \cdot (-4)$.

I hvilke tilfeller ble svaret det riktige tallet?

Oppgave 1.2.3 *Hexadesimale tall:*

Hvor mange bits trengs for å lagre hvert siffer i et hexadesimalt tall? Hvor mange hexadesimale sifre trengs for å lagre verdiene fra og med 0 til og med 500.000? Hvor mange bits er `"tapt"` sammenliknet med å lagre de samme verdiene binært?

Oppgave 1.2.4 *Oktale tall.*

Svar på de samme spørsmålene som i den forrige oppgaven, bare med oktale tall heller enn hexadesimale. Oktale tall, tall i base 8, står beskrevet på side 34 i Log: Mathema bind 1, og i Mughal & al: Java ... side 625.

Oppgave 1.2.5 *Kuber:*

En *kube* er et tall som er ett annet tall opphøyd i tredje (det er volumet av en kube med heltallig sidekant). For eksempel er 27 en kube, siden $27 = 3^3 = 3 \cdot 3 \cdot 3$.

a) Skriv et program som finner alle kubene som er mindre enn 2000.

b) Skriv et program som finner alle tall under 2000 som kan skrives som summen av to kuber (for eksempel $28 = 3^3 + 1^3$). Er det noe tall som kan skrives som en sum av to kuber på to ulike måter?

c) I tegnefilmserien "Futurama" har robotene Bender og Flexo serienumre 2716057 og 3370318, som begge er en sum av to kuber. Skriv et program som finner hvordan disse to tallene kan skrives som en sum av to kuber. **Hint**: Kuber kan også være negative, slik at $26 = 3^3 + (-1)^3$ er en mulighet.

Kapittel 2

Flyttall

Alle de operasjonene vi har sett så langt har bare brukt hele tall. Om vi ønsker å se på tall som kan ha verdier som ikke bare er heltall, er det vanlig å bruke såkalte *flyttall*. På samme måte som det er ulike heltallstyper i Java, er det også ulike flyttallstyper, `float` og `double`. `float` bruker 32 bits, `double` bruker 64 bits og gir muligheter for større tall (blant annet). Vi har sett at resultatet av regneoperasjoner på heltall kan bli galt på datamaskinen, dersom utregningen krever flere bits enn det som er satt av. Det kan også bli galt svar av utregninger med flyttall, og vi skal undersøke hvordan det skjer.

2.1 Regneoperasjoner på flyttall

For å forstå hvilke tall som kan representeres ved de ulike flyttalstypene trenger vi først litt notasjon.

Vitenskapelig notasjon for tall

Vi begynner med å skrive tall på såkalt *vitenskapelig* form. Da har vi et siffer før komma, et passe antall siffer (avhengig av situasjonen) etter komma, og ganger med en potens av 10. For eksempel vil tallet 457,352 skrives som

$$4{,}57352 \cdot 10^2$$

og 0,0002631 vil skrives som

$$2{,}631 \cdot 10^{-4}.$$

Husk at $a^{-b} = 1/a^b$, slik at $10^{-4} = 1/10^4 = 1/10.000 = 0{,}0001$. Det er også vanlig å skrive e eller E for eksponenten her, slik at det siste tallet også kan skrives $2{,}631e-4$ eller $2{,}631E-4$. Vi kommer ikke til å bruke denne notasjonen.

Det er ikke noe spesielt med titallsystemet som gjør at vi kan bruke komma, det kan vi også for andre baser. For binære tall betyr det at vi kan skrive

$$0{,}101_2 = 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = \frac{1}{2} + \frac{1}{8} = 0{,}625.$$

Flyttallstyper i java

Tallene som kan representeres som `float` er tall som er representert ved en binær versjon av vitenskapelig notasjon, der det består av et tall på $N = 24$ bits, som vi skal tenke på som det første sifferet komma de andre sifrene, og et annet tall e med $K = 8$ bits, som vi skal tenke på som eksponenten 2^e. For å vise hvordan dette virker bruker vi lavere verdier for N og K, la oss ta $N = 5$ og $K = 4$. Vi hopper også over problemet med representasjon av negative tall, mer forteller bare at eksponentene er begrenset til å ligge mellom $-(2^{K-1} - 2) = -6$ og $2^{K-1} - 1 = 7$ Da vil for eksempel

$$0{,}1100_2 \cdot 2^3 = (0{,}5 + 0{,}25) \cdot 2^3 = 0{,}75 \cdot 8 = 6.$$

mens

$$1{,}0101_2 \cdot 2^2 = (1 + 0{,}25 + 0{,}0625) \cdot 2^2 = 1{,}3125 \cdot 4 = 5{,}25.$$

Et eksempel med negativ eksponent er

$$0{,}1101_2 \cdot 2^{-6} = (0{,}5 + 0{,}25 + 0{,}0625) \cdot 2^{-6} = 0{,}0126963125.$$

Det største tallet vi kan skrive er

$$1{,}1111_2 \cdot 2^7 = (1 + 0{,}5 + 0{,}25 + 0{,}125 + 0{,}0625) \cdot 2^7 = 248.$$

Formelen for dette, uttrykt ved $N = 5$ og $K = 4$, er

$$Max = (2 - 2^{-(N-1)}) \cdot 2^{2^{K-1}-1} = (2 - 2^{-4}) \cdot 2^7 = 1{,}9375 \cdot 128 = 248.$$

Det minste positive tallet man kan skrive på denne måten er

$$0{,}0001_2 \cdot 2^{-6} = 0{,}0625 \cdot 2^{-6} = 0{,}0009765625.$$

Formelen for dette uttrykt ved N og K er

$$Min = 2^{-(N-1)} \cdot (2^{2^{K-1}-2}) = 2^{-4} \cdot 2^{-6} = 2^{-10} = \frac{1}{1024}.$$

Om vi gjør det samme med $N = 24$ og $K = 8$, som i `float`, finner vi

$$MAX\_VALUE = (2 - 2^{-(N-1)}) \cdot 2^{2^{K-1}-1} = (2 - 2^{-23}) \cdot 2^{127} \approx 3{,}4 \cdot 10^{38}.$$

Den minste verdien er

$$MIN\_VALUE = 2^{-(N-1)} \cdot (2^{2^{K-1}-2}) = 2^{-23} \cdot 2^{-126} = 2^{-149} \approx 1{,}4 \cdot 10^{-45}.$$

Disse to verdiene er innebygde konstanter i Java, av typen `float` (det er tilsvarende for andre typer). Mer presist er dette tilordnet en innkapsling, et objekt av typen `Float`. Se Java-boken side 70-72.

I tillegg til disse verdiene, og de tilsvarende negative tallene, er tallet 0 representert på to måter (som vi kaller positiv og negativ null), det er en representasjon av uendelig ($\pm\infty$), og det er en spesiell verdi som heter NaN, en forkortelse for " Not a Number". I neste avsnitt skal vi se eksempler på hvordan disse symbolene dukker opp.

Verdien av en `double` er bygd opp på samme måte, med $N = 53$ og $K = 11$. Da er

$$MAX\_VALUE = (2 - 2^{-52}) \cdot 2^{1023} \approx 1{,}8 \cdot 10^{308}$$

og

$$MIN\_VALUE = 2^{-52} \cdot 2^{-1022} = 2^{-1074} \approx 4{,}9 \cdot 10^{-324}.$$

Typekonvertering kan gjøres implisitt fra `float` til `double`, men må gjøres eksplisitt den andre veien. Vi kan også konvertere til eller fra heltallstyper, se oppgavene.

Hvor store og hvor små er egentlig grensene for disse talltypene? Til sammenlikning er massen til et elektron omtrent $9{,}2 \cdot 10^{-31} kg$, mens massen til Melkeveien er anslått til $1{,}4 \cdot 10^{42} kg$ (tall fra wikipedia).

Flyttall av typen `double` sies å ha dobbel presisjon. Av og til sier man singel presisjon for å streke under at man bruke `float`.

Regneartene

Addisjon (+), subtraksjon(-), multiplikasjon(*) og divisjon(/) av flyttall gir (omtrent) det svaret man skulle tro, så lenge tallene som inngår ikke blir for små eller for store. Se for eksempel på dette programmet:

```
float a;
float b;
float c;
int n;
a = 1.0f;
b = 10.0f;
c = 0.0f;
n = 100000;
```

```
System.out.printf("1/10 er %f", a/b);
for (int i=0;i<n;i++){
        c += (a/b);
}
System.out.printf(" 100.000 ganger 0,1 er %f \n", n*a/b);
System.out.printf("0,1 +...+0,1 100.000 ganger er %f", c);
```

Resultatet når det kjøres er

```
1/10 er 0.1
100.000 ganger 0,1 er 10000.0
0,1 +...+0,1 100.000 ganger er 9998.557
```

(bortsett fra antall desimaler). I `for`-løkken har vi kjørt gjennom 100.000 iterasjoner, så der har det blir akkumulert en liten feil. Merk også at tallet 10.000 skrives ut som 10000.0 (husk at punktum er den engelskspråkliges komma), for å markere at det er et flyttall og ikke et heltall. Om vi hadde erstattet $b = 10.0f$ med $b = 2.0f$ hadde vi fått riktig svar av begge utregningene (prøv selv!). Hva er egentlig forskjellen?

Oppgaver

Oppgave 2.1.1 *Maksimal- og minimalverdier for ulike typer.*

Ved hjelp av dokumentasjonen for Java, finn og bruk MAX_VALUE og MIN_VALUE for typene `byte`, `short`, `float` og `double`.

Hint: Her er maksimalverdien for `float`

```
Float a = new Float(Float.MAX_VALUE);
```

Se ellers Java-boken side 70-72, i tillegg til dokumentasjonen i Java.

Oppgave 2.1.2 *Bruk av flyttall.*

Finn massene til planetene i solsystemet på nettet, og legg dem sammen. Kan du få Java til å skrive ut planetenes navn og masse på en oversiktlig måte, ved å bruke `System.out.printf`? Kan du få skrevet ut like mange sifre som massene er oppgitt med der du fant dem?

NB Mulig problem: Java tror i utgangspunktet at desimaltall er av typen `double`, så det kan være nødvendig å erklære typen til å være `double` eller bruke `float` og skrive `"f"` etter tallet.

2.2 Vanskeligheter med flyttall

Vi skal nå se på noen av de vanligste problemene som kan oppstå med flyttall.

Eksakt representasjon som flyttall

Noen tall kan representeres eksakt som flyttall. Årsaken til det er at flyttallene internt er representert binært, men mange tall kan ikke skrives presist som binære tall. For å være sikker på at vi forstår hva som skjer, la oss først tenke på titallsystemet.

Om vi skal regne ut 1/20, og skrive svaret som desimaltall, får vi 0,05. Det er eksakt rett.

Om vi skal regne ut 1/3, og skrive svaret som desimaltall, får vi $0{,}33333\cdots$. Om vi avbryter denne utregningen noe sted, vil vi ikke få et helt eksakt riktig svar.

Eksakt rett desimalutvikling for en brøk a/b (der a og b ikke har noen felles faktor) får vi bare dersom de eneste primtallsfaktorene til b er 2 og 5.

I totallsystemet får vi bare rett svar dersom nevneren er en potens av 2. For eksempel er $3/4 = 0{,}75 = 0{,}5 + 0{,}25$, så vi kan skrive

$$\frac{3}{4} = 0{,}11_2$$

som er eksakt rett. Et annet eksempel er $0{,}0625 = 1/16$, som kan skrives som $0{,}0001_2$. Om vi deretter regner ut 1/3 (hvordan kan vi gjøre det?), finner vi at

$$\frac{1}{3} = 0{,}0101010101\cdots_2\,,$$

det skifter altså mellom 0 og 1 i det uendelige. Dermed kan ikke 1/3 representeres eksakt som flyttall, mens 3/4 kan. Utregninger med eksakte representasjoner vil altså gi korrekte svar (i hvert fall addisjon, om ikke tallene er veldig forskjellige), mens utregninger med andre tall kan bli feil.

`Random`-funksjonen i Java, som vi skal bruke en del senere, vil velge ut eksakt representerbare tall, og vi vil derfor ikke få illustrert de vanskeligste problemene (i hvert fall ikke for addisjon) ved å generere tall på denne måten.

Tall med stor størrelsesforskjell

Dersom man regner med tall som har stor størrelsesforskjell, kan feil også oppstå, selv om tallene er eksakt representert som flyttall. Forsøk selv med forskjellige verdier for

n og b i dette programmet:

```
float a;
float b;
float c;
int n;
int m;
a = 1.0f;
b = 2.0f;
c = 0.0f;
n = 25;
m=2;
c=a/b;
for (int i=0;i<n;i++){
   a=a*m;
   System.out.printf("I iterasjon %d er c+a-a %f\n", i, (c+a)-a);
}
```

Her vil det plutselig skje noe galt for `i=22`, men om `b=2.0f` erstattes av for eksempel `b=10.0f`, slik at brøken `c=a/b` ikke kan representeres eksakt, vil problemene komme mer gradvis.

Spesielle flyttall

De spesielle flyttallene dukker opp i uilike situasjoner. Dette programmet produserer de fem ulike spesielle verdiene:

```
System.out.println(1.0f/0.0f);
System.out.println(-1.0f/0.0f);
System.out.println(Math.sqrt(-0.0f));
System.out.println(2.0f-2.0f);
System.out.println(0.0f/0.0f);
```

Kjøring gir

```
Infinity
-Infinity
-0.0
0.0
NaN
```

Merk at

```
System.out.println((-0.0f)==(0.0f));\\
```

gir `true`. Kanskje enda merkeligere er det at

```
float d = 0.0f/0.0f;
System.out.println(d==d);
```

gir `false`, siden "0/0" er representert som NaN, og denne verdien ikke skal kunne brukes til noe.

Oppgaver

Oppgave 2.2.1 *Typekonvertering mellom heltallstyper og flyttallstyper.*

Undersøk hva som skjer om et heltall legges sammen med et flyttall. Prøv videre implisitt og eksplisitt konvertering mellom `int` og `float`, i begge retninger. Sjekk også typekonvertering til `double`, kjør for eksempel dette programmet:

```
double f = 29*0.01;
System.out.println((int) (f*100));
```

Oppgave 2.2.2 *Regneoperasjoner og typekonvertering.*

Undersøk hva som skjer om regneoperasjoner på heltall utføres før eller etter typekonvertering til flyttall. Undersøk deretter hva som skjer om regneoperasjoner på flyttall utføres før eller etter typekonvertering til heltall.

Oppgave 2.2.3 *Den kommutative loven sier at vi kan bytte om på faktorene.*

Kjør dette programmet:

```
float a=4e12f;
float b=0.25e-12f;
float c=5e35f;
System.out.println((a*b)*c);
System.out.println((a*c)*b);
```

Siden vi kan bytte om på faktorene skal de matematiske størrelsene $(a \cdot b) \cdot c = (a \cdot c) \cdot b$. Hvorfor får vi disse svarene? Hva skjer om du fjerner symbolet `f`, og representerer tallene som `double`? Kan du lage et tilsvarende problem med `double`?

Oppgave 2.2.4 *Divisjon og binære tall.*

I teksten brukte vi binær representasjon av 1/3:

$$\frac{1}{3} = 0{,}01010101\cdots_2 .$$

Kontroller at den representasjonen er rett ved å multiplisere med $3 = 11_2$. Prøv å regne ut den tilsvarende representasjonen for 1/5. Hvis du syns dette er vanskelig, forsøk først å regne ut 1/7 i titallsystemet.

Oppgave 2.2.5 *Annengradslikningen.*

Formelen for løsning av annengradslikningen

$$ax^2 + bx + c = 0$$

er

$$x_0 = \frac{-b+\sqrt{b^2-4ac}}{2a} \quad x_1 = \frac{-b-\sqrt{b^2-4ac}}{2a}.$$

Velg passende verdier for a, b, c (bruk typen `float`), og skriv et program som beregner disse to løsningene. For å ta kvadratrot, bruk `Math.sqrt` ("sqrt" er forkortelse for square root). Ta med en test av løsningen, altså regn ut $ax_0^2 + bx_0 + c$ og sjekk at det blir lik null, og ditto for x_1. Prøv programmet ditt på disse fire verdiene for koeffisientene:

a	b	c
1	2	-3
1	2	1
1	1	1
1	-20000	1

I hvilke tilfeller forstår du svaret?

NB Hvis du får feilmeldinger av typen "possible loss of precision", prøv å konvertere tilbake til `float`.

Oppgave 2.2.6 *De spesielle symbolene.*

Undersøk hvordan Java håndterer de fem spesielle verdiene ± 0, $\pm\infty$ og NaN. Lag multiplikasjonstabell, addisjonstabell, subtraksjonstabell og divisjonstabell. Du kan få tak i disse verdiene via innkapsling, se Java-boken side 70 – , ved å definere f.eks.

```
Float minusUendelig = new Float(Float.NEGATIVE_INFINITY);
```

Se ellers Java-dokumentasjon.
Undersøk til slutt om Java behandler de spesielle `double`-verdiene på samme måte som Java behandler de spesielle `Float`-verdiene.

Kapittel 3

Fakulteter og binomialkoeffisienter

Det er mange måter å kode rekursive definisjoner, som for eksempel $n! = n \cdot (n-1)!$ på i Java, noen gode og noen ikke fullt så gode. Vi skal sjekke at vi får det til, og i tillegg passe på hvor langt vi kan gå før vi overstiger grensene for de ulike heltallstypene. Siden binomialkoeffisientene kan regnes ut ved hjelp av fakulteter, er det fristende å regne ut fakultene først, og så dividere. Det er i praksis ingen god løsning, og vi skal se nærmere både på årsaken til problemet, og hvordan vi kan komme unna det.

3.1 Permutasjoner og utvalg

Vi skal først finne ut hvordan uttrykkene $P(n,k)$ og $C(n,k) = \binom{n}{k}$ kan regnes ut direkte ved å bruke formlene. Materialet til den matematiske bakgrunnen her er gjennomgått i matematikken.

Fakultet

En måte å lage fakultetsoperasjonen på er å lage en klasse `fakultet` som bare inneholder en metode som vi kan kalle `resultatet`:

```
public class Fakultet {
        static long resultatet(long n){
                if (n<0){
                        return 0;
                } else {
                        long mellomRegning=1;
                        for (int j=2;j<=n;j++){
                                mellomRegning=mellomRegning*j;
                        }
```

```
                return mellomRegning;
                }
        }
}
```

Her bruker vi definisjonen av $n!$ direkte. Merk at vi gir null til svar for negative verdier, men egentlig burde det vært behandlet annerledes. Hva skjer om $n = 0$ eller $n = 1$?

For å bruke denne utregningen, kaller vi metoden fra `main`, for eksempel som dette:

```
long a;
a = Fakultet.resultatet(20);
System.out.println(a);
```

Kontroller at programmet gir riktige resultater for de lave verdiene av $n!$ du allerede kjenner. For eksempel skal `a = fakultet.resultatet(5);` gi utskriften `120`.

Det er å kaste blår i øynene på dere å bruke typen `long` overalt, egentlig burde bare svaret vært `long`. Vi får galt svar i Java allerede for

$$21! = 51090942171709440000 > 2^{63},$$

som er større enn grensen for `long`.

Permutasjoner

Nå skal vi lage permutasjonsfunksjonen $P = P(n, k)$ på samme måte som vi har laget fakultetsoperasjonen. Vi skriver en egen klasse, der den største forskjellen nå er at vi må ha med to parametre i `resultatet`:

```
public class Permutasjon {
        static long resultatet(long n, long k){
                if (k<0 || n<k){
                        return 0;
                } else{
                long mellomRegning=1;
                for (long j=n; j>(n-k); j--){
                        mellomRegning=mellomRegning*j;
                }
                return mellomRegning;
                }
        }
}
```

Vi returnerer null om $k < 0$ eller $n < k$, ellers bruker vi definisjonen direkte.

For å bruke denne utregningen må vi i `main` inkludere noe liknende dette:

```
long b;
b=Permutasjon.resultatet(10,3);
System.out.println(b);
```

Kontroller også her at svaret blir rett for de små verdiene du kjenner, og kontroller at $P(n,n) = n!$. For eksempel får vi fra `b=Permutasjon.resultatet(12,2);` utskriften 132.

Binomialkoeffisienter - naivt

Vi kunne gått frem på samme måte som for fakulteter og permutasjoner, men siden vi allerede har programsnuttene for disse utregningene kan vi heller kalle dem, som i denne klassen:

```
public class Binomial {
        static long resultatet(long n, long k){
        return Permutasjon.resultatet(n,k)/Fakultet.resultatet(k); /*(1)*
        }
}
```

Vi kan så bruke denne klassen ved å skrive i `main`:

```
long c;
c= Binomial.resultatet(10,3);
System.out.println(c);
```

Kontroller igjen at dette gir rett svar for små verdier av parametrene n, k. For eksempek skal `c= Binomial.resultatet(12,2);` gi svaret 66.

Av ulike årsaker er dette programmet ikke helt tilfredsstillende. For en liten forbedring av dette programmet, som i hvert fall ikke fører til så mange feil under kjøring, se Oppgave 3.2.2.

Den viktigste feilen er at programmet ikke fanger opp brøkregningen, slik at selv om en binomialkoeffisient har moderat størrelse, så kan det være at mellomregningen kommer over grensene for `long`. Bruk for eksempel programmet til å regne ut antall lottorekker på to måter:

a) Antall lottorekker er antall måter vi kan plukke ut 7 av 34 tall på, de tallene vi vil skal gå inn. Det gir at antall lottorekker er $C(34, 7)$.

b) Antall lottorekker er antall måter vi kan plukke ut 27 av 34 tall på, de tallene vi ikke vil skal gå inn. Det gir at antall lottorekker er $C(34, 27)$.

Om disse to utregningene kjøres gjennom programmet vi har skrevet, får vi forskjellig svar. Faktisk påstår programmet at $C(34, 27) = -1$, som opplagt er galt. En første tilnærming til dette problemet er å bruke likheten

$$C(n, k) = C(n, n - k)$$

og modifisere kallet til de andre funksjonene i punkt (1) til

```
return Permutasjon.resultatet(n,Math.min(k,n-k))/
                    Fakultet.resultatet(Math.min(k,n-k));
```

Dette er en enkel løsning, som forbedrer situasjonen noe. Men om vi prøver å regne ut for eksempel $C(44, 17)$ blir svaret -10230, mens det riktige svaret er

$$C(44, 17) = 686353797976.$$

Dette skulle vi klart å regne ut, siden det er mye mindre enn grensen for `long`. Om vi skal klare å løse dette problemet må vi tenke på en helt annen måte.

3.2 Binomialkoeffisienter - rekursivt

Vi skal finne en måre å regne ut flere binomialkoeffisienter på, uten å komme over grensene for heltallstypene.

Fakulteter rekursivt

La oss først ta en ny titt på fakultetene. Vi skal gange sammen alle tallene opp til og med n for å finne $n!$. Dermed er $n! = n \cdot (n-1)!$, det er en rekursiv definisjon som vi kan bruke direkte i Java. La oss regne ut fakultetene opp til $n = 20$, og lagre dem i en tabell. Vi kan starte på 0, siden $0! = 1$ også er definert. Vi trenger derfor en tabell med $n + 1 = 21$ plasser.

```
int nMaks =20;
long[] fakultetene = new long[nMaks+1];
fakultetene[0]=1; //(1)
for (int i=1;i<(nMaks +1);i++){ //(2)
        fakultetene[i]=i*fakultetene[i-1];
}
for (int i=0;i<(nMaks+1);i++){//(3)
        System.out.printf("%d! = %d\n", i, fakultetene[i]);
}
```

Her bruker vi definisjonen rekursivt, men passer på at det første tallet blir rett (Punkt (1)). Vi bruker derfor rekursjonen først fra `i=1` i Punkt (2). Når vi så skal skrive ut fakultetene, går vi gjennom hele tabellen fra `i=0` (Punkt (3)).

Utskriften blir

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
```

Vi ser at vi nærmer oss grensen for `long`.

Utregning av binomialkoeffisientene

Den formelen vi skal bruke er

$$C(n,k) = C(n-1,k) + C(n-1,k-1).$$

Om vi tenker på C som å fortelle noe om utvalg, kan vi argumentere for denne formelen på denne måten: Vi skal velge k objekter av n. Se på det første objektet, som er med i utvalget eller ikke. Om det er med i utvalget, er det $k-1$ objekter igjen å velge blant de resterende $n-1$. Om det første objektet ikke er med, er det k objekter igjen å velge blant de resterende $n-1$.

Dermed kan vi finne $C(n,k)$ ved hjelp av C-er med lavere n. Det er praktisk å sette opp en tabell av `long`, tenk på dette som å være tall ordnet i et rektangel av størrelse $(n-k+1)\times(k+1)$ (det er litt vridd i forhold til Pascals trekant, men noe enklere å kode). Vi bruker også at

$$C(n,0) = C(n,n) = 1$$

for å komme i gang. Her er koden som erklærer og initialiserer denne tabellen, og så bruker rekursjonen, for å regne ut $C(10,3)$:

```
int n=10;
int k=3;
long[][] Pascal = new long[n-k+1][k+1];
for (int i=0; i<n-k+1;i++){
        Pascal[i][0]=1;
}
for (int j=1;j<k+1;j++){
        Pascal[0][j]=1;
}
for (int i=1;i<n-k+1;i++){
        for(int j=1;j<k+1;j++){
                Pascal[i][j]=Pascal[i-1][j]+Pascal[i][j-1];
        }
}
```

Selv om dette er litt vridd sammenliknet med Pascals trekant, kan vi trekke ut binomialkoeffisienten $C(n,k)$ ved å skrive

```
System.out.printf("Binomialkoeffisienten C(%d,%d) er %d",\
 n, k, Pascal[n-k][k]);
```

Det kan være litt kjelkete å lese denne tabellen (på grunn av vridningen). Prøv selv å forstå hvilke binomialkoeffisienter som står hvor i tabellen `Pascal`. Her er koden som skriver dataene ut i firkant:

```
for (int i=0;i<n-k+1;i++){
        for(int j=0;j<k+1;j++){
        System.out.printf(" %d ", Pascal[i][j]);
        }
        System.out.printf("\n");
}
```

Oppgave 3.2.6 ber deg om å skrive ut binomialkoeffisientene i en trekant.

Oppgaver

Oppgave 3.2.1 *Logaritmer av fakulteter:*

Skriv et program som kan regne ut

$$\ln n!$$

og skrive ut svaret. Javakoden for den naturlige logaritmen er `Math.log`.

Hvor stort kan svaret bli før vi får problemer med grensene avhengig av typen til heltallet n? Kan du forbedre utregningen ved å bruke den multiplikative egenskapen til logaritmen? Du kan være fornøyd med svaret om du kan svare på dette spørsmålet: Er $\ln 2000!$ større eller mindre enn 14.000?

Oppgave 3.2.2 *Forbedring av den første binomialkoeffisient-utregningen:*

Hva skjer om det første programmet for utregning av binomialkoeffisienter får negative tall som inputs? Kontroller utregningen, og modifiser programmet slik at svaret blir 0 i denne situasjonen.

Oppgave 3.2.3 *Færre klasser.*

Erstatt klassene `Fakultet`, `Permutasjon` og `Binomial` med én klasse `KombinatorikkFunksjoner`. Denne klassen skal inneholde metoder for å regne ut disse tre operasjonene. Kontroller at det blir samme svar i utregninger med denne ene klassen som den opprinnelige løsningen med tre klasser.

Oppgave 3.2.4 *Catalan-tall.*

Catalan-tallene C_n er definert ved at

$$C_n = \frac{1}{n+1}\binom{2n}{n}.$$

Lag en klasse (eller legg til en metode i klassen `KombinatorikkFunksjoner`) som regner ut disse tallene. Se `http://www-math.mit.edu/~rstan/ec/catadd.pdf` for en liste over fenomener som angår disse tallene.

Oppgave 3.2.5 *Store binomialkoeffisienter.*

Kontroller at det rekursive programmet for utregning av binomialkoeffisienter også virker på store binomialkoeffisienter, som for eksempel $C(44, 17)$. Hva er det minste eksempelet du kan finne (lavest n, k) slik at også dette programmet gir galt svar for $C(n, k)$?

Oppgave 3.2.6 *Utskrift av binomialkoeffisienter.*

Modifiser programmet som regner ut binomialkoeffisientene rekursivt slik at utskriften blir trekantet som Pascals trekant. Prøv deg frem, og finn den største trekanten som ser ok ut på din datamaskin.

Kapittel 4

Divisjon og rest

For flyttall i Java oppfører divisjon seg slik man skulle tro (bortsett fra mulige avrundingsfeil):

```
double a = 243.2;
double b = 21.45;
System.out.printf("f/%f=%f", a,b, a/b);
```

gir svaret `243,200000/21,450000=11,337995`. For heltall er det litt annerledes, divisjon av heltall (unntatt divisjon med 0) gir nemlig alltid et heltall til svar i Java!

4.1 De elementære operasjonene / og %

Vi skal først se hvilke fenomener som dukker opp med heltallsdivisjonen og modulregningen.

Resultatet av divisjon

Heltallsdivisjon gir heltallsdelen av svaret, så Java sier for eksempel at

$$11/3 = 3.$$

Det rundes altså nedover. På samme måte er

$$-11/3 = 11/(-3) = -3.$$

Vi kan tenke på det som skjer for positive og for negative tall samlet som å "runde mot null". Helt presist kan vi si at resultatet av

$$n/m$$

er d dersom $md \leq n < (d+1)n$. Om vi først konverterer til en flyttallstype, får vi det korrekte svaret:

```
System.out.println(11/3);
System.out.println((float)11/(float)(3));
```

gir `3` og `3.6666667`. Hvor blir det av de siste 0,6666$\cdots$? Java tar ikke vare på dem, men det er en annen kommando som brukes for å få tak i den. Hvis vi deler tall for hånd blir resultatet noe sånt som dette:

$$\begin{array}{rrrcrcl} 1 & 2 & 0 & : & 7 & = & 17 + \frac{1}{7} \\ & 7 & & & & & \\ \hline & 5 & 0 & & & & \\ & 4 & 9 & & & & \\ \hline & & 1 & & & & \end{array}$$

der vi slutter når vi kommer under 7, og får derfor resten 1. Om vi vil skrive dette bare ved hjelp av hele tall, uten brøken 1/7, kan vi gange hele uttrykket med 7 og få

$$120 = 17 \cdot 7 + 1.$$

Det er denne fremgangsmåten Java tar som utgangspunkt, og gir altså 17 som svar på 120/7. Det generelle uttrykket kan vi skrive (for positive tall, i det minste), som at gitt to tall n og m, med $m \neq 0$, finnes det bestemte tall q (for "quotient") og r (for " rest" eller " remainder ") slik at

$$n = qm + r \text{ og } 0 \leq r < m.$$

Her vil n/m bli evaluert til q i Java. Det at divisjonen går opp betyr at resten er lik 0.

Rest og modulregning

I likningen

$$120 = 17 \cdot 7 + 1$$

representerer 1 resten når 120 deles på 7. Den kan vi få tak i i Java ved å skrive %, prosenttegnet. Vi kan teste at `System.out.println(120%7)` gir svaret `1`. Hvilke tall er det som gir rest 1 når vi deler på 7?

$$1,\ 8,\ 15,\ 22,\ 29, \cdots,\ n \cdot 7 + 1, \cdots .$$

Vi kaller også resten for "120 modulo 7", og regnereglene for operasjonen `%` kalles modulregning. De viktigste regnereglene er disse (se Mathema bind 1 side 30):

a) Selve definisjonen er at gitt to tall n og m, med $m \neq 0$, så er n modulo m lik r i likningen

$$n = qm + r \text{ og } 0 \leq r < m.$$

b) Sum:

$$n\%m + l\%m = (n + l)\%m$$

dersom $n\%m + l\%m < m$. Ellers kan vi trekke fra m, eller eventuelt ta $\%m$ en gang til:

$$(n\%m + l\%m)\%m = (n + l)\%m.$$

c) Produkt: Som for sum, men det er sjeldnere at $n\%m \cdot l\%m < m$. Vi skriver derfor direkte

$$(n\%m \cdot l\%m)\%m = (n \cdot l)\%m.$$

For å ta noen eksempler:

$$15\%4 + 2\%4 = 3 + 2 = 5$$

og

$$(15 + 4)\%4 = 19\%4 = 1 = 5\%4.$$

For multiplikasjon er

$$15\%4 \cdot 2\%4 = 3 \cdot 2 = 6$$

og

$$(15 \cdot 2)\%4 = 30\%4 = 2 = 6\%4.$$

Multiplikasjonstabeller

Vi skal nå sette opp to multiplikasjonstabeller i Java, en modulo 18, og en modulo 19. Siden vi kan få alle restene modulo 18 (for eksempel) ved å dele tallene $0, 1, 2, \cdots, 17$ på 18, trenger vi en 18×18-gangetabell. For å illustrere fremgangsmåten lager vi først gangetabellen modulo 4 for hånd. Da skal vi gange tallene $0, 1, 2, 3$ med hverandre, dele svaret på 4, og plukke ut resten. Tabellen blir

$(*)\%4$	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

Vi koder denne fremgangsmåten ved å bruke en størrelse `int antall` som kan være 18 eller 19, slik at vi kan bruke koden to ganger. Koden for å initialisere en tabell som kan inneholde svaret, og fylle den med de rette verdiene, er

```
int antall = 18;
int[][] Gangetabell = new int[antall][antall];
for (int i=0;i<antall;i++){
        for (int j=0;j<antall;j++){
                Gangetabell[i][j]=(i*j)%antall;
        }
}
```

Etter at dette er gjort kan vi skrive ut gangetabellen ved å skrive

```
for (int i=0;i<antall;i++){
        for (int j=0;j<antall;j++){
                System.out.printf("%2d ", Gangetabell[i][j]);
        }
        System.out.println();
}
```

Jeg har endret litt på dette for å få med også hvilke tall som ganges sammen, og da blir utskriften

```
    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
-----------------------------------------------------------
 0 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 1 | 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
 2 | 0  2  4  6  8 10 12 14 16  0  2  4  6  8 10 12 14 16
 3 | 0  3  6  9 12 15  0  3  6  9 12 15  0  3  6  9 12 15
 4 | 0  4  8 12 16  2  6 10 14  0  4  8 12 16  2  6 10 14
 5 | 0  5 10 15  2  7 12 17  4  9 14  1  6 11 16  3  8 13
 6 | 0  6 12  0  6 12  0  6 12  0  6 12  0  6 12  0  6 12
 7 | 0  7 14  3 10 17  6 13  2  9 16  5 12  1  8 15  4 11
 8 | 0  8 16  6 14  4 12  2 10  0  8 16  6 14  4 12  2 10
 9 | 0  9  0  9  0  9  0  9  0  9  0  9  0  9  0  9  0  9
10 | 0 10  2 12  4 14  6 16  8  0 10  2 12  4 14  6 16  8
11 | 0 11  4 15  8  1 12  5 16  9  2 13  6 17 10  3 14  7
12 | 0 12  6  0 12  6  0 12  6  0 12  6  0 12  6  0 12  6
13 | 0 13  8  3 16 11  6  1 14  9  4 17 12  7  2 15 10  5
14 | 0 14 10  6  2 16 12  8  4  0 14 10  6  2 16 12  8  4
15 | 0 15 12  9  6  3  0 15 12  9  6  3  0 15 12  9  6  3
16 | 0 16 14 12 10  8  6  4  2  0 16 14 12 10  8  6  4  2
17 | 0 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1
```

Om vi så endrer bare den første linjen til

```
int antall = 19;
```

trenger vi ikke endre noe annet av koden. Kjøring og utskrift gir da

```
     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
-----------------------------------------------------------
 0 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 1 | 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
 2 | 0  2  4  6  8 10 12 14 16 18  1  3  5  7  9 11 13 15 17
 3 | 0  3  6  9 12 15 18  2  5  8 11 14 17  1  4  7 10 13 16
 4 | 0  4  8 12 16  1  5  9 13 17  2  6 10 14 18  3  7 11 15
 5 | 0  5 10 15  1  6 11 16  2  7 12 17  3  8 13 18  4  9 14
 6 | 0  6 12 18  5 11 17  4 10 16  3  9 15  2  8 14  1  7 13
 7 | 0  7 14  2  9 16  4 11 18  6 13  1  8 15  3 10 17  5 12
 8 | 0  8 16  5 13  2 10 18  7 15  4 12  1  9 17  6 14  3 11
 9 | 0  9 18  8 17  7 16  6 15  5 14  4 13  3 12  2 11  1 10
10 | 0 10  1 11  2 12  3 13  4 14  5 15  6 16  7 17  8 18  9
11 | 0 11  3 14  6 17  9  1 12  4 15  7 18 10  2 13  5 16  8
12 | 0 12  5 17 10  3 15  8  1 13  6 18 11  4 16  9  2 14  7
13 | 0 13  7  1 14  8  2 15  9  3 16 10  4 17 11  5 18 12  6
14 | 0 14  9  4 18 13  8  3 17 12  7  2 16 11  6  1 15 10  5
15 | 0 15 11  7  3 18 14 10  6  2 17 13  9  5  1 16 12  8  4
16 | 0 16 13 10  7  4  1 17 14 11  8  5  2 18 15 12  9  6  3
17 | 0 17 15 13 11  9  7  5  3  1 18 16 14 12 10  8  6  4  2
18 | 0 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1
```

Hva er likt og hva er forskjellig i de to tabellene? Det som gjør den store forskjellen er at 19 er et *primtall*, altså et tall som ikke kan skrives som et produkt av to mindre heltall.

Javas heltallstyper med et fast antall bits regner modulo 2 opphøyd i antall bits, se Oppgave 4.3.3. Konsekvens av dette: Bruk typen `int`, som har 32 bits. Da vil Java regne modulo 2^{32}. Hvis vi definerer en `int` med verdi $2^{16} = 65536$, vil

$$2^{16} \cdot 2^{16} = 2^{16+16} = 2^{32} \text{ ha rest } 0 \text{ modulo } 2^{32}.$$

Test denne kodelinjen i Java:

```
System.out.println(65536*65536);
```

Kan du lage et tilsvarende fenomen med `long`?

Divisjon på null

For flyttall får vi ikke kompileringsfeil om vi forsøker å dele på null. Svaret blir enten $\pm\infty$ eller NaN. Disse spesielle verdiene finnes ikke for heltallstypene, så da får vi kompileringsfeil ved å forsøke å dele på null:

```
System.out.println(0.0/0.0);
System.out.println(1.0/0.0);
System.out.println(1/0);
```

gir

```
NaN
Infinity
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Kode etter divisjon med null vil ikke bli evaluert. Siden operasjonen % også innebærer en divisjon, får vi heller ikke lov til å skrive

```
System.out.println(5%0); // Kompileringsfeil
```

En siste observasjon er at divisjon med 1 alltid går opp, slik at for eksempel $23/1 = 23$ og $23\%1 = 0$.

4.2 Felles divisorer

Når vi jobber med divisjon er den enkleste situasjonen at divisjonen går opp. Hvis den ikke gjør det, kan det allikevel være at vi kan forkorte noe, slik at brøken vi får til svar er minst mulig. Det vil svare til at det tallet vi regner modulo blir mindre. For eksempel er

$$24\%18 = 6 \text{ og } 24 = 1 \cdot 18 + 6,$$

men vi kan forenkle dette litt først:

$$\frac{24}{18} = \frac{4 \cdot 6}{3 \cdot 6} = \frac{4}{3},$$

og vi får så

$$4\%3 = 1 \text{ og } 4 = 1 \cdot 3 + 1.$$

Helltallsdivisjonen i Java vil gi samme svar for 24/18 og 4/3, men siden vi har forkortet 6 får vi lavere tall å jobbe med. Hvordan kan vi finne det største tallet vi kan forkorte med? Hvis tallene som inngår er litt større, er det ikke så lett å se svaret direkte. Se for eksempel på 120 og 35. Om vi deler det største på det minste, og skriver utregningen bare med heltall, ser det slik ut:

$$120 = 3 \cdot 35 + 15.$$

Hvis vi kan dele 120 og 35 på noe, kan vi også dele resten 15 på det samme tallet! Og omvendt, om 35 og 15 kan deles på noe, må også 120 kunne deles på det samme tallet! Dermed vil det største tallet vi kan dele både 120 og 35 på være det samme

som det største tallet vi kan dele både 35 og 15 på! Vi har med andre ord byttet ut problemet med et tilsvarende problem, med lavere tall, og med samme svar! Vi kan derfor dele 35 på 15 for å komme videre:

$$35 = 2 \cdot 15 + 5.$$

Igjen vil et tall kunne dele både 35 og 15 hvis og bare hvis det både kan dele 15 og 5. Vi deler igjen:

$$15 = 3 \cdot 5 + 0.$$

Denne divisjonen gikk opp, og vi kan konkludere med at 5 er det største tallet som deler både 5 og 15, derfor det største som deler 15 og 35, derfor det største som deler 35 og 120. Svaret er altså at det største tallet som deler både 120 og 35 er 5.

Største felles divisor og Euklids algoritme

Det største tallet som deler to tall kalles tallenes *største felles divisor*. Det forkortes ofte *gcd* (="greatest common divisor"). Utregningen over viser da at

$$gcd(120, 35) = 5.$$

Utregningen kan formaliseres som en algoritme, ved å si: $gcd(a, b)$ regnes ut ved å dele det største av a og b på det minste (anta at $a > b$). Hvis divisjonen går opp er det minste tallet lik $gcd(a, b)$, altså $gcd(a, b) = b$. Ellers beholder vi det minste tallet b og resten r i divisjonen, og sier at $gcd(a, b) = gcd(b, r)$. Siden resten $r < b$ vil tallene som inngår nå være lavere, og vi kan gjøre dette gjentatte ganger. Før eller siden går divisjonen opp, og vi ender opp med å finne den største felles divisoren som den siste resten før det går opp.

Denne fremgangsmåten er ganske gammel, og kalles ofte Euklids algoritme (men den er eldre enn fra Euklids tid). Koden for den kommer her, og kan kanskje være litt forvirrende, vi skal se en mer oversiktlig variant snart.

```
public class Euklid {
    static long gcd(long a, long b){ // Vi skal finne gcd av a og b
    long q; //kvotient
    long r0, r1, r2; //rester (1)
    long svar;
    q = 1;
    r0 = Math.max(a,b); //(2)
    r1 = Math.min(a,b);
    r2 = r0%r1; // (3)
    svar=r1;
```

```
    while (r2>0){ // Resten er ulik 0
        svar = r2; //(4)
        r2 = r0% r1;//(4)
//      System.out.printf("%d = %d*%d+%d\n", r0, r0/r1, r1, r2); // (5)
        r0 = r1; //(1)
        r1 = r2;
    }
    return svar; //(6)
}
}
```

Vi trenger flere variable for å lagre restene, siden de skal bytte oppgave fra et skritt i algoritmen til det neste, se de to punktene merket (1). For å være sikker på at vi starter med å dele det største på det minste, bruker vi funksjonene `Math.max` og `Math.min` i punkt (2). I punkt (3) settes `r2` lik den første resten. Så lenge divisjonen ikke går opp, altså så lenge resten er større enn 0, deler vi `r0` på `r1`. Om denne divisjonen går opp, må vi returnere den forrige resten. Det er derfor den lagres som `svar` i Punkt (4). Punkt (5) har jeg kommentert ut. Om dette tas med, skriver algoritmen også ut alle skrittene i mellomregningen. Det kan være praktisk å ta med mens man skal lære seg algoritmen, men er i veien senere. Endelig kommer vi ut av løkken når divisjonen har gått opp (altså når `r2` er null), og returnerer `svar`, som nå inneholder den siste resten før divisjonen gikk opp.

Kjøring av (skriv dette i `main`)

```
System.out.println(Euklid.gcd(120,35));
```

gir svaret 5, om vi tar med kommentarlinjen merket (5) i koden, gir kjøringen

```
120 = 3*35+15
35 = 2*15+5
15 = 3*5+0
5
```

Denne algoritmen kan også kodes rekursivt, den kan kalle seg selv. Se på denne klassen:

```
public class gcd {
        static long gcd(long a, long b){
                long a1, b1;
                a1=Math.max(a,b);
                b1=Math.min(a,b);
                if (a1%b1 ==0){
                        return b1; //(1)
```

```
                }
                else
                {return gcd.gcd(b1, a1%b1); //(2)
                }
        }
}
```

Dersom divisjonen går opp returneres det minste tallet i Punkt (1). Om diivisjonen ikke går opp kalles `gcd.gcd` igjen med det minste tallet og resten. Denne måten å kode programmet på er kan skje nærmere måten vi faktisk tenker, og det blir færre linjer med kode. Hvilket alternativ man velger avhenger blant annet av smak, men også av krav til effektivitet. Mer om det senere!

4.3 Felles multiplum

På samme måte som vi kan spørre om to tall har en felles divisor, kan vi spørre om de har et felles multiplum. Det er altså et tall man kan få ved å gange det ene tallet med noe, og ved å gange det andre tallet med noe. For eksempel er 100 et felles multiplum av 2 og 25, siden $100 = 50 \cdot 2$ og $100 = 4 \cdot 25$. Et felles multiplum er alltid lett å finne: bare gang sammen de to tallene. Kan vi finne et mindre?

Ta de to tallene 4 og 6. Produktet av dem er 24, men også 12 er et felles multiplum. Det er let å sjekke at ingen lavere tall er et felles multiplum, så 12 er det minste felles multiplum av 4 og 6. Vi skriver ofte lcm = ("least common multiple") for dette, så

$$lcm(4, 6) = 12.$$

Om vi skal legge sammen brøk, er den mest hensiktsmessige fellesnevneren det minste felles multiplum av nevnerne. For eksempel er

$$\frac{1}{4} + \frac{5}{6} = \frac{3}{12} + \frac{10}{12} = \frac{13}{12}.$$

Hvordan kan vi regne ut dette tallet? Den enkleste måten, nå som vi allerede har en metode for å regne ut gcd, er å observere at

$$gcd(a, b) \cdot lcm(a, b) = a \cdot b.$$

For eksempel er $gcd(4, 6) = 2$ og $lcm(4, 6) = 12$, som passer med $2 \cdot 12 = 4 \cdot 6$. Vi kan derfor definere en ny metode i klassen `Euklid` (eller i `gcd`) som

```
static long lcm(long a, long b){
    return (a*b/Euklid.gcd(a,b));
}
```

Merk at heltallsdivisjonen her går opp, siden $gcd(a, b)$ er en faktor i a (og også i b). Faktisk vil også $lcm(a, b)/gcd(a, b)$ alltid gå opp.

Oppgaver

Oppgave 4.3.1 *Primtall.*

Husk at et primtall er et tall som ikke kan skrives som et produkt av to lavere heltall. For eksempel er 11 et primtall, mens $14 = 2 \cdot 7$ ikke er det. Skriv et program som undersøker om et heltall er et primtall eller ikke. Bruk programmet til å avgjøre hvilke(t) av tallene 11, 14, 49, 31999, 32001, 32003, 32005, 32007 som er primtall. **Hint**: Hvis tallet *ikke* er et primtall, må det kunne deles på et tall som er mindre enn eller lik kvadratroten av tallet.

Oppgave 4.3.2 *Modulo 8.*

Lag en gangetabell modulo 8.

Oppgave 4.3.3 *Modulo med negative rester.*

Skriv en gangetabell modulo 16, men bruk restene

$$-8, -7, \cdots, 0, \cdots 6, 7$$

heller enn $0, 1, \cdots, 15$. Sammenlikn med heltallstypen med 4 bits vi undersøkte i slutten av Kapittel 1. **Hint**: Hvis resten modulo 16 er ≥ 8, trekk fra 16 for å få den negative resten.

Oppgave 4.3.4 *Den manglende kronen:*

Tre venner leier seg inn på et (billig) hotell, og betaler 10 kroner hver til hotelleieren for de tre rommene, til sammen 30 kroner. Hotelleieren syns allikevel at han har krevd for mye betalt, og ber pikkoloen levere tilbake 5 kroner til de tre vennene. Pikkoloen (som ikke er så god i hoderegning) deler 5 på 3 i Java, og gir tilbake 1 krone til hver av vennene. Da har vennene betalt til sammen 27 kroner, og pikkoloen har 2. Hvor er den siste kronen?

Kapittel 5

Funksjoner og mengder

I dette kapittelet skal vi raskt se på hva en funksjon er, hva en mengde er, og hvordan Java takler disse begrepene. Vi skal undersøke hvorvidt funksjoner er veldefinerte, om en funksjon er enentydig, om den er på, og så videre. Funksjonsbegrepet er sentralt både i matematikk og i informatikk, så dette emnet passer bra til vårt kurs.

5.1 Mengder

En mengde består av forskjellige elementer. Vi skriver vanligvis mengder ved å liste opp elementene

$$A = \{a,\, b,\, \cdots ,\, \text{å}\}$$

eller ved å angi en regel

$$Jamn = \{n \text{ heltall } |n\%2 == 0\}.$$

Her er det brukt en blanding av matematikk-notasjon og Java-notasjon, men det bør være mulig å forstå hva mengdene A og $Jamn$ består av. Vi kan også beskrive disse mengdene på andre måter:

$$A = \{\text{norske bokstaver}\}$$

$$Jamn = \{\cdots ,-4,\, -2,\, 0\, 2,\, 4,\, \cdots\}$$

Merk at Java har et mengdebegrep (i Java-boken side 327), men vi skal gjøre ting litt enklere for oss ved å bruke tabeller. Strengt tatt kan ikke en mengde ha repetisjoner, mens tabeller kan ha det, men vi vil implisitt sikre oss mot eventuelle problemer dette vil medføre. Vi vil også stort sett se på mengder av tall, slik at sammenlikninger kan brukes, og gi de svarene vi regner med. For å la datamaskinen jobbe med

mengdene, vil vi også bruke *endelige* mengder. Dermed vil både A (bokstaver, ikke tall) og $Jamn$ (ikke endelig) være mengder vi ikke skal jobbe med, mens

$$\{0, 2, \cdots, 98, 100\}$$

og

$$\{p \text{ primtall} \mid p < 100\}$$

er eksempler på mengder vi vil jobbe med.

De tre viktigste operasjonene vi har for mengder er snitt, union og komplement. Kort sagt er snittet av to mengder det som er med i begge og unionen er det som er med i minst én av mengdene. En delmengde har et utvalg av elementene i en større mengde. Komplementet til delmengden består av de elementene i den store mengden som ikke er med i delmengden. Disse operasjonene står beskrevet i Kapittel 2.2 i Mathema.

For å ta noen eksempler, la $U = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ være de ti første tallene, og la oss se på delmengder av U.

$$A = \{x \in U | x \leq 5\} = \{0, 1, 2, 3, 4, 5\},$$

$$B = \{x \in U | x \text{ primtall}\} = \{2, 3, 5, 7\}$$

Da er A snitt B, skrevet $A \bigcap B$:

$$A \bigcap B = \{2, 3, 5\}$$

A union B, skrevet $A \bigcup B$:

$$A \bigcup B = \{0, 1, 2, 3, 4, 5, 7\}$$

Komplementet til A, $\overline{A}$, er

$$\overline{A} = \{6, 7, 8, 9\}$$

siden den store mengden U er gitt.

Programmering av operasjonene

For å utføre operasjonene, trenger vi å kunne sjekke om en verdi forekommer i en tabell. Det finnes mange måter å gjøre dette på, og dere vil lære om det i programmeringsfaget. Foreløpig tar vi en svært enkel, ikke særlig effektiv, variant. Bedre metoder vil dere lære for eksempel i algoritmefaget.

Vi starter med å definere noen mengder, slik at vi har noe å jobbe med. Mengdene vi definerer i Java nå er de samme vi allerede har sett på, men vi endrer navn.

```
int maks = 10; //  (1)
int[] univers = new int[maks];
for (int i=0;i<maks;i++){
        univers[i]=i;
}
int[] mengdeA = {0,1,2,3,4,5};// (2)
boolean[] mediA = new boolean[maks]; //(3)
int[] mengdeB = {2,3,5,7}; // (4)
boolean[] mediB = new boolean[maks]; //(3)
```

Vi må sette av 10 plasser til `univers` (Punkt (1)), siden vi skal ha tallene fra og med 0 til og med 9. For mengdene `mengdeA` og `mengdeB` lister vi opp alle elementene (punktene (2) og (4)). For å holde styr på hvilke elementer som er med, oppretter vi også en tabell av `boolean` til hver av dem (Punkt (3)), som skal holde styr på hvilke av verdiene fra `univers` som faktisk er i `mengdeA` eller `mengdeB`. For eksempel vil vi at `mediA[3]` skal være `true`, siden det fjerde elementet i `univers` er med i `mengdeA`. Her er en måte å fylle disse tabellene på:

```
for (int tall: univers){
        boolean aTest = false; //
        for (int aTall:mengdeA){
                if (tall==aTall){
                        aTest=true;
                }
        }
        mediA[tall]=aTest;
}
```

og tilsvarende for `mediB`. Denne koden er ikke særlig effektiv. Etter at en verdi er funnet, fortsetter vi å sjekke de andre verdiene. Vi har heller ikke brukt det at listene består av tall som er ordnet i stigende rekkefølge.

Fra de boolske tabellene `mediA` og `mediB` kan vi så definere snitt og union. Husk at `&&` er boolsk **og**, som gir sant bare hvis begge verdiene er sanne. Siden både `mediA[5]` og `mediB[5]` er sanne, vil `mediA[5]&&mediB[5]` være sann, så vi har funnet en verdi i snittet. Denne koden skriver ut snittet av `mengdeA` og `mengdeB`:

```
for (int tall:univers){
        if (mediA[tall] && mediB[tall]){
                System.out.printf("%d er i A snitt B\n", tall);
        }
}
```

Unionen finner vi ved å bruker boolsk **eller**, som i Java skrives `||`. Den gir sann hvis minst en av verdiene er sanne. For eksempel er `mediA[4]` sann og `mediB[4]` falsk, så `mediA[4]||mediB[4]` er sann, og vi har funnet en verdi i unionen. Se Oppgave 5.1.1.

Komplementet finner vi ved å bruke logisk negasjon, som i Java skrives som `!`. Den gir sann hvis verdien den brukes på er falsk, og omvendt. For eksempel er `mediA[6]` falsk, så `!mediA[6]` er sann, og vi har funnet en verdi i komplementet. Se Oppgave 5.1.2.

Med disse grunnoperasjonene kan mer kompliserte uttrykk konstrueres, om nødvendig.

Oppgaver

Oppgave 5.1.1 *Union:*

Skriv et program som finner og skriver ut unionen av to mengder (A og B fra teksten).

Oppgave 5.1.2 *Komplement:*

Skriv et program som finner og skriver ut komplementet til en delmengde (A fra teksten).

Oppgave 5.1.3 *De Morgans lover.*

De Morgans lover finnes i Mathema side 68 (mengder) eller Java-boken side 42 (boolsk). Skriv et program som sjekker at disse lovene holder for mengdene A og B fra teksten.

NB: Det siste eksempelet i avsnittet om de Morgans lover i Java-boken (rett før 2.11 Løkker, side 42) stemmer ikke. Hva er feil?

5.2 Funksjoner

En *funksjon* er en regel som til hvert element i en mengde tilordner et element i en annen mengde. Vi skal som eksempel bruke mengden U fra forrige avsnitt, og se på funksjoner som går fra denne mengden til seg selv. Vi har altså

$$U = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

og skal se på funksjoner som er definert her. Mer generelt er en *relasjon* på U en hvilken som helst delmengde av par av elementer fra U, altså mengden som består

$(0,0)$, $(0,1)$, $\cdots (3,7), \cdots (9,9)$, og en funksjon f er en spesielle relasjon som består av parene

$$(0, f(0)),\ (1, f(1)), \cdots, (9, f(9)).$$

Vi skal ikke tenke så mye på generelle relasjoner, men se Oppgave 5.2.1.

Et eksempel på en funksjon vi kan bruke er å fiksere et tall, for eksempel 4, og la $f(x) =$ resten når $4x$ deles på 10. Det virker, siden U akkurat består av verdiene som kan forekomme som rester modulo 10. Vi kan skrive ut disse funksjonsverdiene ved å bruke koden

```
int multMed = 4;
for (int tall:univers){
        System.out.printf("f(%d) = %d\n", tall, (tall*multMed)%maks);
}
```

`univers` og `maks` er som før, `univers` er tallene 0 til 9, og `maks`= 10.

Vi er egentlig interesserte i å lagre verdiene, slik at funksjonen beholdes i minnet. Derfor vil vi lage en dobbel tabell av typen `int[10][2]` som består av parene $(a, f(a))$. Den kan vi lage på denne måten:

```
        int[][] funksjonsTabell = new int[maks][2];
        for (int tall:univers){
                funksjonsTabell[tall][0] = tall; // tallet er a
                funksjonsTabell[tall][1] = (tall*multMed)%maks; //f(a)
        }
```

Da kan vi telle opp hvor mange ulike verdier funksjonen har ved å bruke denne koden:

```
int antallVerdier =0;
for (int verdi:univers){
        boolean fTest = false;
        for (int tall:univers){
                if (funksjonsTabell[tall][1]==verdi){//(1)
                        fTest = true;
                }
        }
        if (fTest){
                antallVerdier++;
        }
}
System.out.printf("Antall verdier er %d\n", antallVerdier);
```

Den virkelige testen er i Punkt (1). For hver mulige verdi sjekker vi om det finnes et tall slik at f av dette tallet er lik den mulige verdien. I så fall økes variabelen `antallVerdier` med 1.

Egenskaper ved funksjoner

Vi får mer interessante funksjoner ved å hente argumentene og verdiene fra forskjellige mengder. La oss omdøpe mengden U til `fraMengde`, og definere en ny mengde som funksjonen skal gå til, `tilMengde`, som vi tenker på som rester modulo 5:

$$tilMengde = \{0, 1, 2, 3, 4\}$$

La oss bruke den samme funksjonen vi hadde, men nå ta resten modulo 5 heller enn modulo 10. Vi ser altså på funksjonen som tar et tall fra `fraMengde`, et tall mellom 0 og 9, ganger det med 4, og tar resten modulo 5. Her er koden som intialiserer alt, og definerer funksjonen.

```
int maksFraMengde = 10; // ti minste tall
int maksTilMengde = 6; // seks minste tall
int multMed = 4;
int[] fraMengde = new int[maksFraMengde];
for (int i=0;i<maksFraMengde;i++){
        fraMengde[i]=i;
}
int[] tilMengde = new int[maksTilMengde];
for (int i=0;i<(maksTilMengde);i++){
        tilMengde[i]=i;
}
int[][] funksjonsTabell = new int[maksFraMengde][2];
for (int tall:fraMengde){
        funksjonsTabell[tall][0] = tall;
        funksjonsTabell[tall][1] = (tall*multMed)%maksTilMengde;
}
```

En funksjon kalles *surjektiv* eller *på* dersom alle elementene i mengden den går til, i dette tilfellet `tilMengde`, forekommer som $f(noe)$. Om vi teller opp antall ulike verdier, vil funksjonen være på dersom antallet verdier er lik antallet elementer i mengden funksjonen går til. Vi telte opp verdiene for en funksjon fra U til U tidligere, men det skal bare en liten endring til for å takle den nye situasjonen:

```
        int antallVerdier =0;
        for (int verdi:tilMengde){
                boolean fTest = false;
```

```
                for (int tall:fraMengde){
                        if (funksjonsTabell[tall][1]==verdi){
                                fTest = true;
                        }
                }
                if (fTest){
                        antallVerdier++;
                }
        }
        System.out.printf("Antall verdier er %d\n", antallVerdier);
```

Vi kan selvsagt også be Java sjekke om antallet er maksimalt, altså om antallet verdier er *maksTilMengde* (eller `tilMengde.length`). Det er bare testen

```
if (antallVerdier == maksTilMengde){
        System.out.println("Funksjonen er surjektiv");
}
```

Et annet viktig begrep er om en funksjon er injektiv (også kalt 1-1 eller enentydig). Det vil si at alle verdiene er ulike, altså at samme verdi ikke kommer for to ulike argumenter. For eksempel er ikke $f(x) = x^2$ injektiv, siden for eksempel $(-4)^2 = 4^2$, mens $f(x) = 2 * x$ er injektiv. For oss vil det nå si at antallet verdier er lik antall elementer i mengden vi går fra, altså *maksFraMengde*. Vi tester som før:

```
if (antallVerdier == maksFraMengde){
        System.out.println("Funksjonen er injektiv");
}
```

Om en funksjon både er injektiv og surjektiv, sies den å være bijektiv (eller 1-1 eller enentydig). Da oppretter funksjonen en korrespondanse mellom de to mengdene, slik at hvert element i hver mengde er i korrespondanse med nøyaktig ett element i den andre mengden. Det kan vi teste ved å kombinere de to testene vi har sett:

```
if (antallVerdier == maksTilMengde && antallVerdier == maksFraMengde){
        System.out.println("Funksjonen er bijektiv");
}
```

Siden vi ser på endelige mengder kan koblingen mellom injektive, surjektive og bijektive funksjoner sies litt sterkere, se Oppgave 5.2.2.

Oppgaver

For flere av oppgavene kan dere finne gode eksempler ved å variere `maksTilMengde`, `maksFraMengde` og `multMed` fra teksten.

Oppgave 5.2.1 *Relasjoner og funksjoner:*

Gitt en vilkårlig relasjon på U, skriv et program som sjekker om det er en funksjon. Det er to ting som må være på plass: Alle elementene i U må være med i definisjonsmengden til f (så alle a forekommer i et par $(a, f(a))$). Og hvert element i U kan bare være med i ett slikt par. Kontroller programmet ved å bruke det på eksempler du vet er funksjoner og eksempler du vet ikke er funksjoner.

Oppgave 5.2.2 *Injektivitet, surjektivitet og bijektivitet for endelige mengder.*

La $f : U \to V$ være en funksjon mellom to endelige mengder. Begrunn disse påstandene:

a) Hvis f er injektiv har V minst like mange elementer som U.

b) Hvis f er surjektiv har U minst like mange elementer som V.

c) Hvis f er bijektiv har U og V like mange elementer.

d) Hvis U og V har like mange elementer, er f injektiv hvis og bare hvis f er surjektiv hvis og bare hvis f er bijektiv.

Oppgave 5.2.3 *Inverse funksjoner.*

Hvis $f : U \to V$ er bijektiv, har den en inversfunksjon, se Mathema s. 88. Lag et program som sjekker om en funksjon er bijektiv, og i så fall finner inversfunksjonen.

Oppgave 5.2.4 *Sammensatte funksjoner.*

Om $f : U \to V$ og $g : V \to W$ er to funksjoner, kan vi danne den sammensatte funksjonen $g \circ f : U \to W$. Lag et program som finner en sammensatt funksjon. Sjekk i eksempler:

a) Hvis $g \circ f$ er injektiv, så er f injektiv.

b) Hvis $g \circ f$ er surjektiv, så er g surjektiv.

Kapittel 6

Programmering av heltall

I dette kapittelet skal vi programmere heltallene, slik at vi kan få Java til å regne med så store heltall vi vil ønske. Vi skal også programmere rasjonale tall. Det finnes selvsagt pakker for å gjøre dette, men vi vil forhåpentligvis lære ting av at vi gjør det selv. I tillegg skal vi se på rasjonale tall (brøk), og skal se at vi lettest kan kode dette som objekter.

6.1 Rasjonale tall

Vi skal lagre en brøk a/b som et par av heltall, som vi (treffende nok) kaller teller og nevner. Ved å bruke regnereglene for brøk kan vi fortelle Java hvordan vi vil at disse tallene skal håndtere pluss, gange og så videre. I tillegg vil vi redusere brøk, altså forkorte mest mulig, slik at vi får brøken representert ved minst mulige tall. Fordi vi allerede har implementert Euklids algoritme kan vi bruke den også i vårt nye program.

Representasjon av rasjonale tall

Et rasjonalt tall består av to heltall, telleren og nevneren. Nevneren kan ikke være null (det ville svare til å dele på null). Vi kan derfor definere en enkel klasse for å lagre et rasjonalt tall som

```
public class Brok {
        long teller;
    long nevner; //skal aldri =0
    Brok(){
        teller = 0;
        nevner = 1;
    }
```

```
        Brok(long over, long under){
        if (under != 0){
        this.nevner = under;
        this.teller = over;}
        else {System.out.format ("Nevneren kan ikke vaere null");
        }
    }
}
```

En brøk som opprettes uten verdier settes lik 0/1.

Til denne klassen skal vi lage flere ekstra metoder. Vi kan initialisere rasjonale tall (i `main`) ved å skrive f.eks.

```
        Brok enBrok = new Brok(14,16);
        Brok annenBrok = new Brok (3,0);
```

Den andre brøken vil programmet protestere mot, så vi bytter heller ut med

```
        Brok annenBrok = new Brok(3,2);
```

For å få tak i telleren til en brøk kan vi legge inn metoden

```
    long hentTeller(){
        return teller;
    }
```

og tilsvarende for `nevner`. I `main` kan vi da ha

```
System.out.printf("Teller til enBrok er %d\n", enBrok.hentTeller());
System.out.printf("Nevner til enBrok er %d\n", enBrok.hentNevner());
```

Vi bør også kunne skrive brøken ut noenlunde pent, denne metoden setter opp en brøkstrek som er like lang som det lengste tallet. Ikke ta denne koden så tungt, det er en detalj for oss.

```
    void skrivBrok(){
        String MAX = ((Long) Math.max(teller, nevner)).toString();
        int lengde = MAX.length();
        System.out.format("%-20d\n", teller);
        for (int i=0;i<lengde;i++){
            System.out.format("-");
        }
        System.out.format("\n%-20d\n", nevner);
    }
```

Grunnen til at tallet 20 dukker opp her, er at vi bruker typen `long` som ikke har mer enn 19 sifre.

Operasjoner på rasjonale tall

Vi skal definere følgende operasjoner for rasjonale tall: addisjon, multiplikasjon, divisjon, subtraksjon og forkorting. Java lar oss ikke definere symbolene `+`, `*` og så videre for egne klasser, så vi må finne på navn til operasjonene selv. Forkortingen innebærer å forkorte teller og nevner med største felles divisor, så da bruker vi Euklids algoritme som vi allerede har programmert. For addisjon trenger vi også fellesnevner, som vi fikk gratis da vi fant største felles divisor.

Vi starter med å legge inn klassen `Euclid` (denne så vi på i kapittelet om divisjon, gå tilbake dit om du trenger repetisjon):

```
public class Euclid {
        // gcd = storste felles divisor
        static long gcd(long a, long b){
                long a1, b1;
                a1=Math.max(a,b);
                b1=Math.min(a,b);
                if (a1%b1 ==0){
                        return b1; //
                }
                else
                {return Euclid.gcd(b1, a1%b1); //
                }
        }
        // lcm = fellesnevner = minste felles multiplum
            static long lcm(long a, long b){
        return (a*b/Euclid.gcd(a,b));
    }
}
```

For å forkorte mest mulig, som vi kan si er å redusere en brøk, definerer vi en ny metode i `Brok`:

```
    static Brok reduser(Brok enBrok){
        long enTeller = enBrok.hentTeller();
        long enNevner = enBrok.hentNevner();
        long gcd;
        gcd = Euclid.gcd(enTeller,enNevner);
        Brok redusertBrok = new Brok(enTeller/gcd,enNevner/gcd);
        return redusertBrok;
    }
```

Vi oppretter altså en ny brøk der telleren og nevneren er forkortet med største felles divisor.

Om vi nå skriver i `main`:

```
Brok enBrok = new Brok(14,16);
Brok enRedusertBrok = new Brok();
enRedusertBrok = Brok.reduser(enBrok);
enBrok.skrivBrok();
enRedusertBrok.skrivBrok();
```

får vi skrevet ut brøken 14/16 og så den forkortede brøken 7/8. Fordelen med å redusere brøk er at vi får lavere tall å jobbe med. I flere av de andre operasjonene vi skal se på vil vi redusere i mellomregningen.

La oss først se på multiplikasjon. Husk at brøk multipliseres ved at tellerne ganges sammen og nevnerne ganges sammen hver for seg:

$$\frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd}.$$

Vi kan gjøre dette i Java ved å bruke denne koden:

```
static Brok produkt(Brok enBrok, Brok toBrok){
    long enTeller = enBrok.hentTeller();
    long toTeller = toBrok.hentTeller();
    long enNevner = enBrok.hentNevner();
    long toNevner = toBrok.hentNevner();
    Brok ganget = new Brok(enTeller*toTeller, enNevner*toNevner);
    ganget = Brok.reduser(ganget);
    return ganget;
}
```

Vi henter tellere og nevnere, og ganger dem sammen hver for seg. Så reduserer vi svaret.

Divisjon fungerer på omtrent samme måte, men vi må snu den brøken vi skal dele på:

$$\frac{\frac{a}{b}}{\frac{c}{d}} = \frac{a}{b} \cdot \frac{d}{c} = \frac{ad}{bc}.$$

Se Oppgave 6.1.1.

Addisjon er litt vanskeligere, fordi vi trenger fellesnevner. Om vi skriver *lcm* for fellesnevneren (least common multiple), er formelen denne:

$$\frac{a}{b} + \frac{c}{d} = \frac{a}{b} \cdot \frac{lcm(b,d)/b}{lcm(b,d)/b} + \frac{c}{d} \cdot \frac{lcm(b,d)/d}{lcm(b,d)/d} = \frac{a \cdot lcm(b,d)/b + c \cdot lcm(b,d)/d}{lcm(b,d)}.$$

```
static Brok sum(Brok enBrok, Brok toBrok){
    long enTeller = enBrok.hentTeller();
    long toTeller = toBrok.hentTeller();
    long enNevner = enBrok.hentNevner();
    long toNevner = toBrok.hentNevner();
    long fellesNevner = Euclid.lcm(enNevner, toNevner);
    Brok summen = new Brok(enTeller*fellesNevner/enNevner +
            toTeller*fellesNevner/toNevner ,fellesNevner);
    summen = Brok.reduser(summen);
    return summen;
}
```

Vi bruker her formelen direkte, og reduserer svaret.

Subtraksjon virker på samme måte, men den enkleste måten å innføre subtraksjon på er kanskje å bytte fortegn i telleren til den ene brøken, som i

```
Brok minusBrok = new Brok(-enBrok.hentTeller(), enBrok.hentNevner());
minusBrok.skrivBrok();
```

Se Oppgave 6.1.2.

Til slutt skal vi teste når to brøker er like. Java lar oss ikke teste dette med `==`, se for eksempel på

```
Brok test1 = new Brok(4,5);
Brok test2 = new Brok(4,5);
System.out.println(test1==test2);
```

Utskriften her blir `false`. Vi må også ta hensyn til at brøker ikke trenger være reduserte, slik at $3/2 = 15/10$. For å omgjøre det til heltallsoperasjoner kan vi bruke at

$$\frac{a}{b} = \frac{c}{d} \text{ er ekvivalent med } ad = bc.$$

En måte å ordne dette på er ved å bruke denne koden:

```
boolean equals(Brok annenBrok){
    boolean erLik;
    long annenTeller = annenBrok.hentTeller();
    long annenNevner = annenBrok.hentNevner();
            erLik =
             (this.teller*annenNevner)==(this.nevner*annenTeller);
    return erLik;
}
```

Om vi da kjører

```
Brok enBrok = new Brok(14,16);
Brok annenBrok = new Brok (3,2);
Brok enRedusertBrok = new Brok();
enRedusertBrok = Brok.reduser(enBrok);
System.out.println(enBrok.equals(annenBrok));
System.out.println(enBrok.equals(enRedusertBrok));
```

får vi `false` fra den første testen, og `true` for den andre.

La oss til slutt observere at alt vi har gjort med rasjonale tall lider under de samme problemene som heltallstypene, vi kan få gale svar om tallene som inngår er for store.

Oppgaver

Oppgave 6.1.1 *Divisjon av brøk:*

Lag en metode `static Brok divisjon(Brok enBrok, Brok annenBrok)` som returnerer enBrok/annenBrok. Følg mønsteret for produkt i teksten. Husk spesielt å redusere svaret til slutt.

Oppgave 6.1.2 *Subtraksjon:*

Lag en metode `static Brok differanse(Brok enBrok, Brok annenBrok)` som returnerer enBrok-annenBrok. Gjør det på to måter, enten ved å kalle `Brok.sum` eller ved å bruke formelen direkte. Hvilken metode foretrekker du?

6.2 Addisjon av heltall

Vi skal først se på addisjon, siden multiplikasjon bruker addisjon i mellomregningen.

La oss se på et lite program, som adderer to heltall lagret som `char[]` (character array) som består kun av sifre, og som gir summen av de to tallene til svar. Svaret er også representert som `char[]`.

Java bruker ascii-verdiene til heltallene, siden `char`-typen internt er representert ved disse tallene. Ascii står for American Standard Code for Information Interchange, og angir tallkoder for vanlig brukte tegn. Java bruker egentlig Unicode, men for vårt bruk er ascii nok (og talltegnene er representert på samme måte i ascii og unicode). Utdraget fra ascii-tabellen som vi skal bruke er

Tegn	ascii	Tegn	ascii
'0'	48	'5'	53
'1'	49	'6'	54
'2'	50	'7'	55
'3'	51	'8'	56
'4'	52	'9'	57

Det betyr at vi kan få en `char` som representerer et siffer ved å konvertere fra heltall mellom 48 og 57. For å få sifferet 7 kan vi skrive

```
int j= 55;
char c = (char) j;
System.out.printf("Er %d = %c?\n", j, c); // litt dumt
```

Da vil c ha verdi lik sifferet 7. Mer elegant er det kanskje å bruke `'0'` heller enn 48, slik at sifferet 7 kan skrives

```
char c = (char)(7+'0');
```

Dette kan gjøre koden enklere å lese, og vi velger å bruke dette. Husk allikevel at Java tenker på `'0'` som å være 48.

La oss så lage et par tall med hundre tilfeldige siffer. For å kunne bruke `Random` må vi skrive

```
import java.util.Random;
```

og lage en generator ved

```
Random generator = new Random();
```

De to tallene vi vil plusse kan vi definere som `char[]` med hundre (for eksempel) plasser. Vi fyller dem med tilfeldige sifre med det samme:

```
char[] summand1 = new char[100];
char[] summand2 = new char[100];
for (int i=0;i<100;i++){
        summand1[i] = (char)(generator.nextInt(10)+'0');
        summand2[i] = (char)(generator.nextInt(10)+'0');
}
```

Random-funksjonen er laget slik at `generator.nextInt(10)` gir et av tallene $\{0, 1, \cdots, 10-1\}$, som vi så konverterer til tilsvarende `char` ved å legge til `'0'`.

Så skal vi legge sammen disse tallene. La oss først definere summen som skal holde svaret:

```
char[] sum = new char[101];
```

Merk at summen må ha en ekstra plass, siden vi kan få et siffer mer ved å legge sammen to tall. Siden vi skal legge sammen enerne først, lager dette litt krøll. For summandene er enerplassen posisjon 99, mens det for summen er plass 100. Derfor får vi en forskyvning i indeksen. Her er kode for å legge sammen tallene:

```
int mente = 0;
for (int i=100;i>0; i--){
        int j = summand1[i-1]+summand2[i-1]+mente -'0';
        if(j<='9'){
                sum[i] = (char)j;
                mente = 0;
        } else {
                sum[i] = (char)(j-10);
                mente = 1;
        }
}
sum[0]=(char)(mente +'0');
```

Variabelen `mente` lagrer det som er in mente, og vi må behandle det første sifferet spesielt til slutt. For å skrive ut svaret kan vi for eksempel skrive

```
System.out.print("  ");
System.out.println(summand1);
System.out.print(" +");
System.out.println(summand2);
System.out.print("=");
System.out.println(sum);
```

Java klarer nemlig å forstå at en `char` skal skrives ut som om det var en streng. Det kan allikevel skje rare ting dersom dere forsøker med diverse formatering. Eksperimenter og se hva som skjer!

Det vi har gjort er ikke så veldig elegant, men vi ser i hvert fall at vi klarer å addere tall som er større enn de tallene Java lar oss håndtere direkte. Et problem er at vi fortsatt bare aksepterer et fast antall sifre.

Hvis vi ville subtrahere kunne vi gjort noe liknende; den største forskjellen ville vært at vi måtte passet på at ascii-verdiene ikke kom for lavt, da måtte vi låne, og vi måtte hatt en måte å passe på fortegnet til svaret.

Oppgaver

Oppgave 6.2.1 *Ledende nuller:*

Modifiser programmet for summering av to store tall slik at utskriften ikke tar med sifferet 0 i starten. Hvis s starter 003456... skal programmet bare skrive ut 3456..., og tilsvarende for s1 og s2. Pass på at enerplassen i de forskjellige tallene står rett under hverandre!

Oppgave 6.2.2 *Subtraksjon*

Skriv et program som lager to tilfeldige tall på 100 sifre og skriver ut differansen mellom dem.

6.3 Multiplikasjon

Multiplikasjon kan vi gjøre ved å bruke addisjon gjentatte ganger, slik som i utregningen 1.1. I tillegg må vi passe på at svaret kan få flere sifre enn bare ett mer. Om du har gjort Oppgave 1.1.2 vet du at produktet av et tall med n sifre og et tall med m sifre maksimalt kan ha $n + m$ sifre. Vi må altså passe på å sette av dobbelt så mange plasser til svaret som vi setter av til hver av faktorene. For å gjøre ting enkelt fyller vi opp med nuller. Verdien til `antallTegn` er antallet tegn i faktorene. Her er koden som intitaliserer faktorene og produktet, lar faktorene bli vilkårlige, og ellers fyller inn med nuller.

```
int antallTegn = 4;
char[] faktor1 = new char[2*antallTegn];
char[] faktor2 = new char[2*antallTegn];
char[] produkt = new char[2*antallTegn];
Random generator = new Random();
for (int i=0;i<antallTegn;i++){
        faktor1[i] = '0';
        faktor1[i+antallTegn] = (char)(generator.nextInt(10)+'0');
        faktor2[i] = '0';
        faktor2[i+antallTegn] = (char)(generator.nextInt(10)+'0');
}
for (int i=0;i<2*antallTegn;i++){
        produkt[i]='0';
}
```

Vi kan dele opp multiplikasjonen i biter, ved først å få til å multiplisere et tall med mange sifre med et ettsifret tall. Dette legger vi i klassen `multSiffer`:

```
        static char[] mult(char siffer, char[] tall){
                char[] product = new char[tall.length];
                int mente =0;
```

```
            for (int i=tall.length-1;i>0; i--){
                int j = (siffer -'0')*(tall[i]-'0')+ mente+'0';
                product[i]=(char)((j-'0')%10+'0');
                mente = (j-'0')/10;
            }
            product[0]=(char)(mente + '0');
            return product;
}
```

Sammenliknet med det vi gjorde med addisjon tidligere, kan vi nå få større tall in mente. Den riktige verdien `mente = (j-'0')/10;` plukkes ut, og resten ved divisjonen er det rette tegnet i `product[i]=(char)((j-'0')%10+'0');`.

I tillegg til å gange med ett siffer, må vi kunne gange med 10, for så å gange svaret med neste siffer. Å gange med ti er bare å flytte sifrene ett hakk, og legge inn en 0 på slutten. Legg inn denne metoden i samme klasse `multSiffer`:

```
static char[] multMedTi(char[] tall){
        char[] gangerTi = new char[tall.length];
        for (int i = 0;i<tall.length-1;i++){
                gangerTi[i]=tall[i+1];
        }
        gangerTi[tall.length-1] = '0';
        return gangerTi;
}
```

Om vi så legger inn addisjonen fra forrige seksjon i en klasse `summen`, i en metode som også heter `summen`, kan vi prøve å gange de to tallene ved å kjøre denne koden:

```
for (int i=0;i<antallTegn;i++){
        produkt = summen.summen(multSiffer.mult
                (faktor1[2*antallTegn-1-i], faktor2), produkt);
        faktor2 = multSiffer.multMedTi(faktor2);
}
```

Når `i=0` multipliseres enerne i `faktor1` med `faktor2`. Når `i=1` er `faktor2` ganget med 10, og dette tallet ganges så med sifferet på tierplassen i `faktor1`, og sånn fortsetter det gjennom alle sifrene i `faktor1` (unntatt de nullene vi har innledet med). `produkt` oppdateres fortløpende, slik at vi til slutt forhåpentligvis får produktet ut.

Det ligger en vanskelighet i hvordan lengden av tallene skal behandles, siden den var hardkodet i addisjonen. I tillegg vil resultatet av addisjonen vi brukte alltid ha første siffer lik 0 eller 1, siden vi hadde satt av nøyaktig én plass ekstra til svaret. For å ordne disse to tingene kan vi lage klassen `summen` på denne måten:

```
public class summen {
static char[] summen(char[] tall1,char[] tall2){
        char[] sum = new char[tall1.length];
        int mente = 0;
        for (int i=tall1.length-1;i>-1; i--){
              int j = tall1[i]+tall2[i]+mente -'0';
              if(j<(10+'0')){
                                  sum[i] = (char)j;
                                  mente = 0;
              } else {
                                  sum[i] = (char)(j-10);
                                  mente = 1;
             }
        }
//sum[0]= (char) (mente + '0');
return sum;
}
}
```

Vi har nå fjernet `i=0` som en spesiell verdi, og tatt den med i løkken (der det nå står `i>-1`). Linjen som er kommentert bort er linjen som behandlet `i=0` tidligere.

Ved å endre på `antallTegn` får vi nå muligheten til å multiplisere sammen veldig store tall, inputs på 10.000 sifre er helt uproblematisk på min maskin (men det tar en stund å skrive det ut på skjermen).

Oppgaver

Oppgave 6.3.1 *Ledende nuller:*

Modifiser programmet fra teksten, slik at vi får svaret skrevet ut på en enkel og oversiktlig måte, og slik at de ledende nullene vi har innført ikke blir med i utskriften.

Oppgave 6.3.2 *Les inn tall:*

Modifiser programmet slik at inputtallene leses fra tastaturet. Hvis faktorene er kortere enn det antallet sifre du har avsatt, fyll inn med ledende nuller. **Hint:** Bruk `Scanner` og `nextLine` til å lese inn en streng fra tastaturet, og `.charAt(i)` for å få tak i det `i`-te tegnet.

6.4 Klassen Vector og tall med ukjent størrelse

Vi har sett at vi kan addere og multiplisere tall med en gitt størrelse, ved å representere tallene som `char[]`. Ulempen er at vi må sette av rett størrelse til tallene på forhånd, eventuelt med en sikkerhetsmargin. Java har en mulighet for å sette av plass dynamisk, altså bruke akkurat passe mye plass til hvert tall. Dette gjøres gjennom å bruke `Vector`, som vi skal beskrive nå. Dere skal lære mye mer om slike konstruksjoner senere, nå tar vi bare med det vi trenger for å lage et program som skal addere to tall med vilkårlig lengde. Merk at vår bruk av `Vector` ikke har noen direkte kobling til vektorene dere møter i matematikken.

En `Vector<Character>` av tegn kan vi tenke på som en `char[]` der vi ikke har angitt lengden på forhånd. Det er noen viktige forskjeller, og vi lister opp det vi trenger her. Man kan lage `Vector` med andre typer enn `Character` (ved å sette typenavnet i skarpe parenteser), men det har vi ikke bruk for akkurat nå. Se Oppgave 6.4.3 for et eksempel.

- `Vector<Character>` bruker `Character`, ikke `char`, så vi må passe på noen detaljer med det (se Java-boken side 70).
- For å legge nye elementer inn i en `Vector` bruker vi `add(tegn)`. Det forlenger vår `Vector` med 1, og legger det nye elementet inn bakerst. Om vi heller vil ha det inn først, kan vi bruke `add(0, tegn)` (eller en annen indeks enn 0 for å plassere det på et bestemt sted). Det er dessverre ganske kostbart, så ideelt sett vil vi ønske å bare bruke `add(tegn)`.
- For å hente verdier fra en `Vector` må vi bruke `elementAt(i)`, der `i` er indeksen til elementet vi vil hente. Som vanlig har det første elementet indeks 0.
- `length` er byttet ut med `size`.

I tillegg, siden vi bruker `Character`, kan vi bruke `charValue` for å få tilbake selve tegnet representert av et `Character`-objekt.

Programmet vi skal se på leser inn to tall av vilkårlig lengde fra tastaturet, og returnerer summen deres. For å gjøre koden enklest mulig vil vi igjen legge inn en del ekstra nuller.

Først må vi huske å importere `Vector`, og også `Scanner` siden vi skal ha input:

```
import java.util.Vector;
import java.util.Scanner;
```

Deklarasjon av det vi trenger:

```
Scanner tastatur = new Scanner(System.in);
Vector<Character> summand1 = new Vector<Character>();
Vector<Character> summand2 = new Vector<Character>();
Vector<Character> summen = new Vector<Character>();
```

Merk spesielt at vi ikke har angitt noen størrelse for de tre instansene av `Vector<Character>`. Deretter leser vi inn de to tallene som skal legges sammen, og fyller dem inn i `summand1` og `summand2` ved å bruke `add`:

```
System.out.println("Summand 1: ");
String input1 = tastatur.nextLine();
System.out.println("Summand 2: ");
String input2 = tastatur.nextLine();
// vektoren summand1 holder det ene tallet
for (int i=0;i<input1.length();i++){
        summand1.add(input1.charAt(i));
}
// vektoren summand2 det andre
for (int i=0; i<input2.length();i++){
        summand2.add(input2.charAt(i));
}
```

For å få tallene til å ha samme lengde fyller vi det korteste med nuller. Siden vi ikke vet hvilket av tallene som er lengst i utgangspunktet, må vi tenke oss litt om. Her er en mulig løsning:

```
//Fyll det korteste med nuller
while (summand1.size()>summand2.size()){
        summand2.add(0,'0');
}
while (summand1.size()<summand2.size()){
        summand1.add(0,'0');
}
```

Ikke glem at `char` må skrives med merker, `'0'` og ikke bare `0`. Etter at vi har fylt ut med nuller, vil `size` være den samme for begge summandene, så vi gir denne felles verdien navnet `lengde`. Deretter bruker vi omtrent akkurat samme kode som vi tidligere brukte for `char[]`, med noen få endringer.

```
int lengde = summand1.size();// felles lengde
int mente = 0;
for (int i=lengde-1;i>-1; i--){
        int j = summand1.elementAt(i).charValue()+
```

```
                summand2.elementAt(i).charValue()+mente -'0'; //(1)
        if(j<= '9'){
                summen.add(0, (char)j );
                mente = 0;
        } else {
                summen.add(0,(char)(j-10));
                mente = 1;
        }
}
if (mente != 0){ // (2)
summen.add(0,(char)(mente + '0'));
}
```

I Punkt 1 må vi bruke `elementAt` og `charValue`. I Punkt 2 legger vi ikke til et ekstra siffer bortsett fra om det trengs (altså hvis det ikke er null). Nå holder `summen` svaret av utregningen, og det eneste som gjenstår er å skrive ut svaret. Husk igjen at vi må bruke `elementAt`:

```
System.out.println("Vi regner ut summen: ");
for (int i=0;i<lengde;i++){
        System.out.print(summand1.elementAt(i));
}
System.out.print("\n + \n");
for (int i=0;i<lengde;i++){
        System.out.print(summand2.elementAt(i));
}
System.out.print("\n = \n");
for (int i=0;i<summen.size();i++){
        System.out.print(summen.elementAt(i));
}
```

Prøv å kjøre det endelige programmet, både med små tall hvor du lett kan se om svaret blir rett, og med lange tall som viser at programmet takler store inndata.

Oppgaver

Oppgave 6.4.1 *Multiplikasjon:*

Brukoppsettet for addisjon med `Vector`, samt det vi gjorde for multiplikasjon med `char[]`, til å multiplisere to vilkårlige heltall som leses inn fra tastaturet.

Oppgave 6.4.2 *Ledende nuller:*

En `Vector` kan også bli kortere, ved å bruke `remove`. Modifiser programmet i teksten ved å fjerne alle ledende nuller før svaret skrives ut. Sjekk at `size` faktisk går ned!

Oppgave 6.4.3 *Gang sammen flere tall:*

Bruk `Vector<Integer>` til å lagre et ukjent antall tall, alle forskjellig fra null, som leses inn fra tastaturet (bruk input 0 for å markere at det siste tallet er skrevet inn), og gang dem sammen.

Oppgave 6.4.4 *Unngå* `add(0,-)`*:*

Prøv å modifisere programmet slik at vi slipper å bruke den kostbare `add(0,tegn)`, men heller bare `add(tegn)` som setter nye elementer inn bakerst.

Kapittel 7

Grafbibliotek

Vi skal se på litt elementær grafikk, og for å lette det arbeidet har vi (det vil si Remy Monsen) laget et bibliotek med funksjoner for å tegne grafer. Vi skal først se på noen klassiske eksempler, grafer til funksjoner dere har sett på videregående. Deretter skal vi tegne litt mer spesielle ting, som et par fraktaler, strekkoder og projeksjoner av tredimensjonale objekter.

7.1 Selve biblioteket

Vi importerer et ikke-standard bibliotek i JCreator ved å velge "Project" og "Project Settings" fra menyen. Deretter går vi til "Required libraries", trykker på "New" og "Add archive". Det åpner for filvalg, og vi kan velge filen med biblioteket.

Den aktuelle filen ligger på It's learning som graf.jar. Last den ned til en passende mappe, og prøv det som er angitt over.

Dette biblioteket introduserer tre klasser, som heter `Graf`, `GrafSamling` og `Punkt`. I tillegg til at biblioteket må legges til som over, bruker vi `import` som for eksempel

```
import graf.Graf;
import graf.Punkt;
```

for å importere disse to klassene. Vi må lage lister over punkter for å få dette biblioteket til å tegne grafen. Mer avansert programvare kan få beskjeden "Tegn grafen til sinus", og så velger programmet selv hvilke punkter som trengs.

Innledende eksempel: parabel

Her er et eksempelprogram som tegner en parabel med likning $y = x^2$.

```
import graf.Graf;
import graf.Punkt;
// legg resten i main
Graf parabel = new Graf();
int antallPunkt = 100; // (1)
double endepunkt = 3.0; // (2)
double xKoord; // (3)
double yKoord;
xKoord = -endepunkt;
while (xKoord<endepunkt){
        yKoord = xKoord*xKoord;// (4)
        Punkt p = new Punkt(xKoord, yKoord);
        parabel.leggTilPunkt(p);
        xKoord += 2*endepunkt/antallPunkt;// (5)
}
parabel.visVindu();//(6)
```

Punkt (1) velger hvor mange punkter vi vil bruke for å tegne grafen. Det er mange måter man kan gjøre dette på, og dere vil se flere eksempler etterhvert. I punkt (2) velger vi `endepunkt` for å tegne grafen fra `-endepunkt` til `+endepunkt`, altså at x-koordinaten går over dette området. Merk at lengden på området er $2 \cdot endepunkt$. Variablene `xKoord` og `yKoord` (Punkt (3)) har opplagt betydning, og vi initialiserer `xKoord` til å være minus endepunktet. Inne i løkken regner vi ut `yKoord` ved å bruke funksjonen "Opphøyd i annen" (Punkt (4)), og lager et `Punkt` med gitte koordinater. Dette punktet legges til parabelen, og vi oppdaterer til slutt `xKoord` i Punkt (5). Her legger vi til lengden på området vi skal tegne grafen over, og deler på antallet punkter vi vil ha. Dermed får vi riktig antall punkter. Til slutt tegner vi grafen ved å bruke `visVindu`.

Merk at det høyre endepunktet faktisk ikke blir med her (prøv med ti punkter heller enn hundre). Hvorfor skjer det?

Det er ganske klart at vi bare ved å endre den ene linjen i Punkt (4) kan bytte ut denne funksjonen med en hvilken som helst annen funksjon vi kan beskrive i Java, se Oppgave 7.1.2.

Geometriske figurer

Grafbiblioteket har en metode som heter `settTegnLinje`. Om vi setter denne til `true` vil det trekkes linjer mellom punktene, i den rekkefølgen de angis. Dette kan vi bruke til å tegne en trekant, for eksempel:

```
double pi = 3.14;
Graf trekant = new Graf();
Punkt[] hjorner = new Punkt[3];
trekant.settTegnLinje(true);
for (int i=0;i<3;i++){
        Punkt p = new Punkt(Math.cos(i*2*pi/3), Math.sin(i*2*pi/3));
        hjorner[i]=p;
}
trekant.leggTilPunkter(hjorner);
trekant.visVindu();
```

Uttrykket med sinus og kosinus sier at hjørnene i trekanten skal velges på en sirkel med radius 1, og at de skal liogge jevnt fordelt.

Koden over har en litt uheldig mangel: Vi får bare tegnet to streker! For å få en tredje strek må vi repetere det første punktet (som altså også må være det fjerde); da får vi også en strek fra det tredje tilbake til det første punktet. Den enkleste måten å orden dette på er bare å la indeksen løpe ett hakk lenger!

```
Graf trekant2 = new Graf();
Punkt[] hjorner2 = new Punkt[4];
trekant2.settTegnLinje(true);
for (int i=0;i<4;i++){ // 3 gir samme punkt som 0
        Punkt p = new Punkt(Math.cos(i*2*pi/3), Math.sin(i*2*pi/3));
        hjorner2[i]=p;
}
trekant2.leggTilPunkter(hjorner2);
trekant2.visVindu();
```

Dette virker siden $\cos(x+2\pi) = \cos(x)$ og tilsvarende for sin. Det er lett å modifisere denne koden slik at vi får tegnet mangekanter med et annet antall hjørner. Prøv for eksempel denne:

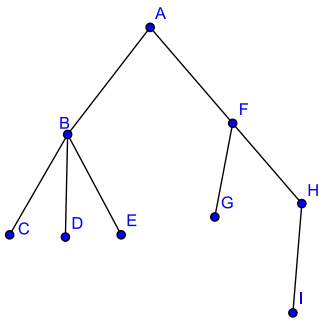
```
Graf tolvkant = new Graf();
Punkt[] tolvhjorner = new Punkt[13];
tolvkant.settTegnLinje(true);
for (int i=0;i<13;i++){ // 12 gir samme punkt som 0
        Punkt p = new Punkt(Math.cos(i*2*pi/12), Math.sin(i*2*pi/12));
        tolvhjorner[i]=p;
}
tolvkant.leggTilPunkter(tolvhjorner);
tolvkant.visVindu();
```

Om vi har en regulær mangekant med veldig mange kanter, kan vi knapt se forskjell mellom mangekanten og en sirkel. Om vi ikke vil ha en mangekant, kan vi tegne punktene såpass tett at det ser ut som en sirkel. Koden `Punkt(Math.cos(i*2*pi/12), Math.sin(i*2*pi/12))` plukker akkurat ut et punkt på en sirkel med radius 1 og sentrum i origo. Ved å bytte ut 12 med et stort tall, og la `i` gå fra 0 til det store tallet, kan vi få så mange punkter på sirkelen som vi måtte ønske. Se Oppgave 7.1.1

Trær

I denne seksjonen skal vi tegne et tre i samme forstand som dere lærer om i matematikken (en sammenhengende graf uten sykler). For å bruke biblioteket vårt må vi angi koordinater for alle hjørnene i treet, og så passe på at vi får tegnet streker mellom dem i rett rekkefølge. Tenk på det som å tegne opp hele treet uten å løfte blyanten fra arket!

Her er treet jeg vil bruke:



For å komme gjennom dette treet uten å løfte blyanten fra arket kan vi bruke rekkefølgen ABCBDBEBAFGFHI. Siden poenget med et tre er å se hvilke hjørner som henger sammen, er ikke koordinatene veldig viktige. Vi definerer punktene med omtrent passende koordinater (det er hentet fra programmet jeg brukte til å tegne treet i utgangspunktet), slik at tegningen ser omtrent ut som det vi ser på figuren, og så legger vi til punktene i den rekkefølgen som er angitt for å få tegnet strekene. Her er koden:

```
// Punkter med koordinater fra tegning
Punkt punktA = new Punkt(3.3, 4.9);
Punkt punktB = new Punkt(2.5, 3.9);
```

```
Punkt punktC = new Punkt(2,3);
Punkt punktD = new Punkt(2.5, 3);
Punkt punktE = new Punkt(3,3);
Punkt punktF = new Punkt(4,4);
Punkt punktG = new Punkt(3.9,3);
Punkt punktH = new Punkt(4.6,3);
Punkt punktI = new Punkt(4.5, 2.3);
Graf treet = new Graf();
treet.settTegnLinje(true);
//vi legger til punktene som i teksten
treet.leggTilPunkt(punktA);
treet.leggTilPunkt(punktB);
treet.leggTilPunkt(punktC);
treet.leggTilPunkt(punktB);
treet.leggTilPunkt(punktD);
treet.leggTilPunkt(punktB);
treet.leggTilPunkt(punktE);
treet.leggTilPunkt(punktB);
treet.leggTilPunkt(punktA);
treet.leggTilPunkt(punktF);
treet.leggTilPunkt(punktG);
treet.leggTilPunkt(punktF);
treet.leggTilPunkt(punktH);
treet.leggTilPunkt(punktI);
treet.visVindu();
```

Vi bør være litt misfornøyde med det vi akkurat har gjort. To grunner er at det er mye arbeid å skrive inn alle punktene, og vi burde kunne lagt til punktene bare ved å angi rekkefølgen på en eller annen måte. Løsningen på det første problemet ligger i filbehandling. Det hadde vært en fordel å skrive koordinatene i en fil, som vi så kunne lese og bruke til å definere punktene. Å hente informasjon fra fil vil dere lære om mot slutten av semesteret. Det andre problemet kunne vi løst allerede, ved å bruke `Punkt[]` og slå opp i tabellen ved hjelp av inputstrengen ABCBDBEBAFGFHI, men vi vil ikke forfølge det her.

Oppgaver

Oppgave 7.1.1 *Sirkel:*

Tegn en sirkel med radius 1 og sentrum i origo på to måter. Ta med nok punkter til at sirkelen ser tett ut (bruk kode av typen `Punkt(Math.cos(i*2*pi/12), Math.sin(i*2*pi/12))`)

med og uten å sette `settTegnLinje` til sann. For de to variantene, hvor mange punkter må du bruke for at det skal se noenlunde ok ut?

Tegn deretter en sirkel med angitt radius r og sentrum i et angitt punkt (a, b).

Oppgave 7.1.2 *Graftegning:*

Tegn grafene til funksjonene $f(x) = \sin x$, $g(x) = 2^x$, $h(x) = e^x$ ved å erstatte bare en linje i programmet fra teksten.

Tegn så grafene til $k(x) = \ln x$ og $l(x) = \log_2 x$. Her må det litt større endringer til, siden definisjonsmengden til disse funksjonene bare består av positive tall.

Hint: Bruk `Math.noe` for å finne funksjonene i Java.

Oppgave 7.1.3 *Tretegning:*

Tegn treet med 8 hjørner $ABCDEFGH$ der kantene er AB, AC, AD, AE, CF, CG og CH. Velg passe koordinater for punktene (tegn det gjerne på papir først)

7.2 Akser og flere grafer

Grafbiblioteket sjekker hvilke punkter vi angir, og bestemmet rekkeviddene for koordinatene basert på det. Men vi kan også overstyre dette, og angi grensene selv. Da må vi først skru av den automatiske utregningen i grafen `grafNavn` ved å skrive

```
grafNavn.settAutoSkala(false);
```

og så angi verdiene ved

```
grafNavn.settXMaks(2.2);
grafNavn.settYMaks(4.2);
grafNavn.settXMin(-1.2);
grafNavn.settYMin(-0.7);
```

om vi vil at x skal gå fra $-1{,}2$ til 2,2 og y fra $-0{,}7$ til 4,2. Om deler av grafen havner utenfor disse grensene, vil vi ikke få se alt.

Biblioteket tilbyt også en klasse `GrafSamling` som kan tegne flere grafer i samme vindu. Hvis autoskaleringen er slått på vil vinduets dimensjoner være bestemt av den første figuren, og om den andre (eller tredje, eller fjerde ...) er større vil det være uheldig, så vi vil vanligvis sette grensene selv i denne situasjonen.

Her er et eksempel som tegner trekanten og tolvkanten fra forrige seksjon i samme vindu.

```
import graf.Graf;
import graf.Punkt;
import graf.GrafSamling;
// resten inne i main
Graf trekant = new Graf();
Punkt[] trehjorner = new Punkt[4];
trekant.settTegnLinje(true);
for (int i=0;i<4;i++){ // 3 gir samme punkt som 0
        Punkt p = new Punkt(Math.cos(i*2*pi/3), Math.sin(i*2*pi/3));
        trehjorner2[i]=p;
}
trekant.leggTilPunkter(trehjorner);
Graf tolvkant = new Graf();
Punkt[] tolvhjorner = new Punkt[13];
tolvkant.settTegnLinje(true);
for (int i=0;i<13;i++){ // 12 gir samme punkt som 0
        Punkt p = new Punkt(Math.cos(i*2*pi/12), Math.sin(i*2*pi/12));
        tolvhjorner[i]=p;
}
tolvkant.leggTilPunkter(tolvhjorner);
GrafSamling begge = new GrafSamling();
begge.leggTilGraf(trekant);
begge.leggTilGraf(tolvkant);
begge.visVindu();
```

Om vi kjører denne koden vil den venstre delen av tolvkanten havne utenfor det vinduet som vises. Om vi bytter om på linjene, og legger til tolvkanten først, ser det bedre ut. Forsøk begge deler!
Om vi vil være sikre, bør vi slå av autoskaleringen på trekanten (om den legges inn først), og så sette grensene manuelt:

```
//legges inn der trekanten defineres
trekant.settAutoSkala(false);
trekant.settXMin(-1.2);
trekant.settXMaks(1.2);
trekant.settYMin(-1.2);
trekant.settYMaks(1.2);
```

Da vil vi ha satt av nok plass til begge figurene, også om trekanten legges inn i grafsamlingen først.

Oppgaver

Oppgave 7.2.1 *Farger:*

Finn ut fra dokumentasjonen hvordan fargene virker i grafbiblioteket. Tegn trekanten og tolvkanten i forskjellige farger!

Oppgave 7.2.2 *Skogtegning:*

Tegn en (liten) skog som består av to trær. Hjørnene er ABCDEFG, og kantene er AB, AC, DE, DF og DG.

7.3 To kompliserte eksempler

Vi skal nå se på to eksempler som er mer kompliserte. Hensikten med denne seksjonen er å vise at med en gang vi kan tegne enkeltpunkter kan vi tegne hva som helst. Det første eksempelet leser inn en tallkode (som 9788251921794, koden for Mathema 1) og koder det om til strekkoder. Det andre eksempelet lager et tredimensjonalt geometrisk objekt (en smultring) og tegner et todimensjonalt bilde av det.

Strekkoder

En strekkode er en grafisk representasjon av et 13-sifret tall. Det er konstruert for å kunne tolkes raskt og enkelt av maskiner (i kassen på butikken), som så bearbeider informasjonen og finner pris og varenavn fra en database.

Det første sifferet tegnes ikke, men forteller hvordan de neste seks sifrene skal kodes. De siste seks sifrene kodes alltid over et bestemt mønster. Strekene som er lenger enn de andre (start, slutt og midt i), ser alltid like ut. Kodingen er bit for bit, og hvert siffer kodes med 7 bits, som er kodet svart og hvit. Det er mye sikring, altså! For de siste seks sifrene er for eksempel 0 representert ved

$$0 = \{hvit, hvit, hvit, svart, svart, hvit, svart\}$$

For å representere dette enkelt lar vi svart svare til `true` og hvit til `false`. Dermed skal vi bare tegne noe når det står `true`. Om vi lagrer koden for å representere sifrene i en egen klasse, kalt sifferKode, med bare én metode

```
static boolean[] koding(int siffer)
//{.. returnerer en streng av 7 boolske verdier tilordnet hvert siffer}
```

kan vi fylle inn den siste delen av strekkoden ved

```
boolean[] del2 = new boolean[42];
int[] andreSekvens ={9,2,1,7,9,4};
for (int i=0;i<6;i++){
        hvertSiffer = sifferKode.koding(andreSekvens[i]);
        for (int j=0;j<7;j++){
                del2[7*i+j] = !hvertSiffer[j];
        }
}
```

Vi må sette av 42 plasser siden hvert av de seks sifrene tar opp 7 plasser.

For de første seks sifrene (unntatt det innledende) er det ulike regler, basert på hva det første sifferet er. For alle bøker er det første sifferet 9, så for å gjøre ting enkelt bruker vi bare det tilfellet. Koden vi bruker vil altså ikke virke for strekkoder som ikke starter med 9. `del1` og `forsteSekvens` har tilsvarende betydning som før, men vi fyller verdiene i `del1` ved denne koden:

```
for (int i=0;i<6;i++){
        hvertSiffer = sifferKode.koding(forsteSekvens[i]);
        for (int j=0;j<7;j++){
                if (i==0 || i== 3|| i==5){
                        del1[7*i+j] = hvertSiffer[j];
                } else{
                        del1[7*i+j]= !hvertSiffer[6-j];
                }
        }
}
```

La oss bare ta dette som en svart boks nå, men den som er interessert i å se mønsteret for andre tall, henvises til å søke opp `barcode` og `EAN-13` på wikipedia.

For å skrive dette ut i en graf tegner vi mange punkter. De lange strekene tegnes alltid på samme måte, de to delene tegnes basert på de boolske verdiene. Jeg har valgt midtpunktene til hver enkelt strek som et heltall, og om vi skal ha svart i f.eks. 13 fyller jeg ut med punkter fra $13-0{,}5 = 12{,}5$ til $13+0{,}4 = 13{,}4$. Det gjør at svarte streker ved siden av hverandre går i ett, og vi får et inntrykk av tykkere streker. Her er begynnelsen av koden, som tegner de første lange strekene og kodingen av de første seks sifrene:

```
Graf utKoden = new Graf();
double height = 4.0; //hoeyden paa strekene
int antallPunkt = 200; //passe tett til at figuren ser bra ut
// Vi skriver ut forste guard fra -45 til -43
```

```
// k brukes for aa faa litt tykkere streker
for(int j=-(antallPunkt/8); j<antallPunkt;j++){
        for (int k=-5;k<5;k++){
                Punkt p = new Punkt(-45.0+0.1*k,
                        (height*j)/antallPunkt);
                utKoden.leggTilPunkt(p);
        }
}
for(int j=-(antallPunkt/8); j<antallPunkt;j++){
        for (int k= -5;k<5; k++){
                Punkt p = new Punkt(-43+0.1*k,
                        (height*j)/antallPunkt);
                utKoden.leggTilPunkt(p);
        }
}
//Vi skriver ut del1 fra -42 til -1
for (int i =0;i<42;i++){
        if(del1[i]){
                for (int j=0;j<antallPunkt;j++){
                        for (int k=-4;k<5;k++){
                                Punkt q = new Punkt(-42+i+0.1*k,
                                 height*j/antallPunkt);
                                utKoden.leggTilPunkt(q);
                        }
                }
        }
}
```

De lange strekene har fått en ekstra åttendel under aksen. Om resten av strekkoden skrives ut på samme måte (fullstendig kode på It's learning), og vi til slutt viser vinduet `utKoden.visVindu();`, ser vi forhåpentligvis at strekkoden stemmer overens med det vi ser bak på boken.

Torus

En torus er et tredimensjonalt objekt, det ser i utgangspunktet ut som en smultring. Vi skal konstruere en torus i rommet, projisere den ned i et plan, og så tegne bildet i dette planet. Dette er en realistisk beskrivelse av hvordan datamaskinen håndterer tredimensjonal grafikk, dog noe forenklet. Vi kan tenke på torusen som at vi starter med en (stor) sirkel, og så lar en (liten) sirkel svinge rundt slik at sentrum ligger på den første sirkelen. Det objektet vi får skåret ut, er selve torusen. Vi kan parametrisere

en sirkel med radius r_1 som

$$(x, y) = (r_1 \cos(\alpha), r_1 \sin(\alpha))$$

der vinkelen α går fra 0 til 2π (eller fra 0° til 360°). Om vi så legger til en sirkel med radius r_2, som står vinkelrett på $xy-$planet og som har sentrum i det bestemte punktet, vil parametriseringen av denne sirkelen være

$$(x, y, z) = (r_1 \cos(\alpha)(1 + r_2 \cos(\beta)), r_1 \sin(\alpha)(1 + r_2 \cos(\beta)), r_2 \sin(\beta)).$$

Når både α og β går fra 0 til 2π får vi hele torusen.

Koden for å bruke denne parametriseringen, med et gitt antall punkter for hver sirkel, er

```
double r1 = 1.0; // stor radius
double r2 = 0.3; // liten radius
double pi = 3.14;
int antall = 100; // antall kontrollpunkt per sirkel
double[][] torusPunkter = new double[antall*antall][3];
for (int i=0;i<antall;i++){
      for (int j=0;j< antall; j++){
           int indeks = antall*i+j;
           // Standard parametrisering av torusen
           torusPunkter[indeks][0] = r1*Math.cos(i*2*pi/antall)*
                        (1+r2*Math.cos(j*2*pi/antall));
           torusPunkter[indeks][1] = r1*Math.sin(i*2*pi/antall)*
                        (1+r2*Math.cos(j*2*pi/antall));
           torusPunkter[indeks][2] = r2*Math.sin(j*2*pi/antall);
      }
}
```

Detaljene her er ikke så veldig viktige, vi trenger bare et noenlunde fornuftig tredimensjonalt objekt som vi forstår godt. Vi velger så et plan vi skal projisere dette til, slik at vi får noe vi kan tegne. Planet er bestemt av to passende vektorer, her er koden som velger dem.

```
double[] planVektor1 = new double[3]; // Ortonormalt koordinatsystem
double[] planVektor2 = new double[3]; // for projeksjonsplanet
planVektor1[0] = 1/Math.sqrt(2);
planVektor1[1] = 0.5;
planVektor1[2] = 0.5;
planVektor2[0] = 0.0;
planVektor2[1] = -1/Math.sqrt(2);
planVektor2[2] = 1/Math.sqrt(2);
```

Til slutt oppretter vi en graf og legger til alle punktene etterhvert som de blir projisert ned i planet:

```
Graf torusen = new Graf();
for (int i=0;i<antall*antall;i++){
        double koord1 = 0.0;
        double koord2 = 0.0;
        // Egentlig en matrisemultiplikasjon:
        for (int j=0;j<3;j++){
                koord1+=planVektor1[j]*torusPunkter[i][j];
                koord2+=planVektor2[j]*torusPunkter[i][j];
        }
        Punkt p = new Punkt(koord1, koord2);
        torusen.leggTilPunkt(p);
}
torusen.visVindu();
```

Senere, når vi skal lære om lineær algebra, skal vi forstå selve projeksjonen bedre. Da vil vi lettere forstå hvordan de tredimensjonale objektene egentlig virker, og ved å bruke projeksjon og graftegning får vi forhåpentligvis innsikt i hva som skjer.

Oppgaver

Oppgave 7.3.1 *Strekkoder med andre første sifre:*

Se på `barcodes`, og på `EAN-13` (som er navnet på strekkodessytemet vi bruker) på wikipedia, og prøv å endre koden for strekkoder slik at det stemmer når første siffer er noe annet enn 9. Prøv for eksempel med 7, som er det norske varer er kodet med som første siffer. Sjekk svaret ved å sammenlikne med noe du har kjøpt, og som du kan se strekkoden på!

Oppgave 7.3.2 *Modifikasjon av torusen:*

Prøv å multiplisere med 2 forskjellige steder i formlene for torusen (for eksempel bare i $z-$koordinaten). Kan du ved hjelp av grafen forstå hva som har skjedd tredimensjonalt?

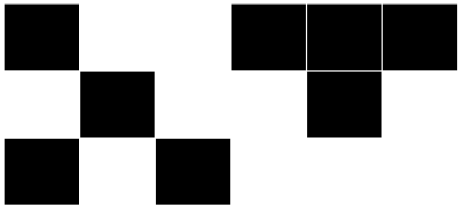
Kapittel 8

Fraktaler og repeterte operasjoner

I dette kapittelet skal vi se på tre eksempler på kompliserte prosesser, som kan forstås punkt for punkt. Det første eksempelet er egentlig ikke fraktalt (Game of Life), det andre er en repetisjon på en trekantkonstruksjon (Koch-kurven), og det tredje er et klassisk eksempel på en fraktal (Mandelbrotmengden). For de to første eksemplene skal vbi gå gjennom alle detaljene, for det tredje må vi akseptere noe kode om komplekse tall som vi tar for gitt.

8.1 Game of Life

Conways Game of Life er ikke egentlig et spill, men en prosess som gjentas lokalt vilkårlig mange ganger. Vi starter med et spillebrett med firkantmønster på, og markerer hvert felt på spillebrettet med svart (levende) eller hvit (død). For eksempel representerer dette et spillebrett med åtte levende:



Naboene til et felt er de åtte feltene som omgir det (over, under, venstre, høyre, og fire retninger på skrå).

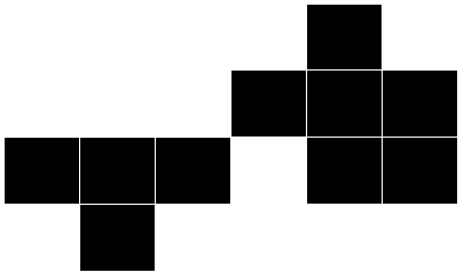
Hver gang tiden øker med 1, oppdateres spillebrettet etter følgende regler:

Et levende felt med to eller tre naboer forblir levende.

Et dødt felt med akkurat tre naboer blir levende.

Alle andre felter er døde.

Om vi bruker disse reglene på Figur 8.1 får vi situasjonen i Figur 8.1.



Om vi skal kode dette i Java må vi først bestemme oss for størrelsen på spillebrettet. For å gjøre koden litt enklere vil vi drepe ting som kommer til kanten, men det finnes andre løsninger på dette (man kan for eksempel tenke seg at brettet er limt fast på en torus, slik at det ikke er noen kanter). Vi trenger også å bestemme hvordan vi vil lagre verdiene levende/død for hvert felt på brettet. La oss velge kvadratiske spillebrett, lagret som `boolean[]`, med svart/levende som `true` og hvit/død som `false`. Da kan vi lage spillebrettet, og en startoppstilling, som

```
int brettKant = 8;
boolean[][] brett = new boolean[brettKant][brettKant];
brett[4][4] = true;
brett[4][3] = true;
brett[4][5] = true;
brett[3][2] = true;
brett[3][3] = true;
brett[3][4] = true;
```

Vi ha altså markert alle de levende feltene, og brukt at resten er satt til `false` bak kulissene. I dette startoppsettet har vi satt opp seks levende felter omtrent midt på brettet. Det vil være en del arbeid å sette opp starten, særlig om vi skal ha et stort brett. Hvis vi for eksempel hadde hatt et grafisk grensesnitt, der brukeren kunne markert de levende feltene med musen, ville dette vært enklere.

For å gå fram et steg må vi oppdatere brettet etter reglene. Da må vi først kunne telle opp antall naboer, og så bruke denne informasjonen til å oppdatere brettet. Vi

lager derfor en klasse `oppdater` som inneholder to metoder, en for å regne ut antall naboer (som mellomregning), og en metode som gir hele oppdateringen.

Her er den fullstendige koden for å telle opp antall naboer.

```
static int naboTall(boolean[][] brettet, int xKoord, int yKoord){
        int antall =0;
        for (int i=-1;i<2;i++){
                for (int j=-1;j<2;j++){
                        if (brettet[xKoord+i][yKoord+j]){
                                antall++;
                        }
                }
        }
        if (brettet[xKoord][yKoord]){
                antall--;//Punktet selv er ingen nabo
        }
        return antall;
}
```

Merk at punktet selv telles med om det er levende ((i,j)=(0,0)), så vi må eventuelt redusere antallet igjen.

Oppdateringen av brettet bruke så reglene for levende felter:

```
static boolean[][] nyttBrett(boolean[][] gammeltBrett){
        int stor = gammeltBrett.length;
        boolean[][] brettpluss = new boolean[stor][stor];
        for (int i =1;i<(stor-1);i++){
                for (int j=1;j<(stor-1);j++){
                        boolean overlever =false;
                        if (gammeltBrett[i][j] &&
                         (oppdater.naboTall(gammeltBrett, i,j))==2){
                                overlever=true;
                        }
                        if (gammeltBrett[i][j] &&
                         (oppdater.naboTall(gammeltBrett, i,j))==3){
                                overlever=true;
                        }
                        if (gammeltBrett[i][j]==false &&
                         (oppdater.naboTall(gammeltBrett, i,j))==3){
                                overlever=true;
                        }
        brettpluss[i][j] = overlever;
```

```
                }
        }
        return brettpluss;
}
```

Det er én ting man bør bite seg merke i her: Vi gjør aldri noe med kanten av brettet, altså det som svarer til `i=0` eller `i=stor` (eller tilsvarende for `j`). Det er fordi vi dreper det som kommer til kanten.

For å skrive en passende firkant lager vi en klasse feltTegning, som inneholder en enkelt metode for å tegne en passe stor firkant:

```
import graf.Graf;
import graf.Punkt;
public class feltTegning {
        static void lagFirkant(Graf brettet, int xKoord, int yKoord){
                double tetthet = 20.0;
                for (int i=0;i<tetthet;i++){
                        for (int j=0;j<tetthet;j++){
                                Punkt p = new Punkt(xKoord+i/tetthet,
                                 yKoord+j/tetthet);
                                brettet.leggTilPunkt(p);
                        }
                }
        }
}
```

Firkanten tegnes med nedre venstre hjørne i punktet (x, y).

I `main` styrer vi så hvor mange ganger vi skal oppdatere, og hvordan vi skal presentere svaret.

Vi går altså videre med det initialiserte brettet, og oppdaterer det et visst antall ganger. Siden oppdateringen tenkes å skje ved faste tidspunkter, kan vi kalle variabelen som angir antall oppdateringer for `tid`.

```
int tid = 9;
for (int t=0;t<tid;t++){
        brett = oppdater.nyttBrett(brett);
}
```

Vi initialiserer grafen, og fyller inn firkantene, på denne måten:

```
Graf brettTegning = new Graf();
```

```
brettTegning.settAutoSkala(false);
brettTegning.settXMin(-0.2);
brettTegning.settXMaks(brettKant + 0.2);
brettTegning.settYMin(-0.2);
brettTegning.settYMaks(brettKant + 0.2);
for (int i =1;i<(brettKant-1);i++){
        for (int j=1;j<(brettKant-1);j++){
                if(brett[i][j]){
                        feltTegning.lagFirkant(brettTegning,i,j);
                }
        }
}
brettTegning.visVindu();
```

Størrelsen på grafen er satt for å få tegningen til å se passe fin ut.

Oppgaver

Oppgave 8.1.1 *Varier antall oppdateringer på eksempelet i teksten:*

Prøv enten å tegne en graf for hver oppdatering, eller kjør programmet flere ganger med `tid` en større for hver gang (hvis det ellers blir for rotete med mange vinduer). Kan du gjette oppførselen mot uendelig?

Oppgave 8.1.2 *Skriv ut antall naboer:*

Siden klassen `oppdater` har en metode for å finne antallet naboer, skriv ut på skjermen antall naboer for hvert felt i brettet. Hint: Bruk denne formen:

```
for (int i =1;i<(brettKant-1);i++){
        for (int j=1;j<(brettKant-1);j++){
                // Sett inn koden her
        }
}
```

Oppgave 8.1.3 *Mønstre kan brukes til å generere brettet:*

Sett `brettKant=17` og initialiser brettet på denne måten:

```
int[] liste1 = {2,7,9,14};
int[] liste2 = {4,5,6,10,11,12};
for (int x:liste1){
        for (int y:liste2){
```

```
                brett[x][y]=true;
                brett[y][x]=true;
        }
}
```

Det er i utgangspunktet 48 levende felter. Undersøk hva som skjer når tiden går og dette brettet oppdateres gjentatte ganger.

Oppgave 8.1.4 *Variert input:*

Forsøk med varierende input, altså ulike startbrett, for å se hvordan spillet utvikler seg. Ting å prøve på er: Hvor tett kan du få de levende? Kan du lage eksempler der alt dør? Kan du lage eksempler der ting flytter seg bortover brettet?

Tips: Det finnes mange kjente konstellasjoner, ved å søke på nettet kan man finne interessante inputs.

Oppgave 8.1.5 *Rundt brett:*

Vi kan modifisere spille brettet ved å kreve at man kommer tilbake fra venstre om man forsvinner ut til høyre, og man kommer tilbake øverst om man forsvinner nederst. Det innebærer at om `siste` er den høyeste indeksen i tabellen skal naboene finnes med indeks `siste`, `siste-1` og `0`. På samme måte har felter med indeks `0` flere naboer. Forsøk å implementere dette. En test på om du har fått det til: Søk opp game of life"og gliderfor å finne inputdata med 5 levende felter, som da flytter seg på skrå rundt og rundt dette brettet!

Oppgave 8.1.6 *Tilfeldige brett:*

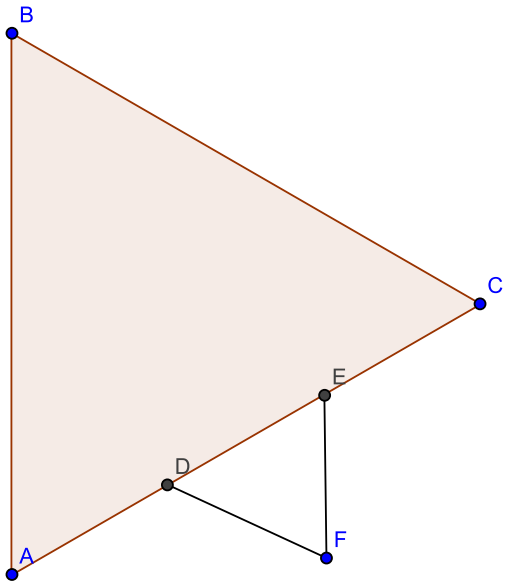
Vi kan lage et tilfeldig startbrett ved å bruke `Random`-funksjonen til Java. De fleste tilfedlig genererte brett vil sannsynligvis dø ut ganske fort. Forsøk å sette en prosentsats for hvor sannsynlig det skal være at et felt på startbrettet er levende, og se hvilken prosentsats som fører til at det går lengst før alt dør. Hint: Bruk random-metoden `.nextInt(100)` og velg levende dersom verdien er under prosentsatsen.

8.2 Koch-snøflaket

Koch-snøflaket er en av de enkleste fraktalene, og viser hvordan store mønstre kan sees repetert i mindre målestokk.

Start med en likesidet trekant. For hver side, tenk deg en likesidet trekant som festes midt på (på utsiden av den opprinnelige trekanten), med sider akkurat en

tredjedel av den du startet med. Behold den ytterste kanten, men fjern den midterste tredjedelen av den gamle siden.



I trekanten ABC starter vi med kanten AC, lager den likesidete trekanten DEF midt på AC, fjerner kanten DE, og tenker av vi skal gå ADFEC heller enn direkte AC. Vi gjør det samme for de andre sidene.

Den nye figuren består av 12 punkter med 12 kanter mellom. For hver av disse 12 kantene kan vi repetere prosessen, ved å erstatte den midtre tredjedelen med en likesidet trekant som stikker ut av figuren. For hver av de 12 kantene har vi altså lagt til tre nye punkter, i tillegg til de vi hadde fra før, så den nye figuren har 48 punkter og 48 kanter. Denne prosessen kan så fortsettes i det uendelige.

Hvor mye av dette kan vi tegne på datamaskinen ved å bruke grafbiblioteket vårt? Den iterative prosessen egner seg godt for implementering, men vi må først være sikre på matematikken som ligger bak utvelgelsen av punktene. Det å tegne en likesidet trekant har vi allerede gjort, så vi har i hvert fall et startpunkt. For å iterere trenger vi å kunne velge de tre nye punktene på en kant, vi må altså finne DEF om vi har gitt A og C på figuren. Vi må også passe på å legge til punktene til grafen i riktig rekkefølge, slik at strekene kommer der de skal.

Så anta vi har gitt to punkter $P = (p_x, p_y)$ og $Q = (q_x, q_y)$. Vi skal finne tre nye punkter; to av dem er ganske enkle: Det første punktet, R, ligger en tredjedel av veien mellom P og Q, så koordinatene til det er

$$R = (r_x, r_y) = (p_x, p_y) + \frac{1}{3}(q_x - p_x, q_y - p_y) = (\frac{2}{3}p_x + \frac{1}{3}q_x, \frac{2}{3}p_y + \frac{1}{3}q_y).$$

Det tredje punktet, T, ligger to tredjedeler av veien mellom P og Q, og vi finner tilsvarende at

$$T = (t_x, t_y) = (\frac{1}{3}p_x + \frac{2}{3}q_x, \frac{1}{3}p_y + \frac{2}{3}q_y).$$

For å finne det andre punktet, S, som ikke ligger på linjestykket mellom P og Q, må vi bruke litt trigonometri. Om vi ser på vinkelen med toppunkt i R, med et bein pekende mot Q og ett pekende mot S, er denne vinkelen $\pi/3$ (siden trekanten er likesidet, dette svarer til 60°). Det betyr at avstanden fra S til linjen PQ er sidekanten i den nye trekanten ganget med $\sin \pi/3 = \sqrt{3}/2$. Sidekanten er en tredjedel av avstanden mellom P og Q. En vektor som peker i riktig retning, og som har denne avstanden, er $\frac{1}{3}(q_y - p_y, p_x - q_x)$. Merk rekkefølgen på koordinatene her! Om vi kombinerer dette med at fotpunktet til S på PQ er midtpunktet, finner vi koordinatene til S som

$$\begin{aligned} S = (s_x, s_y) &= \tfrac{1}{2}(q_x + p_x, q_y + p_y) + \tfrac{\sqrt{3}}{2}\tfrac{1}{3}(q_y - p_y, p_x - q_x) = \\ &= (\tfrac{1}{2}(q_x + p_x) + \tfrac{\sqrt{3}}{6}(q_y - p_y), \tfrac{1}{2}(q_y + p_y) + \tfrac{\sqrt{3}}{6}(p_x - q_x)). \end{aligned}$$

Resten av arbeidet består av å holde styr på disse formlene.

En iterasjon

La oss først se på hva som skal til for å foreta *en enkelt* iterasjon, med start fra trekanten vi har generert tidligere. Husk å inkludere grafbiblioteket, og å importere `Graf` og `Punkt`. Punktene for trekanten er

```
double pi = 3.14;
Punkt[] trekantPunkt = new Punkt[4];
for (int i=0;i<4;i++){
        Punkt p = new Punkt(Math.cos(i*2*pi/3), Math.sin(i*2*pi/3));
        trekantPunkt[i]=p;
}
```

Deretter deler vi opp hver av de tre kantene, og legger til punkter ved å bruke formlene vi har funnet:

```
Punkt[] Koch1Punkt = new Punkt[13];
for (int i =0;i<3;i++){
        Punkt p = new Punkt(1,0);
        Punkt q = new Punkt(1,0);
        p = trekantPunkt[i];
        q = trekantPunkt[i+1];
        Punkt R = new Punkt(2*p.hentX()/3+q.hentX()/3,
          2*p.hentY()/3+q.hentY()/3);
        Punkt S = new Punkt((q.hentX()+p.hentX())/2+
          Math.sqrt(3)*(q.hentY()-p.hentY())/6,
          (q.hentY()+p.hentY())/2+
          Math.sqrt(3)*(p.hentX()-q.hentX())/6);
        Punkt T = new Punkt(p.hentX()/3+2*q.hentX()/3,
          p.hentY()/3+2*q.hentY()/3);
        Koch1Punkt[4*i]=p;
        Koch1Punkt[4*i+1]=R;
        Koch1Punkt[4*i+2]=S;
        Koch1Punkt[4*i+3]=T;
}
Koch1Punkt[12]=trekantPunkt[3];
```

Formlene for de tre nye punktene er såpass kompliserte at de går over flere linjer, men det er der det egentlige arbeidet er utført. Merk at punktet q ikke legges til, siden det vil være det første punktet i neste gjennomløping av løkken. Men dermed må også det aller siste punktet legges til utenfor løkken.

Vi kan tegne grafen ved å legge til punktene `Koch1Punkt`:

```
Graf Koch1 = new Graf();
Koch1.settTegnLinje(true);
Koch1.leggTilPunkter(Koch1Punkt);
Koch1.visVindu();
```

Den fullstendige konstruksjonen

Om vi skal iterere dette må vi holde styr på hvor mange punkter som trengs etter et gitt antall iterasjoner. Merk at det alltid er like mange punkter som linjestykker, men at det første punktet må tas med to ganger for å få lukket kurven. For hver kant legger vi til tre nye punkter, i tillegg til de punktene vi har fra før betyr det at vi ganger antall punkter med 4 i hver iterasjon. Siden vi starter med en trekant, må vi sette av

$$3 \cdot 4^i + 1$$

punkter i iterasjon nummer i. Vi starter med trekanten, og en passe stor dobbel tabell til å holde styr på alle punktene vi trenger.

```
int iterasjoner = 5;
Punkt[][] KochPunkt = new Punkt[iterasjoner+1]
                          [3*(int)Math.pow(4,iterasjoner)+1];
for (int i=0;i<4;i++){
        Punkt p = new Punkt(Math.cos(i*2*pi/3), Math.sin(i*2*pi/3));
        KochPunkt[0][i] = p;
}
```

Nå ligger trekanten i tabellen `KochPunkt[0]`. Den endelige kurven vår skal vi finne fra den siste tabellen, `KochPunkt[iterasjoner]`. Vi itererer på akkurat samme måte som da vi tok én iterasjon tidligere, men vi må huske på at antallet punkter øker med hver iterasjon, og at det siste punktet hele tiden må legges til spesielt. Her er koden:

```
for (int i=1;i<=iterasjoner;i++){
        //Lag KochPunkt[i] basert paa KochPunkt[i-1]
        for (int j = 0;j<3*Math.pow(4,i-1);j++){
                Punkt p = new Punkt(1,0);
                Punkt q = new Punkt(1,0);
                p = KochPunkt[i-1][j];
                q = KochPunkt[i-1][j+1];
                Punkt R = new Punkt(2*p.hentX()/3+q.hentX()/3,
                 2*p.hentY()/3+q.hentY()/3);
                Punkt S = new Punkt((q.hentX()+p.hentX())/2+
                 Math.sqrt(3)*(q.hentY()-p.hentY())/6,
                 (q.hentY()+p.hentY())/2+
                 Math.sqrt(3)*(p.hentX()-q.hentX())/6);
                Punkt T = new Punkt(p.hentX()/3+2*q.hentX()/3,
                 p.hentY()/3+2*q.hentY()/3);
                KochPunkt[i][4*j]=p;
                KochPunkt[i][4*j+1]=R;
```

```
                KochPunkt[i][4*j+2]=S;
                KochPunkt[i][4*j+3]=T;
            }
            KochPunkt[i][3*(int)Math.pow(4,i)]=
               KochPunkt[i-1][3*(int)Math.pow(4,i-1)];
}
```

Nå ligger den $i$te itererte Koch-kurven i `KochPunkt[i]`, og vi kan skrive ut den siste ved

```
Graf KochFull = new Graf();
KochFull.settPunktRadius(1);
KochFull.settTegnLinje(true);
KochFull.leggTilPunkter(KochPunkt[iterasjoner]);
KochFull.visVindu();
```

Grensekurven

Når vi itererer over denne konstruksjonen uendelig mange ganger, vil vi i grensen få en kurve med uendelig lengde, men som ligger i et endelig område i planet.

Hvis den opprinnelige trekanten har sidekanter av lengde l, vil kurven etter første iterasjon ha lengde $4l/3$. Det er fordi vi erstatter en tredjedel av hver sidekant med noe som er dobbelt så langt, så lengden ganges med

$$(1 - \frac{1}{3}) + 2 \cdot \frac{1}{3} = \frac{4}{3}.$$

Den samme faktoren ganges lengden med hver gang, så etter n iterasjoner er lengden $(4/3)^n l$, som går mot uendelig.

Grensekurven kalles *Koch-kurven*. Den er altså spesiell ved at den er uendelig lang, men lever i et begrenset område. Det er en såkalt enkel kurve, det vil si at den deler planet i to deler (innsiden og utsiden), der innsiden har endelig areal.

Oppgaver

Oppgave 8.2.1 *Et skjevt snøflak:*

Forsøk å endre på snøflaket bare ved å endre på den opprinnelige trekanten, slik at den ikke lenger er likesidet.

Oppgave 8.2.2 *Endret vektor :*

I utledningen av koordinatene til punktet S, som ikke ligger på linjen mellom P og Q, la vi sammen to vektorer. Bytt fortegn på vektoren som har faktoren $\sqrt{3}/2$ og se hva som skjer!

8.3 Mandelbrotmengden

Mandelbrotmengden er en av de mest berømte fraktalene, og har en enkel beskrivelse ved hjelp av komplekse tall. Komplekse tall er et stort tema i seg selv, men vi trenger bare noen ganske få egenskaper ved dem for å tegne mandelbrotmengden.

Komplekse tall - en innføring på 2 minutter

Et *komplekst tall* z består av to reelle (vanlige) tall a og b, som skrives samlet som $z = a + bi$. For eksempel er $2 + 3i$ og $0{,}7 - 1{,}4i$ komplekse tall. De legges sammen på den opplagte måten:

$$2 + 3i + 0{,}7 - 1{,}4i = 2{,}7 + 1{,}6i.$$

Multiplikasjon er vanskeligere, men der innfører vi en regel med at $i \cdot i$ erstattes med -1, og så ordnes alt deretter:

$$(a + bi)(c + di) = ac + adi + bci + bdii = ac - bd + (ad + bc)i.$$

I eksempelet er

$$\begin{aligned}
&(2 + 3i)(0{,}7 - 1{,}4i) = \\
&2 \cdot 0{,}7 + 2 \cdot (-1{,}4)i + 3 \cdot 0{,}7i + 3 \cdot (-1{,}4)ii = \\
&1{,}4 - 2{,}8i + 2{,}1i + 4{,}2 = \\
&5{,}6 - 0{,}7i
\end{aligned}$$

Vi kan representere et komplekst tall som et punkt i planet, eller en vektor. På den måten kan vi tilordne en *lengde* $\sqrt{a^2 + b^2}$. For eksempel er lengden av $2 + 3i$ lik $\sqrt{2^2 + 3^2} = \sqrt{13}$.

Definisjon av mandelbrotmengden

Start med et komplekst tall $z = a + bi$. Lag en følge av komplekse tall ved å starte med dette, og på hvert skritt gange det foregående med seg selv, og legge til det opprinnelige z:

$$\begin{aligned}
z_0 &= z \\
z_1 &= z_0^2 + z \\
z_2 &= z_1^2 + z \\
z_3 &= z_2^2 + z \\
&\vdots \\
z_n &= z_{n-1}^2 + z
\end{aligned}$$

Vi lar punktet (a, b) være hvitt om lengden i denne følgen går mot uendelig, svart ellers. Mandelbrotmengden består av de svarte punktene, altså de punktene der følgen ikke går mot uendelig.

Implementering av mandelbrotmengden

For å kunne få datamaskinen til å forstå mandelbrotmengden må vi først innføre komplekse tall. Det finnes mange pakker som gjør dette, og dere trenger ikke bry dere om detaljene her; den aktuelle klassen ligger på It's learning. Det eneste vi trenger å vite er at i klassen `Kompleks` kan vi sette verdien til et tall ved å angi a og b, og vi kan sjekke om det ligger i mandelbrotmengden ved å bruke den boolske metoden `mediMandelbrot`. Metoden er ikke perfekt, og den avhenger av en parameter, men det skjer bak kulissene.

Her tester vi om $z = 0{,}1 + 0{,}2i$ er med i mandelbrotmengden:

```
Kompleks tallz = new Kompleks();
tallz.settVerdi(0.1, 0.2);
System.out.println("Er tallet med i mandelbrotmengden?");
System.out.printf("%b", tallz.mediMandelbrot());
```

For å få tegnet mandelbrotmengden må vi teste en passende mengde punkter, og så legge til de punktene som er med i mengden til en graf. Her er kode for å tegne mengden over gridpunkter med tillegg `steg` i hver retning, i firkanten der x går fra `startx` til `stoppx` og tilsvarende for y.

```
double steg = 0.01;
Graf Mandelbrot = new Graf();
Mandelbrot.settPunktRadius(1);
double startx = -2.0;
double stoppx = 2.0;
double starty = -2.0;
double stoppy = 2.0;
double x = startx;
double y = starty;
while (x<stoppx){
        x+=steg;
        while (y<stoppy){
                y+=steg;
                Kompleks c = new Kompleks();
                c.settVerdi(x,y);
                if (c.mediMandelbrot()){
                        Punkt q = new Punkt(x,y);
```

```
                    Mandelbrot.leggTilPunkt(q);
                }
            }
            y = starty;
        }
        Mandelbrot.visVindu();
```

Se oppgavene for varianter.

Oppgaver

Oppgave 8.3.1 *Zoom:*

Velg et område på grafen, og lag et utsnitt av mandelbrotmengden på dette området. Repeter. Fortsett inntil du enten får problemer med oppløsningen, eller du ser mønstre som likner på det opprinnelige.

Oppgave 8.3.2 *Sett skala:*

Grafbiblioteket er laget slik at det setter grensene for vinduet basert på de svarte punktene. Det kan gjøre at utsnitt av mandelbrotmengden, som i forrige oppgave, kan bli skjeve. Modifiser programmet slik at vinduet som vises passer med grensene du har definert (`xstart` og så videre).

Oppgave 8.3.3 *Er 49 omtrent lik uendelig?*

Inne i klassen `Kompleks` ligger det en løkke opp til 50, og en test om antall iterasjoner er lik 49 (se programsnutten på It's learning). Juster begge disse tallene samtidig. Kan du på noen måte se forskjell på grafene om dette økes? Forsøk å zoome langt inn. Det testen faktisk sjekker er om vi etter 50 iterasjoner har kommer utenfor en sirkel med radius 2, og om vi ikke har det tipper vi at vi aldri vil gå mot uendelig.

Kapittel 9

Lineær algebra

I dette kapittelet skal vi se på mange fenomener fra lineær algebra. For å få datamaskinen til å jobbe med vektorer, må vi innføre koordinater. Da er det til gjengjeld lett å definere for eksempel en vektor i $\mathbb{R}^3$:

```
double[] enVektor = {1.0, -3.4, 2.8};
```

Vi skal programmere de vanligste operasjonene som skalarprodukt og vektorprodukt, og noen av algoritmene som Gauss-Jordan-eliminasjon (rekkereduksjon) og Gram-Schmidt (for å finne gode koordinatsystemer). Vi skal studere lineære transformasjoner som rotasjoner, speilinger og projeksjoner, og kombinere dette med å tegne figurer i grafbibilioteket.

9.1 Grunnleggende vektoroperasjoner

Vi kommer til å forutsette at alle vektorene vi jobber med er tredimensjonale. Dersom man i en eller annen sammenheng har spesielt bruk for vektorer i planet, for eksempel, kan man sette den tredje koordinaten til null. For å få mest mulig fleksibilitet i utregningene, kommer alle vektorer og skalarer til å være av typen `double`. La oss så se på de operasjonene vi har mest bruk for. Vi angir kode for de forskjellige operasjonene, men det må selvsagt ordnes i klasser på en hensiktsmessig måte. Når noen metoder bruker andre metoder i mellomregningen, kaller jeg dem ved å referere til en klasse `produkter`.

Operasjoner på én vektor

Lengden av en vektor $\vec{v} = (v_1, v_2, v_3)$ er $|\vec{v}| = \sqrt{v_1^2 + v_2^2 + v_3^2}$. Vi må bare huske at kvadratroten finnes som `Math.sqrt`, og så er koden

```
static double lengde(double[] enVektor){
        double resultatet =0.0;
        for (int i=0;i<3;i++){
                resultatet+= enVektor[i]*enVektor[i];
        }
        resultatet = Math.sqrt(resultatet);
        return resultatet;
}
```

Vi kan også gange en vektor med et tall (en skalar), ved å gange alle komponentene med den samme skalaren. Det er også ganske enkelt:

```
static double[] skalarMult(double enSkalar, double[] enVektor){
        double[] resultatet = new double[3];
        for (int i=0;i<3;i++){
                resultatet[i]=enSkalar*enVektor[i];
        }
        return resultatet;
}
```

Operasjoner på to vektorer

Summen av to vektorer tas komponent for komponent:

```
static double[] sumVektor(double[] enVektor, double[] annenVektor){
        double[] resultatet = new double[3];
        for (int i=0;i<3;i++){
                resultatet[i] = enVektor[i]+annenVektor[i];
        }
        return resultatet;
}
```

Det er to varianter av produkt mellom vektorer, skalarproduktet og vektorproduktet. De kalles også for prikkproduktet og kryssproduktet, henholdvis, på grunn av symbolene som vanligvis brukes for dem.

I matematisk notasjon, med $\vec{v} = (v_1, v_2, v_3)$ og $\vec{u} = (u_1, u_2, u_3)$, er skalarproduktet

$$\vec{v} \bullet \vec{u} = v_1u_1 + v_2u_2 + v_3u_3.$$

Vi koder det som

```
static double skalarProdukt(double[] enVektor, double[] annenVektor){
        double resultat = 0.0;
```

```
        for (int i=0;i<3;i++){
                resultat += enVektor[i]*annenVektor[i];
        }
        return resultat;
}
```

Svaret her er altså et tall, en skalar.

Vektorproduktet er

$$\vec{v} \times \vec{u} = (v_2u_3 - v_3u_2,\ -(v_1u_3 - v_3u_1),\ v_1u_2 - v_2u_1).$$

Heller enn å huske denne formelen, er det praktisk å huske andre måter å regne det ut på (for eksempel ved å sette opp en determinant), men en direkte formel er grei å bruke for datamaskinen, og kan da kodes like lett som det foregående:

```
static double[] vektorProdukt (double[] enVektor, double[] annenVektor){
        double[] resultatet = new double[3];
        resultatet[0] = enVektor[1]*annenVektor[2]
          -enVektor[2]*annenVektor[1];
        resultatet[1] = -(enVektor[0]*annenVektor[2]
          -enVektor[2]*annenVektor[0]);
        resultatet[2] = enVektor[0]*annenVektor[1]
          -enVektor[1]*annenVektor[0];
        return resultatet;
}
```

Disse to produktene har også alternative beskrivelser ved hjelp av trigonometriske funksjoner. For eksempel er

$$\vec{v} \bullet \vec{u} = |\vec{v}||\vec{u}| \cos(\alpha),$$

der α er vinkelen mellom de to vektorene. Siden vektorproduktet faktisk er en vektor, må vi også angi en retning. Vektorproduktet står vinkelrett på begge de to vektorene som multipliseres, og peker i den retningen som gjør at $\vec{v}, \vec{u}, \vec{v} \times \vec{u}$ danner et høyresystem. Lengden er

$$|\vec{v} \times \vec{u}| = |\vec{v}||\vec{u}| \sin(\alpha)$$

der α fortsatt er vinkelen mellom de to vektorene.

Vi kan bruke de to formlene for skalarproduktet til å regne ut vinkelen mellom to vektorer. I matematisk formulering er

$$\alpha = \arccos\left[\frac{\vec{v} \bullet \vec{u}}{|\vec{v}||\vec{u}|}\right].$$

arccos, eller *arcus cosinus*, er den inverse funksjonen til cosinus. Alt som står på høyre side har vi allerede kodet, så vi kan skrive dette i Java på én linje (lang, riktignok):

```
static double vinkel (double[] enVektor, double[] annenVektor){
  return Math.acos(produkter.skalarProdukt(enVektor, annenVektor)/
  (produkter.lengde(enVektor)*produkter.lengde(annenVektor)));
}
```

Vinkelen angis i radianer, slik at vektorer som står vinkelrett på hverandre har vinkel $\pi/2$, for eksempel.

Om vi har gitt to vektorer, $\vec{u}$ og $\vec{v}$, kan vi skrive $\vec{u}$ på en entydig måte som en sum av en vektor parallell med $\vec{v}$ og en vektor vinkelrett på $\vec{v}$. Den første av disse kalles *vektorprojeksjonen* av $\vec{u}$ på $\vec{v}$, og vi kan regne ut (ved trekantberegninger) at formelen for denne vektorprojeksjonen er

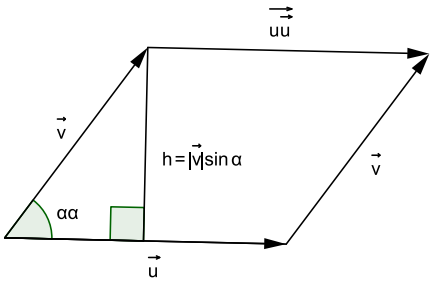
$$\frac{\vec{u} \bullet \vec{v}}{|\vec{v}|^2}\vec{v}.$$

Om vi får noe negativt ganget med $\vec{v}$, betyr det at den vektorprojeksjonen peker i motsatt retning av $\vec{v}$. For den vinkelrette komponenten kan vi så trekke fra vektorer, se Oppgave 9.1.3.

Vi presiserer ofte at en projeksjon skal oppfattes som en vektorprojeksjon, fordi det også finnes et begrep for *skalarprojeksjon*, som er lengden av vektorprojeksjonen, eventuelt med fortegn.

Areal og volum

I parallellogrammet utspent av to vektorer $\vec{u}$ og $\vec{v}$ ser vi at grunnlinjen har lengde $|\vec{u}|$ mens høyden har lengde $|\vec{v}|\sin(\alpha)$:



Arealet er altså lengden av vektorproduktet.

Om vi i tillegg har en tredje vektor $\vec{w}$ kan vi se på parallellepipedet utspent av dem. Da er grunnflaten det vi akkurat har regnet ut, altså lengden av vektorproduktet. Høyden får vi ved å projisere $\vec{w}$ på vektorproduktet, som altså har lengde $|\vec{w}|\cos(\beta)$, der β er vinkelen mellom $\vec{w}$ og $\vec{u} \times \vec{v}$. Om vi samler dette får vi at volumet er

$$|\vec{w} \bullet (\vec{u} \times \vec{v})|$$

Vi må ha på absoluttverditegn fordi fortegnet på det vi regner ut avhenger av orienteringen av de tre vektorene, mens volumet skal være positivt. Denne størrelsen kalles ofte *trippelproduktet*.

Oppgaver

Oppgave 9.1.1 *Grader:*

Modifiser utregningen av vinkelen mellom to vektorer, slik at det er en ekstra mulighet til å få vinkelen i grader (men default skal være radianer).

Oppgave 9.1.2 *Vinkel fra vektorprodukt:*

Forsøk å få ut vinkelen mellom to vektorer fra de to formlene for vektorproduktet. Hva kan gå galt?

Oppgave 9.1.3 *Vinkelrett komponent:*

Basert på vektorprojeksjon, finn et uttrykk for den vinkelrette komponenten av $\vec{u}$ med hensyn på $\vec{v}$. Skriv kode for å regne dette ut.

Oppgave 9.1.4 *Avrundingsfeil:*

Siden vi arbeider med flyttall må vi regne med at det hele tiden introduseres avrundingsfeil i operasjonene våre. For eksempel er det sant at to vektorer (ulike nullvektoren) står vinkelrett på hverandre hvis og bare hvis skalarproduktet mellom dem er null. Men på grunn av avrundingsfeil vil våre skalarprodukter i Java nesten aldri bli null. Løs dette problemet! Hint: Sett en skranke (for eksempel $\epsilon = 0{,}0001$) og si at noe er null dersom absoluttverdien av det er mindre enn ϵ. (Tegnet ϵ er den greske bokstaven epsilon).

9.2 Grunnleggende matriseoperasjoner

På grunn av at vi primært er interessert i vektorer i rommet, kommer (nesten) alle matrisene våre til å være $3 \times 3-$matriser. Det er to unntak: Vi skal også se på $2 \times 3-$matriser for å forstå projeksjoner ned i planet (eller ut på skjermen), og vi skal se på mer generelle matriser i forbindelse med likningssystemer i neste seksjon.

En matrise er altså i utgangspunktet av type `double[3][3]`. Matriser kan adderes og ganges med en konstant (så mengden av $3 \times 3-$matriser danner et nidimensjonalt vektorrom), og koden for det er ganske lik koden for vektorer. Se Oppgave 9.2.1.

Matrisemultiplikasjon

Matrisemultiplikasjon er definert ved at hver rekke i den første matrisen ganges (det vil si, vi tar skalarprodukt) med hver søyle i den andre matrisen. Om vi husker koden for skalarmultiplikasjon av vektorer er koden for å multiplisere matriser

```
static double[][] matriseMult (double[][] enMatrise,
          double[][] annenMatrise){
        double[][] resultatet = new double[3][3];
        double[] rekke = new double[3];
        double[] soyle = new double[3];
        for (int i=0;i<3;i++){
                rekke[0] = enMatrise[i][0];
                rekke[1] = enMatrise[i][1];
```

```
                rekke[2] = enMatrise[i][2];
                for (int j=0;j<3;j++){
                        soyle [0] = annenMatrise[0][j];
                        soyle[1] = annenMatrise[1][j];
                        soyle[2] = annenMatrise[2][j];
                        resultatet[i][j]= produkter.
                         skalarProdukt(rekke, soyle);
                }
        }
        return resultatet;
}
```

Matriser virker på vektorer ved de samme prinsippene som for matrisemultiplikasjon, med at hver rekke i matrisen ganges (altså skalarprodukt) med vektoren, og vi får en ny vektor. Koden for det er veldig lik, vi må bare huske på typene:

```
static double[] matriseGangerVektor (double[][] enMatrise,
          double[] enVektor){
        double[] resultatet = new double[3];
        double[] rekke = new double[3];
        for (int i=0;i<3;i++){
                rekke[0] = enMatrise[i][0];
                rekke[1] = enMatrise[i][1];
                rekke[2] = enMatrise[i][2];
                resultatet[i]=produkter.skalarProdukt(rekke, enVektor);
        }
        return resultatet;
}
```

Transponert, determinant og invers

Om vi bytter om rekkene og søylene i en matrise, kalles den nye matrisen for den transponerte. I en dobbelløkke svarer det til å bytte om de to indeksene, så vi kan få den transponerte ut på denne måten:

```
static double[][] transponert (double[][] enMatrise){
        double[][] resultatet = new double[3][3];
        for (int i=0;i<3;i++){
                for (int j=0;j<3;j++){
                        resultatet[i][j]= enMatrise[j][i];
                }
        }
```

```
        return resultatet;
}
```

En av de viktigste operasjonene for matriser er determinanten. Her, som for vektorproduktet i forrige seksjon, er formelen vi skal kode ikke det vi vil huske når vi skal regne det ut for hånd, men heller en fremgangsmåte. For en $2 \times 2-$matrise er determinanten gitt ved formelen

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc.$$

Formelen for determinanten til en $3 \times 3-$matrise er

$$\det(A) = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = aei + bfg + cdh - afh - bdi - ceg.$$

For å kode dette må vi passe på indeksene, ellers burde det være ganske rett frem, se Oppgave 9.2.2.

Dersom determinanten til en matrise A er forskjellig fra null, vil matrisen ha en invers A^{-1}. Det betyr at

$$AA^{-1} = A^{-1}A = I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Igjen er det flere måter å regne ut inversen på, men den metoden vi skal bruke er via kofaktormatrisen. Det er en matrise som alltid kan defineres, og for å få inversmatrisen må vi dele på determinanten (om den er ulik null). Vi må også transponere kofaktormatrisen for å få den inverse, men siden vi uansett skal kode dette ganske hardt, dropper vi egne symboler for kofaktormatrisen og går rett til formelen for inversen. Formelen for inversen bruker to steg: For hver plass (i, j) i matrisen, la $A_{ij} = (-1)^{i+j} \cdot \det$, der determinanten er $2 \times 2-$determinanten til matrisen vi får ved å fjerne rekke i og søyle j. Inversmatrisen er da

$$A^{-1} = \frac{1}{\det(A)} \begin{bmatrix} A_{11} & A_{21} & A_{31} \\ A_{12} & A_{22} & A_{32} \\ A_{13} & A_{23} & A_{33} \end{bmatrix}.$$

9.2.1 Oppgaver

Oppgave 9.2.1 *Sum og skalarmultiplikasjon:*

Skriv metoder for å addere to matriser, og for å gange en matrise med en skalar.

Oppgave 9.2.2 *Determinanter:*

Skriv koden for å regne ut determinanten til en $3 \times 3-$matrise ved å bruke formelen fra teksten. Skriv deretter kode for å regne ut determinanten til en $3 \times 3-$matrise i to steg, først ved å regne ut determinanten til $2 \times 2-$matriser, og så ved å bruke utvikling langs første rekke.

9.3 Løsning av lineære likningssystemer

NB: Dette avsnittet er ikke pensum i høst! For å skrive programmet trengs derfor en god del egenarbeid, som er sammenfattet i det siste avsnittet med tilhørende oppgaver

For å løse likningssystemer der alle likningene som inngår er lineære, kan vi bruke en reduksjonsprosess som kalles *Gauss-Jordan-eliminasjon* eller mer uformelt *rekke-reduksjon.*. Denne reduksjonsprosessen består i å bruke tre enkle regler gjentatte ganger:

a) Å multiplisere en likning med en konstant ulik null endrer ikke løsningsmengden.

b) Å bytte rekkefølgen på likningene endrer ikke løsningsmengden.

c) Å erstatte en likning med den samme likningen pluss et multiplum av en av de andre endrer ikke løsningsmengden.

Et likningssystem av typen

$$\begin{aligned} ax + by &= e \\ cx + dy &= f \end{aligned}$$

tilordnes en matrise

$$\begin{bmatrix} a & b & e \\ c & d & f \end{bmatrix}.$$

Reglene for likningssystemet fører til tilsvarende regler for matrisen:

a) Å multiplisere en rekke med en konstant ulik null endrer ikke løsningsmengden.

b) Å bytte rekkefølgen på rekkene endrer ikke løsningsmengden.

c) Å erstatte en rekke med den samme rekken pluss et multiplum av en av de andre endrer ikke løsningsmengden.

Disse grunnleggende rekkeoperasjonene kan brukes til å overføre en matrise til redusert form, som er kjennetegnet ved disse reglene:

i) I hver rekke er det første (fra venstre) tallet ulik null et ettall. Disse ettallene kalles *ledende enere*.

ii) En ledende ener i en gitt rekke står lenger til venstre enn ledende enere i rekker lenger nede.

iii) Over hver ledende ener, i samme søyle, står det bare nuller.

iv) Eventuelle rekker med bare nuller er samlet nederst.

Fra denne endelige oppstillingen kan vi lese av løsningsmengden på likningssystemet. Presist har vi:

i) Dersom det er en ledende ener i den siste søylen (svarende til konstantene, altså høyresiden i likningssystemet), har vi ingen løsning. Vi forutsetter i de neste punktene at dette ikke skjer.

ii) Hver søyle uten ledende ener gir en fri variabel, som vi kan oppfatte som en parameter som gir løsninger uansett valg av verdi for denne parameteren.

iii) Hver søyle med ledende ener gir en variabel hvis verdi er bestemt av de frie variablene og konstantleddene. Hvis den ledende eneren står i søyle j vil den tilsvarende variabelen x_j være uttrykt ved hjelp av de frie variablene med høyere indeks enn j.

Eksempler på reduserte matriser og tolkning av løsningen:

$$\begin{bmatrix} 1 & 0 & 2 & 1 \\ 0 & 1 & 0{,}5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ Ingen løsning!}$$

Her er det en ledende ener i den siste søylen.

$$\begin{bmatrix} 1 & 0 & 2 & 1 \\ 0 & 1 & 0{,}5 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \; x_3 \text{ parameter, } \begin{matrix} x_1 = 1 - 2x_3 \\ x_2 = -0{,}5x_3 \end{matrix}$$

En hel rekke med nuller svarer til en likning av typen $0 = 0$, som ikke gir så mye den ene veien eller den andre.

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0{,}5 \end{bmatrix} \quad \begin{matrix} x_1 = 1 \\ x_2 = 0 \\ x_3 = 0{,}5 \end{matrix} .$$

Gauss-Jordan-eliminasjon

For å gjennomføre reduksjonen starter vi oppe til venstre og arbeider oss nedover mot høyre. Hvis tallet oppe til venstre er ulik null, ganger vi hele rekken med 1 delt på tallet (den første rekkeoperasjonen). Ellers prøver vi å finne et annet tall i første søyle som er ulik null, bytter om på rekkene slik at dette tallet kommer oppe til venstre, og gjennomfører den samme prosessen. Når vi så har fått en ledende ener oppe til venstre, tar vi for hvert tall under denne ledende eneren en rekkeoperasjon hvor vi erstatter rekken med et multiplum av den første rekken slik at det første tallet i rekken blir null.

Dersom det ikke var noe tall ulikt null i den første søylen, beholder vi en søyle med bare nuller til venstre, og gjentar prosessen på (den mindre) matrisen hvor vi har glemt den første søylen. Dersom vi nå har en ledende ener oppe til venstre, og ellers bare nuller i den venstre søylen, gjentar vi prosessen på (den mindre) matrisen hvor vi har glemt den første søylen og den første rekken.

Advarsel: Vi skal flere steder teste om noe er lik null. Siden vi arbeider med `double` kan vi få avrundingsfeil, så vi må passe på at et tall som er ganske lite bør oppfattes som null.

Denne prosessen stopper når vi ikke har flere steder å lete etter ledende enere, altså når vi har kommet ned til høyre i likningssystemet. Deretter tar vi, for hver ledende ener, rekkeoperasjonen med å erstatte en rekke med et multiplum av en annen for å få nuller over de ledende enerne.

Implementasjon av Gauss-Jordan

Vi trenger metoder for de tre rekkeoperasjonene. Hver for seg er de ganske enkle å implementere, se Oppgave 9.3.1. Når de tre metodene er på plass, må vi følge gangen i Gauss-Jordan-eliminasjonen.

Først må vi, for hver søyle fra venstre til høyre, lokalisere et tall ulikt null (som skal endre til en ledende ener), som er lenger nede enn tidligere plasserte ledende enere. Så må rekken med tallet vi har valgt plasseres øverst blant de ledige plassene, og vi må gjøre det om til en ledende ener. Deretter må alle plassene under den ledende eneren gjøres om til nuller. Se Oppgave 9.3.2. Vi lager samtidig en tabell over plasseringene til de ledende enerne.

Vi skal så baklengs gjennom matrisen, fra høyre til venstre. For hver ledende ener (som vi har laget tabell over), fjern alt som står over den ledende eneren. Se Oppgave 9.3.3.

Det eneste som gjenstår er å skrive ut et pent svar, som beskriver løsningen av likningssystemet direkte. Se Oppgave 9.3.4.

Oppgaver

Oppgave 9.3.1 *Rekkeoperasjoner:*

Vi skal lage tre metoder.

a) Lag først en metode som erstatter en rekke med en konstant ganget med rekken. Input skal være `(double[][] enMatrise, int rekkeNummer, double skalar)`, og svaret skal være av typen `double[][]`. Sjekk at konstanten er ulik null.

b) Skriv deretter en metode som bytter om to rekker. Input skal være av typen `(double[][] enMatrise, int rekke1, int rekke2)`, svaret av typen `double[][]`.

c) Skriv til slutt en metode som erstatter en rekke med rekken pluss et multiplum av en annen rekke. Input skal være av typen `(double[][] enMatrise, int rekkeSkalEndres, int rekkeSkalLeggesTil, double multiplum)`, svaret igjen av typen `double[][]`.

Oppgave 9.3.2 *Ledende enere/trappeform:*

Vi skal implementere fremgangsmåten for å finne de ledende enerne, og bruke dem til å få nuller nedenfor i rekkene. Samtidig skal vi lage en tabell over søylenumrene med ledende enere. Hvis matrisen vår er en $n \times m-$matrise, kan vi maksimalt ha $min(m, n)$ antall ledende enere, så det gir en grense for en løkke. Anta at det i skritt $i-1$ i løkken ble plassert en ledende ener i søyle j. I skritt i i løkken trenger vi ikke ta hensyn til de $i-1$ første rekkene, siden de allerede har ledende enere. Velg en rekke, med rekkenummer minst i, som har et tall ulik null i søyle $j+1$. Om det ikke finnes, øk søylenummeret med 1. Når et slik tall er funnet, bytt om på rekkene så denne rekken blir rekke i (om den allerede er rekke i trenger du ikke bytte om). Legg til søylenummeret, k, i tabellen over ledende enere. For hver rekke med rekkenummer større enn i, erstatt rekken med rekken pluss et multiplum av rekken med den nye ledende eneren, slik at det blir null i søyle k i rekken. Gjenta inntil søylenummeret eller rekkenummeret har kommet utenfor dimensjonene til matrisen.

Matrisen sies å være på *trappeform* etter denne prosessen.

Oppgave 9.3.3 *Redusert trappeform.*

Bruk tabellen over ledende enere baklengs, og husk at indeksen i tabellen gir rekkenummeret, verdien søylenummeret, til de ledende enerne. For hver ledende ener, bruk rekkeoperasjonen å erstatte en rekke med rekken pluss et multiplum av en annen, til å få nuller over de ledende enerne. **NB:** Dersom tabellen ikke gjennomleses baklengs blir svaret galt.

Etter denne prosessen sies matrisen å våre på *redusert trappeform.*

Oppgave 9.3.4 *Skriv ut svaret:*

Fra den reduserte trappeformen, skriv ut svaret på likningssystemet. List opp de frie variablene, og for hver annen variabel, skriv ut denne variabelen uttrykt ved hjelp av konstanter og de frie variablene. For eksempel, om den reduserte matrisen er

$$\begin{bmatrix} 1 & 2 & 5 \\ 0 & 0 & 0 \end{bmatrix}$$

skal programmet skrive ut at x_2 er fri mens $x_1 = -2x_2 + 5$.

Oppgave 9.3.5 *Alt i ett:*

Juster programmet fra Oppgave 9.3.2 slik at Oppgave 9.3.3 blir unødvendig. Hva er fordelene og ulempene med dette?

Oppgave 9.3.6 *Avrundingsfeil:*

For å unngå avrundingsfeil, velg det største tallet som kandidat for ledende ener i Oppgave 9.3.2. Sett opp en test som antar tallene bør være null om maksimum er mindre enn en skranke (for eksempel 0,00001).

9.4 Ortogonale koordinatsystemer og Gram-Schmidt

To vektorer kalles *ortogonale* dersom de står vinkelrett på hverandre. Vi kan sjekke det ved å sjekke om skalarproduktet mellom dem er null. Dersom vi har tre vektorer $\vec{u}, \vec{v}, \vec{w}$, slik at hvert par av dem er ortogonale, kan vi observere noe spesielt ved å gange sammen matrisen med vektorene som rekker og matrisen med vektorene som søyler:

$$\begin{bmatrix} u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \\ w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} u_1 & v_1 & w_1 \\ u_2 & v_2 & w_2 \\ u_3 & v_3 & w_3 \end{bmatrix} = \begin{bmatrix} \vec{u} \bullet \vec{u} & \vec{u} \bullet \vec{v} & \vec{u} \bullet \vec{w} \\ \vec{v} \bullet \vec{u} & \vec{v} \bullet \vec{v} & \vec{v} \bullet \vec{w} \\ \vec{w} \bullet \vec{u} & \vec{w} \bullet \vec{v} & \vec{w} \bullet \vec{w} \end{bmatrix}$$

Siden alle skalarproduktene utenfor diagonalen er lik null, får vi en diagonalmatrise. Det som står langs diagonalen er i tillegg kvadratet av lengdene til vektorene, så om vi i tillegg krever at $|\vec{u}| = 1$, og tilsvarende for de to andre vektorene, får vi

$$\begin{bmatrix} u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \\ w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} u_1 & v_1 & w_1 \\ u_2 & v_2 & w_2 \\ u_3 & v_3 & w_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Slike matriser kalles *ortogonale*, og har altså søyler som er parvis vinkelrette og av lengde 1. De har videre egenskapen at den inverse matrisen er lik den transponerte (som er mye lettere å regne ut).

Det er ofte hensiktsmessig å velge basiser for underrom slik at basisvektorene er parvis ortogonale og av lengde 1. Her er måten å gjøre det på for 1,2 og 3 dimensjoner. Framgangsmåten kan itereres, og utvides til flere dimensjoner. Det er denne teknikken som kalles for Gram-Schmidt.

Endimensjonal Gram-Schmidt:
Gitt en vektor $\vec{v} \neq \vec{0}$. Siden det bare er én vektor vil det ikke være noen betingelse om ortogonalitet, men vi ønsker at den skal ha lengde 1. Gram-Schmidt i 1 dimensjon består altså bare i å dele på lengden:

$$\vec{v} \text{ erstattes av } \frac{\vec{v}}{|\vec{v}|}.$$

Alle vektorer på linjen (gjennom origo) utspent av $\vec{v}$ kan skrives som et multiplum av $\vec{v}$. Siden vi nå har at $|\vec{v}| = 1$ kan vi si dette litt sterkere: Enhver vektor på denne linjen kan skrives som enten pluss eller minus lengden sin ganget med $\vec{v}$.

Todimensjonal Gram-Schmidt:
Gitt to vektorer $\vec{v}$ og $\vec{u}$ som ikke er parallelle (og ingen av dem null). Først bruker vi endimensjonal Gram-Schmidt på den første, så $\vec{v}$ erstattes med $\vec{v}/|\vec{v}|$. Deretter erstatter vi $\vec{u}$ med den ortogonale komponenten med hensyn til $\vec{v}$, og til slutt deler vi på dennes lengde.

$$\begin{aligned} &\vec{v} \text{ erstattes av } \tfrac{\vec{v}}{|\vec{v}|} \\ &\vec{u} \text{ erstattes av } \vec{u} - \vec{u} \bullet \vec{v} \cdot \vec{v} \\ &\vec{u} \text{ erstattes av } \tfrac{\vec{u}}{|\vec{u}|} \end{aligned}$$

Den ortogonale komponenten får vi ved å trekke vekk projeksjonen av $\vec{u}$ på $\vec{v}$. Siden vi allerede har at $|\vec{v}| = 1$ får vi en litt enklere form enn vanlig på denne projeksjonen. Alle vektorer på planet (gjennom origo) utspent av de to vektorene $\vec{v}$ og $\vec{u}$ kan nå uttrykkes som $\vec{x} = a\vec{v} + b\vec{u}$ der Pythagoras' teorem gjelder: $|\vec{x}| = \sqrt{a^2 + b^2}$.

Tredimensjonal Gram-Schmidt:
Gitt tre vektorer $\vec{v}$, $\vec{u}$ og $\vec{w}$ som ikke ligger i samme plan. Vi bruker først todimensjonal Gram-Schmidt på de to første, og så erstatter vi $\vec{w}$ med den ortogonale komponenten med hensyn til $\vec{v}$, så med den ortogonale komponenten med hensyn til $\vec{u}$, og endelig deler vi på lengden. De to ortogonale komponentene kan slås sammen i én likning, så vi gjør det:

$$\begin{aligned}
&\vec{v} \text{ erstattes av } \frac{\vec{v}}{|\vec{v}|}\\
&\vec{u} \text{ erstattes av } \vec{u} - \vec{u} \bullet \vec{v} \cdot \vec{v}\\
&\vec{u} \text{ erstattes av } \frac{\vec{u}}{|\vec{u}|}\\
&\vec{w} \text{ erstattes av } \vec{w} - \vec{w} \bullet \vec{v} \cdot \vec{v} - \vec{w} \bullet \vec{u} \cdot \vec{u}\\
&\vec{w} \text{ erstattes av } \frac{\vec{w}}{|\vec{w}|}
\end{aligned}$$

Som i det todimensjonale tilfellet vil en basis produsert av Gram-Schmidt gi Pythagoras' teorem: Dersom $\vec{x} = a\vec{v} + b\vec{u} + c\vec{w}$ er lengden $|\vec{x}| = \sqrt{a^2 + b^2 + c^2}$.

Oppgaver

Oppgave 9.4.1 *1D Gram-Schmidt:*

Lag et program som utfører Gram-Schmidt på 1 vektor. Input skal være av typen `double[]`, og det skal resultatet også være. Sjekk om vektoren er null først!

Oppgave 9.4.2 *2D Gram-Schmidt:*

Lag et program som utfører Gram-Schmidt på 2 vektorer. Input skal være 2 verdier av typen `double[]`, og resultatet skal være en verdi av samme type. Sjekk om vektorene er parallelle først!

Oppgave 9.4.3 *3D Gram-Schmidt:*

Lag et program som utfører Gram-Schmidt på 3 vektorer. Input skal være 3 verdier av typen `double[]`, og resultatet skal være en verdi av samme type. Sjekk om vektorene ligger i samme plan først! Hint: Tre vektorer i trerommet ligger i samme plan hvis og bare hvis determinanten til matrisen med dem som søyler er null.

Oppgave 9.4.4 *Sjekk for ortogonalitet.*

Skriv et program som sjekker om en samling vektorer er ortogonale, altså om de står parvis vinkelrett på hverandre, og om de har lengde 1.

9.5 Spesielle matriser og transformasjoner

Vi har sett at matriser kan ganges med vektorer, slik at vi får nye vektorer ut, og om vi utsetter en figur for dette vil vi få en vridd, strukket, kanskje speilet variant av den opprinnelige figuren. Men vi vil jo ofte gjøre det motsatte, altså ut fra en beskrivelse av hva vi ønsker skal skje med en figur, trenger vi en matrise som gjør nettopp det! I denne seksjonen skal vi se på noen eksempler på dette.

Strekking/komprimering langs en akse

Dersom vi ønsker å doble alt som skjer i $x-$retningen, kan vi multiplisere med denne matrisen:

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Om vi multipliserer med en generell vektor vil $x-$komponenten ganges med 2, mens de to andre koordinatene bevares. Om vi ønsket å komprimere, for eksempel halvere $x-$retningen, kunne vi oppnådd det ved å bytte ut 2 med 0,5, og så la resten være som det er.

Mer generelt er

$$\begin{bmatrix} a & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

en strekking eller komprimering i $x-$retning med en faktor a (som inkluderer at $x-$retningen kan speiles om $a < 0$, og glemmes om $a = 0$),

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

tilsvarende i $y-$retning, og

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & c \end{bmatrix}$$

tilsvarende i $z-$retning. Endelig er

$$\begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix}$$

en samtidig strekking/komprimering langs hver av aksene.

Som et spesialtilfelle kan vi se på speilinger i koordinatplanene: Speiling i yz-planet svarer til at $x-$koordinaten skifter fortegn. Det betyr at vi skal ha matrisen

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Om -1 står på plassen til y dreier det seg om en speiling i xz-planet, og om den er på plassen til z er det speiling i $xy-$planet.

Rotasjon om en akse

I xy-planet er en rotasjon om origo med en vinkel α gitt ved matrisen

$$\begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix}.$$

Det kan verifiseres ved en elementær trekantbetraktning (merk at i geometri er elementær og enkel ikke synonymer). Om vi lar $z-$aksen peke rett opp fra arket, slik at vi får et høyresystem, kan vi la $z-$koordinaten være uforandret, og la α ha samme mening. Det gir matrisen for rotasjon om $z-$aksen:

$$\begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Om tommelen på høyre hånd peker i positiv $z-$retning, svarer dette til en rotasjon om $z-$aksen α radianer i den retningen du griper med høyre hånd.

Tilsvarende matriser for rotasjon om $x-$aksen og $y-$aksen likner ganske mye, vi må bare passe på fortegn på grunn av at vi ønsker den samme tolkningen ved hjelp av høyre hånd. Rotasjon om $x-$aksen:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}.$$

Rotasjon om $y-$aksen:

$$\begin{bmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix}.$$

Parallellprojeksjon

Den enkleste måten å se en parallellprojeksjon på, er å starte med parallellprojeksjonen ned i xy-planet. Det svarer til å glemme z-koordinaten, så matrisen for det er

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Om vi har et annet plan, og ønsker parallellprojeksjonen vinkelrett på det planet, må vi først være enige om hvordan det skal representeres. En vanlig måte å angi et plan på, er å angi to (ikke-parallelle) vektorer som ligger i planet. Vi antar videre at disse to vektorene er ortogonale og av lengde 1; om de ikke er det utsetter vi dem for Gram-Schmidt, som vi allerede har på plass. La de to vektorene være $\vec{v}$ og $\vec{u}$. Vi kan utvide til en basis for hele rommet ved å bruke en normalvektor $\vec{n} = \vec{v} \times \vec{u}$ som den tredje basisvektoren. Vi kan tenke på projeksjonen på planet ved først å skifte basis til $\vec{v}$, $\vec{u}$, $\vec{n}$, så glemme $\vec{n}$ slik vi glemte z nettopp, og endelig skifte tilbake til de opprinnelige xyz-koordinatene (standardkoordinatene). Siden overgangsmatrisen fra de nye koordinatene til standardkoordinatene er matrisen med de tre vektorene som søyler, og siden vektorene er ortogonale og av lengde 1, er overgangsmatrisen fra standardkoordinatene til de nye koordinatene bare den transponerte (vi slipper å regne ut inversmatrisen). Dermed er projeksjonsmatrisen gitt ved matriseproduktet

$$\begin{bmatrix} v_1 & u_1 & n_1 \\ v_2 & u_2 & n_2 \\ v_3 & u_3 & n_3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 & v_2 & v_3 \\ u_1 & u_2 & u_3 \\ n_1 & n_2 & n_3 \end{bmatrix}.$$

Om dette multipliseres ut, vil vi se at koordinatene til $\vec{n}$ forsvinner fra uttrykket, som passer med at den vektoren skal glemmes. En annen interessant variant får vi ved å la være å gå tilbake til standardkoordinatene, men heller beholde koordinatene i $\vec{v}$ og $\vec{u}$ (med $\vec{n}$ glemt). Det kan være fornuftig om vi vil tenke på planet vi projiserer i som skjermen, og dermed bruke disse koordinatene i grafbiblioteket. Det svarer til å multiplisere sammen de to siste matrisene, og så anvende produktet på en vektor. Vi får

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_1 & v_2 & v_3 \\ u_1 & u_2 & u_3 \\ n_1 & n_2 & n_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} v_1 & v_2 & v_3 \\ u_1 & u_2 & u_3 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} \vec{v} \bullet \vec{x} \\ \vec{v} \bullet \vec{x} \\ 0 \end{bmatrix}.$$

Koordinatene i bildet får vi altså ved å ta skalarproduktet av en variabel vektor $\vec{x}$ og de to basisvektorene $\vec{v}$ og $\vec{u}$! Merk spesielt at dette bare virker fordi vi valgte koordinatene til å være ortogonale og av lengde 1.

9.6 Utvidet eksempel: Stanford bunny

Vi skal nå sette opp et noenlunde reelt tredimensjonalt eksempel, en skannet kanin. Den skannede informasjonen ligger i en fil med navn `"kanin.txt"`, som ligger på It's learning. Denne filen er en lett bearbeidet versjon av en fil jeg fant på `"http://graphics.stanford.edu/"`. Kaninen omtales vanligvis som The Stanford bunny!

Først må vi lese inn koordinatene til punktene fra fil. Dette er et av de siste temaene dere skal innom i programmeringsfaget i høst, så jeg vil ikke kommentere koden som leser inn informasjonen særlig utførlig. Filen er formatert slik at hver linje består av x, y og z koordinatene med mellomrom mellom. Unntaket er den første linjen, som forteller hvor mange linjer filen består av. Vi leser inn verdiene linje for linje, og lagrer dem i en tabell av tabeller `double[][3]`, der tretallet er de tre koordinatene, og den første indeksen er antallet linjer i filen.

Her er koden (med litt uklart innrykk):

```
import java.io.*;
public class kanin {
        final static char TEGN_MELLOM_KOORD = ' ';
        public static void main(String[] args) {
// try{} catch{} er unntakshandtering, jf. TOD062
try{
        FileReader tekstFilLeser = new FileReader("kanin.txt");
        BufferedReader tekstLeser = new BufferedReader(
                tekstFilLeser);
        //Forste linje i filen forteller antall linjer,
        // ett punkt per linje
        int antallPunkter = Integer.parseInt(tekstLeser.
                readLine());
        //x,y og z-koordinater:
        double[][] kaninPunkter = new double[antallPunkter][3];
        for (int i =0;i<antallPunkter;i++){
                String linje = tekstLeser.readLine();
                //Finner mellomrommene
                int koordinatIndeks1 = linje.indexOf(
                  TEGN_MELLOM_KOORD);
                int koordinatIndeks2 = linje.indexOf(
                  TEGN_MELLOM_KOORD, koordinatIndeks1+1);
                // Parser tall mellom mellomrom
```

```
                    kaninPunkter[i][0] = Double.parseDouble(linje.
                      substring(0,koordinatIndeks1));
                    kaninPunkter[i][1] = Double.parseDouble(linje.
                      substring(koordinatIndeks1 +1,koordinatIndeks2));
                    kaninPunkter[i][2] = Double.parseDouble(linje.
                      substring(koordinatIndeks2+1));
            }
tekstLeser.close();
// Tegn kaninen innenfor try{}
}catch (Exception e){//Catch exception if any
  System.err.println("Error: " + e.getMessage());
  }
 }
}
```

Når vi så har denne tabellen med punkter, kan vi utsette hver vektor for en matriseoperasjon. Vi prøver med de forskjellige matrisene fra Seksjon 9.5. I tillegg vil vi ønske å tegne filen i grafbiblioteket, slik at vi til hver vektor må kunne plukke ut to tall som kan brukes. Fra avsnittet om parallellprojeksjon vet vi at vi kan få det til ved å velge to vektorer som er ortogonale og av lengde 1, og så ta skalarprodukt med dem. Fordi vi skal fokusere på det tredimensjonale, vil vi som standard bruke $x-$ og $y-$koordinatene, slik at vi alltid ser det samme utsnittet, men det tredimensjonale objektet er endret. Vi kan tegne den opprinnelige grafen ved å (husk å inkludere grafbiblioteket)

```
Graf kanin = new Graf();
 for (int i=0;i<antallPunkter;i++){
        double x,y;
        x= kaninPunkter[i][0];
        y= kaninPunkter[i][1];
        Punkt p = new Punkt(x,y);
        kanin.leggTilPunkt(p);
 }
 kanin.visVindu();
```

Det kan også være aktuelt å sette punktradius til 1, og å angi grensene for x og y eksplisitt.

Om vi så skal endre på kaninen ved hjelp av en matrise, husk at vi har implementert matrisen ganger vektor tidligere. La M være matrisen vi skal bruke, og se på denne koden:

```
for (int i=0;i<antallPunkter;i++){
```

```
        kaninPunkter[i] = matriseOperasjoner.matriseGangerVektor(
                          M, kaninPunkter[i]);
}
```

Deretter kan kaninen tegnes igjen.

Oppgavene ber deg prøve dette med forskjellige matriser.

Oppgaver

Oppgave 9.6.1 *Rotasjon:*

Bruk rotasjonsmatriser som M og undersøk hvordan bildet av kaninen endrer seg. Prøv med alle de tre aksene.

Oppgave 9.6.2 *Speiling:*

Bruk speilingsmatriser som M og undersøk hvordan bildet av kaninen endrer seg. Prøv med alle de tre aksene.

Oppgave 9.6.3 *Strekking:*

Bruk strekking eller komprimering i matrisen M og undersøk hvordan bildet av kaninen endrer seg. Prøv med strekkinger langs alle de tre aksene. **NB:** For å se dette skikkelig bør autskaleringen skrus av i grafbiblioteket.

Oppgave 9.6.4 *Translasjon:*

Denne operasjonen er ikke uttrykt ved hjelp av en matrisemultiplikasjon (men se neste seksjon). Heller enn å gange med matrisen M i løkken for oppdatering av kaninpunktene, prøv å legge til en fast vektor, og undersøk hvordan bildet av kaninen endrer seg. Prøv translasjon langs alle aksene. **NB:** For å se dette skikkelig bør autskaleringen skrus av i grafbiblioteket.

Oppgave 9.6.5 *Sammensetning og matrisemultiplikasjon.*

Hvis M er en transformasjon, og M' en annen, vil matrisemultiplikasjonen $M'M$ svare til transformasjonen som bruker M først og så M'. Undersøk hva som skjer med kaninen om den roteres først, og så strekkes, eller om den strekkes først, og så roteres. Dette viser at matrisemultiplikasjon ikke er kommutativt!

Oppgave 9.6.6 *Smultring:*

Finn tilbake til punktene for smultringen fra Seksjon 7.3. Prøv oppgavene over også på torusen. Unngå å bruke det spesielle planet som ble valgt der, men prøv å få gode projeksjoner i $xy-$planet ved å utføre passende transformasjoner.

9.7 Homogene koordinater

Homogene koordinater er en generell konstruksjon i matematikk, som i dag er tilpasset datagrafikk og lagt inn som standard i hardware (GPU). Vi skal ikke gå så langt i denne retningen, bare ta med nok til at vi kan finne en projeksjon fra et punkt på et plan, slik at vi får fram perspektiv i projeksjoner.

Homogene koordinater er egentlig koordinater i et såkalte *projektive rom*, men vi skal tenke litt enklere på dem her. Vi bruker fire tall for hvert punkt i rommet, heller enn de vanlige tre. Men vi lar punktene være ekvivalensklasser av slike firetupler av tall, der to firetupler er ekvivalente dersom det ene er et multiplum $\neq 0$ av det andre. For eksempel er

$$(1, 2, 4, 7) \text{ og } (0{,}7,\ 1{,}4,\ 2{,}8,\ 4{,}9)$$

ekvivalente, fordi det andre tuppelet er 0,7 ganger det første. De vanlige punktene i rommet, med vanlig avstand seg i mellom, kan vi tenke enklest på ved å velge det siste tallet lik 1. Firetuplene over representerer altså det "vanligepunktet

$$\left(\frac{1}{7}, \frac{2}{7}, \frac{4}{7}\right).$$

Alle transformasjonene vi har sett på kan utvides til firedimensjonale matriser ved å legge til nuller nederst og til høyre, bortsett fra et ettall helt nede i hjørnet. I tillegg kan translasjoner oppfattes som multiplisering av $4 \times 4-$matriser fordi

$$\begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x+a \\ y+b \\ z+c \\ 1 \end{bmatrix}.$$

Vi kan også få til perspektiv i projeksjoner, og det gjør at vi nærmer oss poenget med det vi gjør i dag.

I framstillingen av kaninen i forrige avsnitt projiserte vi til $xy-$planet. Men øyet, eller kamera, var egentlig plassert uendelig langt borte i $z-$retningen, slik at projeksjonen rett og slett glemte $z-$koordinaten. Reelt sett vil øyet befinne seg i en endelig avstand fra projeksjonsplanet, og $z-$koordinaten vil ikke glemmes helt: Den overlever via *perspektiv*.

For å være helt presise kan vi anta at øyet ser nedover $z-$aksen, og er plassert i origo. La oss videre anta at projeksjonsplanet er parallelt med xy-planet og plassert

i $z = -d$. Da er projeksjonsmatrisen, som $4 \times 4-$matrise,

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\frac{1}{d} & 0 \end{bmatrix}$$

Om vi bruker denne på (x, y, z) får vi

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\frac{1}{d} & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ -\frac{z}{d} \end{bmatrix}.$$

For å forstå hvorfor dette virker, bør man se på formlike trekanter. Se Oppgave 9.7.4 For å få dette på standardformen, med 1 som siste koordinat, må vi gange gjennom med $-\frac{d}{z}$. Projeksjonspunktet er altså

$$\begin{bmatrix} -dx/z \\ -dy/z \\ -d \\ 1 \end{bmatrix}$$

og vi ønsker bare å ta ut de to første koordinatene. Vi ser fra svaret at vi kan kode dette enklere direkte, ved å endre på koden

```
for (int i=0;i<antallPunkter;i++){
        double x,y;
        x= kaninPunkter[i][0];
        y= kaninPunkter[i][1];
        Punkt p = new Punkt(x,y);
        kanin.leggTilPunkt(p);
 }
```

til å ta inn over seg en ny skalar `d`, av typen `double`, sett den lik et passende tall. Perspektivet får vi da inn ved å sette

```
for (int i=0;i<antallPunkter;i++){
        double x,y,z;
        x= kaninPunkter[i][0];
        y= kaninPunkter[i][1];
        z= kaninPunkter[i][2]-1;// (1)
        Punkt p = new Punkt(-d*x/z,-d*y/z);
        kanin.leggTilPunkt(p);
 }
```

Det er et problem om noen av punktene har $z-$koordinat akkurat null, eller svært nær null. Siden det er tilfellet med punktene i kaninskanningen, kan vi flytte kaninen vekk fra $z = 0$ først. Det er det som skjer i punkt (1).

Oppgaver

Oppgave 9.7.1 *Tegn kanin i perspektiv:*

Velg en passende d og utfør projeksjonen med perspektiv. Siden mange av $z-$koordinatene til kaninen er omtrent lik null, kan det være hensiktsmessig å utføre en translasjon først, som i teksten.

Oppgave 9.7.2 *Sammenstilling med andre transformasjoner:*

Kombiner matriseoperasjonene fra forrige seksjon med perspektivprojeksjonen, og tegn speilede og roterte kaniner (for eksempel).

Oppgave 9.7.3 *Perspektiv i torusen:*

Tegn torusen ved perspektiv, jamfør Oppgave 9.6.6. Kan du få det til slik at det er klart hva som er på framsiden og hva som er på baksiden av tegningen?

Oppgave 9.7.4 *Formlike trekanter:*

Hopp over y-koordinaten, og tegn opp trekanter som skal vise projeksjonen av et punkt på linjen $z = -d$ med utgangspunkt i origo. Bruk formlikhet til å verifisere formlene for $x-$koordinaten fra teksten.

Kapittel 10

Innleveringer

10.1 Innlevering 1

Fristen for å levere inn oppgaven er fredag i neste uke (9. september). Etter forelesningen tirsdag (30. august) skal alle være delt inn i grupper på tre, dere skal levere en besvarelse for hver gruppe.

Besvarelsen skal inneholde både programmene og resultatet av kjøring, samt svar på spørsmålene som ikke direkte er programmeringsoppgaver.

Lever besvarelsen i én fil, der programmene som inngår skal være komplette (slik at jeg kan klippe og lime). Pass på at programmene skriver ut både inndataene og svaret på en forståelig måte, og at koden er kommentert slik at jeg kan forstå hva dere gjør og hva dere tenker.

Fristen for å levere inn oppgaven er fredag i neste uke (9. september). Jeg skal se på oppgavene i løpet av helgen, og så vil dere bli spurt om å presentere noen punkter hver fra oppgaven på forelesningen på tirsdagen uken etter (13. september). Etter presentasjonen vil innleveringen bli godkjent.

NB: Ikke nøl med å spørre meg om hjelp! Jeg vet at noen av tingene som inngår er nye og vanskelige for mange. Hjelpen kan godt inkludere å gi en linje eller to med kode om dere står fast!

Oppgave 1

Skriv et program som plukker ut første siffer i en `double`, og lagrer det som en `char`. Her kan vi få bruk for metoder som `Double.toString`, som konverterer en `double`

til `String`, og `enStreng.charAt(0)`, som plukker ut det første tegnet i strengen `enStreng`. For eksempel vil

```
System.out.println((Double.toString(234.94)).charAt(0));
```

gi svaret 2.
Finn en liste på nettet over

Gruppe 1: 20 elvers lengde (km)

Gruppe 2: 20 elvers lengde ($miles$)

Gruppe 3: 20 europeiske lands areal (km^2)

Gruppe 4: 20 europeiske lands areal ($miles^2$)

Gruppe 5 : 20 asiatiske lands areal (km^2)

Gruppe 6 : 20 asiatiske lands areal ($miles^2$)

Gruppe 7: 20 afrikanske lands areal (km^2).

Gruppe 8: 20 afrikanske lands areal ($miles^2$)

Gruppe 9: 20 amerikanske lands areal (km^2)

Gruppe 10: 20 amerikanskje lands areal ($miles^2$)

Gruppe 11: 20 fjells høyde (m)

Gruppe 12: 20 fjells høyde ($feet$)

Gruppe 13: 20 lands innbyggertall

Gruppe 14: 20 hovedsteders innbyggertall

Lag en liste med disse tallene i Java, og tell opp hvor mange ganger vært av sifrene $1, 2, \cdots, 9$ forekommer som første siffer i tallet.

Oppgave 2

Lag en tabell over tall av typen `long`, der du velger ett tall med ett siffer, ett tall med to siffer, ett tall med tre siffer, og så videre. Stopp ved 19 siffer. Skriv ut en gangetabell for disse tallene. Marker etterpå hvilke svar som ble rette!

Konverter så alle tallene til `double`, og lag en ny gangetabell. Ble alt rett?

Oppgave 3

Skriv et program som regner ut Fibonacci-tallene $\{F_0, F_1, F_2, \cdots\}$. Tallene er definert ved at $F_0 = 0, F_1 = 1$ og for alle tall videre ($n \geq 2$) er $F_n = F_{n-1} + F_{n-2}$.

En formel for disse tallene er

$$F_n = \frac{\phi_0^n - \phi_1^n}{\sqrt{5}},$$

der

$$\phi_0 = \frac{1+\sqrt{5}}{2} \text{ og } \phi_1 = \frac{1-\sqrt{5}}{2}.$$

Lag to tabeller, en som bruker definisjonen direkte, og en som bruker formelen, og kontroller om de gir samme svar opp til det femtiende fibonaccitallet F_{50}.

Oppgave 4

Løs oppgave 2.2.6 om de spesielle flyttallene (lag én av tabellene for hver av typene `float` og `double`).

10.2 Innlevering 2

Den andre innleveringen har frist tirsdag 4. oktober, og vi skal gjennomgå den tirsdag 11. oktober. Da skal altså hver gruppe presentere en av oppgavene på forelesning. Som sist skal dere levere i grupper på tre.

Rutinene for innlevering er skjerpet noe: Om nødvendig kan dere be meg om utsatt frist på forhånd, men dersom fristen e passert og dere ikke har levert, blir det ikke godkjent. Det vil være mulig å skrive en søknad til instituttleder om å få oppgaven vurdert allikevel, men det vil dere nok helst unngå!

Oppgave 1

En gammel metode for å finne primtall kalles *Eratosthenes' såld*. Metoden tar et (positivt) heltall n som input, og returnerer en liste over alle primtallene opp til og med n. Lag først en liste over alle tallene 2, 3, 4, $\cdots$, n. Marker 2 som primtall, og kryss ut alle tallene 4, 6, 8, 10, $\cdots$ som er delelig med 2. Marker det første tallet etter 2 som ikke er krysset ut, altså 3, som primtall, og kryss ut alle tallene 6, 9, 12, $\cdots$ som er delelig med 3. Fortsett inntil alle tallene som ikke er krysset ut er primtall.

Oppgaven deres er

a) Finn ut hvor lenge man må holde på før alle tallene som står igjen på listen (altså som ikke er krysset ut) er primtall. Svaret skal begrunnes matematisk.

b) Bruk metoden for hånd, for å lage en liste over primtall under $n = 50$. Hvor mange primtall under 50 er det?

c) Implementer Eratosthenes' såld. Resultatet av programmet skal være en liste med primtall, men den vil være for lang til at det er hensiktsmessig å skrive den ut til skjermen. Programmet skal skrive ut det største primtallet som er funnet, grensen for hvor dere har lett etter primtall (altså n), og antall primtall mindre enn n.

d) Hva er det største tallet n datamaskinen din takler som input om programmet ikke skal kjøre lenger enn ett minutt?

Oppgave 2

I kapittel 6 er det definert en klasse for rasjonale tall (forelesning 23/9). Skriv fire nye boolske metoder for denne klassen, som skal gi svar på om et slikt tall (altså et objekt i denne klassen) er større enn, større enn eller lik, mindre enn eller mindre enn eller lik et annet tall.

Oppgave 3

Vi skal nå addere heltall ved å sette av det minimale antall bits som trengs. Siden en boolsk variabel bare kan ha to verdier, kan vi tenke på `boolean[]` som å passe til dette (Java bruker faktisk mer plass en én bit til en `boolean`, men vi kan tenke på det som én bit i denne oppgaven). Vi tenker på `true` som 1 og `false` som 0. For eksempel vil vi med fem bits representere

$$9 = 01001 = \{false, true, false, false, true\}$$

og

$$-7 = 11001 = \{true, true, false, false, true\}.$$

Skriv et program som setter av nok plass til å lagre to heltall i området $-n$ til n, og som adderer to slike tall. Input, som dere kan hardkode om dere vil, er n og de to binære tallene (velg dem positive, altså med første siffer lik null) som skal adderes. Output skal være svaret, altså summen av de to tallene, samt en advarsel dersom svaret er feil. Bruk samme framgangsmåte (toerkomplement) som vi har sett Java bruker for `short`, `int` og `long`, slik at svaret kan bli galt om tallene er nær grensen n.

Som en ekstra utfordring kan de som vil også multiplisere de to tallene og/eller ta input i titallsystemet.

Register