Stockholms matematiska cirkel

Datorernas matematik

Daniel Ahlsén
Joar Bagge

Institutionen för matematik, KTH och
Matematiska institutionen, Stockholms universitet
2019–2020

Stockholms matematiska cirkel genom tiderna
(tidigare KTH:s matematiska cirkel)

2019-2020	Datorernas matematik
2018-2019	Grafteori med inriktning på färgläggning
2017-2018	Geometriska konstruktioner
2016-2017	Vad är ett tal?
2015-2016	Fraktaler
2014-2015	Polytoper
2013-2014	Grupper, mönster och symmetrier
2012-2013	Den matematiska analysens grunder
2011-2012	Diofantiska ekvationer
2010-2011	Polynom
2009-2010	Hyperbolisk geometri
2008-2009	Talteori
2007-2008	Sannolikhetsteori
2006-2007	Gruppteori
2005-2006	Vad är ett tal?
2004-2005	Integraler
2003-2004	Linjär algebra och bioinformatik
2002-2003	Algebra och kryptografi
2001-2002	Analysens grunder
2000-2001	Talföljder, rekursioner och iterationer
1999-2000	Linjära avbildningar

Innehåll

Lista över grekiska alfabetet v

Några ord på vägen vi

1 Vad är matematik, egentligen? 1

1.1 Mängder . . . 2
1.2 Funktioner . . . 5
1.3 Logik . . . 7
1.4 Matematiska bevis . . . 11

2 Hur kan en dator räkna? 19

2.1 Binära tal och osignerade heltal . . . 19
2.2 Överflöde och kongruensräkning . . . 22
2.3 En additionsmaskin för heltal . . . 25
2.4 Signerade heltal . . . 28

3 Tal med decimaler 33

3.1 Binärbråk och flyttal . . . 33
3.2 Ekvationslösning med intervallhalvering . . . 40
3.3 Bevis av satsen om mellanliggande värden . . . 45

4 Tärningen är kastad 48

4.1 Pseudoslumptal . . . 48
4.2 Tillämpningar av slumptal . . . 54

5 Formella språk 60

5.1 Tecken, alfabet och ord . . . 60
5.2 Språk över alfabet . . . 62
5.3 Reguljära språk . . . 65

6 Tillståndsmaskiner 69

6.1 Deterministiska tillståndsmaskiner 70
6.2 Icke-deterministiska tillståndsmaskiner 75
6.3 Delmängdskonstruktionen 79

7 Tillståndsmaskinernas språk **90**
7.1 Reguljära språk avgörs av tillståndsmaskiner 90
7.2 Tillståndsmaskinernas språk är reguljära 92
7.3 Särskiljning av ord 98

Lösningar till udda övningsuppgifter **107**

Förslag till vidare läsning **130**

Sakregister **131**

Lista över grekiska alfabetet

A	α	alfa
B	β	beta
Γ	γ	gamma
Δ	δ	delta
E	ε	epsilon
Z	ζ	zeta
H	η	eta
Θ	θ	theta
I	ι	iota
K	κ	kappa
Λ	λ	lambda
M	μ	my
N	ν	ny
Ξ	ξ	xi
O	o	omikron
Π	π	pi
P	ρ	rho
Σ	σ	sigma
T	τ	tau
Υ	υ	ypsilon
Φ	ϕ	fi
X	χ	chi
Ψ	ψ	psi
Ω	ω	omega

Några ord på vägen

Detta kompendium är skrivet för att användas som kurslitteratur till Stockholms matematiska cirkel under läsåret 2019–2020 och består av sju kapitel.

Kompendiet är inte tänkt att läsas enbart på egen hand, utan ska ses som ett skriftligt komplement till undervisningen på de sju föreläsningarna. Alla elever rekommenderas att läsa igenom varje kapitel själv innan föreläsningen. Det är inte nödvändigt att förstå alla detaljer vid den första genomläsningen.

Som de flesta matematiska skrifter på högre nivå är kompendiet kompakt skrivet. Detta innebär att man i allmänhet inte kan läsa det som en vanlig bok. Istället bör man pröva nya satser och definitioner genom att på egen hand exemplifiera. Därmed uppnår man oftast en mycket bättre förståelse av vad dessa satser och deras bevis går ut på.

Till varje kapitel finns ett antal övningsuppgifter. De udda övningarna har lösningar längst bak i kompendiet. Syftet med dessa är att eleverna ska kunna lösa dem och på egen hand kontrollera att de förstått materialet. Övningar med jämna nummer saknar facit och kan användas som examination. Det rekommenderas dock att man försöker lösa dessa uppgifter även om man inte examineras på dem. Om man kör fast kan man alltid fråga en kompis, en lärare på sin skola eller någon av författarna. Under årets gång kommer det att finnas övningstillfällen där eleverna kan jobba med uppgifterna, själva eller i grupp, och få hjälp av oss.

De övningsuppgifter som är något svårare markeras med en stjärna ($\star$). Uppgifter som är extra utmanande markeras med två stjärnor ($\star\star$). Det finns även programmeringsuppgifter för elever som kan programmera sedan tidigare, vilka markeras med **P** framför uppgiftsnumret. Programmering är inget förkunskapskrav för kursen, och kommer inte heller läras ut under kursen, så programmeringsuppgifterna bör ses som helt och hållet valfria.

Övningarna kan ha många olika möjliga lösningar och det som står i facit bör endast ses som ett förslag.

Några ord om Cirkeln

Stockholms matematiska cirkel är en kurs för matematikintresserade gymnasieelever, som arrangeras av Kungliga Tekniska högskolan och Stockholms universitet. Cirkeln startade 1999 och firar alltså 20 år detta läsår. Vid starten hade den namnet KTH:s matematiska cirkel och hölls i KTH:s ensamma regi. Ambitionen med Cirkeln är att sprida kunskap om matematiken och dess användningsområden utöver vad eleverna får genom gymnasiekurser, och att etablera ett närmare samarbete mellan gymnasieskolan och högskolan. Cirkeln skall särskilt stimulera elevernas matematikintresse och inspirera dem till fortsatta naturvetenskapliga och matematiska studier.

Till varje kurs skrivs ett kompendium som distribueras gratis till eleverna. Detta material, föreläsningsschema och övrig information om Stockholms matematiska cirkel finns tillgängligt på

www.math-stockholm.se/cirkel

Cirkeln godkänns ofta som en gymnasiekurs eller som matematisk breddning på gymnasieskolorna. Det är upp till varje skola att godkänna Cirkeln som en kurs och det är lärarna från varje skola som sätter betyg på kursen. Lärarna är självklart också välkomna till Cirkeln och många har kommit överens med sin egen skola om att få Cirkeln godkänd som fortbildning eller som undervisning.

Vi vill gärna understryka att föreläsningarna är öppna för alla gymnasieelever, lärare eller andra matematikintresserade.

Vi har avsiktligt valt materialet för att ge eleverna en inblick i matematisk teori och tankesätt och presenterar därför både några huvudsatser inom varje område och bevisen för dessa resultat. Vi har också som målsättning att bevisa alla satser som används om de inte kan förutsättas bekanta av elever från gymnasiet. Detta, och att flera ämnen är på universitetsnivå, gör att lärarna och eleverna kan uppleva programmet som tungt, och alltför långt över gymnasienivån. Det bör därför inte ses som ett krav att lärarna och eleverna skall behärska ämnet fullt ut och att lära in det på samma sätt som gymnasiekurserna. Det viktigaste är att eleverna kommer i kontakt med teoretisk matematik och får en inblick i *matematikens väsen*. Vår förhoppning är att lärarna med denna utgångspunkt skall ha lättare att upplysa intresserade elever om Stockholms matematiska cirkel och övertyga skolledarna om vikten av att låta både elever och lärare delta i programmet.

Några ord om betygssättning

Då Cirkeln inte är utformad på samma sätt som andra gymnasiekurser i matematik kan betygsättningen bli problematiskt, om samma standard som för ordinarie gymnasiekurser används. Utgångspunkten bör istället vara att eleverna skall få insikt i matematiken genom att gå på föreläsningarna och att eleven gör sitt bästa för att förstå materialet och lösa uppgifterna. Självklart betyder det mycket vad eleverna har lärt av materialet i kursen, men lärarna kan bara förvänta sig att ett fåtal elever behärskar ämnet fullt ut.

Författarna, sommaren 2019

1 Vad är matematik, egentligen?

Temat för årets upplaga av cirkeln är *datorernas matematik*. Vi kommer att studera olika aspekter av datorer, med fokus på metoder och teori för att utföra matematiska beräkningar med hjälp av datorer. För att studera detta kommer vi använda matematiken som verktyg, och i detta första kapitel förklarar vi vad vi egentligen menar med matematik.

Den moderna matematiken, som vetenskap, sägs ofta bestå av definitioner, satser och bevis. En *definition* slår fast vad ett visst ord ska betyda. Till exempel kan vi definiera *jämna tal* och *udda tal* på följande sätt.

Definition 1.0.1. Ett heltal n kallas för ett *jämnt tal* om $n = 2k$ för något heltal k. △

Definition 1.0.2. Ett heltal n kallas för ett *udda tal* om $n = 2k + 1$ för något heltal k. △

Läsaren är förstås bekant med udda och jämna tal sedan tidigare, och kan kontrollera att ovanstående definitioner stämmer överens med läsarens egen uppfattning om vad udda och jämna tal är. Observera att vi inte har definierat begreppet *heltal*, men vi antar att detta redan är ett välkänt begrepp.

En *sats* är ett påstående som har bevisats vara sant. En sats hänger alltså ihop med ett *matematiskt bevis*, vilket är en följd av logiska resonemang som visar varför påståendet är sant. Ett exempel följer nedan.

Sats 1.0.3. *Om n är ett jämnt tal så är $n + 1$ ett udda tal. Om n istället är ett udda tal så är $n + 1$ ett jämnt tal.*

De flesta håller säkert med om att satsens påstående är sant. Den som är tveksam kanske provar att addera 1 till några heltal, och ser att påståendet verkar stämma. I *empiriska vetenskaper*, såsom fysik och andra naturvetenskaper, är det precis så man gör för att testa teorier – man utför experiment eller observationer för att se om teorin verkar stämma eller inte. Om teorin stämmer i tillräckligt många fall anses den vara bra nog för att använda.

I matematiken räcker det inte. Vi måste visa att påståendet *alltid* är sant, i *alla* tänkbara fall. För att göra det måste vi använda oss av definitionerna av jämna och udda tal, det vill säga Definition 1.0.1 och 1.0.2.

Bevis av Sats 1.0.3. Antag först att n är ett jämnt tal. Enligt definitionen finns då ett heltal k sådant att $n = 2k$. Då måste $n + 1$ vara ett udda tal, eftersom $n + 1 = 2k + 1$, vilket enligt definitionen betyder att $n + 1$ är udda.

Antag nu istället att n är ett udda tal. Då finns det ett heltal k sådant att $n = 2k + 1$. Men då är $n + 1 = 2k + 2 = 2(k + 1)$, vilket är ett jämnt tal eftersom $k + 1$ är ett heltal. □

Vi kommer återkomma till påståenden och sanning i avsnitt 1.3, och se fler exempel på bevis i avsnitt 1.4. Innan dess ska vi introducera två av de mest grundläggande begreppen inom matematiken, nämligen *mängder* (avsnitt 1.1) och *funktioner* (avsnitt 1.2).

1.1 Mängder

En *mängd* är en samling av objekt. Objekten som ingår i mängden brukar kallas *element*. Elementen kan vara vad som helst, till exempel kan de vara tal, bokstäver eller ord. För att beskriva en mängd kan vi helt enkelt lista alla element inom klammerparenteser $\{\}$. Till exempel är

$$\{1, 2, \mathrm{a}, \mathrm{b}, \tfrac{5}{7}, \pi\}$$

mängden som består av elementen 1, 2, a, b, $\frac{5}{7}$ och π. Två mängder är lika med varandra om de innehåller samma element. Ordningen på elementen spelar ingen roll, och inte heller hur många gånger ett element listas. Till exempel är

$$\{0, 1\} = \{1, 0\} = \{0, 0, 1, 1, 0\} = \{1, 1, 1, 1, 1, 1, 0\}$$

mängden som innehåller de två elementen 0 och 1. *Antalet element* i en mängd M betecknas med $|M|$. Alltså är $|\{0, 1\}| = 2$, men även $|\{0, 0, 1, 1, 0\}| = 2$ eftersom vi bortser från om samma element listas flera gånger.

För att säga att en mängd M innehåller ett visst element x kan vi skriva $x \in M$, vilket utläses "x tillhör M" eller "x är i M".

Exempel 1.1.1. En mängd kan innehålla andra mängder som element. Ett exempel är mängden

$$M = \{5, \{0, 4, 2\}, \{-1, 1\}, 2\}.$$

Mängden M består av elementen 2, 5, $\{0, 4, 2\}$ och $\{-1, 1\}$. Alltså är $|M| = 4$. Vi kan till exempel skriva $5 \in M$ och $\{0, 4, 2\} \in M$. Däremot är *inte* 0 eller 4 element i M. ▲

Definition 1.1.2. Mängden som inte innehåller några element alls, det vill säga mängden $\{\}$, kallas *den tomma mängden* och betecknas $\emptyset$. △

Det finns också mängder som innehåller oändligt många element. Ett exempel är mängden av alla heltal, som brukar betecknas $\mathbb{Z}$. (Valet av bokstaven Z kommer från det tyska ordet *Zahl*, som betyder tal.) För en oändlig mängd kan vi förstås inte lista alla element, men i vissa fall kan vi beskriva mängden som en talföljd. Till exempel är

$$\mathbb{Z} = \{0, 1, -1, 2, -2, 3, -3, \ldots\},$$

där de tre punkterna betyder att uppräkningen fortsätter enligt samma mönster i all oändlighet. Några andra viktiga oändliga mängder är de *rationella talen* $\mathbb{Q}$ som är tal på formen $\frac{a}{b}$ där $a, b \in \mathbb{Z}$ och $b \neq 0$, samt de *reella talen* $\mathbb{R}$ som är de tal som finns på en oändligt lång sammanhängande tallinje.[1]

[1] De reella talen $\mathbb{R}$ består dels av de rationella talen, dels av de irrationella talen (vilket är de tal som *inte* kan skrivas på formen $\frac{a}{b}$ där a och b är heltal). De irrationella talen behövs för att fylla i de "glapp" som annars skulle uppstå i tallinjen. Det går att definiera mängden av reella tal på ett mycket mer rigoröst sätt än vad vi gjort här, men det kräver mer arbete och var temat för Cirkeln 2016–2017 (*Vad är ett tal?*), vars kompendium den intresserade kan ladda ner från Cirkelns hemsida `www.math-stockholm.se/cirkel`.

Ett annat sätt att beskriva mängder är genom att välja ut de element från en tidigare känd mängd som uppfyller ett visst villkor. Mängden av element i M som uppfyller ett visst villkor skrivs

$$\{x \in M \mid \text{villkor på } x\}.$$

Till exempel kan vi låta

$$A = \{x \in \mathbb{Z} \mid x > 0\}, \tag{1.1}$$

vilket vi kan utläsa som ”mängden av x i $\mathbb{Z}$ sådana att x är större än noll”, eller helt enkelt ”mängden av heltal som är större än noll”. Mängden A består alltså av de positiva heltalen. Vi kan fortsätta och låta

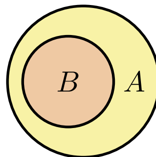
$$B = \{x \in A \mid x \text{ är ett udda tal}\}. \tag{1.2}$$

Mängden B består av alla positiva udda tal.

Definition 1.1.3. Låt A och B vara mängder. Om alla element i B också är element i A sägs B vara en *delmängd* av A. Detta betecknas $B \subseteq A$. △

Till exempel, om A och B definieras som i ekvation (1.1) och (1.2) ovan så är $A \subseteq \mathbb{Z}$ och $B \subseteq A$. Dessutom är förstås $B \subseteq \mathbb{Z}$. Vidare är $\mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R}$.

Ett sätt att illustrera relationer mellan mängder är med hjälp av *venndiagram* (döpta efter den brittiske matematikern John Venn). I ett venndiagram representeras varje mängd av en cirkel. Att B är en delmängd av A illustreras genom att cirkeln för B ritas inuti cirkeln för A.



Figur 1.1: Venndiagram för delmängd $B \subseteq A$.

Mängder kan också beskrivas genom att ange på vilken form elementen ska vara. Till exempel kan mängden av jämna tal (kom ihåg Definition 1.0.1) skrivas

$$\{2k \mid k \in \mathbb{Z}\},$$

vilket utläses ”mängden av element på formen $2k$, där k tillhör $\mathbb{Z}$”. Denna mängd kan också beskrivas med hjälp av ett villkor som

$$\{n \in \mathbb{Z} \mid n \text{ är ett jämnt tal}\}$$

eller som

$$\{n \in \mathbb{Z} \mid n = 2k,\ k \in \mathbb{Z}\}.$$

Mängden av udda tal (Definition 1.0.2) kan på samma sätt skrivas

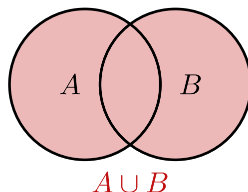
$$\{2k + 1 \mid k \in \mathbb{Z}\}.$$

Vi ska nu definiera fyra olika sätt att kombinera två mängder, med så kallade *mängdoperationer*.

Definition 1.1.4. Låt A och B vara mängder. Mängden av alla element som tillhör *någon* av mängderna A och B kallas *unionen* av A och B, och betecknas $A \cup B$. △

Om till exempel $A = \{1, 2, 3\}$ och $B = \{3, 4, 5\}$ så är $A \cup B = \{1, 2, 3, 4, 5\}$.

I venndiagrammet för $A \cup B$ ritar vi A och B som delvis överlappande cirklar, och allt som tillhör någon av mängderna A och B är färglagt.

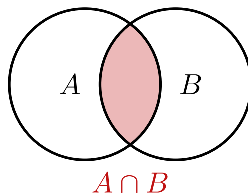


Figur 1.2: Venndiagram för union $A \cup B$.

Definition 1.1.5. Låt A och B vara mängder. Mängden av alla element som tillhör *båda* mängderna A och B kallas *snittet* av A och B, och betecknas $A \cap B$. △

Om som ovan $A = \{1, 2, 3\}$ och $B = \{3, 4, 5\}$ så är $A \cap B = \{3\}$.

I venndiagrammet för $A \cap B$ färglägger vi bara själva överlappet mellan A och B.

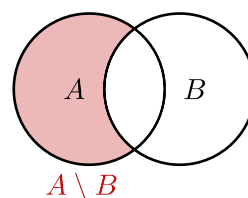


Figur 1.3: Venndiagram för snitt $A \cap B$.

Definition 1.1.6. Låt A och B vara mängder. Mängden av alla element som tillhör A men *inte* B kallas *differensen* (ibland *mängddifferensen*) mellan A och B, och betecknas $A \setminus B$. △

Om $A = \{1, 2, 3\}$ och $B = \{3, 4, 5\}$ så är $A \setminus B = \{1, 2\}$.

I venndiagrammet för $A \setminus B$ färgläggs endast den del av A som inte överlappar B.



Figur 1.4: Venndiagram för differens $A \setminus B$.

Givet två mängder A och B kan vi definiera ett *ordnat par* (a, b) där $a \in A$ och $b \in B$. Mängden av alla sådana ordnade par betecknas

$$A \times B = \{(a, b) \mid a \in A,\ b \in B\}$$

och kallas den *kartesiska produkten* av A och B. I ett ordnat par spelar ordningen roll, så i allmänhet är $(a, b) \neq (b, a)$ och $A \times B \neq B \times A$.

Exempel 1.1.7. Om $A = \{1, 2\}$ och $B = \{1, 5, 9\}$ så är den kartesiska produkten $A \times B$ mängden

$$\begin{aligned} A \times B = \{&(1,1),\ (1,5),\ (1,9),\\ &(2,1),\ (2,5),\ (2,9)\}. \end{aligned}$$

Den kartesiska produkten $B \times A$ är mängden

$$\begin{aligned} B \times A = \{&(1,1),\ (1,2),\\ &(5,1),\ (5,2),\\ &(9,1),\ (9,2)\}. \end{aligned}$$

Observera att $|A \times B| = |B \times A| = 6 = 2 \cdot 3 = |A| \cdot |B|$. (Att detta alltid gäller visas i Övning 1.16.) Dock är mängderna $A \times B$ och $B \times A$ inte lika med varandra. ▲

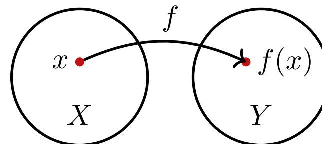
Om $A = B$ skriver vi ofta $A \times A = A^2$. Till exempel är $\mathbb{R}^2$ mängden av ordnade par av reella tal

$$\mathbb{R}^2 = \{(x, y) \mid x, y \in \mathbb{R}\},$$

vilket läsaren kanske känner igen som talplanet.

1.2 Funktioner

En *funktion* består av två mängder X och Y samt en regel som till varje element $x \in X$ entydigt tilldelar precis ett element i Y. Mängden X kallas funktionens *definitionsmängd* och Y kallas funktionens *målmängd*. För att säga att f är en funktion från mängden X till mängden Y kan vi skriva $f : X \to Y$. Vi skriver $f(x)$ för att beteckna det element i Y som funktionen f tilldelar $x \in X$. Vi kallar x för *indata* till funktionen och $f(x)$ för *utdata*.



Figur 1.5: Illustration av en funktion $f : X \to Y$.

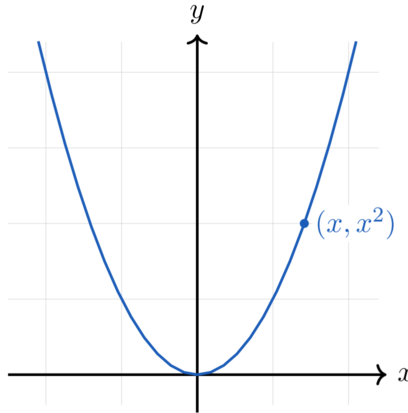
Definition 1.2.1. Givet en funktion $f : X \to Y$ definierar vi *grafen* till funktionen f som mängden $\{(x, f(x)) \mid x \in X\}$. △

Notera att grafen till f är en delmängd av $X \times Y$.

Exempel 1.2.2. En funktion $f : \mathbb{R} \to \mathbb{R}$ definieras genom att till varje $x \in \mathbb{R}$ tilldela talet $f(x) = x^2 \in \mathbb{R}$. Funktionen f kvadrerar alltså sitt indata. Grafen till denna funktion är mängden

$$\{(x, x^2) \mid x \in \mathbb{R}\} \subseteq \mathbb{R}^2,$$

vilket är en parabel. En del av denna parabel visas i Figur 1.6. ▲



Figur 1.6: En del av grafen till funktionen $f(x) = x^2$.

Exempel 1.2.3. Vi kan definiera en funktion $g : X \to \mathbb{Z}$ där X är mängden av ord som finns i Svenska Akademiens ordbok och $g(x)$ är antalet bokstäver i ordet $x \in X$. Till exempel är

$$g(\text{abborre}) = 7, \qquad g(\text{kamelopard}) = 10, \qquad g(\text{sjakal}) = 6, \qquad g(\text{i}) = 1.$$

Däremot är vår funktion g inte definierad för ordet Torbjörn eftersom det inte finns i Svenska Akademiens ordbok. ▲

Exempel 1.2.4. Definitionsmängden X till en funktion f kan förstås vara en kartesisk produkt $X = A \times B$. I så fall består indata till funktionen f av ordnade par (a, b) där $a \in A$ och $b \in B$. Funktionens utdata skrivs då $f(a, b)$. Ett exempel är funktionen $h : \mathbb{Z} \times \mathbb{Z} \to \{0, 1\}$ som definieras av att $h(a, b) = 1$ om $a = kb$ för något heltal k och $h(a, b) = 0$ annars. Funktionen h visar alltså om a är en multipel av b. Till exempel är

$$h(1, 1) = 1, \qquad h(6, 3) = 1, \qquad h(6, 4) = 0, \qquad h(-2, 0) = 0.$$

Givet funktionen h kan vi gå vidare och definiera funktionen $s : \mathbb{Z} \to \{0, 1\}$ genom att låta $s(x) = h(x, 2)$ för $x \in \mathbb{Z}$. Funktionen s visar om x är ett jämnt tal. ▲

Funktioner kan definieras till exempel genom att man anger en formel (som för $f(x) = x^2$), genom att man listar alla funktionsvärden i en tabell (vilket bara fungerar om definitionsmängden är ändlig), eller genom att man anger funktionens regel på något annat sätt. Ett speciellt sätt att definiera funktioner på är genom en *rekursiv* definition, vilket betyder att funktionen definieras med hjälp av referenser till sig själv, som i följande exempel.

Exempel 1.2.5. Låt $\mathbb{Z}_{\geqslant 0} = \{x \in \mathbb{Z} \mid x \geqslant 0\}$ vara mängden av ickenegativa heltal. Vi definierar en funktion $F : \mathbb{Z}_{\geqslant 0} \to \mathbb{Z}_{\geqslant 0}$ rekursivt genom

$$F(n) = \begin{cases} 0 & \text{om } n = 0, \\ 1 & \text{om } n = 1, \\ F(n-1) + F(n-2) & \text{om } n \geqslant 2. \end{cases}$$

Funktionen F definierar en talföljd $F(0)$, $F(1)$, $F(2)$, $F(3)$, $\ldots$ och så vidare. Det är viktigt att definitionen innehåller ett eller några *basfall* som den rekursiva definitionen kan falla tillbaka på, i detta fall $F(0) = 0$ och $F(1) = 1$. Med hjälp av dessa kan vi räkna ut övriga funktionsvärden,

$$\begin{aligned} F(2) &= F(1) + F(0) = 1 + 0 = 1, \\ F(3) &= F(2) + F(1) = 1 + 1 = 2, \\ F(4) &= F(3) + F(2) = 2 + 1 = 3, \\ F(5) &= F(4) + F(3) = 3 + 2 = 5, \end{aligned}$$

och så vidare. Den här specifika talföljden har fått namnet *Fibonaccis talföljd* efter den italienske matematikern Leonardo Fibonacci som levde på 1200-talet. De första 20 talen i talföljden, upp till och med $F(19)$, är

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181.$$

Talet $F(n)$ kallas det n:te Fibonaccitalet. ▲

En funktion måste vara *förutsägbar* och alltså alltid ge samma utdata för ett visst indata. En process som ger slumpmässigt utdata kan alltså inte vara en funktion i matematisk mening. Ibland beror slumpmässighet på att den indata som finns inte beskriver processen fullständigt. Till exempel *är* en väderleksprognos för imorgon en funktion av observationer som gjorts vid olika väderstationer (prognosen beräknas ju utifrån observationerna). Det *faktiska* vädret imorgon är dock *inte* en funktion av dessa observationer, eftersom det kan bero på andra faktorer som inte är kända.

Till sist definierar vi likhet mellan funktioner.

Definition 1.2.6. Två funktioner $f : X \to Y$ och $g : U \to V$ är lika med varandra om $X = U$, $Y = V$, och $f(x) = g(x)$ för alla $x \in X$. Om så är fallet skriver vi $f = g$. △

Observera att två funktioner inte kan vara lika med varandra om de har olika definitionsmängder eller målmängder.

1.3 Logik

Ett *påstående* är inom klassisk logik[2] något som antingen är *sant* eller *falskt*. Nedan följer fyra exempel på påståenden:

[2]Det finns olika grenar inom logiken, men i detta kompendium håller vi oss till klassisk logik, där påståenden antingen är sanna eller falska.

(i) $1 + 1 = 2$,

(ii) $5 > 6$,

(iii) $|\{1, 1, 1\}| = 3$,

(iv) Januari har 31 dagar.

Vi ser att (i) och (iv) är sanna påståenden, medan (ii) och (iii) är falska (kom ihåg att $|\{1, 1, 1\}| = |\{1\}| = 1$). Följande uttryck

- $1 + 1$
- Tomtar och troll
- $\{1, 1, 1\}^2$

är *inte* påståenden; de är varken sanna eller falska.

I datorsammanhang låter man ofta talet 0 betyda falskt och talet 1 betyda sant. Vi kommer använda denna konvention, och i detta kompendium kommer ett påstående P alltså ha värdet 0 om det är ett falskt påstående och värdet 1 om det är sant. Att skriva $P = 1$ är alltså samma sak som att säga "P är sant".

Definition 1.3.1. Vi inför beteckningen $\mathbb{B} = \{0, 1\}$. Om $x \in \mathbb{B}$ kallas x för en *Boolesk variabel.* $\triangle$

En Boolesk variabel har alltså antingen värdet 0 (falskt) eller 1 (sant). Alla påståenden är Booleska variabler. George Boole var en brittisk matematiker verksam på 1800-talet, som införde olika sätt att kombinera Booleska variabler, så kallad *Boolesk algebra.* Den Booleska algebran använder bland annat följande symboler.

Definition 1.3.2. Låt P och Q vara två Booleska variabler. Då är $P \vee Q$ en ny Boolesk variabel som har värdet 1 om *någon* av P och Q har värdet 1, och annars värdet 0. $\triangle$

Definition 1.3.3. Låt P och Q vara två Booleska variabler. Då är $P \wedge Q$ en ny Boolesk variabel som har värdet 1 om *både* P och Q har värdet 1, och annars värdet 0. $\triangle$

Symbolen $\vee$ utläses "eller" och symbolen $\wedge$ utläses "och". Vi kan alltså se $P \vee Q$ som påståendet "P eller Q", och $P \wedge Q$ som påståendet "P och Q". I datorsammanhang används ofta de engelska orden OR för $\vee$ och AND för $\wedge$.

Exempel 1.3.4. Nedan följer fyra påståenden:

(i) $(1 + 2 = 3) \wedge (5 > 4)$,

(ii) $(0 = 0) \vee (5 = 6)$,

(iii) $(6 = 2 \cdot 3) \wedge (x = 2) \wedge (x = 3)$,

(iv) $\Big((1 = 1) \vee (5 \in \mathbb{B})\Big) \wedge (\text{Vinkelsumman i en triangel är } 90°)$.

Påstående (i) är sant eftersom både $1 + 2 = 3$ och $5 > 4$ är sant. Påstående (ii) är sant eftersom $0 = 0$ är sant. Påstående (iii) är falskt eftersom $x = 2$ och $x = 3$ omöjligen kan vara sanna samtidigt oavsett värde på x. Påstående (iv) är falskt eftersom vinkelsumman i en triangel inte är 90°.

Om vi kombinerar $\vee$ och $\wedge$ i samma påstående, som i (iv), behöver vi använda parenteser för att visa i vilken ordning vi ska läsa delpåståendena. Faktum är att påståendet

(iv′) $(1 = 1) \vee \Big((5 \in \mathbb{B}) \wedge (\text{Vinkelsumman i en triangel är } 90°)\Big)$

är sant. ▲

Lägg märke till att $\vee$ och $\wedge$ i själva verket är två funktioner med definitionsmängd $\mathbb{B} \times \mathbb{B}$ och målmängd $\mathbb{B}$. Funktionernas indata är alltså ett ordnat par (P, Q) och utdata är den Booleska variabeln $P \vee Q$ respektive $P \wedge Q$. Ett vanligt sätt att visa vilka värden sådana här Booleska funktioner antar för olika indata är med hjälp av *sanningstabeller*. Sanningstabellerna för $P \vee Q$ och $P \wedge Q$ visas nedan.

P	Q	$P \vee Q$
0	0	0
1	0	1
0	1	1
1	1	1

P	Q	$P \wedge Q$
0	0	0
1	0	0
0	1	0
1	1	1

Vi kan även använda sanningtabeller för att analysera mer komplicerade sammansatta påståenden. Ett exempel följer nedan för $(P \vee Q) \wedge R$.

P	Q	R	$P \vee Q$	$(P \vee Q) \wedge R$
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
0	0	1	0	0
1	1	0	1	0
1	0	1	1	1
0	1	1	1	1
1	1	1	1	1

Två andra viktiga Booleska funktioner är *negation* och *implikation*. Dessa definieras på följande sätt.

Definition 1.3.5. Låt P vara en Boolesk variabel. Då är $\neg P$ en ny Boolesk variabel som har värdet 1 om P har värdet 0, och 0 om P har värdet 1. △

Definition 1.3.6. Låt P och Q vara två Booleska variabler. Då är $P \implies Q$ en ny Boolesk variabel som har värdet 0 om P har värdet 1 samtidigt som Q har värdet 0, och annars värdet 1. △

Symbolen $\neg$ utläses ”icke” (på engelska: NOT) medan $P \implies Q$ kan utläsas antingen ”P implicerar Q” eller ”om P så Q”. Sanningstabellerna för dessa funktioner visas nedan.

P	$\neg P$
0	1
1	0

P	Q	$P \implies Q$
0	0	1
1	0	0
0	1	1
1	1	1

Exempel 1.3.7. Implikation ($\implies$) är mycket vanligt i matematiska påståenden. Nedan följer några exempel.

(i) ”Om n är ett jämnt tal så är $n + 1$ ett udda tal”, vilket kan skrivas
$(n \text{ är ett jämnt tal}) \implies (n + 1 \text{ är ett udda tal})$

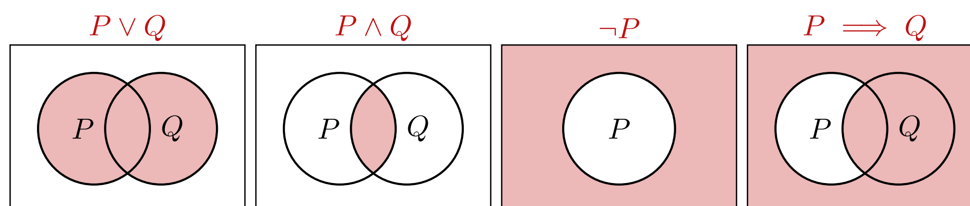
(ii) ”Om n är ett heltal så är n ett jämnt eller udda tal”, vilket kan skrivas
$(n \in \mathbb{Z}) \implies \big((n \text{ är ett jämnt tal}) \vee (n \text{ är ett udda tal})\big)$

(iii) ”Om B är en delmängd av A, och C är en delmängd av B, så är C en delmängd av A”, vilket kan skrivas
$\big((B \subseteq A) \wedge (C \subseteq B)\big) \implies (C \subseteq A)$

(iv) ”Om 5 tillhör den tomma mängden så är $1 = 2$”, vilket kan skrivas
$(5 \in \emptyset) \implies (1 = 2)$

Alla dessa påståenden är sanna. Påstående (i) bevisade vi som Sats 1.0.3, och påstående (ii) kommer vi att bevisa som Sats 1.4.5. Påstående (iii) bevisas i Övning 1.11. Påstående (iv) är sant eftersom $P \implies Q$ alltid är sant när P är falskt, vilket kan ses i sanningstabellen. ▲

Vi kan också använda venndiagram för att illustrera Booleska funktioner. Vi ritar då upp en cirkel för varje indatavariabel så att cirklarna delvis överlappar, och tänker oss att varje variabel har värdet 1 inuti sin respektive cirkel och 0 utanför. Vi färglägger sedan de områden som motsvarar att funktionen i fråga ger 1 som utdata. Se Figur 1.7.



Figur 1.7: Venndiagram för några Booleska funktioner.

Med hjälp av venndiagrammen ovan kan vi tolka de Booleska funktionerna som mängdoperationer, och vi ser att funktionerna $\vee$ och $\wedge$ motsvarar mängdoperationerna union ($\cup$) respektive snitt ($\cap$); jämför med Figur 1.2 och 1.3.

1.4 Matematiska bevis

Vi har redan sett ett exempel på en sats och tillhörande bevis i och med Sats 1.0.3 på sida 1. Ett sådant bevis, där man med hjälp av definitioner och tidigare satser tar sig direkt från satsens antaganden till dess slutsats, kallas ett *direkt bevis*. Några fler exempel på direkta bevis kommer här.

Sats 1.4.1. *$(P \implies Q) = ((\neg P) \vee Q)$ för alla $P, Q \in \mathbb{B}$.*

Bevis. Låt oss jämföra sanningstabellerna för $P \implies Q$ samt $(\neg P) \vee Q$.

P	Q	$P \implies Q$
0	0	1
1	0	0
0	1	1
1	1	1

P	Q	$\neg P$	$(\neg P) \vee Q$
0	0	1	1
1	0	0	0
0	1	1	1
1	1	0	1

Detta visar att $P \implies Q$ och $(\neg P) \vee Q$ alltid har samma värde. $\square$

Sats 1.4.2. *Låt X och Y vara ändliga mängder, med $|X| = k$ och $|Y| = n$. Antalet möjliga olika funktioner från X till Y är n^k.*

Bevis. En funktion $f : X \to Y$ måste ge ett värde $f(x) \in Y$ till varje element $x \in X$. Det finns $|X| = k$ olika element i X, och alla dessa ska få ett funktionsvärde. Antalet möjliga olika funktionsvärden är $|Y| = n$. Vi måste alltså välja k stycken funktionsvärden, och varje värde kan väljas på n olika sätt. Enligt *multiplikationsprincipen*[3] är det totala antalet möjliga sätt att välja funktionsvärdena på

$$\underbrace{n \cdot n \cdot n \cdots n}_{k \text{ stycken faktorer}} = n^k.$$

$\square$

En annan typ av bevis är *motsägelsebevis* (även kallade *indirekta bevis*), där man bevisar ett påstående P genom att visa att motsatsen, $\neg P$, leder till en motsägelse, alltså en omöjlighet. Eftersom $\neg P$ leder till en omöjlighet så kan inte $\neg P$ vara sant, och därför måste P vara sant.[4] Ett enkelt exempel följer på nästa sida.

[3] Multiplikationsprincipen säger att om det finns a sätt att göra ett första val på och b sätt att göra ett andra val på, så finns det ab sätt att tillsammans göra de två valen på. I detta fall har vi k val att göra, och varje val kan göras på n sätt.

[4] Denna princip kan formuleras med logiska symboler på följande sätt: Om vi kan visa att implikationen $(\neg P) \implies Q$ är sann, och Q uppenbart är falskt, så måste även $\neg P$ vara falskt (för $0 \implies 0$ är sant, men $1 \implies 0$ är falskt), och alltså är P sant.

Sats 1.4.3. *Det finns inget minsta positivt rationellt tal.*

Bevis. Ett positivt rationellt tal kan skrivas på formen a/b där både a och b är positiva heltal. Låt oss anta motsatsen till satsens påstående, alltså att det *finns* ett minsta positivt rationellt tal. Låt oss kalla detta tal x, och skriva $x = a/b$ där a och b är positiva heltal. Betrakta talet $y = x/2 = a/(2b)$. Även y är ett positivt rationellt tal, och y är uppenbarligen mindre än x. Detta är en motsägelse, för x var det minsta positiva rationella talet. Alltså måste antagandet att det finns ett minsta positivt rationellt tal vara falskt. □

Ett välkänt motsägelsebevis används för att visa att $\sqrt{2}$ inte är ett rationellt tal.

Sats 1.4.4. *$\sqrt{2}$ är inte ett rationellt tal.*

Bevis. Antag motsatsen, det vill säga att $\sqrt{2}$ är ett rationellt tal. Då kan vi skriva

$$\sqrt{2} = \frac{a}{b}, \tag{1.3}$$

där a och b är heltal och $b \neq 0$. Dessutom kan vi anta att bråket a/b är förkortat så långt det går, så att a och b inte har några gemensamma delare (heltalsfaktorer) förutom 1. Genom att kvadrera ekvation (1.3) och multiplicera med b^2 fås

$$2b^2 = a^2, \tag{1.4}$$

vilket visar att a^2 är ett jämnt tal eftersom b^2 är ett heltal. Enligt Övning 1.6 är också a ett jämnt tal, låt säga att $a = 2k$ för något $k \in \mathbb{Z}$. Kvadrering ger då $a^2 = (2k)^2 = 4k^2$. Vi sätter in detta i ekvation (1.4) och får efter division med 2

$$b^2 = 2k^2,$$

vilket säger att även b^2 är ett jämnt tal. Övning 1.6 ger att b är ett jämnt tal, men då är både a och b jämna tal och har alltså 2 som gemensam delare. Detta motsäger antagandet vi gjorde om att bråket a/b var förkortat så långt det går. Alltså kan $\sqrt{2}$ inte vara ett rationellt tal. □

Till sist tar vi upp en typ av bevis som kallas för *induktionsbevis*. I detta fall vill vi bevisa en *följd* av påståenden P_1, P_2, P_3, ... och så vidare i all oändlighet. Vi gör detta genom att

(i) Bevisa att det första påståendet P_1 är sant,

(ii) Bevisa att $P_k \implies P_{k+1}$, det vill säga om P_k är sant så är även P_{k+1} sant.

Vi bevisar alltså först P_1, och (ii) leder därefter till att P_2 måste vara sant, vilket leder till att P_3 måste vara sant, och så vidare i all oändlighet. Steg (i) kallas för *bassteg* eller *basfall*, och steg (ii) kallas för *induktionssteg*.

Vi kan använda induktion för att bevisa följande.

Sats 1.4.5. *Alla heltal är udda eller jämna tal.*

Bevis. För att kunna använda induktion börjar vi med att formulera om påståendet till följande: ”n är ett udda eller jämnt tal”, där n är ett heltal. Vi kallar detta påstående P_n.

(i) Vi börjar med att välja ett basfall, alltså ett första påstående att bevisa. Till exempel kan vi välja P_0, påståendet ”0 är ett udda eller jämnt tal”. Faktum är att 0 är ett jämnt tal eftersom $0 = 2 \cdot 0$. Alltså är P_0 sant.

(ii) Vi ska nu bevisa induktionssteget, det vill säga att om P_k är sant så är P_{k+1} sant. Det vi ska bevisa är att om k är udda eller jämnt så är även $k+1$ udda eller jämnt. Här kan vi använda Sats 1.0.3. Vi delar upp beviset i två fall.

Fall 1: Om k är jämnt så ger Sats 1.0.3 att $k + 1$ är udda.

Fall 2: Om k är udda så ger Sats 1.0.3 att $k + 1$ är jämnt.

I båda fallen gäller att $k + 1$ är udda eller jämnt, vilket var det vi ville bevisa.

Detta bevisar att alla heltal större än eller lika med 0 är udda eller jämna. Beviset för negativa heltal lämnas som Övning 1.13. □

Sammanfattningsvis kunde vi alltså visa att 0 är jämnt, och därför måste 1 vara udda, vilket betyder att 2 måste vara jämnt, och så vidare.

I en annan variant av induktionsbevis använder man sig av de två stegen

(i) Bevisa att det första påståendet P_1 är sant,

(ii) Bevisa att $(P_1 \wedge P_2 \wedge \cdots \wedge P_k) \implies P_{k+1}$.

I induktionssteget antar man alltså att *alla* de tidigare påståendena är sanna, inte bara det senaste. Detta kallas ibland *stark induktion*. Principen är densamma som för vanliga induktionsbevis: om vi bevisat att P_1 är sant följer det att P_2 är sant. Då är både P_1 och P_2 sanna, och det följer att P_3 är sant. Då vet vi att P_1, P_2 och P_3 är sanna, så P_4 är sant, och så vidare. Ibland kan det behövas *flera* basfall, och då kan steg (i) till exempel bestå i att bevisa att både P_1 och P_2 är sanna (eller alla P_n upp till ett bestämt värde på n).

Vi avslutar med ett bevis där stark induktion används.

Definition 1.4.6. Ett heltal större än 1 kallas för ett *primtal* om det inte kan uttryckas som en produkt av två mindre positiva heltal. △

Till exempel är talen 2, 3, 5 och 7 primtal. Däremot är inte 4 eller 6 primtal, eftersom $4 = 2 \cdot 2$ och $6 = 2 \cdot 3$.

Sats 1.4.7. *Varje heltal större än 1 är antingen ett primtal, eller en produkt av flera primtal.*

Bevis. Låt P_n vara påståendet ”n är antingen ett primtal eller en produkt av flera primtal”, för heltal $n \geqslant 2$. Vi använder (stark) induktion.

(i) Basfallet blir P_2, vilket är sant eftersom 2 är ett primtal.

(ii) För induktionssteget antar vi att P_2, P_3, $\ldots$, P_k är sanna upp till något tal k. Vi ska bevisa P_{k+1}, att $k+1$ antingen är ett primtal eller en produkt av flera primtal.

Fall 1: Om $k+1$ är ett primtal så är vi klara.

Fall 2: Om $k+1$ inte är ett primtal måste vi bevisa att $k+1$ är en produkt av flera primtal. Eftersom $k+1$ inte är ett primtal är $k+1 = a \cdot b$, där a och b är positiva heltal mindre än $k+1$, enligt Definition 1.4.6. Varken a eller b kan vara lika med 1, eftersom det andra talet då skulle vara lika med $k+1$. Talen a och b är alltså heltal större än 1 men mindre än $k+1$. Enligt vårt induktionsantagande är P_a och P_b då sanna; a är antingen ett primtal eller en produkt av primtal, och detsamma gäller b. Alltså är $k+1 = a \cdot b$ en produkt av flera primtal.

Därmed har vi bevisat satsen med hjälp av (stark) induktion. □

Övningar

Grundläggande uppgifter

Övning 1.1. Lista alla element i följande mängder.

(i) $A = \{x \in \mathbb{Z} \mid 0 < x + 2 \leqslant 4\}$

(ii) $B = \{x \in \mathbb{Z} \mid (x^2 = 4) \vee (x^3 = 27)\}$

(iii) $C = \{2x + 5 \mid x \in B\}$

(iv) $A \cap C$

(v) $(A \cup B) \setminus C$

Övning 1.2. Låt $A = \{1, 2, 5, \pi, \{1, 2\}, \{9, 27\}\}$. Vilka av följande mängder är delmängder till A?

$$B = \{\pi, 5\}$$
$$C = \{1, 1, 2\}$$
$$D = \{\pi, -\pi\}$$
$$E = \{9, 27\}$$
$$F = \{\{9, 27\}\}$$
$$G = \{1, 2, 5, \pi, \{1, 2\}, \{9, 27\}\}$$

Övning 1.3. Är de två mängderna lika med varandra? Om de är lika, motivera varför! Om de inte är lika, ge ett exempel på ett element som finns i den ena mängden, men inte i den andra.

(i) $\{0, 1\}$ och $\{\{0, 1\}\}$

(ii) $\{x \in \mathbb{Z} \mid x^2 = 4\}$ och $\{x \in \mathbb{Z} \mid x^3 = 8\}$

(iii) $\{1 - x \mid x \in \mathbb{B}\}$ och $\{x^2 \mid x \in \mathbb{B}\}$

(iv) $\{2x + 1 \mid x \in \mathbb{Z}\}$ och $\{2x - 1 \mid x \in \mathbb{Z}\}$

(v) $\{\sqrt{x^2} \mid x \in \mathbb{Z}\}$ och $\{x \in \mathbb{Z} \mid x \geqslant 0\}$

Övning 1.4. Bevisa att ett heltal inte kan vara både udda och jämnt på samma gång.

Övning 1.5. Bevisa att det inte finns något största heltal.

Övning 1.6. Låt x vara ett heltal. Bevisa att om x^2 är ett jämnt tal så är även x ett jämnt tal.

Övning 1.7. Vilka av följande regler definierar funktioner i matematisk mening? För de regler som *inte* definierar funktioner, förklara varför de inte är funktioner.

(i) $f : \mathbb{Z} \to \mathbb{Z}$ tar indata x och ger som utdata $f(x) = x$ med 50 % sannolikhet och $f(x) = x + 1$ med 50 % sannolikhet.

(ii) $f : \mathbb{Z} \to \mathbb{Z}$ tar indata x och ger som utdata $f(x) = 2$.

(iii) $f : \mathbb{Z} \to \mathbb{Z}$ tar indata x och ger som utdata $f(x)$ antalet femmor i talet x då x skrivs i det vanliga decimala talsystemet (det vill säga så som vi brukar skriva tal med siffrorna 0 till 9).

(iv) $f : \mathbb{Z} \to \{0, 1\}$ tar indata x och ger utdata $f(x) = 1$ om det idag är den x:te dagen i månaden, annars $f(x) = 0$.

(v) Låt T vara mängden av alla möjliga trianglar i planet. $f : T \to \mathbb{R}$ tar en triangel x som indata och ger som utdata $f(x)$ triangelns area.

(vi) $f : \{1, 2\} \to \mathbb{R}$ tar indata x och ger utdata $f(x) = 5$ om $x = 1$ och $f(x) = 2$ om $x = 2$.

(vii) $f : \mathbb{R} \to \mathbb{R}$ tar indata x och ger utdata $f(x) = 1$ om x är ett rationellt tal, annars $f(x) = 0$.

Övning 1.8. Vilka av följande är påståenden? Vilka av påståendena är sanna?

(i) Vinkelsumman i en triangel

(ii) Helium är ett grundämne

(iii) $6 = 2 + 3$

(iv) $\frac{0}{0}$

(v) $5 \wedge 3$

(vi) $\emptyset \subseteq \{4, 6, 8\}$

Övning 1.9. Är de två funktionerna f och g lika med varandra? Om de inte är lika med varandra, förklara varför!

(i) $f : \mathbb{R} \to \mathbb{Z}$ som ges av

$$f(x) = \begin{cases} 1 & \text{om } x \in \mathbb{Z}, \\ 0 & \text{annars,} \end{cases}$$

och $g : \mathbb{Z} \to \mathbb{Z}$ som ges av $g(x) = 1$ för alla $x \in \mathbb{Z}$.

(ii) $f : \mathbb{R} \to \mathbb{R}$ som ges av $f(x) = 2x$ och $g : \mathbb{Z} \to \mathbb{Z}$ som ges av $g(x) = 2x$.

(iii) $f : \mathbb{Z} \to \mathbb{Z}$ som ges av $f(x) = \sqrt{x^2}$ och $g : \mathbb{Z} \to \mathbb{Z}$ som ges av $g(x) = x$.

(iv) $f : \mathbb{Z} \to \mathbb{Z}$ som ges av

$$f(x) = \begin{cases} 0 & \text{om } x = 0, \\ f(x-1) + 1 & \text{om } x > 0, \\ f(x+1) - 1 & \text{om } x < 0, \end{cases}$$

och $g : \mathbb{Z} \to \mathbb{Z}$ som ges av $g(x) = x$ för alla $x \in \mathbb{Z}$.

Övning 1.10. Låt P och Q vara Booleska variabler och bevisa att

(i) $\neg(P \wedge Q) = (\neg P) \vee (\neg Q)$,

(ii) $\neg(P \vee Q) = (\neg P) \wedge (\neg Q)$.

(Dessa likheter kallas *De Morgans lagar* efter matematikern Augustus De Morgan.)

Övning 1.11. Låt A, B och C vara mängder. Visa med hjälp av definitionen av delmängd att om $B \subseteq A$ och $C \subseteq B$ så är $C \subseteq A$.

Övning 1.12 ($\star$). Låt $n \geqslant 0$ vara ett heltal. Visa att summan av de n första tvåpotenserna är lika med $2^{n+1} - 1$. Alltså, visa att

$$2^0 + 2^1 + 2^2 + \ldots + 2^n = 2^{n+1} - 1.$$

Övning 1.13. Slutför beviset av Sats 1.4.5 genom att visa att även alla negativa heltal är udda eller jämna tal.

Mer avancerade uppgifter

Övning 1.14. Bevisa att

(i) summan av två udda tal är jämn,

(ii) produkten av två udda tal är udda.

Övning 1.15 ($\star\star$). Bevisa att det finns oändligt många primtal.

Övning 1.16. Bevisa att om A och B är ändliga mängder så är

$$|A \times B| = |A| \cdot |B|.$$

Övning 1.17. Hur många olika funktioner finns det från $\mathbb{B} \times \mathbb{B}$ till $\mathbb{B}$?

Övning 1.18. En funktion $\veebar$ från $\mathbb{B} \times \mathbb{B}$ till $\mathbb{B}$ kan definieras genom att låta $P \veebar Q$ ha värdet 1 om P eller Q, men *inte båda*, har värdet 1. Funktionen $\veebar$ kallas XOR (exclusive OR) och har sanningstabellen nedan.

P	Q	$P \veebar Q$
0	0	0
1	0	1
0	1	1
1	1	0

(i) Hur ser venndiagrammet för $P \veebar Q$ ut?

(ii) Skriv upp sanningstabellen för $(P \veebar Q) \veebar R$.

(iii) Visa att $P \veebar Q$ kan uttryckas med hjälp av funktionerna $\vee$, $\wedge$ och $\neg$.

Övning 1.19 ($\star$)**.** Vi kan definiera ytterligare en funktion från $\mathbb{B} \times \mathbb{B}$ till $\mathbb{B}$ genom att sätta

$$P \uparrow Q = \neg(P \wedge Q)$$

för alla $P, Q \in \mathbb{B}$. Funktionen $\uparrow$ kallas NAND (NOT AND) och har sanningstabellen nedan.

P	Q	$P \uparrow Q$
0	0	1
1	0	1
0	1	1
1	1	0

Ett intressant faktum är att *alla* möjliga Booleska funktioner kan uttryckas som kombinationer av endast NAND. Till exempel är

$$\neg P = P \uparrow P,$$

vilket lätt kan verifieras med en sanningstabell.

(i) Bevisa att $P \wedge Q = (P \uparrow Q) \uparrow (P \uparrow Q)$.

(ii) Bevisa att även $P \vee Q$ kan uttryckas som kombinationer av $\uparrow$.

(iii) Hur skulle man kunna bevisa att alla möjliga funktioner från $\mathbb{B} \times \mathbb{B}$ till $\mathbb{B}$ kan uttryckas som kombinationer av $\uparrow$?

Övning 1.20. Låt A och B vara mängder. Visa med hjälp av definitionen av delmängd att om $A \subseteq B$ och $B \subseteq A$ så är $A = B$. (Detta är ett vanligt sätt att visa att två mängder är lika med varandra.)

Övning 1.21. Låt A, B och C vara mängder. Bevisa att likheterna

(i) $(A \cup B) \setminus C = (A \setminus C) \cup (B \setminus C)$

(ii) $C \setminus (A \cup B) = (C \setminus A) \cap (C \setminus B)$

alltid gäller, det vill säga att mängderna i vänsterledet och högerledet innehåller samma element.

Övning 1.22 ($\star$)**.** Låt A och B vara mängder. Bevisa att

$$(A \times B) \cap (B \times A) = (A \cap B) \times (A \cap B).$$

Övning 1.23 ($\star\star$)**.** Låt F vara Fibonaccifunktionen från Exempel 1.2.5. Bevisa att

$$F(n) = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}} \qquad \text{för alla } n \in \mathbb{Z}_{\geqslant 0} = \{x \in \mathbb{Z} \mid x \geqslant 0\},$$

där $\phi = (1+\sqrt{5})/2$ är det *gyllene snittet*, som löser ekvationen $\phi^2 = \phi + 1$.

Programmeringsuppgifter

Övning 1.P1. Skriv ett program som beräknar och skriver ut det n:te Fibonaccitalet, alltså $F(n)$ från Exempel 1.2.5. Använd programmet för att räkna ut $F(5)$, $F(10)$, $F(20)$ och $F(40)$.

Övning 1.P2 ($\star$)**.** Skriv ett program som beräknar och skriver ut kvoten $F(k+1)/F(k)$ för $k = 1, 2, \ldots, n$, där n är ett givet positivt heltal. Använd programmet för att skriva ut kvoterna upp till och med $n = 30$. Vad verkar hända?

2 Hur kan en dator räkna?

I en dator lagras all information som en följd av ettor och nollor. I datorns minne finns långa rader av små elektroniska komponenter som kan växla mellan hög spänning (vilket tolkas som 1) och låg spänning (vilket tolkas som 0). På samma sätt kan information överföras, till exempel genom en optisk fiber där ljussignalen är stark (tolkas som 1) eller svag (tolkas som 0). En följd av ettor och nollor kan tolkas på olika sätt, till exempel som ett tal ('71'), ett tecken i en text ('G'), eller färgen på en pixel i en bild.

Detta kapitel och nästa handlar om hur datorer kan lagra tal och räkna med dem. Grundprincipen är densamma i all modern elektronik – vi talar alltså inte bara om ”vanliga” datorer, utan även om mobiltelefoner, spelkonsoler, miniräknare, mikrovågsugnar, digitala klockor och så vidare. Detta kapitel handlar om heltal, nästa kapitel om tal med decimaler.

Ibland kan datorernas sätt att räkna på leda till förvånande resultat. Till exempel kan, under vissa omständigheter, $255 + 1$ bli 0 istället för 256, och på samma sätt kan $0 - 1$ bli 255. Detta återkommer vi till i avsnitt 2.2, men först ska vi gå igenom hur en dator över huvud taget lagrar tal.

2.1 Binära tal och osignerade heltal

Datorer kan alltså bara lagra ettor och nollor, vilket betyder att de måste använda det binära talsystemet. För att förstå hur det binära talsystemet fungerar börjar vi med att repetera hur vårt vanliga *decimala talsystem* fungerar. När vi skriver ett tal som 3405 menar vi egentligen

$$\begin{aligned}3405 &= 3 \cdot 10^3 + 4 \cdot 10^2 + 0 \cdot 10^1 + 5 \cdot 10^0 \\ &= 3 \cdot 1000 + 4 \cdot 100 + 0 \cdot 10 + 5 \cdot 1.\end{aligned}$$

Siffrornas position i talet 3405 avgör vilken tiopotens varje siffra ska multipliceras med, från 10^0 för entalssiffran längst till höger och så vidare. *Basen* i det decimala talsystemet är 10, vilket återspeglas i att vi använder potenser av talet 10, och har tio olika siffror (0, 1, 2, 3, 4, 5, 6, 7, 8 och 9).

I det *binära talsystemet* är basen 2, och vi använder då istället potenser av talet 2, och har endast två olika siffror (0 och 1). Ett exempel på ett tal skrivet i det binära talsystemet är

$$\begin{aligned}11010_2 &= 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\ &= 1 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 \\ &= 16 + 8 + 2 = 26, \qquad (2.1)\end{aligned}$$

vilket som vi ser är talet 26. Talet 26 är detsamma oavsett om vi skriver det som 26 (i bas 10) eller 11010_2 (i bas 2); det är bara olika sätt att skriva talet på. Vi använder en liten tvåa ($_2$) till höger för att visa att ett tal är skrivet i bas 2.

Både det binära och det decimala talsystemet kan användas för att skriva alla heltal. Om vi har ett tal skrivet i bas 2 så är det lätt att omvandla talet till

bas 10; vi skriver bara ut tvåpotenserna och adderar dem, som vi gjorde i ekvation (2.1). För att omvandla ett tal skrivet i bas 10 till bas 2 kan vi göra som i följande exempel.

Exempel 2.1.1. Skriv talet 3405 i bas 2.

Vi ska bestämma vilka tvåpotenser talet 3405 är uppbyggt av. Vi kan börja med att hitta den största tvåpotensen som är mindre än eller lika med 3405. Följande lista visar alla tvåpotenser mindre än 10000.

$$\begin{array}{ll} 2^0 = 1 & 2^7 = 128 \\ 2^1 = 2 & 2^8 = 256 \\ 2^2 = 4 & 2^9 = 512 \\ 2^3 = 8 & 2^{10} = 1024 \\ 2^4 = 16 & 2^{11} = 2048 \\ 2^5 = 32 & 2^{12} = 4096 \\ 2^6 = 64 & 2^{13} = 8192 \end{array}$$

Tydligen är $2^{11} = 2048$ den största tvåpotensen mindre än eller lika med 3405. Skillnaden är $3405 - 2048 = 1357$. Detta betyder att

$$3405 = 2^{11} + 1357.$$

Vi gör likadant med talet 1357 och ser att $2^{10} = 1024$ är den största tvåpotensen mindre än eller lika med 1357. Skillnaden är $1357 - 1024 = 333$. Nu vet vi alltså att

$$3405 = 2^{11} + 2^{10} + 333.$$

På samma sätt är $333 = 256 + 77$, och $77 = 64 + 13$. Till sist är $13 = 8 + 5$, och $5 = 4 + 1$. Om vi nu sätter ihop allt som vi kommit fram till ser vi att

$$\begin{aligned} 3405 &= 2^{11} + 2^{10} + 2^8 + 2^6 + 2^3 + 2^2 + 2^0 \\ &= 1 \cdot 2^{11} + 1 \cdot 2^{10} + 0 \cdot 2^9 + 1 \cdot 2^8 + 0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + \\ &\quad + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0. \end{aligned}$$

Detta betyder att $3405 = 1101\,0100\,1101_2$. ▲

Om vi har två tal skrivna i bas 2 så kan vi addera, subtrahera, multiplicera eller dividera dem för hand precis som vi är vana att göra med tal skrivna i bas 10.

Exempel 2.1.2. Beräkna summan av $27 = 11011_2$ och $51 = 110011_2$.

Vi kan göra en vanlig additionsuppställning, och gå från höger till vänster och summera en sifferposition i taget. (De små understrukna ettorna nedan är minnessiffror som flyttas över till positionen till vänster eftersom resultatet i nuvarande position blev $2 = 10_2$ eller större.)

Steg 0.

$$\begin{array}{r} 110011_2 \\ +\quad 11011_2 \\ \hline \end{array}$$

Steg 1.

$$\begin{array}{r} \underline{1} \\ 110011_2 \\ +\quad 11011_2 \\ \hline 0_2 \end{array}$$

Steg 2.
$$\begin{array}{r} {\scriptstyle 1\,1} \\ 110011_2 \\ +\quad 11011_2 \\ \hline 10_2 \end{array}$$

Steg 3.
$$\begin{array}{r} {\scriptstyle 1\,1} \\ 110011_2 \\ +\quad 11011_2 \\ \hline 110_2 \end{array}$$

Steg 4.
$$\begin{array}{r} {\scriptstyle 1\,1} \\ 110011_2 \\ +\quad 11011_2 \\ \hline 1110_2 \end{array}$$

Steg 5.
$$\begin{array}{r} {\scriptstyle 1\,1\,1} \\ 110011_2 \\ +\quad 11011_2 \\ \hline 01110_2 \end{array}$$

Steg 6.
$$\begin{array}{r} {\scriptstyle 1\,1\,1\,1} \\ 110011_2 \\ +\quad 11011_2 \\ \hline 001110_2 \end{array}$$

Steg 7.
$$\begin{array}{r} {\scriptstyle 1\,1\,1\,1} \\ 110011_2 \\ +\quad 11011_2 \\ \hline 1001110_2 \end{array}$$

Summan är alltså 1001110_2. I bas 10 blir talet $2^6 + 2^3 + 2^2 + 2^1 = 78$, vilket såklart är $27 + 51$. För oss människor är det förmodligen lättare att genomföra additionen direkt i bas 10 istället. För en dator är det istället snabbare att använda bas 2. Vi ska titta närmare på hur man faktiskt kan konstruera en maskin som adderar heltal i bas 2 i avsnitt 2.3. ▲

Osignerade heltal. En siffra i ett tal skrivet i bas 2 kallas en *bit* (från engelskans *binary digit*, alltså ”binär siffra”). Datorer har en begränsad mängd minne, och därför lagras tal oftast med ett bestämt antal bitar, till exempel 8 bitar. Vi kan tänka oss att datorn har 8 stycken lådor som var och en kan innehålla antingen en etta eller nolla, som i figuren nedan.

0	0	1	1	0	1	0	0	$_2$
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	

Talet som lagras här är $00110100_2 = 110100_2 = 52$ (de inledande nollorna påverkar ju inte talets värde). Ett tal som lagras på det här sättet med N bitar (där N är ett positivt heltal) kallas ett *N-bitars osignerat heltal* (på engelska *N-bit unsigned integer*). Ordet *osignerat* betyder att vi inte har lagrat något tecken (*sign* på engelska), alltså information om huruvida det är ett negativt eller positivt tal. Osignerade heltal kan därför inte vara negativa. Följande sats berättar vilka värden som ett N-bitars osignerat heltal kan anta.

Sats 2.1.3. *Låt N vara ett positivt heltal. Ett N-bitars osignerat heltal kan anta 2^N olika värden, nämligen*

$$0, 1, \dots, 2^N - 1.$$

Bevis. Vi bevisar påståendet med hjälp av induktion över N.

I basfallet är $N = 1$ och påståendet är uppenbarligen sant eftersom 1 bit kan anta $2 = 2^1$ olika värden, och dessa värden är 0 och $1 = 2^1 - 1$.

Antag att påståendet är sant för $N = k$, alltså att ett k-bitars osignerat heltal kan anta de 2^k olika värdena $0, 1, \dots, 2^k - 1$. Vi lägger nu till en bit längst till

vänster, så att vi får ett $k+1$-bitars osignerat heltal. Om den nya biten sätts till 0 kan vi lagra alla tal från tidigare, alltså de 2^k värdena $0, 1, \dots, 2^k - 1$. Om den nya biten sätts till 1 kan vi lagra de 2^k nya värdena

$$2^k, 2^k + 1, \dots, 2^k + 2^k - 1$$

(de gamla värdena plus 2^k, vilket är den tvåpotens som multipliceras med den nya biten). Vi ser att det totalt blir $2^k + 2^k = 2^{k+1}$ värden, nämligen

$$0, 1, \dots, 2^k - 1, 2^k, 2^k + 1, \dots, 2^{k+1} - 1.$$

Därmed är påståendet sant även för $N = k + 1$ och vi har bevisat satsen med hjälp av induktion. □

Exempel 2.1.4. Ett 8-bitars osignerat heltal kan anta $2^8 = 256$ olika värden, nämligen $0, 1, \dots, 255$. (En följd av 8 bitar brukar kallas en *oktett* eller en *byte*. En byte kan alltså anta 256 olika värden.)

Ett 32-bitars osignerat heltal kan anta $2^{32} = 4\,294\,967\,296$ (över 4 miljarder) olika värden, från 0 till $4\,294\,967\,295$. (Detta motsvarar 4 byte.)

Ett 64-bitars osignerat heltal kan anta $2^{64} = 18\,446\,744\,073\,709\,551\,616$ (ett tal med 20 siffror i bas 10) olika värden, från 0 till $2^{64} - 1$. (Detta motsvarar 8 byte.) ▲

2.2 Överflöde och kongruensräkning

Vad händer om värdet som ska lagras i ett N-bitars osignerat heltal är för stort? Som exempel, låt oss säga att vi använder 4-bitars osignerade heltal och vill addera talen $1011_2 = 11$ och $1100_2 = 12$. Summan blir $10111_2 = 23$, men detta värde kan ju inte lagras med endast 4 bitar som de ursprungliga talen.

$$\begin{array}{rc}
 & \boxed{1}\boxed{0}\boxed{1}\boxed{1}_2 \\
+ & \boxed{1}\boxed{1}\boxed{0}\boxed{0}_2 \\ \hline
= & {\color{red}1}\;\boxed{0}\boxed{1}\boxed{1}\boxed{1}_2 \\
 & 2^4\;\; 2^3\;\; 2^2\;\; 2^1\;\; 2^0
\end{array}$$

Detta kallas för *överflöde* (på engelska *overflow*), och innebär att man försöker lagra ett värde som är antingen för stort eller för litet för det antal bitar som man arbetar med (i fallet med 4 bitar antingen större än 15 eller mindre än 0).

Hur överflöde hanteras beror på vilken typ av dator och även vilket programmeringsspråk man använder. Vanligtvis händer något av följande:

(i) Datorn utökar automatiskt antalet bitar som används för att lagra talet så att det får plats. Resultatet i exemplet ovan kan då bli $\boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{1}_2$ med 5 bitar, eller också $\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{1}_2$ med 8 bitar, eller i allmänhet ett M-bitars osignerat heltal där M är vilket heltal som helst stort nog för att värdet ska kunna lagras. (Så här fungerar till exempel programmeringsspråket Python.)

(ii) Datorn ändrar värdet på talet till det största som går att lagra (om talet var för stort), eller det minsta som går att lagra (om talet var för litet), utan att ändra antalet bitar. I exemplet ovan skulle resultatet bli $\boxed{1|1|1|1}_2$ det vill säga 15, vilket är det största tal som kan lagras med 4 bitar. (Det här kallas på engelska *clamping* eller *saturation* och används bland annat ofta av grafikkort.)

(iii) Datorn slänger bort de siffror som inte får plats och lagrar alltså bara siffrorna längst till höger i talet (de *minst signifikanta* siffrorna). I exemplet ovan blir resultatet $\boxed{0|1|1|1}_2$ det vill säga 7. Det här är samma sak som *kongruensräkning* modulo 2^N, vilket vi kommer gå in mer på strax. (Det här används bland annat av programmeringsspråket C.)

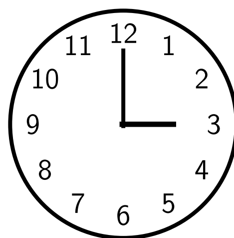
Metod (i), där datorn automatiskt utökar antalet bitar så att talet får plats, har förstås en stor fördel i och med att summan blir vad man förväntar sig. Det finns också en nackdel, nämligen att det är långsammare än att behålla ett fixt antal bitar. Metod (ii) och (iii) är alltså snabbare, och av dessa är (iii), som vi sa är detsamma som kongruensräkning, den vanligaste. Vi ska därför fördjupa oss lite i kongruensräkning, eller moduloräkning som det också kallas.

Definition 2.2.1. Låt a och b vara heltal, och låt n vara ett positivt heltal. Vi säger att a och b är *kongruenta modulo* n om de uppfyller $a = b + kn$ för något heltal k. Om så är fallet skriver vi

$$a \equiv b \pmod{n},$$

vilket utläses "a är kongruent med b modulo n".[5] △

Exempel 2.2.2. Kongruensräkning brukar jämföras med hur en vanlig analog 12-timmarsklocka fungerar. Om klockan är 3:00 på natten står timvisaren på 3, som i Figur 2.1.



Figur 2.1: Analog klocka som visar klockan 3.

Efter 12 timmar är klockan 15:00, men timvisaren står återigen på 3, och klockan ser alltså likadan ut. Detta beror på att $15 = 3 + 1 \cdot 12$, vilket innebär att $15 \equiv 3 \pmod{12}$ enligt Definition 2.2.1. Talen 15 och 3 är alltså kongruenta modulo 12. ▲

[5]Kongruens kan också definieras som att $a \equiv b \pmod{n}$ om a och b har samma rest vid division med n. Detta innebär att $a = pn + r$ och $b = qn + r$, där p och q är heltal och r är ett heltal som uppfyller $0 \leqslant r \leqslant n - 1$. Denna definition är likvärdig med vår definition, vilket visas i Övning 2.7.

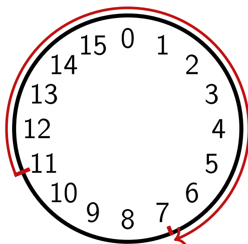
Definition 2.2.3. Låt n vara ett positivt heltal. Givet ett heltal a definierar vi $\text{Mod}_n(a)$ som det tal i mängden $\{0, 1, \dots, n-1\}$ som uppfyller

$$\text{Mod}_n(a) \equiv a \pmod{n}.$$

Notera att Mod_n är en funktion från $\mathbb{Z}$ till mängden $\{0, 1, \dots, n-1\}$. △

Till exempel är $\text{Mod}_{12}(15) = 3$ och $\text{Mod}_{12}(3) = 3$, medan $\text{Mod}_{12}(12) = 0$.

Exempel 2.2.4. Nu återvänder vi till de N-bitars osignerade heltalen. Ett sådant tal kan enligt Sats 2.1.3 anta 2^N olika värden och motsvarar därför en ”klocka” med 2^N timmar. I exemplet med 4 bitar blir det $2^4 = 16$ timmar (från 0 till 15), som i Figur 2.2.



Figur 2.2: En klocka med 16 timmar.

Figuren visar också att om vi som tidigare ska addera talen $1011_2 = 11$ och $1100_2 = 12$ (det vill säga vi startar på klockan 11 och går 12 steg medurs) så hamnar vi på $7 = 0111_2$, samma svar som metod (iii) gav. Detta beror på att $23 = 7 + 1 \cdot 16$, så $23 \equiv 7 \pmod{16}$ och $\text{Mod}_{16}(23) = 7$. ▲

Exempel 2.2.5. Vi kan lagra $255 = 1111\,1111_2$ och $1 = 0000\,0001_2$ som 8-bitars osignerade heltal. Om vi adderar talen får vi $256 = 1\,0000\,0000_2$, vilket inte kan lagras med 8 bitar. Om överflödet hanteras med metod (iii) ska den första ettan i talet slängas bort, och resultatet blir alltså $0000\,0000_2 = 0$. Eftersom $256 = 0 + 1 \cdot 2^8$ så är $\text{Mod}_{2^8}(256) = 0$. ▲

Exempel 2.2.6. Subtraktion av osignerade heltal kan också leda till överflöde. Till exempel blir $11 - 12 = -1$, men negativa värden kan ju inte lagras i ett osignerat heltal. Om metod (ii) används för att hantera överflödet lagras istället det minsta möjliga värdet, nämligen 0.

Om däremot metod (iii) används lagras istället talet $x = \text{Mod}_{2^N}(-1)$, det vill säga det heltal x mellan mellan 0 och $2^N - 1$ som uppfyller $x \equiv -1 \pmod{2^N}$. Enligt Definition 2.2.1 ska x alltså uppfylla

$$x = -1 + k \cdot 2^N.$$

Eftersom dessutom $0 \leqslant x \leqslant 2^N - 1$ måste vi välja $k = 1$, så att $x = 2^N - 1$. Talet x beror alltså på N, antalet bitar. Om $N = 4$ som i Exempel 2.2.4 blir värdet som lagras $x = 15$ (jämför med Figur 2.2). ▲

Ett begrepp som är relaterat till kongruensräkning är delbarhet. Ett sätt att beskriva det är att ett heltal a delar ett annat heltal b när divisionen b/a går jämt ut.

Definition 2.2.7. Ett nollskilt heltal a *delar* ett annat heltal b ifall det finns ett heltal k så att $ak = b$. Man säger också att b *är delbart* med a. Talet k kallas för *kvoten* mellan a och b. △

Man kan formulera detta som att a delar b ifall b ingår i a:s multiplikationstabell. Ett annat sätt att formulera delbarhet är att det nollskilda heltalet a delar heltalet b ifall kvoten $k = b/a$ är ett heltal, eftersom

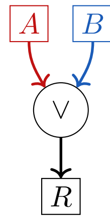
$$ak = a\frac{b}{a} = b.$$

I Övning 2.10 ges ett ytterligare sätt att beskriva delbarhet, den här gången i termer av kongruenser.

Exempel 2.2.8. Talet 5 delar 25 med kvot 5, eftersom $5 \cdot 5 = 25$. Talet 6 delar inte 21, eftersom $21/6 = 7/2$ inte är ett heltal. ▲

2.3 En additionsmaskin för heltal

De Booleska funktionerna $\vee$, $\wedge$, $\neg$ och $\Longrightarrow$ som vi lärde oss om i kapitel 1 kan alla konstrueras som elektroniska komponenter, och kallas då *logiska grindar*. En logisk grind tar binära indatasignaler (som var och en har värdet 0 eller 1) och ger en binär utdatasignal. Till exempel visas i Figur 2.3 en OR-grind som representerar den Booleska funktionen ”eller”.[6]

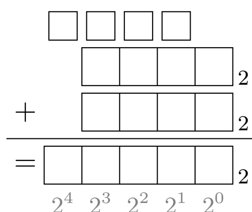


Figur 2.3: En OR-grind, som ger utsignalen $R = A \vee B$.

Den del av datorn som utför heltalsberäkningar kallas ALU efter engelskans *arithmetic logic unit* (ungefär ”aritmetisk-logisk enhet”) och är en del av datorns processor. ALU:n består i princip av en massa sammankopplade logiska grindar, som används för att exempelvis addera eller multiplicera heltal.

Vi ska nu visa hur det är möjligt att med hjälp av logiska grindar bygga en maskin som adderar två heltal. Vi antar att två N-bitars osignerade heltal ska adderas. För att undvika överflöde ska summan ha $N+1$ bitar, vilket illustreras i fallet $N = 4$ i Figur 2.4 (det värsta fallet här är $1111_2 + 1111_2 = 11110_2$, så 5 bitar räcker för summan).

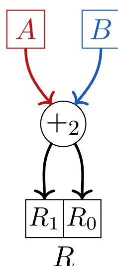
[6]Som Övning 1.19 visade så kan alla Booleska funktioner uttryckas som kombinationer av funktionen NAND. Det räcker alltså att kunna konstruera en elektronisk NAND-grind och koppla ihop flera sådana.



Figur 2.4: Additionsuppställning för addition av 4-bitars osignerade heltal. De mindre rutorna är platser för minnessiffror.

Vi kommer bygga upp additionsmaskinen i tre steg. Först bygger vi maskiner som kan addera en enda kolumn i Figur 2.4, alltså två eller tre bitar åt gången.

Steg 1. Vi bygger först en maskin som kan addera två fristående bitar. Vi kallar denna maskin för $+_2$, och den ska fungera som Figur 2.5 visar, alltså ta två bitar A och B som indata och ge tillbaka ett 2-bitars osignerat heltal R som utdata. Vi kallar bitarna i talet R för R_1 (biten 2^1) och R_0 (biten 2^0).



Figur 2.5: Maskin som adderar två fristående bitar.

Målet är att ta reda på hur maskinen $+_2$ kan byggas upp av logiska grindar. Vi kan använda en variant av en sanningstabell för att se vilka värden R_1 och R_0 ska ha; se Tabell 2.6.

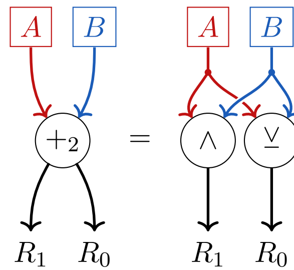
A	B	R	R_1	R_0
0	0	00_2	0	0
1	0	01_2	0	1
0	1	01_2	0	1
1	1	10_2	1	0

Tabell 2.6: Visar vilka värden R_1 och R_0 antar som funktion av A och B.

Som vi ser i Tabell 2.6 ska R_1 vara 1 precis då både A och B är 1. Detta betyder förstås att $R_1 = A \wedge B$. Vi ser också att R_0 ska vara 1 då antingen A eller B är 1, men *inte båda*. Detta stämmer precis in på funktionen XOR ($\veebar$) som vi introducerade i Övning 1.18 och vars sanningstabell ser ut så här:

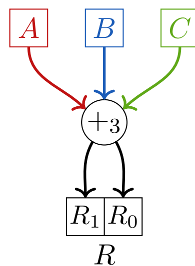
P	Q	$P \veebar Q$
0	0	0
1	0	1
0	1	1
1	1	0

Alltså är $R_0 = A \veebar B$. Vi har därmed kommit fram till att $+_2$-maskinen kan konstrueras med hjälp av en AND-grind och en XOR-grind enligt Figur 2.7.



Figur 2.7: Konstruktion av maskinen $+_2$ med en AND-grind och en XOR-grind.

Steg 2. Nästa steg är att bygga en maskin, som vi kallar för $+_3$, som kan addera *tre* fristående bitar. Vi behöver den tredje biten för att kunna ta med en minnessiffra när vi summerar en kolumn. Figur 2.8 visar hur maskinen $+_3$ ska fungera; den ska ta tre bitar A, B och C som indata och ge tillbaka ett 2-bitars osignerat heltal R som utdata, med bitar R_1 och R_0 som tidigare. Tabell 2.9 visar vilka värden R_1 och R_0 antar i det här fallet.



Figur 2.8: Maskin som adderar tre fristående bitar.

A	B	C	R	R_1	R_0
0	0	0	00_2	0	0
1	0	0	01_2	0	1
0	1	0	01_2	0	1
0	0	1	01_2	0	1
1	1	0	10_2	1	0
1	0	1	10_2	1	0
0	1	1	10_2	1	0
1	1	1	11_2	1	1

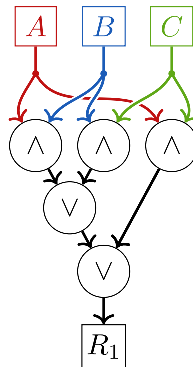
Tabell 2.9: Visar vilka värden R_1 och R_0 antar som funktion av A, B och C.

Tydligen ska R_1 vara 1 precis då två eller fler av indatabitarna är 1. Ett sätt att skriva detta på är $R_1 = (A \wedge B) \vee (B \wedge C) \vee (A \wedge C)$, vilket är sant precis då (minst) två av bitarna är 1. När vi kopplar ihop motsvarande logiska grindar får vi komma ihåg att $P \vee Q \vee R$ egentligen betyder $(P \vee Q) \vee R$ (alternativt

$P \vee (Q \vee R)$; hur parenteserna placeras spelar ingen roll i detta fall), så att

$$R_1 = \Big((A \wedge B) \vee (B \wedge C)\Big) \vee (A \wedge C).$$

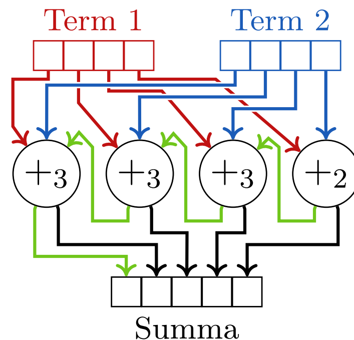
Detta uttryck översatt till logiska grindar ger den maskin som visas i Figur 2.10.



Figur 2.10: En maskin som beräknar den första biten R_1.

Att konstruera en maskin som beräknar den andra biten R_0 i utdata från $+_3$ lämnas som Övning 2.14.

Steg 3. Vi kan nu sätta ihop flera $+_3$-maskiner och en $+_2$-maskin för att genomföra additionen av två 4-bitars osignerade heltal. Kom ihåg additionsuppställningen i Figur 2.4 och hur vi gjorde för att addera tal i Exempel 2.1.2; vi börjar med kolumnen längst till höger och arbetar oss vänsterut medan vi summerar en kolumn i taget. Additionsmaskinen visas i Figur 2.11. Varje kolumn i additionsuppställningen motsvaras av en $+_3$-maskin, förutom den längst till höger som motsvaras av en $+_2$-maskin. De gröna pilarna visar hur minnessiffrorna skickas vidare i maskinen.



Figur 2.11: Den slutliga additionsmaskinen för 4-bitarstal.

2.4 Signerade heltal

Vi har nu sett osignerade heltal, som bara kan vara ickenegativa. Om man vill kunna lagra även negativa tal måste man använda *signerade heltal* (på engelska *signed integers*). I ett signerat heltal lagras även information som berättar om talet är positivt eller negativt. Detta kan göras på olika sätt.

Det enklaste sättet är med en *separat teckenbit* som är 0 för positiva tal och 1 för negativa tal. Det brukar vara biten längst till vänster som används som teckenbit. Ett 8-bitars signerat heltal kan då se ut som

$$\underset{\mathrm{T}}{\boxed{1}}\,\underset{2^6}{\boxed{0}}\underset{2^5}{\boxed{1}}\underset{2^4}{\boxed{1}}\underset{2^3}{\boxed{0}}\underset{2^2}{\boxed{0}}\underset{2^1}{\boxed{1}}\underset{2^0}{\boxed{0}}{}_2 = -110010_2 = -50$$

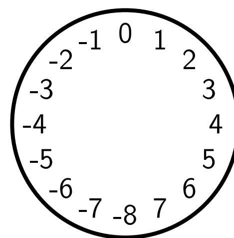
eller

$$\underset{\mathrm{T}}{\boxed{0}}\,\underset{2^6}{\boxed{0}}\underset{2^5}{\boxed{1}}\underset{2^4}{\boxed{1}}\underset{2^3}{\boxed{0}}\underset{2^2}{\boxed{0}}\underset{2^1}{\boxed{1}}\underset{2^0}{\boxed{0}}{}_2 = 110010_2 = 50.$$

Eftersom en bit används som teckenbit är det bara 7 bitar kvar för att lagra talets faktiska värde. En lite märklig egenskap är att både en ”positiv” nolla och en ”negativ” nolla kan lagras, nämligen $\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}{}_2$ och $\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}{}_2$. Naturligtvis är $0 = -0$, så dessa representerar egentligen samma tal. Därmed kan ett N-bitars signerat heltal med separat teckenbit lagra endast $2^N - 1$ genuint olika värden, från $-(2^{N-1} - 1)$ till $2^{N-1} - 1$.

Ett annat sätt att lagra signerade heltal är på så kallad *tvåkomplementsform*, vilket är baserat på kongruensräkning. Alla N bitar används då som för ett vanligt osignerat tal, men biten längst till vänster (den *mest signifikanta* biten) spelar en speciell roll. Om den biten är 0 har talet samma värde som motsvarande osignerade heltal (vilket är ickenegativt). Om den istället är 1 är talets värde $x - 2^N$ (vilket är negativt), där x är motsvarande osignerade heltals värde. Med 4 bitar är till exempel $\boxed{0}\boxed{1}\boxed{0}\boxed{1}{}_2 = 0101_2 = 5$ eftersom första biten är noll, medan $\boxed{1}\boxed{1}\boxed{0}\boxed{1}{}_2 = 1101_2 - 2^4 = 13 - 16 = -3$ eftersom den första biten är ett.

Ett enkelt sätt att visualisera tal på tvåkomplementsform är att tänka sig en klocka med 2^N timmar, där 0 är högst upp, talen på högra sidan är positiva och talen på vänstra sidan är negativa, som i Figur 2.12 (talet längst ner på klockan blir också negativt med den regel vi beskriv tidigare).



Figur 2.12: En klocka med 16 timmar för tvåkomplementsform.

Om vi jämför Figur 2.12 med Figur 2.2 (det vill säga motsvarande klocka för osignerade heltal) så ser vi att -3 och 13 är på samma plats i de två figurerna. Detta beror förstås på att $13 \equiv -3 \pmod{16}$.

Exempel 2.4.1. Som vi såg tidigare lagras talet 50 som $\boxed{0}\,\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{0}{}_2$ och talet -50 som $\boxed{1}\,\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{0}{}_2$ när de lagras som 8-bitars signerade

heltal med separat teckenbit. På tvåkomplementsform blir talet 50 detsamma, det vill säga $\boxed{0|0|1|1|0|0|1|0}_2$. Eftersom $-50 \equiv 206 \pmod{2^8}$ och 206 är $1100\,1110_2$ i bas 2 så lagras talet -50 som $\boxed{1|1|0|0|1|1|1|0}_2$ på tvåkomplementsform. ▲

En fördel med tvåkomplementsform är att talet noll bara kan lagras på ett sätt, till skillnad från metoden med separat teckenbit. Därför kan signerade tal på tvåkomplementsform lagra ett värde mer jämfört med tal med separat teckenbit. En annan fördel är att additionsmaskiner som den vi konstruerade i Figur 2.11 går att använda direkt även med negativa tal (medan om talet lagras med separat teckenbit måste negativa tal hanteras som specialfall).

Exempel 2.4.2. Tabell 2.13 visar vilket värde ett 4-bitarstal har beroende på om det tolkas som ett osignerat heltal, ett signerat heltal med separat teckenbit eller ett signerat heltal på tvåkomplementsform.

Bitar	Osignerat	Signerat[a]	Signerat[b]
0000_2	0	0	0
0001_2	1	1	1
0010_2	2	2	2
0011_2	3	3	3
0100_2	4	4	4
0101_2	5	5	5
0110_2	6	6	6
0111_2	7	7	7
1000_2	8	-0	-8
1001_2	9	-1	-7
1010_2	10	-2	-6
1011_2	11	-3	-5
1100_2	12	-4	-4
1101_2	13	-5	-3
1110_2	14	-6	-2
1111_2	15	-7	-1

[a] Med separat teckenbit
[b] På tvåkomplementsform

Tabell 2.13: Olika sätt att tolka 4 bitar som ett heltal.

▲

Övningar

Övning 2.1. Skriv talet 42 i bas 2.

Övning 2.2. Vilket tal i bas 10 representerar det binära talet 110101_2?

Övning 2.3. Hur många olika värden kan ett 5-bitars osignerat heltal anta?

Övning 2.4. En discgolfare ska programmera en dator att lagra sina resultat. Antalet kast på en runda kan variera från 18 till 180 och ska lagras i ett osignerat heltal. Hur många bitar behöver talet vara?

Övning 2.5. I datorsammanhang används ibland bas 16, med de 16 siffrorna 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E och F (där A = 10, B = 11 och så vidare till F = 15). Tal i bas 16 kallas för *hexadecimala tal*. Till exempel är $\mathrm{FF}_{16} = 15 \cdot 16^1 + 15 \cdot 16^0 = 255$.

(i) Skriv talet 32 i bas 16.

(ii) Vilket tal motsvarar det hexidecimala talet $2\mathrm{B}1_{16}$?

(iii) Skriv talet 60 i bas 16.

(iv) Visa att en siffra i bas 16 motsvarar 4 bitar.

Övning 2.6. Sant eller falskt?

(i) 6 är kongruent med 16 modulo 5.

(ii) 8 är kongruent med 5 modulo 2.

(iii) 120 är kongruent med 184 modulo 32.

Övning 2.7 ($\star$)**.** Låt a och b vara heltal och låt n vara ett positivt heltal. Visa att de två påståendena (i) och (ii) nedan implicerar varandra, det vill säga att (i) $\implies$ (ii) och att (ii) $\implies$ (i).

(i) Det finns heltal p, q och r, där $0 \leqslant r \leqslant n-1$, sådana att $a = pn + r$ och $b = qn + r$.

(ii) Det finns ett heltal k sådant att $a = b + kn$ (vilket ju är vår definition av $a \equiv b \pmod{n}$).

Övning 2.8.

(i) Delar 9 talet 162?

(ii) Delar 7 talet 80?

Övning 2.9. Bevisa att om tre heltal a, b och c är sådana att a delar b och b delar c, så delar a även c.

Övning 2.10. Bevisa att om ett heltal a delar ett heltal b, så är b kongruent med 0 modulo a.

Övning 2.11. Ett polynom P delar ett annat polynom Q ifall det finns ett polynom K så att $PK = Q$. Visa att polynomet $x - 1$ delar polynomet $x^2 - 1$.

Övning 2.12. Konstruera en maskin som givet A och B beräknar biten R i följande tabell.

A	B	R
0	0	0
0	1	0
1	0	1
1	1	0

Maskinen ska vara uppbyggd av sammankopplade logiska grindar, och de grindar som du kan använda är OR ($\vee$), AND ($\wedge$), NOT ($\neg$) samt XOR ($\veebar$).

Övning 2.13 ($\star$)**.** Säg att $A_1, \ldots, A_n$ är bitar, och att A_i är en särskilt utvald av dessa. Låt R_i vara biten som är 1 när $A_i = 1$ och alla andra bitar är 0, och annars 0.

Kan du konstruera en maskin som beräknar R_i? Maskinen ska vara uppbyggd av sammankopplade logiska grindar, och de grindar som du kan använda är OR ($\vee$), AND ($\wedge$), NOT ($\neg$) samt XOR ($\veebar$).

Tips: Lös Övning 2.12 och generalisera resonemanget.

Övning 2.14. Konstruera en maskin som givet A, B och C beräknar biten R_0 i Tabell 2.9. Din maskin ska vara uppbyggd av sammankopplade logiska grindar, och de grindar som du kan använda är OR ($\vee$), AND ($\wedge$), NOT ($\neg$) samt XOR ($\veebar$).

Övning 2.15. Konstruera en maskin som givet A, B och C beräknar biten R i följande tabell.

A	B	C	R
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Maskinen ska vara uppbyggd av sammankopplade logiska grindar, och de grindar som du kan använda är OR ($\vee$), AND ($\wedge$), NOT ($\neg$) samt XOR ($\veebar$).

3 Tal med decimaler

I det förra kapitlet såg vi hur datorer kan räkna med heltal. I många sammanhang är talen man arbetar med inte heltal, utan tal med decimaler (det vill säga reella tal, eller rationella tal). Tal med decimaler kan vara extra knepiga, vilket följande exempel visar.

Exempel 3.0.1. Om man i programmeringsspråket Python skriver[7]

```
0.1 + 0.2
```

får man inte exakt 0.3 som svar, utan

```
0.30000000000000004.
```

Om man vidare skriver

```
0.1 + 0.2 - 0.3
```

blir svaret inte 0 utan

```
5.551115123125783e-17,
```

vilket ska tolkas som $5.551\,115\,123\,125\,783 \cdot 10^{-17}$. ▲

För att förstå varför det blir så här måste vi lära oss om hur datorer lagrar tal med decimaler, vilket är temat för nästa avsnitt.

3.1 Binärbråk och flyttal

När vi skriver ett tal med decimaler i bas 10 kan vi tolka siffrorna till höger om decimaltecknet med hjälp av tiopotenser med negativa exponenter. Talet 42.15 betyder till exempel

$$\begin{aligned} 42.15 &= 4 \cdot 10^1 + 2 \cdot 10^0 + 1 \cdot 10^{-1} + 5 \cdot 10^{-2} \\ &= 4 \cdot 10 + 2 \cdot 1 + 1 \cdot 0.1 + 5 \cdot 0.01. \end{aligned}$$

I bas 2 gör vi på samma sätt. Talet 101.01_2 betyder alltså

$$\begin{aligned} 101.01_2 &= 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} \\ &= 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 + 0 \cdot 0.5 + 1 \cdot 0.25 \\ &= 4 + 1 + 0.25 = 5.25. \end{aligned} \tag{3.1}$$

Punkten[8] som vi använder för att separera heltalsdelen från bråkdelen i ett tal kallas för *binärpunkt* i bas 2 och *decimalpunkt* i bas 10.

[7]Python är ett populärt programmeringsspråk som kan laddas ner kostnadsfritt. Det går även att prova Python direkt i webbläsaren på adressen `https://www.python.org/shell/`. Exemplet är inte unikt för Python, utan liknande saker kommer inträffa i de flesta andra programmeringsspråk.

[8]På svenska brukar man använda kommatecken, medan punkt är vanligast på engelska. I detta kompendium använder vi punkt eftersom det är vanligast i datorsammanhang och i de flesta programmeringsspråk.

Rationella tal som skrivs i bas 2 med binärpunkt kallas för *binärbråk*. Talet 101.01_2 är alltså ett exempel på ett binärbråk. Det är lätt att omvandla binärbråk till decimaltal (i bas 10), på samma sätt som vi gjorde i ekvation (3.1). Binärbråk med ett ändligt antal siffor motsvaras alltid av decimaltal med ett ändligt antal decimaler, vilket följer ur Övning 3.4. Det omvända gäller däremot *inte*, vilket visas av följande exempel.

Exempel 3.1.1. Omvandla decimaltalet 0.1 (en tiondel) till bas 2.

Talet 0.1 kan skrivas som bråket $1/10$. Vi kan omvandla täljaren och nämnaren till bas 2 och utföra divisionen, till exempel med liggande stolen (i bas 2). Bråket skrivs om till

$$\frac{1}{10} = \frac{1_2}{1010_2}.$$

Om läsaren inte kommer ihåg liggande stolen kommer här en snabb repetition.

Steg 0. Vi skriver upp täljaren och nämnaren i uppställningen för liggande stolen. Kvoten (i bas 2) kommer växa fram på raden ovanför täljaren.

$$\begin{array}{l|l} \hline 1_2 & 1010_2 \\ \cline{2-2} \end{array}$$

Steg 1. För att få fram entalssiffran i kvoten ställer vi oss frågan: Hur många gånger går $1010_2 = 10$ i $1_2 = 1$? Svaret är 0, vilket är entalssiffran. Vi skriver upp detta ovanför täljaren, räknar ut resten ($1_2 - 0 \cdot 1010_2 = 1_2$) och flyttar ner en nolla från täljaren.

$$\begin{array}{l|l} 0_2 & \\ \hline 1.0_2 & 1010_2 \\ \cline{2-2} -0_2 & \\ \hline \;10_2 & \end{array}$$

Steg 2. För att få nästa siffra frågar vi: Hur många gånger går $1010_2 = 10$ i $10_2 = 2$? Återigen är svaret 0, så vi skriver upp det och räknar ut resten ($10_2 - 0 \cdot 1010_2 = 10_2$) och flyttar ner en till nolla från täljaren.

$$\begin{array}{l|l} 0.0_2 & \\ \hline 1.00_2 & 1010_2 \\ \cline{2-2} -0_2 & \\ \hline \;10_2 & \\ -0_2 & \\ \hline \;100_2 & \end{array}$$

Steg 3. Hur många gånger går $1010_2 = 10$ i $100_2 = 4$? Svaret är 0 igen. Vi hoppar nu över några steg som fortsätter på samma sätt, och kommer då till läget som visas i uppställningen på nästa sida.

$$
\begin{array}{l|l}
0.000_2 & \\
\hline
1.0000_2 & 1010_2 \\
\hline
-0_2 & \\
\hline
10_2 & \\
-0_2 & \\
\hline
100_2 & \\
-0_2 & \\
\hline
1000_2 & \\
-0_2 & \\
\hline
10000_2 &
\end{array}
$$

Steg 4. Nu ska vi räkna ut den fjärde siffran efter binärpunkten, och vi ställer oss frågan: Hur många gånger går $1010_2 = 10$ i $10000_2 = 16$? Svaret är 1, vilket vi skriver upp som den fjärde siffran. Resten är $6 = 110_2$, vilket vi skriver upp längst ner, och flyttar ner en nolla från täljaren.

$$
\begin{array}{l|l}
0.0001_2 & \\
\hline
1.00000_2 & 1010_2 \\
\hline
-0_2 & \\
\hline
10_2 & \\
-0_2 & \\
\hline
100_2 & \\
-0_2 & \\
\hline
1000_2 & \\
-0_2 & \\
\hline
10000_2 & \\
-1010_2 & \\
\hline
1100_2 &
\end{array}
$$

Steg 5. Hur många gånger går $1010_2 = 10$ i $1100_2 = 12$? Svaret är 1, vilket vi skriver upp som den femte siffran. Resten är $2 = 10_2$, vilket vi skriver upp längst ner, och så flyttar vi ner en till nolla från täljaren.

$$
\begin{array}{l|l}
0.00011_2 & \\
\hline
1.000000_2 & 1010_2 \\
\hline
-0_2 & \\
\hline
10_2 & \\
-0_2 & \\
\hline
100_2 & \\
-0_2 & \\
\hline
1000_2 & \\
-0_2 & \\
\hline
10000_2 & \\
-1010_2 & \\
\hline
1100_2 & \\
-1010_2 & \\
\hline
100_2 &
\end{array}
$$

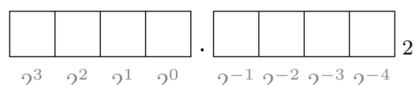
Nu har vi fått ett läge där resten är samma som i ett tidigare steg, nämligen i steget där vi räknade ut den andra siffran efter binärpunkten (vilket visas med

den blå pilen). Detta betyder att fortsättningen kommer vara en upprepning av vad som skedde mellan dessa steg (om du är tveksam, prova att göra några fler steg!). Kvoten blir alltså

$$0.000110011001100110011001100110011\ldots_2 = 0.0\overline{0011}_2,$$

där ett streck över siffror betyder att de upprepas i all oändlighet. Sammanfattningsvis har vi visat att $0.1 = 0.0\overline{0011}_2$. Observera att det senare talet har oändligt många siffror efter binärpunkten. ▲

Fixtal och flyttal. Det enklaste sättet att lagra binärbråk på är att ha ett bestämt antal bitar innan binärpunkten och ett bestämt antal bitar efter, till exempel 4 + 4 bitar som i Figur 3.1.



Figur 3.1: Ett 4 + 4-bitars ickenegativt fixtal.

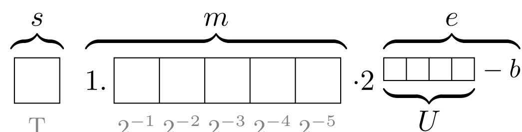
Tal som lagras på detta sätt kallas för *fixtal* (på engelska *fixed-point numbers*). Ett mer flexibelt sätt att lagra binärbråk på är *flyttal* (på engelska *floating-point numbers*), vilket också är det vanligaste. Ett flyttal är på formen

$$s \cdot m \cdot 2^e,$$

och består alltså av de tre delarna s, m och e, med följande betydelse:

- Talet s är flyttalets *tecken* (+1 eller −1) och lagras med 1 bit, där värdet 0 betyder positivt tecken och värdet 1 negativt tecken (precis som för signerade heltal med separat teckenbit).
- Talet m kallas *mantissa* (eller ibland *signifikand*) och är ett rationellt tal som uppfyller $1 \leqslant m < 2$. Detta tal visar vilka siffror flyttalet innehåller (i bas 2).
- Talet e kallas *exponent* och är ett heltal. Detta tal visar flyttalets storleksordning (i bas 2).

Både m och e lagras med ett bestämt antal bitar och kan alltså anta ett begränsat antal värden. Ett exempel med 5 bitar för mantissan och 4 bitar för exponenten visas i Figur 3.2 (vi förklarar strax vad U och b är).



Figur 3.2: Lagring av ett flyttal med 1 bit för tecknet s, 5 bitar för mantissan m och 4 bitar för exponenten e, alltså totalt 10 bitar.

Eftersom mantissan uppfyller $1 \leqslant m < 2$ kommer heltalsdelen av m alltid vara 1, så den behöver inte lagras (därför har ettan innan binärpunkten ingen egen låda i Figur 3.2).

Exponenten e lagras på ett speciellt sätt, nämligen som ett osignerat (alltså ickenegativt) heltal U som en konstant b subtraheras ifrån (så $e = U - b$).[9] Vi kommer i fortsättningen för det mesta bortse från hur exponenten lagras.

Exempel 3.1.2. Låt oss välja 5 bitar för mantissan som i Figur 3.2.

- Talet 1 kan skrivas som $1.00000_2 \cdot 2^0$ och lagras alltså genom att sätta mantissan till $m = 1.00000_2$ och exponenten till $e = 0$; tecknet är $s = +1$ vilket lagras genom att sätta teckenbiten till 0.
- Talet 3 kan skrivas som $11_2 = 1.10000_2 \cdot 2^1$ och lagras alltså genom att sätta mantissan $m = 1.10000_2$ och exponenten $e = 1$; teckenbiten är fortfarande 0.
- Talet -0.75 skrivs i bas 2 som $-0.11_2 = -1.10000_2 \cdot 2^{-1}$ och lagras alltså genom att sätta $m = 1.10000_2$ och $e = -1$, medan teckenbiten sätts till 1 eftersom tecknet är negativt.
- Som vi visade i Exempel 3.1.1 blir talet 0.1 (en tiondel) i bas 2

$$0.0001\,1001\,1001\,1001\,1001\,1001\,1001\,1\ldots_2 = 0.0\overline{0011}_2,$$

där strecket över siffrorna betyder att de upprepas i all oändlighet. Detta tal har oändligt många bitar och kan alltså inte lagras exakt som ett flyttal (eller som ett fixtal för den delen). Talet kan skrivas som

$$1.1001\,1001\,1001\,1001\,1001\,1001\,1001\,1\ldots_2 \cdot 2^{-4},$$

vilket visar att exponenten ska vara $e = -4$, medan mantissan måste avrundas till $m = 1.10011_2$ med fem bitar efter binärpunken. Talet 0.1 lagras alltså som

$$1.10011_2 \cdot 2^{-4},$$

vilket egentligen är lika med 0.099609375 i bas 10. Detta visar att tal som inte kan lagras exakt som flyttal måste avrundas (i bas 2), vilket leder till *avrundningsfel*. ▲

Alla tal som kan lagras exakt som flyttal är rationella, vilket visas i Övning 3.9. Däremot kan inte *alla* rationella tal lagras exakt, som vi såg i exemplet ovan. Oavsett hur många bitar som används för att lagra ett flyttal är det bara ett ändligt antal olika värden som kan lagras (men det finns ju oändligt många rationella tal). Detta betyder att flyttal har en begränsad *precision*; det finns en gräns för hur liten skillnaden mellan två flyttal kan vara. Denna gräns uttrycks med hjälp av *maskinepsilon* (som ibland kallas *maskinprecision*), som definieras enligt nedan.

[9] Konstanten b brukar väljas som $b = M/2 - 1$, där M är antalet olika värden som talet U kan anta ($M = 2^N$ om N bitar används för att lagra U). Detta val innebär att $-b \leqslant e \leqslant b+1$. Om till exempel $N = 4$ så är $b = 7$, så $-7 \leqslant e \leqslant 8$.

Definition 3.1.3. Givet en flyttalstyp så definieras *maskinepsilon* $\varepsilon_{\text{mach}}$ som

$$\varepsilon_{\text{mach}} = s - 1,$$

där s är det minsta tal större än 1 som kan lagras med flyttalstypen. △

Med en ”flyttalstyp” menar vi helt enkelt att man har valt hur många bitar som används för att lagra mantissan och exponenten. Maskinepsilon beror alltså på hur många bitar som används. Följande sats preciserar detta.

Sats 3.1.4. *Om N_m bitar används för att lagra mantissan i ett flyttal (exklusive ettan till vänster om binärpunkten) så är maskinepsilon $\varepsilon_{\text{mach}} = 2^{-N_m}$.*

Bevis. Talet 1 lagras med mantissan $1.000\ldots000_2$ (med N_m nollor efter binärpunkten) och exponenten 0. Det minsta talet större än 1 som kan lagras får vi genom att öka den minst signifikanta biten i mantissan (biten längst till höger), vilket alltså blir $s = 1.000\ldots001_2$ (exponenten är fortfarande 0). Skillnaden är $s-1 = 0.000\ldots001_2$, alltså en etta i bit nummer N_m efter binärpunkten. Denna bit motsvarar 2^{-N_m}, vilket innebär att $\varepsilon_{\text{mach}} = s - 1 = 2^{-N_m}$. □

Ursprungligen var antalet bitar som användes för att lagra mantissan och exponenten en egenskap hos själva datorn, så varje dator hade en bestämd flyttalstyp som inte gick att ändra. Maskinepsilon berodde alltså på datorn (”maskinen”), vilket förklarar dess namn. Nu för tiden kan en dator använda olika flyttalstyper. Det finns en standard för flyttal[10] som bland annat definierar följande flyttalstyper:

- Typen *binary32* (även kallad *single-precision*) använder 23 bitar för att lagra mantissan och 8 bitar för att lagra exponenten (alltså 32 bitar totalt inklusive teckenbiten).

- Typen *binary64* (även kallad *double-precision*) använder 52 bitar för mantissan och 11 bitar för exponenten (alltså 64 bitar totalt inklusive teckenbiten).

Exempel 3.1.5. För flyttal av typen *binary32* är maskinepsilon

$$\varepsilon_{\text{mach}} = 2^{-23} \approx 1.192093 \cdot 10^{-7}.$$

För flyttal av typen *binary64* är maskinepsilon istället

$$\varepsilon_{\text{mach}} = 2^{-52} \approx 2.220446 \cdot 10^{-16}.$$

För flyttal där mantissan lagras med 5 bitar (som i Exempel 3.1.2) är maskinepsilon

$$\varepsilon_{\text{mach}} = 2^{-5} = 0.03125.$$ ▲

[10]Namnet på standarden är IEEE 754.

Maskinepsilon är ett mått på hur mycket två flyttal som minst kan skilja sig åt. Därmed är det också ett ungefärligt mått på hur stora avrundningsfelen kan bli. I slutet av Exempel 3.1.2 såg vi att om 5 bitar används för mantissan så avrundas talet 0.1 till $1.10011_2 \cdot 2^{-4}$, medan det exakta värdet är

$$0.1 = 1.1001\,1001\,1001\,1001\ldots_2 \cdot 2^{-4}.$$

Felet i mantissan är alltså

$$1.1001\,1001\,1001\,1001\ldots_2 - 1.1001\,1_2 = 0.0000\,0001\,1001\,1001\ldots_2 = 0.00625,$$

vilket är mindre än maskinepsilon 0.03125.

Nu är det dags att analysera vad som egentligen hände i Exempel 3.0.1. I exemplet användes flyttal av typen *binary64*, eftersom det är vad Python använder som standard. Vi kommer nu gå igenom samma beräkning för hand, fast med 20 bitar för mantissan istället för 52 (för att inte behöva skriva ut så många siffror).

Exempel 3.1.6. Låt oss först notera att $0.1 = 0.0\overline{0011}_2$ och $0.2 = 0.\overline{0011}_2$ (att multiplicera med 2 innebär att binärpunkten flyttas ett steg till höger). På flyttalsform blir talen alltså

$$\begin{aligned} 0.1 &= 1.1001\,1001\,1001\,1001\,1001\,1001\ldots_2 \cdot 2^{-4}, \\ 0.2 &= 1.1001\,1001\,1001\,1001\,1001\,1001\ldots_2 \cdot 2^{-3}. \end{aligned}$$

Talen har samma mantissa men olika exponenter. Mantissan måste avrundas till 20 bitar efter binärpunkten, vilket innebär att den (korrekt avrundad) blir

$$m = 1.1001\,1001\,1001\,1001\,1010_2.$$

För att addera två tal måste de först skrivas om så att de har samma exponent. Vi skriver om talet till exponenten -3 och får då

$$\begin{aligned} 0.1 &\approx 0.1100\,1100\,1100\,1100\,1101\,0_2 \cdot 2^{-3}, \\ 0.2 &\approx 1.1001\,1001\,1001\,1001\,1010_2 \cdot 2^{-3}. \end{aligned}$$

(i detta läge avrundar vi inte mantissan till talet 0.1 igen). Om vi nu adderar de två talen får vi

$$10.0110\,0110\,0110\,0110\,0111\,0_2 \cdot 2^{-3}$$

Just nu är detta ett tal med 21 bitar efter binärpunkten, men det måste skrivas om så att mantissan är mindre än 2, alltså

$$1.0011\,0011\,0011\,0011\,0011\,10_2 \cdot 2^{-2}$$

med 22 bitar efter binärpunkten. Nu måste mantissan avrundas till 20 bitar så att talet kan lagras med den valda flyttalstypen. Resultatet blir

$$1.0011\,0011\,0011\,0011\,0100_2 \cdot 2^{-2} \tag{3.2}$$

vilket är lika med 0.30000019... i bas 10. På grund av avrundningsfel blir resultatet inte lika med 0.3. Avrundningen har skett i två steg:

- Talen 0.1 och 0.2 behövde avrundas bara för att kunna lagras som flyttal.
- Efter additionen behövde summan också avrundas för att kunna lagras.

Vad händer när vi sedan försöker subtrahera 0.3 då? Jo, datorn måste nu lagra talet $0.3 = 0.01\overline{0011}_2$ och kommer först skriva talet på formen

$$0.3 = 1.0011\,0011\,0011\,0011\,0011\,0011\ldots_2 \cdot 2^{-2}$$

och därefter avrunda mantissan till 20 bitar efter binärpunkten. Talet som lagras är

$$1.0011\,0011\,0011\,0011\,0011_2 \cdot 2^{-2}. \tag{3.3}$$

Talen som visas i (3.2) och (3.3) är inte lika med varandra! Detta beror på att avrundningarna har skett på olika sätt. (Talet i (3.2) är 0.30000019... men talet i (3.3) är 0.29999995... vilket är lite närmare 0.3.) När talen subtraheras blir resultatet

$$0.0000\,0000\,0000\,0000\,0001_2 \cdot 2^{-2},$$

vilket är $2^{-22} \approx 2.384 \cdot 10^{-7}$. Observera att detta värde är maskinepsilon $\varepsilon_{\text{mach}}$ multiplicerat med 2^{-2}. Precis samma sak händer i Exempel 3.0.1 fast med 52 bitar för mantissan ($2^{-52} \cdot 2^{-2} = 2^{-54} \approx 5.551\,115 \cdot 10^{-17}$). ▲

Den uppmärksamma läsaren kanske har noterat att det inte går att lagra talet 0 i ett flyttal på det sätt som vi beskrivit hittills, eftersom mantissan alltid är positiv. För att komma runt detta har man bestämt att det minsta möjliga värdet för exponenten reserveras med en speciell betydelse för att kunna lagra talet 0. På samma sätt reserveras det största möjliga värdet för exponenten för att kunna lagra oändligheten (∞), som ju egentligen inte är ett tal.

3.2 Ekvationslösning med intervallhalvering

Hittills har vi sett hur en dator kan lagra olika typer av tal och addera dem, till exempel med additionsmaskinen från avsnitt 2.3. Liknande (mer komplicerade) maskiner kan byggas för att multiplicera tal. Därmed går det även att beräkna potenser a^n, där a är ett flyttal och n är ett positivt heltal, med hjälp av upprepad multiplikation. För att utföra ännu mer komplicerade beräkningar kan man använda *ekvationslösning*. En *ekvation* med en obekant x är en likhet mellan två funktioner av x, alltså

$$g(x) = h(x), \tag{3.4}$$

där g och h är funktioner med de reella talen $\mathbb{R}$ både som definitionsmängd och målmängd (i detta kompendium begränsar vi oss till fallet $x \in \mathbb{R}$). Ett tal $x \in \mathbb{R}$ som uppfyller (3.4) kallas en *lösning* till ekvationen. En ekvation som (3.4) kan alltid skrivas om på formen

$$f(x) = 0 \tag{3.5}$$

(till exempel genom att subtrahera $h(x)$ från ekvation (3.4) och definiera $f(x) = g(x) - h(x)$). Ett tal $x \in \mathbb{R}$ som uppfyller (3.5) kallas ett *nollställe* till funktionen f. Att lösa en ekvation är alltså detsamma som att hitta ett nollställe till en funktion.

Exempel 3.2.1. Antag att vi vill räkna ut $x = \sqrt{2}$. Genom att kvadrera x får vi ekvationen

$$x^2 = 2. \tag{3.6}$$

Denna ekvation har två lösningar, nämligen $\sqrt{2}$ och $-\sqrt{2}$. Vi kan subtrahera 2 och därmed få ekvationen

$$x^2 - 2 = 0.$$

Denna ekvation är på formen $f(x) = 0$, där funktionen $f : \mathbb{R} \to \mathbb{R}$ i detta fall ges av $f(x) = x^2 - 2$. Eftersom lösningarna till (3.6) är desamma som nollställena till f så vet vi att funktionen f har precis två nollställen. Om vi kan bestämma det positiva nollstället till f så har vi alltså räknat ut $\sqrt{2}$. ▲

För att noggrannt bestämma nollställen till en funktion $f : \mathbb{R} \to \mathbb{R}$ kan vi använda en metod som kallas *intervallhalvering* (eller ibland *bisektionsmetoden*). Metoden går ut på att hitta ett intervall som man vet att det finns ett nollställe i, och sedan förfina det. Vi ska börja med att ge en ordentlig definition av ordet intervall.

Definition 3.2.2. Ett *intervall* är en mängd på formen

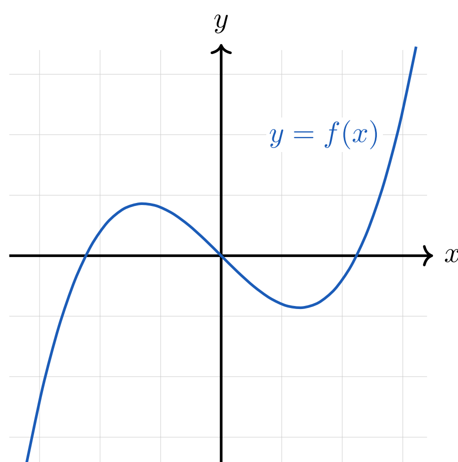
$$\{x \in \mathbb{R} \mid a \leqslant x \leqslant b\},$$

där a och b är reella tal som uppfyller $a \leqslant b$. Intervallet $\{x \in \mathbb{R} \mid a \leqslant x \leqslant b\}$ kan även betecknas $[a, b]$. △

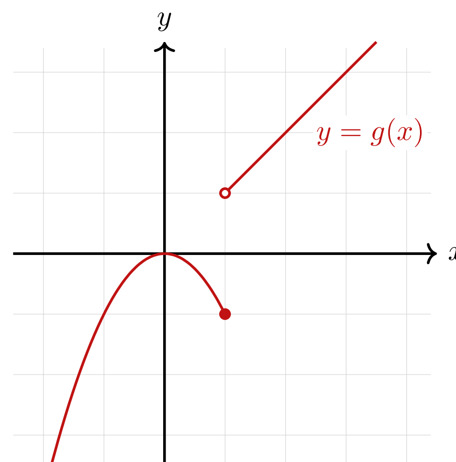
Intervallhalveringsmetoden fungerar bara om f är en *kontinuerlig* funktion. Enkelt uttryckt är en kontinuerlig funktion en funktion vars graf är sammanhängande och alltså kan ritas på ett papper utan att lyfta på pennan. Ett exempel på en kontinuerlig funktion är $f(x) = \frac{1}{5}x^3 - x$ som visas i Figur 3.3. En funktion som *inte* är kontinuerlig har hopp i sin graf, som funktionen

$$g(x) = \begin{cases} -x^2 & \text{om } x \leqslant 1, \\ x & \text{om } x > 1, \end{cases}$$

som visas i Figur 3.4.



Figur 3.3: Grafen till den kontinuerliga funktionen $f(x) = \frac{1}{5}x^3 - x$.



Figur 3.4: Grafen till den ej kontinuerliga funktionen $g(x)$.

En matematisk definition ska helst inte hänvisa till godtyckliga föremål som papper och pennor, som ju kan se ut på olika sätt, såsom vi gjorde ovan. Målet med en definition är snarare att skala bort allt onödigt, och komma fram till själva kärnan i det man vill definiera.[11] För att ordentligt definiera vad en kontinuerlig funktion är måste vi ställa oss frågan: Vad betyder det *egentligen* att grafen till en funktion är sammanhängande?

En vanlig definition av kontinuitet utgår från att då två punkter på x-axeln närmar sig varandra så måste även de motsvarande funktionsvärdena närma sig varandra.

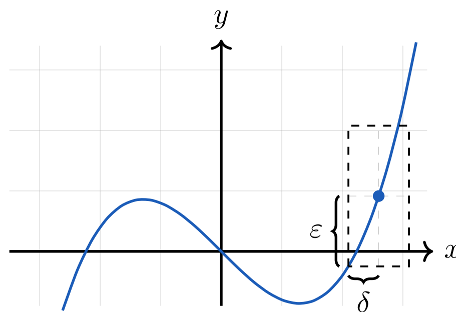
Definition 3.2.3. Låt f vara en funktion från $\mathbb{R}$ till $\mathbb{R}$, och låt $x \in \mathbb{R}$ vara ett godtyckligt reellt tal. Funktionen f kallas *kontinuerlig i punkten* x om skillnaden mellan $f(y)$ och $f(x)$ kan bli hur liten som helst genom att välja talet y tillräckligt nära talet x.

Annorlunda uttryckt: f kallas *kontinuerlig i punkten* x om det för varje tal $\varepsilon > 0$ finns ett tal $\delta > 0$ sådant att $|f(x) - f(y)| < \varepsilon$ då $|x - y| < \delta$. △

Här är $|a|$ *absolutbeloppet* av det reella talet a, det vill säga talets värde bortsett från dess tecken (om a är negativt är $|a| = -a$, annars är $|a| = a$).

Definition 3.2.4. En funktion $f : \mathbb{R} \to \mathbb{R}$ kallas *kontinuerlig* om f är kontinuerlig i punkten x för varje reellt tal x. △

En funktion är alltså kontinuerlig (överallt) om den är kontinuerlig i alla punkter. Talen ε och δ i Definition 3.2.3 kan tolkas grafiskt som i Figur 3.5.



Figur 3.5: Kurvans y-värde håller sig inom avståndet ε från den blå punkten om x-värdet håller sig inom avståndet δ.

För kontinuerliga funktioner gäller följande sats, som även kallas Bolzanos sats.

Sats 3.2.5 (Satsen om mellanliggande värden)**.** *Låt $f : \mathbb{R} \to \mathbb{R}$ vara en kontinuerlig funktion, och låt a och b vara reella tal sådana att $a \leqslant b$. Om det reella talet c ligger mellan $f(a)$ och $f(b)$ så finns ett $x \in [a, b]$ sådant att $f(x) = c$.*

[11] Man skulle kunna säga att det ligger i matematikens natur att utifrån olika konkreta exempel försöka hitta underliggande allmängiltiga principer, och därigenom gå från det konkreta till det abstrakta. De allmänna principerna kan därefter tillämpas på en mängd olika konkreta fall.

Satsen säger alltså att f antar alla värden mellan $f(a)$ och $f(b)$ på intervallet $[a, b]$. En följd av satsen är att om $f(a)$ och $f(b)$ har olika tecken (det ena talet är negativt och det andra positivt) så är $f(x) = 0$ för något $x \in [a, b]$. Alltså finns det ett nollställe till f i intervallet $[a, b]$.

Vi väntar med att bevisa Sats 3.2.5 till avsnitt 3.3. Nu är vi istället redo att beskriva intervallhalveringsmetoden ordentligt. Den går ut på att hitta a och b så att $f(a)$ och $f(b)$ har olika tecken och därefter halvera intervallet upprepade gånger. Vi beskriver metoden som en *algoritm*, det vill säga en följd av instruktioner.

Algoritm 3.2.6 (Intervallhalvering)**.** Låt en kontinuerlig funktion $f : \mathbb{R} \to \mathbb{R}$ vara given, samt två reella tal a och b där $a \leqslant b$ och $f(a) \cdot f(b) < 0$.

1. Sätt $v_1 = a$ och $h_1 = b$. Beräkna mittpunkten $m_1 = (v_1 + h_1)/2$.
2. För $i = 1, 2, 3, \ldots$, gör följande:
   (a) Om $f(v_i) \cdot f(m_i) < 0$, låt $v_{i+1} = v_i$ och $h_{i+1} = m_i$. Gå till (b).
   Om $f(m_i) \cdot f(h_i) < 0$, låt $v_{i+1} = m_i$ och $h_{i+1} = h_i$. Gå till (b).
   Om $f(m_i) = 0$ har vi hittat ett nollställe $x = m_i$. Avbryt.
   (b) Beräkna den nya mittpunkten $m_{i+1} = (v_{i+1} + h_{i+1})/2$.
   Fortsätt sedan till nästa värde på i.

Notera att $f(a) \cdot f(b) < 0$ är ett kompakt sätt att skriva att $f(a)$ och $f(b)$ har olika tecken. I varje steg är vi garanterade att intervallet $[v_i, h_i]$ innehåller ett nollställe till f, eftersom vi i förväg har kontrollerat att $f(v_i)$ och $f(h_i)$ har olika tecken.

Algoritm 3.2.6 fortsätter i all oändlighet om inte mittpunkten någon gång hamnar precis på ett nollställe till f. I praktiken får man avbryta algoritmen när intervallets längd $L_i = h_i - v_i$ har blivit tillräckligt liten. Följande resultat visar hur stort felet kan vara om vi approximerar nollstället med m_i.

Sats 3.2.7. *Låt v_i, m_i och h_i vara som i Algoritm 3.2.6, och låt $L_i = h_i - v_i$. Definiera felet $e_i = |m_i - x|$, där $x \in [v_i, h_i]$ är ett nollställe till f. Då är*

$$e_i \leqslant \frac{L_i}{2}, \qquad i = 1, 2, \ldots \tag{3.7}$$

Om $L = L_1$ så är

$$e_i \leqslant \frac{L}{2^i}, \qquad i = 1, 2, \ldots \tag{3.8}$$

Bevis. Vi vet i varje steg att det finns (minst) ett nollställe $x \in [v_i, h_i]$. Om det finns flera nollställen väljer vi ett av dem som x. Eftersom x ligger i intervallet $[v_i, h_i]$ och m_i är mittpunkten i intervallet så kan x och m_i som mest skilja sig med halva intervallets längd. Detta innebär att

$$e_i = |m_i - x| \leqslant \frac{h_i - v_i}{2} = \frac{L_i}{2},$$

vilket bevisar olikheten (3.7).

För att bevisa olikheten (3.8) räcker det att visa att $L_i = L/2^{i-1}$ för alla positiva heltal i. Detta kan bevisas med hjälp av induktion. För basfallet $i = 1$ blir påståendet $L_1 = L/2^0 = L$, vilket är definitionen av L. Om vi nu antar att

$$L_k = \frac{L}{2^{k-1}}$$

gäller för något positivt heltal k så vet vi att i nästa steg halveras intervallets längd, och därmed är

$$L_{k+1} = \frac{L_k}{2} = \frac{L}{2^{k-1} \cdot 2} = \frac{L}{2^k},$$

vilket visar att likheten gäller även för $k+1$. Därmed har vi visat att $L_i = L/2^{i-1}$ för alla positiva heltal i. Insättning av denna likhet i (3.7) ger (3.8). □

Exempel 3.2.8. För att beräkna $\sqrt{2}$ vill vi använda intervallhalvering för att hitta det positiva nollstället till funktionen $f(x) = x^2 - 2$ (från Exempel 3.2.1). Denna funktion är kontinuerlig, vilket visas i Övning 3.13. Vi väljer $a = 0$ och $b = 2$ och kontrollerar att $f(0) = -2$ och $f(2) = 2$ har olika tecken. Nu använder vi metoden (i tabellen nedan avrundar vi alla tal till 6 decimaler).

i	v_i	h_i	m_i	$f(v_i)$	$f(h_i)$	$f(m_i)$	$L_i/2$
1	0	2	1	-2	2	-1	1
2	1	2	1.5	-1	2	0.25	0.5
3	1	1.5	1.25	-1	0.25	-0.4375	0.25
4	1.25	1.5	1.375	-0.4375	0.25	-0.109375	0.125
5	1.375	1.5	1.4375	-0.109375	0.25	0.066406	0.0625
6	1.375	1.4375	1.40625	-0.109375	0.066406	-0.022461	0.03125
7	1.40625	1.4375	1.421875	-0.022461	0.066406	0.021729	0.015625
8	1.40625	1.421875	1.414062	-0.022461	0.021729	-0.000427	0.007812
9	1.414062	1.421875	1.417969	-0.000427	0.021729	0.010635	0.003906
10	1.414062	1.417969	1.416016	-0.000427	0.010635	0.0051	0.001953
11	1.414062	1.416016	1.415039	-0.000427	0.0051	0.002336	0.000977
12	1.414062	1.415039	1.414551	-0.000427	0.002336	0.000954	0.000488
13	1.414062	1.414551	1.414307	-0.000427	0.000954	0.000263	0.000244
14	1.414062	1.414307	1.414185	-0.000427	0.000263	$-8.2 \cdot 10^{-5}$	0.000122
15	1.414185	1.414307	1.414246	$-8.2 \cdot 10^{-5}$	0.000263	$9.1 \cdot 10^{-5}$	$6.1 \cdot 10^{-5}$
16	1.414185	1.414246	1.414215	$-8.2 \cdot 10^{-5}$	$9.1 \cdot 10^{-5}$	$4.3 \cdot 10^{-6}$	$3.1 \cdot 10^{-5}$
17	1.414185	1.414215	1.4142	$-8.2 \cdot 10^{-5}$	$4.3 \cdot 10^{-6}$	$-3.9 \cdot 10^{-5}$	$1.5 \cdot 10^{-5}$
18	1.4142	1.414215	1.414207	$-3.9 \cdot 10^{-5}$	$4.3 \cdot 10^{-6}$	$-1.7 \cdot 10^{-5}$	$7.6 \cdot 10^{-6}$
19	1.414207	1.414215	1.414211	$-1.7 \cdot 10^{-5}$	$4.3 \cdot 10^{-6}$	$-6.5 \cdot 10^{-6}$	$3.8 \cdot 10^{-6}$
20	1.414211	1.414215	1.414213	$-6.5 \cdot 10^{-6}$	$4.3 \cdot 10^{-6}$	$-1.1 \cdot 10^{-6}$	$1.9 \cdot 10^{-6}$
21	1.414213	1.414215	1.414214	$-1.1 \cdot 10^{-6}$	$4.3 \cdot 10^{-6}$	$1.6 \cdot 10^{-6}$	$9.5 \cdot 10^{-7}$
22	1.414213	1.414214	1.414214	$-1.1 \cdot 10^{-6}$	$1.6 \cdot 10^{-6}$	$2.7 \cdot 10^{-7}$	$4.8 \cdot 10^{-7}$

Efter 22 steg är mittpunkten $m_i \approx 1.414214$, medan det verkliga nollställets värde är $x \approx 1.41421356237$. Alla siffror i approximationen m_i är alltså korrekta (den sista decimalen är också korrekt avrundad). Felet vid det sista steget är som mest $4.8 \cdot 10^{-7}$. ▲

3.3 Bevis av satsen om mellanliggande värden

Vårt mål i detta avsnitt är att bevisa satsen om mellanliggande värden, alltså Sats 3.2.5. Vi kommer behöva använda begreppet övre gräns.

Definition 3.3.1. Låt M vara en icke-tom delmängd till $\mathbb{R}$. Vi kallar $u \in \mathbb{R}$ en *övre gräns* till M om det för alla $x \in M$ gäller att $x \leqslant u$. △

Exempel 3.3.2. Låt $M = \{x \in \mathbb{R} \mid 0 < x < 1\}$. Talet 1 är en övre gräns till M. Även talet 2 är en övre gräns till M. Faktum är att varje $u \geqslant 1$ är en övre gräns till M. ▲

En mängd kan alltså ha flera övre gränser. En fundamental egenskap hos de reella talen är att varje mängd som har en övre gräns också har en *minsta* övre gräns. Vi formulerar detta som följande grundläggande princip.[12]

Princip 3.3.3. *Låt M vara en icke-tom delmängd till $\mathbb{R}$. Om M har en övre gräns så har M en minsta övre gräns.*

Den minsta övre gränsen u^* har egenskapen att om ett reellt tal u är mindre än u^* så kan u inte vara en övre gräns till M. I Exempel 3.3.2 är talet 1 den minsta övre gränsen till M.

Nu kan vi bevisa satsen om mellanliggande värden. Kom ihåg att satsen säger följande:

Låt $f : \mathbb{R} \to \mathbb{R}$ vara en kontinuerlig funktion, och låt a och b vara reella tal sådana att $a \leqslant b$. Om det reella talet c ligger mellan $f(a)$ och $f(b)$ så finns ett $x \in [a, b]$ sådant att $f(x) = c$.

Vi kommer anta att $f(a) \leqslant f(b)$. Om det omvända gäller kan vi byta tecken på funktionen f så att $f(a) \leqslant f(b)$.

Bevis av Sats 3.2.5. Antag att $f(a) \leqslant f(b)$. Välj ett godtyckligt reellt tal c sådant att $f(a) \leqslant c \leqslant f(b)$. Vi delar upp i två fall:

(i) Om $c = f(a)$ eller $c = f(b)$ väljer vi helt enkelt motsvarande ändpunkt ($x = a$ eller $x = b$), vilket leder till att $f(x) = c$. I detta fall behöver vi alltså inte göra något mer.

(ii) Om $f(a) < c < f(b)$ definierar vi mängden

$$M = \{z \in [a, b] \mid f(z) < c\}.$$

Mängden M är inte tom, för $a \in M$. Talet b är en övre gräns till M. Enligt Princip 3.3.3 har M en minsta övre gräns, låt oss kalla den x. Vi ska visa att $f(x) = c$.

A. Antag att $f(x) < c$ och visa att detta leder till en motsägelse. Eftersom $c < f(b)$ är $f(x) < f(b)$, så $x \neq b$. Därmed måste $x < b$. Då f är kontinuerlig kan $|f(y) - f(x)|$ göras godtyckligt litet, mindre än ett

[12] Denna princip hänger ihop med att det inte finns några ”glapp” i tallinjen. Vi kommer inte att bevisa principen här, men detta gjordes i Cirkeln 2016–2017 (*Vad är ett tal?*).

godtyckligt $\varepsilon > 0$, om bara y väljs tillräckligt nära x. Att $|f(y)-f(x)| < \varepsilon$ betyder[13]

$$-\varepsilon < f(y) - f(x) < \varepsilon.$$

Genom att addera $f(x)$ får vi

$$f(x) - \varepsilon < f(y) < f(x) + \varepsilon.$$

Vi kan välja $\varepsilon = c - f(x)$, vilket är positivt eftersom vi antog att $f(x) < c$. Detta innebär att

$$2f(x) - c < f(y) < c,$$

så $f(y) < c$ förutsatt att y är tillräckligt nära x, nämligen närmare än något $\delta > 0$. Några y som uppfyller detta är större än x och uppfyller alltså $x < y < x + \delta$. Om δ väljs tillräckligt litet är $x + \delta < b$, så dessa y tillhör mängden M. Eftersom x är en övre gräns till M så är $y \leqslant x$. Men vi sa ju att $y > x$, så detta är en motsägelse! Alltså måste antagandet att $f(x) < c$ vara fel.

B. Antag istället att $f(x) > c$. Då $f(a) < c$ är $f(x) > f(a)$, så $x > a$. Eftersom f är kontinuerlig kan vi på liknande sätt som i fall A hitta tal y som uppfyller $x - \delta < y < x$ och $f(y) > c$. Men även y är då en övre gräns till M, så detta motsäger att x är den minsta övre gränsen! Alltsa måste antagandet $f(x) > c$ vara fel.

Vi har visat att både $f(x) < c$ och $f(x) > c$ är falskt, så den enda återstående möjligheten är $f(x) = c$. □

Övningar

Övning 3.1. Skriv följande binärbråk som decimaltal.

(i) 1010.0011_2

(ii) 11011.01_2

Övning 3.2. Skriv följande decimaltal som binärbråk.

(i) 9.625

(ii) 25.0625

Övning 3.3. Binärbråksutveckla decimaltalet 0.3.

Övning 3.4 ($\star$)**.** Visa att decimalutvecklingen av ett binärbråk på formen

$$(10_2)^{-(n+1)} = 0.\underbrace{00\cdots00}_{n \text{ stycken}}1_2$$

alltid slutar på 5.

Övning 3.5. Kan följande decimaltal lagras i ett 4+4 ickenegativt fixtal? Om så, ange hur.

[13] $|A| < B$ betyder $-B < A < B$.

(i) 0.1

(ii) 15.25

(iii) 3.03125

Övning 3.6. Hur många värden kan lagras i ett 4+4-bitars ickenegativt fixtal? Vilket är det största och minsta talet?

Övning 3.7. Vilka decimaltal motsvarar flyttalen med följande mantissor och exponenter?

(i) Tecken: 0_2. Mantissa: 1.1011_2. Exponent: 5

(ii) Tecken: 1_2. Mantissa: 1.10011_2. Exponent: 3

(iii) Tecken: 0_2. Mantissa: 1.111_2. Exponent: 1

Övning 3.8 ($\star$). Beskriv hur följande decimaltal lagras i ett flyttal. Hur många bitar behöver flyttalet ha?

(i) 32.625

(ii) 3.03125

(iii) -22.3125

Övning 3.9 ($\star$). Visa att alla tal som kan lagras som ett flyttal utan att avrundas är rationella. Förklara varför detta innebär att $\sqrt{2}$ inte kan lagras exakt i en dator.

Övning 3.10. Bevisa att för alla tal x och y så gäller

$$|xy| = |x||y|.$$

Övning 3.11. Bevisa *triangelolikheten*: för alla tal x och y så gäller

$$|x+y| \leqslant |x| + |y|.$$

Övning 3.12. Rita följande funktioners grafer (antingen för hand eller i en grafritare) och avgör ifall de är kontinuerliga eller ej.

(i)

$$f(x) = \begin{cases} x^2 + x & \text{om } x \geqslant 0 \\ 0 & \text{om } x < 0 \end{cases}$$

(ii)

$$g(x) = \begin{cases} x^2 + 1 & \text{om } x \geqslant 0 \\ -x & \text{om } x < 0 \end{cases}$$

(iii)

$$h(x) = \begin{cases} x^3 + 1 & \text{om } x \geqslant 2 \\ x^2 + 5 & \text{om } x < 2 \end{cases}$$

Övning 3.13 ($\star\star$). Använd Definition 3.2.4 för att visa att funktionen $f(x) = x^2 - 2$ är kontinuerlig. Tips: använd resultaten i Övning 3.11 och 3.10.

Övning 3.14 ($\star$). Använd satsen om mellanliggande värden för att visa att ekvationen $\cos(x) = x$ har en lösning i intervallet $[0, 1]$.

4 Tärningen är kastad

Vad är ett slumpmässigt tal? Är till exempel talet 2 ett slumpmässigt tal? Frågan går inte att besvara eftersom vi inte vet om talet 2 i detta fall kommer från ett tärningskast, eller om det är resultatet av att skriva in $1 + 1$ i miniräknaren. Detta visar att slumpmässighet inte är en egenskap hos ett tal, utan en egenskap hos en *process* som genererar tal (till exempel att kasta en tärning).

Om vi kastar en tärning flera gånger kan vi skriva upp resultatet av varje kast och få en talföljd:

$$4, \quad 1, \quad 6, \quad 2, \quad 3, \quad 5, \quad 4, \quad 6, \quad 3, \quad 3, \quad 6, \quad 4, \quad 5, \quad 3, \quad 2, \quad 2.$$

Om tärningskasten verkligen är slumpmässiga kommer varje tal i talföljden vara oberoende av de andra talen. I så fall borde det inte finnas något mönster i talföljden, och det går inte att förutspå vad nästa tal kommer att bli. Tal som fås från tärningskast är ett exempel på *äkta slumptal*. Andra exempel på äkta slumptal är att singla slant (krona = 1, klave = 0) och att dra papperslappar med tal ur en tombola. Man kan också få äkta slumptal genom att mäta atmosfäriskt brus[14] eller radioaktivt sönderfall.

Att generera äkta slumptal är ofta antingen långsamt eller kräver dyr utrustning. I många sammanhang använder man därför istället datoralgoritmer som skapar talföljder som *ser ut* att vara slumpmässiga, fastän de inte är det. Dessa algoritmiskt genererade slumptal kallas *pseudoslumptal* (*pseudo* kommer från ett grekiskt ord som betyder ”lögnaktig” eller ”falsk”). Pseudoslumptal används till exempel i datorspel, simuleringar, vissa beräkningsmetoder samt inom kryptering. I avsnitt 4.2 berättar vi mer om några tillämpningar av slumptal, men först ska vi titta närmare på hur pseudoslumptal kan genereras.

4.1 Pseudoslumptal

En pseudoslumptalsgenerator (på engelska *pseudorandom number generator*, förkortat PRNG) utgår från ett startvärde x_0 och använder en funktion f för att beräkna en talföljd, enligt nedan.

Definition 4.1.1. En *pseudoslumptalsgenerator* $P = (S, f)$ består av en mängd S som kallas *tillståndsrum* och en funktion $f : S \to S$.

Givet en pseudoslumptalsgenerator P och ett startvärde $x_0 \in S$ som kallas *slumpfrö* definieras en *följd av pseudoslumptal* av

$$x_{n+1} = f(x_n), \qquad n = 0, 1, 2, \ldots \qquad \triangle$$

I detta kompendium är tillståndsrummet S alltid en delmängd av de ickenegativa heltalen, till exempel $S = \{0, 1, 2, 3, 4\}$. Lägg märke till att en pseudoslumptalsgenerator egentligen inte alls är slumpmässig; nästa tal i följden bestäms entydigt av det föregående med hjälp av regeln $x_{n+1} = f(x_n)$. Tricket är att välja en bra funktion f som gör att följden *ser ut* att vara slumpmässig.

[14] På webbsidan `https://www.random.org/` kan man hitta äkta slumptal genererade från atmosfäriskt brus.

En enkel typ av pseudoslumptalsgenerator kallas för linjär kongruensgenerator och är baserad på kongruensräkning (se avsnitt 2.2). En sådan generator definieras av tre tal m, a och b på följande sätt.

Definition 4.1.2. En *linjär kongruensgenerator* $L(m, a, b)$ är en pseudoslumptalsgenerator (S, f) med tillståndsrum $S = \{0, 1, 2, \ldots, m-1\}$, där m är ett positivt heltal, vars funktion f ges av[15]

$$x_{n+1} = f(x_n) = \mathrm{Mod}_m(ax_n + b), \qquad n = 0, 1, 2, \ldots$$

där a och b tillhör S. △

Varje tal x_n är ju här ett heltal som uppfyller $0 \leqslant x_n < m$. Om man istället vill ha slumptal mellan 0 och 1 kan man definiera

$$u_n = \frac{x_n}{m}, \tag{4.1}$$

vilket kommer vara ett reellt tal som uppfyller $0 \leqslant u_n < 1$.

Exempel 4.1.3. Låt $m = 5$, $a = 2$ och $b = 1$. Med startvärde $x_0 = 0$ fås då talföljden

$$\begin{aligned} x_1 &= \mathrm{Mod}_5(2 \cdot 0 + 1) = \mathrm{Mod}_5(1) = 1, \\ x_2 &= \mathrm{Mod}_5(2 \cdot 1 + 1) = \mathrm{Mod}_5(3) = 3, \\ x_3 &= \mathrm{Mod}_5(2 \cdot 3 + 1) = \mathrm{Mod}_5(7) = 2, \\ x_4 &= \mathrm{Mod}_5(2 \cdot 2 + 1) = \mathrm{Mod}_5(5) = 0, \\ x_5 &= \mathrm{Mod}_5(2 \cdot 0 + 1) = \mathrm{Mod}_5(1) = 1, \\ &\vdots \end{aligned}$$

alltså

$$0, \quad 1, \quad 3, \quad 2, \quad 0, \quad 1, \quad 3, \quad 2, \quad 0, \quad 1, \quad 3, \quad 2, \quad 0, \quad 1, \quad 3, \quad 2, \quad \ldots$$

Med startvärde $x_0 = 4$ fås istället talföljden

$$\begin{aligned} x_1 &= \mathrm{Mod}_5(2 \cdot 4 + 1) = \mathrm{Mod}_5(9) = 4, \\ x_2 &= \mathrm{Mod}_5(2 \cdot 4 + 1) = \mathrm{Mod}_5(9) = 4, \\ &\vdots \end{aligned}$$

alltså

$$4, \quad 4, \quad 4, \quad 4, \quad 4, \quad 4, \quad 4, \quad 4, \quad 4, \quad \ldots$$

▲

Ingen av talföljderna i Exempel 4.1.3 ser särskilt slumpmässiga ut! Det som händer är att generatorn går in i en cykel och upprepar samma tal periodiskt. Det är förstås så att så fort generatorn kommer tillbaka till ett tillstånd $x \in S$ som den redan har besökt tidigare så kommer den att upprepa sig själv i all oändlighet. Detta beror på att f är en funktion, och därmed alltid ger samma utdata för ett givet indata. Vi definierar generatorns period enligt följande.

[15] Funktionen Mod_m definierades i Definition 2.2.3.

Definition 4.1.4. Låt $P = (S, f)$ vara en pseudoslumptalsgenerator, och låt $x_0 \in S$ vara ett givet slumpfrö. Generatorn P:s *period* givet x_0 är det minsta positiva heltal p sådant att $x_{n+p} = x_n$ för något $n \in \{0, 1, 2, \ldots\}$. △

I Exempel 4.1.3 är generatorns period 4 för slumpfröet $x_0 = 0$, medan perioden är 1 för slumpfröet $x_0 = 4$.

För att en generator ska vara användbar måste den ha en lång period. Perioden kan dock aldrig vara större än $|S|$, antalet tillstånd (varför inte?). För en linjär kongruensgenerator är $|S| = m$, så den största möjliga perioden är m. Vad perioden faktiskt blir beror på parametrarna m, a och b. Följande sats gäller specialfallet då $b = 0$ och m är ett primtal.

Sats 4.1.5. *Låt $P = L(m, a, b)$ vara en linjär kongruensgenerator där m är ett primtal och $b = 0$. Generatorn P har då perioden $m - 1$ för alla positiva slumpfrön $x_0 \in S$ om a är ett primitivt element modulo m.*

För att förstå satsen (som vi kommer bevisa lite senare) behöver vi följande definition.

Definition 4.1.6. Låt m vara ett positivt heltal. Ett tal $a \in \{1, 2, \ldots, m-1\}$ kallas ett *primitivt element modulo m* om det för varje $k \in \{1, 2, \ldots, m-1\}$ finns ett positivt heltal i sådant att $k = \mathrm{Mod}_m(a^i)$. △

Ett primitivt element är alltså ett tal a sådant att alla positiva heltal mindre än m kan skrivas som en potens av a, modulo m. Eller annorlunda uttryckt, om vi går igenom den oändliga talföljden

$$a^1, \quad a^2, \quad a^3, \quad a^4, \quad \ldots$$

så kommer vi för varje $k \in \{1, 2, \ldots, m-1\}$ hitta något tal i talföljden som är kongruent med k modulo m.

Exempel 4.1.7. Låt $m = 5$. Tabellen nedan visar värdet på a^1, a^2, a^3, a^4, a^5 för $a \in \{1, 2, 3, 4\}$, och deras kongruenser modulo 5.

	a^1	a^2	a^3	a^4	a^5	
$a = 1$	$1 \equiv \mathbf{1}$	$1 \equiv \mathbf{1}$	$1 \equiv \mathbf{1}$	$1 \equiv \mathbf{1}$	$1 \equiv \mathbf{1}$	(mod 5)
$a = 2$	$2 \equiv \mathbf{2}$	$4 \equiv \mathbf{4}$	$8 \equiv \mathbf{3}$	$16 \equiv \mathbf{1}$	$32 \equiv \mathbf{2}$	(mod 5)
$a = 3$	$3 \equiv \mathbf{3}$	$9 \equiv \mathbf{4}$	$27 \equiv \mathbf{2}$	$81 \equiv \mathbf{1}$	$243 \equiv \mathbf{3}$	(mod 5)
$a = 4$	$4 \equiv \mathbf{4}$	$16 \equiv \mathbf{1}$	$64 \equiv \mathbf{4}$	$256 \equiv \mathbf{1}$	$1024 \equiv \mathbf{4}$	(mod 5)

Här kan vi se att 2 och 3 är primitiva element modulo 5. Vi kan också inse att 1 inte kan vara ett primitivt element eftersom $1^i = 1$ för alla heltal i. Vi misstänker att 4 inte heller är ett primitivt element, men för att verkligen visa det behöver vi följande hjälpsats. ▲

Hjälpsats 4.1.8. *Låt m vara ett positivt heltal, och låt x, y och z vara heltal. Om*

$$x \equiv y \pmod{m}$$

så är

$$xz \equiv yz \pmod{m}.$$

Bevis. Kom ihåg att $x \equiv y \pmod{m}$ betyder att det finns ett heltal k sådant att $x = y + km$. Om denna ekvation multipliceras med z fås $xz = yz + kzm$. Eftersom även kz är ett heltal så betyder detta att $xz \equiv yz \pmod{m}$. □

Som ett specialfall av Hjälpsats 4.1.8 med $x = a^i$ och $z = a$ fås att om

$$a^i \equiv y \pmod{m}$$

så är

$$a^{i+1} \equiv ay \pmod{m}.$$

Därmed vet vi direkt att om $4^4 = 256 \equiv 1 \pmod{5}$ så måste $4^5 \equiv 4 \cdot 1 \pmod{5}$. På samma sätt blir $4^6 \equiv 4 \cdot 4 = 16 \equiv 1 \pmod{5}$, och det blir tydligt att 4^i aldrig kommer vara kongruent med 2 eller 3 modulo 5. Ett ordentligt bevis kan göras med hjälp av induktion, vilket vi lämnar som Övning 4.8.

Vi kommer ha nytta även av följande hjälpsats, som berättar när den motsatta riktningen av Hjälpsats 4.1.8 gäller.

Hjälpsats 4.1.9. *Låt m vara ett primtal, och låt x, y och z vara heltal. Antag dessutom att $0 < z < m$. Om*

$$xz \equiv yz \pmod{m}$$

så är

$$x \equiv y \pmod{m}.$$

Bevis. Att $xz \equiv yz \pmod{m}$ innebär att det finns ett heltal k sådant att $xz = yz + km$. Vi kan subtrahera med yz och bryta ut z för att få

$$(x - y)z = km.$$

Detta säger (jämför med Definition 2.2.7) att primtalet m delar talet $(x - y)z$. Om så är fallet måste m dela antingen talet $x - y$ eller talet z.[16] I detta fall kan m inte dela z eftersom $0 < z < m$. Därmed måste m dela $x - y$, vilket innebär att det finns ett heltal j sådant att

$$x - y = jm.$$

Detta betyder att $x = y + jm$, så $x \equiv y \pmod{m}$, vilket slutför beviset. □

Bevis av Sats 4.1.5

Vårt mål är nu att bevisa Sats 4.1.5. Vi börjar med ännu en hjälpsats.

Hjälpsats 4.1.10. *Låt $P = L(m, a, b)$ med $b = 0$ och $a > 0$. Då är*

$$x_n \equiv a^n x_0 \pmod{m}, \qquad n = 0, 1, 2, \ldots$$

[16]Vi använder här ett resultat som kallas *Euklides hjälpsats* och som säger att om ett primtal m delar talet ab, där a och b är heltal, så måste m dela a eller b. Vi kommer använda detta utan bevis.

Bevis. Vi bevisar påståendet med hjälp av induktion. Basfallet $n = 0$ blir

$$x_0 \equiv a^0 x_0 \pmod{m},$$

vilket är sant eftersom $a^0 = 1$ för alla positiva heltal a.

För induktionssteget antar vi att

$$x_k \equiv a^k x_0 \pmod{m} \tag{4.2}$$

är sant för något heltal k. Från Definition 4.1.2 har vi $x_{k+1} = \mathrm{Mod}_m(ax_k)$, vilket speciellt betyder att

$$x_{k+1} \equiv ax_k \pmod{m}. \tag{4.3}$$

Nu använder vi Hjälpsats 4.1.8 som säger att om ekvation (4.2) är sann så är

$$ax_k \equiv a^{k+1} x_0 \pmod{m} \tag{4.4}$$

sann. Ekvation (4.3) och (4.4) tillsammans visar nu att

$$x_{k+1} \equiv a^{k+1} x_0 \pmod{m},$$

vilket visar att ekvation (4.2) är sann även för $k + 1$. Därmed har vi bevisat Hjälpsats 4.1.10. □

Vi är nu redo att äntligen bevisa Sats 4.1.5. För att komma ihåg vad satsen säger repeterar vi den här:

Låt $P = L(m, a, b)$ vara en linjär kongruensgenerator där m är ett primtal och $b = 0$. Generatorn P har då perioden $m - 1$ för alla positiva slumpfrön $x_0 \in S$ om a är ett primitivt element modulo m.

Bevis av Sats 4.1.5. Idén är att visa att a^0, a^1, a^2, $\ldots$, a^{m-2} alla är olika modulo m, och att $a^{m-1} \equiv a^0 \pmod{m}$. Vi visar sedan att detta bevaras om man multiplicerar med x_0.

Steg 1. Eftersom a är ett primitivt element modulo m så finns ett positivt heltal p sådant att $1 \equiv a^p \pmod{m}$. Låt dessutom p vara det minsta positiva heltalet med denna egenskap. För högre exponenter upprepas värdena enligt Hjälpsats 4.1.8, det vill säga $a^{p+1} \equiv a$, $a^{p+2} \equiv a^2$ och så vidare $\pmod{m}$. Detta innebär att a^i som mest kan anta p olika värden modulo m, nämligen

$$\mathrm{Mod}_m(a^1), \quad \mathrm{Mod}_m(a^2), \quad \mathrm{Mod}_m(a^3), \quad \ldots, \quad \mathrm{Mod}_m(a^p). \tag{4.5}$$

Dessa p värden måste verkligen vara olika. För att se det, antag motsatsen, nämligen att $a^i \equiv a^j \pmod{m}$ fastän $1 \leqslant i < j \leqslant p$. Eftersom $0 < a < m$ kan vi använda Hjälpsats 4.1.9 för att "dividera" med a. Om vi gör detta i gånger får vi

$$1 \equiv a^{j-i} \pmod{m}.$$

Nu är $j - i$ ett positivt heltal mindre än p, men p var ju det minsta positiva heltal som uppfyller $1 \equiv a^p \pmod{m}$. Detta är en motsägelse, så alla tal i (4.5) måste vara olika.

Eftersom a är ett primitivt element modulo m måste de $m-1$ talen i mängden $\{1, 2, \ldots, m-1\}$ finnas bland talen i (4.5). Detta innebär att $p \geqslant m-1$. Men p kan inte vara större än $m-1$, för då skulle något tal behöva upprepas (som Övning 4.10 visar kan nämligen inte talet 0 finnas bland talen i (4.5)). Alltså är $p = m-1$.

Steg 2. Vi visar nu att om a^i och a^j är olika modulo m så är även $a^i x_0$ och $a^j x_0$ olika modulo m. Antag motsatsen, nämligen att $a^i x_0 \equiv a^j x_0 \pmod{m}$. Men $0 < x_0 < m$, så enligt Hjälpsats 4.1.9 är då $a^i \equiv a^j \pmod{m}$, vilket säger emot att a^i och a^j är olika modulo m. Alltså är $a^i x_0$ och $a^j x_0$ olika när a^i och a^j är olika.

Hjälpsats 4.1.10 säger ju att $x_n \equiv a^n x_0 \pmod{m}$, så nu har vi alltså sammantaget visat att alla värden

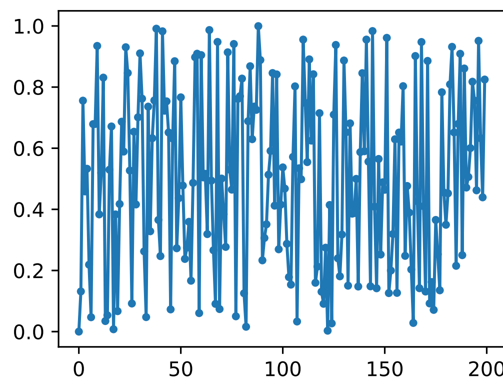
$$x_1, \quad x_2, \quad x_3, \quad \ldots, \quad x_p \qquad\qquad (p = m-1)$$

är olika, och att $x_p = x_0$. Detta visar att perioden för generatorn är p, och eftersom $p = m-1$ har vi bevisat Sats 4.1.5. □

En användbar linjär kongruensgenerator

Eftersom generatorns period inte kan vara större än m vill man välja m mycket stort. För att garantera periodens längd kan man använda Sats 4.1.5, och då måste m dessutom vara ett primtal.

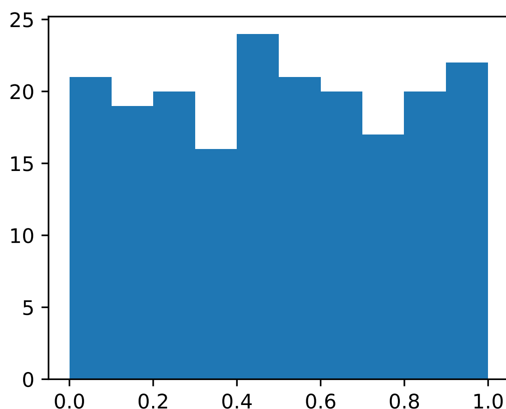
Exempel 4.1.11. Låt en linjär kongruensgenerator ges av $m = 2^{31}-1$, $a = 7^5$ och $b = 0$. Talet m är ett mycket stort primtal[17] ($m = 2\,147\,483\,647$) och talet a är ett primitivt element modulo m, så enligt Sats 4.1.5 är generatorns period $2^{31} - 2$ för alla positiva slumpfrön. Vi provar att generera 200 slumptal $u_n = x_n/m$ med slumpfrö $x_0 = 1$. Resultatet visas i Figur 4.1. Inget uppenbart mönster syns.



Figur 4.1: 200 pseudoslumptal u_n mellan 0 och 1. (På x-axeln visas n, på y-axeln värdet på u_n.)

[17]Ett tal på formen $2^k - 1$ där k är ett heltal kallas ett *Mersennetal*. Ett Mersennetal som även är ett primtal, som $m = 2^{31} - 1$, kallas ett *Mersenneprimtal*. Just detta primtal upptäcktes 1772 av matematikern Leonhard Euler. Det största Mersenneprimtalet som är känt idag (september 2019) upptäcktes 2018 och är det enorma talet $2^{82\,589\,933} - 1$, vilket även är det största primtal man känner till över huvud taget.

Vi kan också undersöka om talen u_n verkar vara jämnt fördelade, med hjälp av ett histogram (Figur 4.2). I histogrammet delas intervallet $[0, 1]$ upp i 10 stycken delintervall med längd 0.1, och så räknas antalet pseudoslumptal som hamnade i varje delintervall. I detta fall ser det någorlunda jämnt ut. Även äkta slumptal kommer ha viss variation i histogrammet, men vi förväntar oss att det blir jämnare och jämnare ju fler slumptal som tas med. ▲



Figur 4.2: Histogram för de 200 pseudoslumptalen u_n. (På x-axeln visas värdet på u_n, på y-axeln antalet pseudoslumptal som hamnade inom varje intervall.)

Exempel 4.1.12. Vi kan använda samma generator som i föregående exempel, men simulera en tärning genom att sätta

$$t_n = \lfloor 6u_n \rfloor + 1.$$

Här betyder $\lfloor x \rfloor$ heltalsdelen av x, det vill säga det man får om man helt enkelt tar bort decimalerna (samma sak som att alltid avrunda nedåt). Eftersom $0 \leqslant u_n < 1$ kommer $0 \leqslant 6u_n < 6$, så $\lfloor 6u_n \rfloor$ kan anta något av värdena $\{0, 1, 2, 3, 4, 5\}$. Genom att addera 1 får vi samma värden som en vanlig tärning kan visa.

Med slumpfrö $x_0 = 1$ blir de 16 första pseudoslumptalen t_n

$$1, \quad 1, \quad 5, \quad 3, \quad 4, \quad 2, \quad 1, \quad 5, \quad 5, \quad 6, \quad 3, \quad 4, \quad 5, \quad 1, \quad 1, \quad 4.$$

Det är förstås svårt att säga bara utifrån detta om vår tärningssimulator är bra eller dålig. Något uppenbart mönster verkar det inte finnas i talföljden i alla fall. ▲

4.2 Tillämpningar av slumptal

Pseudoslumptal används alltså för allt från datorspel till kryptering, och kraven som ställs är olika beroende på tillämpning. Framförallt i kryptering är det extra viktigt att slumptalen är oförutsägbara, eftersom det annars finns en risk att en illvillig person lyckas lista ut de hemliga nycklar som används och därmed kan läsa den krypterade informationen. Därför räknas *kryptografiskt*

säkra pseudoslumptalsgeneratorer som en egen klass av slumptalsgeneratorer. De slumptalsgeneratorer som vi studerade i föregående avsnitt är tyvärr inte kryptografiskt säkra.

Vi ska nu titta närmare på några enkla exempel där slumptal kan vara användbara.

Caesarkrypto. Ett av de enklaste sätten att kryptera text är Caesarkryptot, som användes redan av Julius Caesar. Det går ut på att varje bokstav i texten byts ut mot den bokstav som är X platser framför i alfabetet (modulo alfabetets längd, det vill säga om man går förbi Ö så börjar man om från A). Talet X är ett heltal som hålls hemligt och kallas för *nyckel* (den som ska kunna läsa meddelandet måste förstås också känna till nyckeln). Eftersom nyckeln inte ska gå att gissa så kan det vara lämpligt att generera den slumpmässigt.

Exempel 4.2.1. Låt nyckeln vara $X = 5$. I krypteringsfasen ska då bokstäverna bytas ut enligt a $\to$ f, b $\to$ g, c $\to$ h, ..., å $\to$ c, ä $\to$ d, ö $\to$ e. Till exempel blir meddelandet ”Tärningen är kastad” krypterat som

Ydwsnsljs dw pfxyfi

För att avkryptera meddelandet byter man ut varje bokstav mot den bokstav som är X plaster *bakom* i alfabetet istället. ▲

Caesarkrypto kan även användas i modernare datorsammanhang. Datorer lagrar ju all information som en följd av ettor och nollor, och på alla moderna datorer så grupperas bitarna i grupper om åtta, vilket ju kallas en byte. En byte kan anta 256 olika värden. En fil på en dator kan alltså ses som en följd av byte B_1, B_2, ..., B_N, där N är filens storlek (antal byte). För att kryptera filen kan man välja en nyckel X och beräkna

$$K_i = \mathrm{Mod}_{256}(B_i + X), \qquad i = 1, 2, \ldots, N.$$

Värdena K_i är byten i den krypterade filen. För att avkryptera filen beräknar man

$$B_i = \mathrm{Mod}_{256}(K_i - X), \qquad i = 1, 2, \ldots, N.$$

Caesarkryptot är väldigt enkelt, men det är också lätt att knäcka, framförallt med en dator som snabbt kan testa alla möjliga nycklar X. Av den anledningen ses inte Caesarkryptot som en säker krypteringsmetod nu för tiden.

Slumpvandring. En slumpvandring är en matematisk modell av en partikel som rör sig slumpmässigt. I det enklaste fallet rör sig partikeln längs x-axeln, och dess position x_n vid tiden n ges av

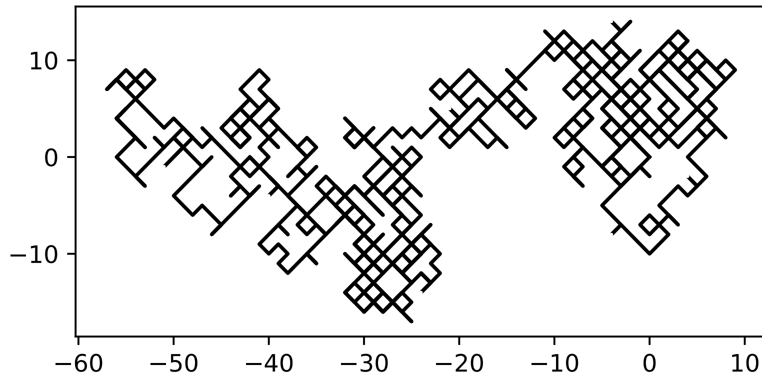
$$x_n = x_{n-1} + D_i, \qquad n = 1, 2, \ldots,$$

där D_i är ett slumptal som till exempel kan anta värden i mängden $\{-1, 1\}$. Partikeln hoppar alltså åt ena eller andra hållet i varje steg. Startpositionen kan sättas till $x_0 = 0$.

Om partikeln rör sig i xy-planet kan dess position (x_n, y_n) vid tiden n ges av

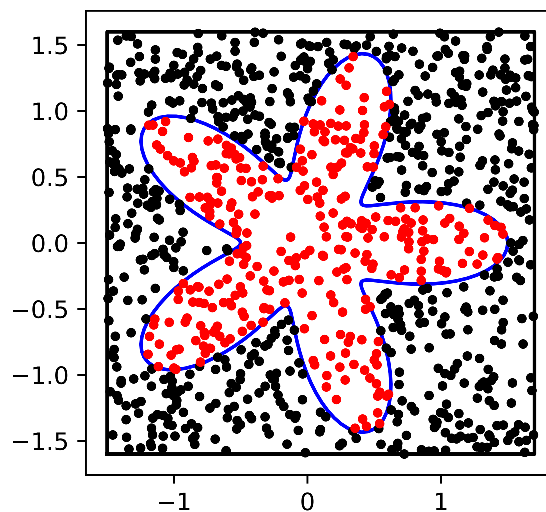
$$\begin{cases} x_n = x_{n-1} + D_i, \\ y_n = y_{n-1} + E_i, \end{cases} \qquad n = 1, 2, \ldots,$$

där både D_i och E_i är slumptal, och startpositionen är $(x_0, y_0) = (0, 0)$. Ett exempel på en sådan slumpvandring visas i Figur 4.3.



Figur 4.3: Slumpvandring med 1000 steg.

Monte Carlo-beräkningar. Ett sätt att beräkna arean av en komplicerad form i planet är att omgärda formen med en fyrkant och slumpa fram punkter inom fyrkanten, som i Figur 4.4.



Figur 4.4: Beräkning av en sjöstjärnas area.

Om vi räknar antalet punkter som hamnar inom den komplicerade formen (färgade röda i Figur 4.4) och dividerar med det totala antalet punkter, så får

vi reda på hur stor formens area är jämfört med fyrkantens area. Eftersom arean för en fyrkant är lätt att räkna ut kan vi från den räkna ut den komplicerade formens area.

I detta fall hamnade 318 punkter av 1000 inuti sjöstjärneformen, vilket betyder att sjöstjärnans area är 0.318 av fyrkantens area. Fyrkantens area är $3.2 \cdot 3.2 = 10.24$ areaenheter, så sjöstjärnans area är

$$0.318 \cdot 10.24 = 3.25632$$

areaenheter.

RSA-kryptering. RSA-kryptering är en betydligt mer avancerad krypteringsmetod jämfört med Caesarkryptot. RSA-kryptering baseras på att det är mycket svårt att bestämma vilka primtal ett stort heltal är en produkt av. Metoden är *asymmetrisk*, vilket betyder att vem som helst kan kryptera ett meddelande med hjälp av en publik nyckel, men för att avkryptera meddelandet behövs en hemlig nyckel. Översiktligt så fungerar det så här:

1. Två stora primtal p och q slumpas fram. De ska hållas hemliga och det är mycket viktigt att de inte går att förutsäga på något sätt.

2. Talet $n = pq$ beräknas. Detta tal är en del av den *publika* nyckeln, och vem som helst får alltså känna till det. Eftersom n är ett stort tal är det svårt att faktorisera, så p och q är fortfarande hemliga.

3. Ett heltal e beräknas på ett specifikt sätt utifrån p och q (vi går inte in på hur här). Detta heltal är den andra delen av den *publika* nyckeln. Det går inte att gissa p och q utifrån e.

4. Ett heltal d beräknas på ett specifikt sätt utifrån e, p och q (vi går inte in på hur här). Detta heltal utgör den *hemliga* nyckeln och ska alltså inte avslöjas för någon, förutom för den som ska kunna avkryptera meddelandena.

När denna process är klar behövs inte primtalen p och q längre. För att kryptera meddelanden behövs talen e och n (som tillsammans utgör den publika nyckeln), och för att avkryptera meddelanden behövs talen d (den hemliga nyckeln) och n. För att kryptera ett heltalsmeddelande m beräknas

$$c = \mathrm{Mod}_n(m^e),$$

där c är det krypterade meddelandet. För att avkryptera meddelandet beräknas

$$m = \mathrm{Mod}_n(c^d),$$

vilket fungerar eftersom $m = \mathrm{Mod}_n((m^e)^d)$ på grund av det speciella sätt som talen e och d valdes på.

Övningar

Övning 4.1. Använd en linjär kongruensgenerator $L(m, a, b)$ med $m = 10$, $a = 3$ och $b = 2$ för att beräkna x_1, x_2, x_3, x_4 och x_5 om $x_0 = 0$. Vad är perioden i detta fall?

Övning 4.2. Använd en linjär kongruensgenerator $L(m, a, b)$ med $m = 10$, $a = 5$ och $b = 5$ för att beräkna x_1, x_2, x_3, x_4 och x_5 om $x_0 = 1$. Vad är perioden i detta fall?

Övning 4.3.

(i) Vad händer om $a = 0$ för en linjär kongruensgenerator $L(m, a, b)$?

(ii) Varför är det ofta ingen bra idé att välja $a = 1$ för en linjär kongruensgenerator $L(m, a, b)$? Ge ett exempel!

Övning 4.4. En annan typ av pseudoslumptalsgenerator kallas *Blum Blum Shub* och ges av

$$x_{n+1} = \mathrm{Mod}_m(x_n^2), \qquad n = 0, 1, 2, \ldots$$

där m är ett positivt heltal. Låt $m = 11$ och $x_0 = 5$ och beräkna x_1, x_2, x_3, x_4 och x_5. Vad blir perioden?

Övning 4.5.

(i) Är 2 ett primitivt element modulo 3?

(ii) Är 2 ett primitivt element modolo 4?

(iii) Vilka tal är primitiva element modulo 7?

Övning 4.6 ($\star$)**.** Antag att $m = pq$ där p och q är heltal större än 1 och mindre än m, och p dessutom är ett primtal. Bevisa följande:

(i) Om ett heltal $a \in \{1, 2, \ldots, m-1\}$ är delbart med p så är $\mathrm{Mod}_m(a^i)$ också delbart med p för varje positivt heltal i.

(ii) Om istället $a \in \{1, 2, \ldots, m-1\}$ *inte* är delbart med p så kan inte $\mathrm{Mod}_m(a^i)$ heller vara delbart med p för något positivt heltal i.

(iii) Använd (i) och (ii) för att visa att det inte kan finnas några primitiva element modulo m om heltalet $m > 1$ inte är ett primtal.

Övning 4.7. Ge ett exempel på att Hjälpsats 4.1.9 inte gäller för alla heltal z sådana att $0 < z < m$ om m inte är ett primtal.

Övning 4.8. Visa med hjälp av induktion att för varje positivt heltal i gäller antingen $4^i \equiv 1 \pmod 5$ eller $4^i \equiv 4 \pmod 5$.

Övning 4.9. Ett Mersenneprimtal är ett tal på formen $2^k - 1$ som dessutom är ett primtal. Hur många Mersenneprimtal finns det som är mindre än 500?

Övning 4.10. Visa att om a är ett primitivt element modulo m så kan det inte finnas något positivt heltal i sådant att $a^i \equiv 0 \pmod{m}$.

Övning 4.11. Du hittar ett hemligt meddelande som lyder

Tupdlipmnt nbufnbujtlb djslfm

Du misstänker att det är ett krypterat meddelande. Vad står det egentligen?

Programmeringsuppgifter

Övning 4.P1. Skriv ett program som använder en linjär kongruensgenerator $L(m, a, b)$ för att skriva ut 20 pseudoslumptal $x_1, \ldots, x_{20}$. Parametrarna m, a och b samt slumpfröet x_0 ska gå att ändra på i programmet. Testa ditt program med $m = 2147483647$, $a = 16807$, $b = 0$ (parametrarna från Exempel 4.1.11) och $x_0 = 1$.

(De flesta programmeringsspråk har inbyggd funktionalitet för att beräkna $\mathrm{Mod}_m(x)$. I Python skrivs $\mathrm{Mod}_m(x)$ till exempel `x % m`.)

Övning 4.P2. Skriv ett program som använder generatorn Blum Blum Shub från Övning 4.4 för att skriva ut 20 pseudoslumptal $x_1, \ldots, x_{20}$. Testa ditt program med $m = 22133009$ och $x_0 = 3$.

Övning 4.P3. Ändra ditt program från Övning 4.P1 så att det istället skriver ut $u_1, \ldots, u_{20}$ där

$$u_n = \frac{x_n}{m}.$$

Övning 4.P4. Ändra ditt program från Övning 4.P3 så att det skriver ut 20 slumpmässiga heltal från mängden $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Övning 4.P5 ($\star$). Skriv ett program som testar om ett heltal a $(0 < a < m)$ är ett primitivt element modulo m, där m är ett positivt heltal. Testa ditt program med $m = 5$ och $a \in \{1, 2, 3, 4\}$. När du ser att programmet fungerar, avgör om 2 eller 3 är ett primitivt element modulo 547.

5 Formella språk

Hittills har detta kompendium handlat om hur man kan använda datorer till att lösa matematiska problem, till exempel att lösa ekvationer eller att generera slumptal.

De kommande kapitlen vänder på detta förhållande. Här studeras istället datorer med hjälp av matematiska verktyg. Mer specifikt så studeras problem som har följande allmänna form.

Givet en mängd, finn en algoritm som avgör ifall ett godtyckligt element ingår i mängden eller inte.

För att detta ska kunna studeras närmare måste två saker specificeras.

(i) Vilka mängder och element som algoritmen ska verka på.

(ii) Vad som avses med en algoritm.

Syftet med detta kapitel är att behandla den första punkten. I Kapitel 6 avhandlas den andra, och i Kapitel 7 vävs dessa ihop för att ge ett svar på beslutsproblemet.

5.1 Tecken, alfabet och ord

Världen är fylld av alfabet. Det mest använda är det latinska alfabetet, ifall man räknar alla varianter. Andra välanvända alfabet är det arabiska, det kyrilliska och det grekiska.

Språkvetenskapligt definieras ett alfabet som en samling tecken som är tänkta att motsvara ett språks grundfonem (de ljud som de talade orden byggs upp av). Ett exempel är att bokstaven a, som motsvarar vokalljudet i ordet *far*.

Här tas ett mer generellt grepp.

Definition 5.1.1. Ett *alfabet* är en icketom, ändlig mängd tecken. △

Alfabet betecknas vanligtvis med det grekiska bokstaven Σ. Ett alfabet behöver inte innehålla bokstäver: mängden $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ är ett alfabet, förutsatt att man betraktar siffrorna som tecken.

Exempel 5.1.2. Det latinska alfabetet är ett alfabet. Ett mindre alfabet är mängden $\{a, b, c\}$. Ett enkelt alfabet är $\{a\}$. ▲

Definition 5.1.3. Ett *ord* är en ändlig följd av tecken ur ett alfabet. Ett ords *längd* är antalet tecken det innehåller. △

Enskilda tecknen i ett alfabet kan ses som ord av längd 1. Ord kan även ha längd 0, och kallas då det tomma ordet.

Definition 5.1.4. Det *tomma ordet* är ordet som saknar tecken och betecknas ε. Det är ett ord i alla alfabet. △

Exempel 5.1.5. Några ord i det latinska alfabetet är *triangel*, *algebra* och *kongruensekvation*. Ur alfabetet $\{a, b\}$ kan man bilda ord som *abaa*, *abba* och *baba*. ▲

Alla följder av tecken är ord, oavsett om de betyder något eller inte. Till exempel är både *abaa* och *triangel* ord i det latinska alfabetet, trots att det förra är meningslöst och det andra refererar till en typ av geometrisk figur. När man ignorerar ordens betydelse säger man att man studerar man deras *formella* eller *syntaktiska* egenskaper.

Genom att sätta samman ord kan man få fler.

Definition 5.1.6. Sammansättningen av två ord $w = \sigma_1 \cdots \sigma_n$ och $u = \tau_1 \cdots \tau_m$ är ordet $\sigma_1 \cdots \sigma_n \tau_1 \cdots \tau_m$, och betecknas $w \circ u$.

När ett ord sammansätts med det tomma ordet (eller omvänt) är resultatet lika med ursprungliga ordet. Man kan skriva det som $\varepsilon \circ w = w = w \circ \varepsilon$. △

Exempel 5.1.7. Sammansättningen av orden *abcd* och *cbab* är $abcd \circ cbab = abcdcbab$. ▲

Ifall $w = \sigma_1 \cdots \sigma_n$ är ett ord, ser man att

$$w = \sigma_1 \circ \sigma_2 \circ \cdots \circ \sigma_n.$$

Varje ord kan alltså skrivas som en sammansättning av dess ingående tecken.

Sammansättning av ord är som multiplikation av två tal: man tar två ord och kombinerar dem för att få ett nytt. Det tomma ordet spelar samma roll som talet 1.

Till skillnad från multiplikation av två tal så spelar det roll i vilken ordning som två ord sammansätts. Till exempel är $aba \circ bab = ababab$, medan $bab \circ aba = bababa$.

Precis som vid multiplikation skrivs i regel inte sammansättningssymbolen $\circ$ ut, utan sammansättningen av w och u betecknas wu.

Definition 5.1.8. Ett ord u är *prefix* till ett ord u ifall det finns ett ord z så att $uz = w$. Ordet u är ett *suffix* ifall det finns ett ord z så att $zu = w$.

Ifall z inte är det tomma ordet, är prefixet eller suffixet *äkta*. △

Alla ord är oäkta suffix och prefix till sig själva. Det tomma ordet är prefix och suffix till alla ord.

Exempel 5.1.9. Ordet *katt* har prefixen ε, *k*, *ka*, *kat* och *katt*, och suffixen ε, *t*, *tt*, *att* och *katt*. ▲

På samma sätt som man definierar potenser av ett tal som upprepad multiplikation av talet med sig självt, definierar man upprepningen av ett ord som upprepad sammansättning av ordet med sig självt.

Definition 5.1.10. Ifall $w = \sigma_1 \cdots \sigma_m$ är ett ord och n ett positivt heltal, är den *n-faldiga upprepningen* av w

$$w^n = (\sigma_1 \cdots \sigma_m)^n = \underbrace{w \cdots w}_{n \text{ stycken}} = \underbrace{(\sigma_1 \cdots \sigma_m)(\sigma_1 \cdots \sigma_m)(\sigma_1 \cdots \sigma_m)}_{n \text{ stycken}}$$

Ifall $n = 0$, är $w^n = w^0 = \varepsilon$. △

Repetioner av ett ord påminner om potenser av ett tal, men är inte riktigt samma sak. Till exempel gäller i allmänhet **inte** att $(wu)^n = w^n u^n$. Ett konkret motexempel är att $(ab)^2$ lika med $abab$, men a^2b^2 är $aabb$. Sambandet gäller däremot ifall orden u och w är lika.

Sats 5.1.11. *Ifall w är ett ord gäller att $w^n w^m = w^{n+m}$ för alla icke-negativa heltal n och m.*

Bevis. Ifall m är 0, får man att

$$w^n \circ w^m = w^n \circ w^0 = w^n \circ \varepsilon = w^n = w^{n+0} = w^{n+m}.$$

Samma bevis fungerar när $n = 0$. Ifall både n och m är större än 0, kan man skriva

$$w^n \circ w^m = \underbrace{w \circ \cdots \circ w}_{n \text{ stycken}} \circ \underbrace{w \circ \cdots \circ w}_{m \text{ stycken}} = \underbrace{w \circ \cdots \circ w}_{n+m \text{ stycken}} = w^{n+m}.$$

□

Sammansättningar och upprepningar är ordoperationer: givet ett eller flera ord som argument ger de ett nytt ord. Man kan tänka sig andra ordoperationer.

Definition 5.1.12. *Reversionen* av ett ord $w = \sigma_1 \cdots \sigma_n$ är ordet $w^{rev} = \sigma_n \cdots \sigma_1$. Reversionen av det tomma ordet är det tomma ordet. △

Exempel 5.1.13. Reversionen av ordet *abbab* är *babba*. Reversionen av ett enskilt tecken är tecknet självt. ▲

5.2 Språk över alfabet

Enskilda ord är i regel av begränsat intresse: det intressanta händer när de samlas i mängder.

Definition 5.2.1. Ett *formellt språk* över ett alfabet är en mängd av ord i detta alfabet. △

Språken som studeras här kallas formella eftersom man enbart tar hänsyn till de ingående ordens formella egenskaper. Språk som används till vardags kallas *naturliga* språk, och studeras bland annat inom språkvetenskap.

Språk, till skillnad från ord och alfabet, kan vara oändliga. Det viktiga är att antalet tecken i varje enskilt ord är ändligt många, och att det bara finns ändligt många olika tecken i orden.

Exempel 5.2.2. Den tomma mängden och mängden som enbart innehåller den tomma strängen är språk över alla alfabet. Samtliga alfabet utgör språk över sig själva, ifall man betraktar tecknen i alfabetet som ord av längd 1.

Mängden $\{a, ac, caba, bb, cab\}$ är ett språk över alfabetet $\{a, b, c\}$. ▲

Definition 5.2.3. *Kleenetillslutningen* av ett alfabet Σ är språket som består av alla möjliga ord över Σ, och betecknas Σ^*. △

Språk är inget annat än mängder, och man kan därför tillämpa mängdoperationer på dem. Givet två språk L_1 och L_2 över Σ definieras därför *unionen*, *snittet*, *differensen* och *komplementet* av dem som

$$L_1 \cup L_2, \quad L_1 \cup L_2, \quad L_1 \setminus L_2 \quad \text{och} \quad \overline{L_1} = \Sigma^* \setminus L_1.$$

Unionen av två språk består av alla ord som ligger i endera språken, medan snittet består av alla ord som ligger i båda språken. Differensen är alla ord som enbart finns i ett av dem, medan komplementet av ett språk består av alla ord som inte ligger i det.

Genom att utnyttja att språken består av ord, kan man definiera ytterligare språkoperationer.

Definition 5.2.4. *Sammansättningen* av två språk L_1 och L_2 är språket

$$\{wu \mid w \in L_1 \text{ och } u \in L_2\}$$

och betecknas L_1L_2. △

Exempel 5.2.5. Språken $L_1 = \{kork, ek\}$ och $L_2 = \{stol, berg\}$ har sammansättningen

$$L_1L_2 = \{korkstol, korkberg, ekstol, ekberg\}.$$

Ifall två språk endast består av ett ord vardera, är sammansättningen av språken lika med språket som består av sammansättning av dessa ord. ▲

Språken $\emptyset$ och $\{\varepsilon\}$ intar en särställning i vid sammansättning av språk. Det tomma språket uppfyller att

$$\emptyset L_1 = L_1\emptyset = \emptyset$$

medan

$$\{\varepsilon\}L_1 = L_1\{\varepsilon\} = L_1.$$

Dessa egenskaper motsvarar de som 0 och 1 har för multiplikation av tal.

I likhet med sammansättning av ord så gäller i regel **inte** att $L_1L_2 = L_2L_1$. Ett konkret exempel är att

$$\{a\}\{b\} = \{ab\} \text{ och } \{b\}\{a\} = \{ba\}$$

är två olika språk.

I likhet med hur man definierar upprepningar av enskilda ord, kan man givet ett språk L definiera det upprepade språket L^n.

Definition 5.2.6. Ifall L är ett språk och n är ett heltal som är större än 0, är den *n-faldiga upprepningen* av L

$$L^n = \underbrace{L \cdots L}_{n \text{ stycken}}.$$

Ifall $n = 0$, är $L^0 = \{\varepsilon\}$. △

Exempel 5.2.7. Ifall $L = \{ab, c\}$, är

$$L^3 = \{ababab, ababc, abcab, abcc, cabab, cabc, ccab, ccc\}. \quad \blacktriangle$$

Övning 5.14 ger en generalisering av Sats 5.1.11 för godtyckliga språk.

Definition 5.2.8. Ifall L är ett språk kallas språket

$$L^* = L^0 \cup L^1 \cup L^2 \cup \cdots = \{\varepsilon\} \cup L \cup L^2 \cup \cdots$$

för *Kleenetillslutningen* av L. △

Ovanstående är en generalisering av Kleenetillslutningen av ett alfabet. Om Σ är ett alfabet (betraktat som ett språk) är Σ^n mängden av ord över Σ av längd n. Eftersom alla ord över Σ har ändlig längd så ligger varje ord i Σ^n för något n. Därför kommer Kleenetillslutning av Σ (sett som ett språk) sammanfalla med den ursprungliga definitionen av Kleenetillslutningen av ett alfabet.

Exempel 5.2.9. Språket $\{a\}$ har Kleenetillslutningen $\{\varepsilon, a, aa, aaa, \ldots\}$, det vill säga samtliga upprepningar av a. Språket $L = \{ab, c\}$ har Kleenetillslutningen

$$L^* = \{\varepsilon\} \cup \{ab, c\} \cup \{abab, abc, cab, cc\} \cup \cdots . \quad \blacktriangle$$

Kleenetillslutningen av ett språk innehåller alltid det tomma ordet eftersom $L^0 = \{\varepsilon\}$. Detta har konsekvensen att $\varnothing^* = \{\varepsilon\}$.

Genom mängdoperationer, sammanfogningar och Kleenetillslutningar kan man bilda en rad olika språk.

Exempel 5.2.10. (i) Språket

$$\{2, 23, 233, 2333, \ldots\}$$

kan skapas genom sammansättning av språken $\{2\}$ och $\{\varepsilon, 3, 33, \ldots\}$. Det senare kan beskrivas som Kleenestillslutningen av $\{3\}$, så

$$\{2, 23, 233, 2333, \ldots\} = \{2\}\left(\{3\}^*\right).$$

(ii) Språket över alfabetet $\{a, b\}$ som består av alla ord som innehåller delordet *bab* kan skrivas som

$$(\{a, b\}^*)\{bab\}(\{a, b\}^*).$$

Varför? Varje ord som innehåller *bab* kan delas upp i tre delar: sekvensen av tecken innan den första förekomsten av *bab*, ordet *bab* och sekvensen av tecken efter den första förekomsten av *bab*. Sekvenserna före och efter den första förekomsten av *bab* kan vara vilka som helst. ▲

Ett annat sätt att bilda nya språk ur gamla är genom prefix och suffix.

Definition 5.2.11. *Prefix-* och *suffixspråket* av ett språk L över Σ är

$$\mathrm{Pre}(L) = \{w \in \Sigma^* \mid w \text{ är ett prefix till något ord i } L\}$$

och

$$\mathrm{Suf}(L) = \{w \in \Sigma^* \mid w \text{ är ett suffix till något ord i } L\}. \qquad \triangle$$

Eftersom alla ord har både det tomma ordet och sig självt som suffix och prefix, så innehåller ett språk L:s prefix- och suffixspråk $L \cup \{\varepsilon\}$ som delmängd.

Exempel 5.2.12. Språket $L = \{katt\}$ har prefixspråket

$$\mathrm{Pre}(L) = \{\varepsilon, k, ka, kat, katt\}$$

och suffixspråket

$$\mathrm{Suf}(L) = \{\varepsilon, t, tt, att, katt\}. \qquad \blacktriangle$$

5.3 Reguljära språk

En geometriker studerar inte enskilda geometriska figurer, utan geometriska figurer i allmänhet. På samma sätt studerar en datavetare i regel inte enskilda formella språk, utan deras allmänna egenskaper.

Ett sätt att göra detta är att dela in de formella språken i klasser, som sedan studeras var för sig. Detta motsvarar att en geometriker studerar olika typer geometriska figurer (trianglar, parallellogram) var för sig.

Utöver att studeras var för sig, kan klasserna även relateras till varandra. Exempelvis kan en geometriker se att alla parallellogram kan delas i två likadana trianglar, genom att dra en diagonal mellan två av dess hörn.

Reguljära språk är en viktig klass av formella språk och är de som studeras i detta kompendium. De är den enklaste klassen av språk som studeras av datavetare.

Definition 5.3.1. De *reguljära språken* över ett alfabet Σ ges av följande rekursiva definition.

(i) Det tomma språket $\emptyset$ är reguljärt.

(ii) Om σ är ett tecken i Σ, är $\{\sigma\}$ reguljärt.

(iii) Ifall L_1 och L_2 är reguljära språk är $L_1 \cup L_2$, $L_1 L_2$ och L_1^* reguljära språk. $\triangle$

För att undvika ett överflöd av parenteser när man beskriver reguljära språk, så används följande bindningsregler:

Först Kleenetillslutning, därefter sammansättning och sist union.

Ett sätt att minnas konventionen är att tänka på Kleenetillslutning som en potens, sammansättning som multiplikation och union som addition. Då är bindningsreglerna desamma som för aritmetiska uttryck.

Exempel 5.3.2. Språket $\{2\}\{3\}^*$ som studerades i Exempel 5.2.10 är reguljärt. Likaledes är språket

$$\{a,b\}^*\{bab\}\{a,b\}^* = (\{a\} \cup \{b\})^* \{bab\} (\{a\} \cup \{b\})^*$$

reguljärt. ▲

Man behöver använda parenteser i ovanstående högerled, för att markera att Kleenetillslutningen ska göras över hela unionen. Språket

$$\{a\} \cup \{b\}^*\{bab\}\{a\} \cup \{b\}^*$$

är ett annat, som bland annat inte innehåller ordet bab.

Mängdklamrarna fyller ingen funktion, och de brukar därför utelämnas. När man gör detta får man *reguljära uttryck*. På engelska kallas reguljära uttryck för *regular expressions*, vilket förkortas som *regex* eller *regexp*.

Exempel 5.3.3. Det reguljära uttrycket för språket $\{2\}\{3\}^*$ är 23^*, medan det reguljära uttrycket för språket

$$(\{a\} \cup \{b\})^* \{bab\} (\{a\} \cup \{b\})^*$$

är $(a \cup b)^* \, bab \, (a \cup b)^*$. ▲

Reguljära uttryck över ett alfabet liknar orden som kan bildas över alfabetet. Men det är viktigt att inte förväxla dem: reguljära uttryck refererar till språk, det vill säga mängder av ord, inte orden i sig.

Varje reguljärt språk beskrivs av ett reguljärt uttryck. Men det är inte unikt: ett reguljärt språk kan beskrivas av många olika reguljära uttryck.

Exempel 5.3.4. (i) Språket över $\{a,b\}$ som består av orden $\{ab, ba\}$ har det reguljära uttrycket $ab \cup ba$.

(ii) Språket över $\{a,b\}$ som består av alla ord som börjar på a har det reguljära uttrycket $a(a \cup b)^*$.

(iii) Ett ord över språket $\{0,1,2,3,4,5,6,7,8,9\}$ representerar ett naturligt tal ifall det är 0 eller inte börjar på 0. Språket som består av alla dessa ord har det reguljära uttrycket

$$0 \cup (1 \cup 2 \cup 3 \cup 4 \cup 5 \cup 6 \cup 7 \cup 8 \cup 9)(0 \cup 1 \cup 2 \cup 3 \cup 4 \cup 5 \cup 6 \cup 7 \cup 8 \cup 9)^*.$$

(iv) Språket som består av alla ord på formen

$$\underbrace{((\cdots(}_{n}\underbrace{))\cdots)}_{n}$$

är *inte* reguljärt.

▲

Informellt så är de reguljära språken de som man kan bygga genom att sammansättning, icke-deterministiskt val och godtycklig upprepning (iteration) ur språk.

Övningar

Övning 5.1. Bevisa att alla språk som består ändligt många ord är reguljära.

Övning 5.2. En datafil innehåller uppmätt tryck med tillhörande tidpunkt för mätning. Tidpunkterna har formatet HH:MM:SS. Skriv ett reguljärt uttryck som väljer ut tiderna mellan 15:00 och 16:59.

Övning 5.3. Ange reguljära uttryck för språket som innehåller alla ord som består av ett jämnt antal a:n följt av ett udda antal b:n.

Övning 5.4. Ange reguljära uttryck för språket som innehåller alla ord som innehåller minst tre a:n och slutar på b.

Övning 5.5. Ange reguljära uttryck för språket som innehåller de ord över $\{0, 1\}$ som representerar binära tal som är delbara med 4.

Övning 5.6. Ange reguljära uttryck för språket som innehåller de ord över $\{0, 1\}$ som representerar binära tal som är kongruenta med 2 modulo 4.

Övning 5.7. Ett språk L har *prefixegenskap* ifall inget ord i L är ett äkta prefix av något annat ord i språket. Ge exempel på ett oändligt språk med prefixegenskapen.

Övning 5.8. Bevisa att två ord över ett alfabet är lika ifall båda är prefix eller suffix av varandra.

Övning 5.9. Bevisa att ifall L är ett språk som innehåller k stycken ord så består L^n av k^n ord. Använd detta för att ta fram ett uttryck för hur många ord

$$L^0 \cup L^1 \cup \cdots \cup L^n$$

innehåller.

Övning 5.10. Bevisa att för godtyckliga språk L gäller

$$(L^*)^* = L^*$$

och

$$L^*L^* = L^*.$$

Övning 5.11. Bevisa att suffixspråken uppfyller

(i) $\mathrm{Suf}(\sigma_1\sigma_2 \cdots \sigma_n) = \{\sigma_1\sigma_2 \cdots \sigma_n, \sigma_2 \cdots \sigma_n, \ldots, \sigma_n, \varepsilon\}$,

(ii) $\mathrm{Suf}(L_1L_2) = (\mathrm{Suf}(L_1)L_2) \cup \mathrm{Suf}(L_2)$,

(iii) $\mathrm{Suf}(L_1 \cup L_2) = \mathrm{Suf}(L_1) \cup \mathrm{Suf}(L_2)$,

(iv) $\mathrm{Suf}(L^*) = \mathrm{Suf}(L)L^*$.

för alla språk L_1 och L_2.

Övning 5.12. Använd ekvationerna i Övning 5.11 för att ge ett uttryck för suffixspråket av $L = ab^*(ab)^*$.

Övning 5.13 ($\star$)**.** Tag fram ekvationer som i Övning 5.11 för prefixspråken och bevisa att det gäller. Beräkna sedan prefixspråket av $ab^*(ab)^*$.

Övning 5.14 ($\star$)**.** Låt $\mathcal{L}$ beteckna mängden av språk över ett alfabet. *Potensfunktionen* $P : \mathcal{L} \times \mathbb{N} \to \mathcal{L}$ definieras genom $P(L, n) = L^n$.

Ge en rekursiv definition av potensfunktionen, och använd den för att bevisa att

$$P(L, n + m) = P(L, n)P(L, m)$$

för alla språk L och naturliga tal n och m.

Övning 5.15. Bevisa att om α, β och γ är reguljära uttryck, så gäller den *distributiva lagen*:

$$\alpha\beta \cup \alpha\gamma = \alpha(\beta \cup \gamma).$$

Övning 5.16 ($\star$)**.** Låt Σ vara ett alfabet. *Reversionsfunktionen* $\mathrm{rev} : \Sigma^* \to \Sigma^*$ avbildar ett ord w på sin reversion w^{rev}.

Ge en rekursiv definition av reversionsfunktionen och använd den för att reversera ordet *abcd*. Bevisa därefter att definitionen är korrekt, det vill säga att

$$\mathrm{rev}(\sigma_1 \cdots \sigma_n) = \sigma_n \cdots \sigma_1$$

för alla tecken $\sigma_1, \ldots, \sigma_n$ i alfabetet.

Övning 5.17 ($\star$)**.** En *palindrom* är ett ord som är lika med sin egen reversion. *Palindromspråket* $P(\Sigma)$ över ett alfabet Σ består av alla palindrom över detta alfabet.

(i) Skriv upp samtliga palindrom över $\{a, b\}$ som består fyra eller färre tecken.

(ii) Ge en rekursiv definition av palindromspråket över $\{a, b\}$.

(iii) Utvidga definitionen i (ii) till ett godtycklig alfabet Σ.

6 Tillståndsmaskiner

Betrakta följande beslutsproblem.

Givet ett reguljärt språk, finn en algoritm som avgör ifall ett godtyckligt ord ingår i det eller inte.

Ord och reguljära språk definierades i förra avsnittet. Här studeras istället algoritmer.

Ordet algoritm kommer från den persiske matematikern al-Khwarizmi. Han uppfann bland annat algebran som självständig matematisk disciplin och gav generella lösningar till första- och andragradsekvationer.

Latinska översättningar av hans böcker introducerade även det indiska positionssystemet i Västeuropa, vari han även beskrev generella metoder för att beräkna exempelvis summor och differenser. Dessa metoder kom att kallas algoritmer.

Löst beskrivet är en algoritm en uppsättning instruktioner som tar indata av en viss typ och producerar utdata av en viss typ. Instruktionerna ska dessutom vara *beräkningsbara*. Beräkningsbarhet kan formaliseras på olika sätt. Den mest kända är *Turingmaskinen*, namngiven efter den brittiske matematikern Alan Turing. Det är en slags teoretisk datormaskin, som är tillräckligt specifik för att studeras med matematiska metoder, men den fångar fortfarande intuitionen av beräkningsbarhet.

Det finns andra formaliseringar av beräkningsbarhet. Men trots att formaliseringarna på ytan är olika, har det visat sig att i princip samtliga är ekvivalenta, och att ingen av dem kan beräkna något som inte kan beräknas av en Turingmaskin. Detta ger stöd åt det som kallas för *Church-Turings tes*. Den säger att de beräkningsbara funktionerna är precis de som kan beräknas av en Turingmaskin.

Eftersom beräkningsbarhet inte är ett matematiskt begrepp kommer tesen aldrig kunna bevisas matematiskt. Man kan se det som ett antagande att Turingmaskinen är en korrekt modellen av vad det innebär att vara beräkningsbar.

I detta kapitel studeras *tillståndsmaskiner*. De är förenklade varianter av Turingmaskiner. Det finns två skäl att göra denna inskränkning.

(i) Tillståndsmaskiner löser beslutsproblemet för reguljära språk.

(ii) Tillståndsmaskiner är enklare att arbeta med.

Man kan tänka att ifall en Turingmaskin är en dator, så är en tillståndsmaskin en dator utan arbetsminne. Den som programmerat kan tänka tillståndsmaskiner är program som inte får spara eller ändra några programvariabler.

I detta kompendium används ibland kortformen *maskin* för tillståndsmaskiner.

6.1 Deterministiska tillståndsmaskiner

En deterministisk tillståndsmaskin över ett alfabet består av en ändlig mängd av tillstånd. Ett av dessa är utvalt som starttillstånd och noll eller fler är utvalda som accepterande tillstånd. Det finns också en övergångsfunktion, som tar ett tillstånd och ett tecken som argument och som returnerar ett tillstånd.

Maskinerna fungerar som en växellåda. De olika växlarna är tillstånden, och övergångsfunktionen beskriver hur den beter sig i varje växel. Mer konkret så specificerar den vilken växel maskinen ska byta till, ifall ett visst tecken kommer när maskinen står i en viss växel.

Övergångsfunktionen gör att maskinen kan drivas av ord över alfabetet. Man kan se orden som en sekvens av kommandon. Maskinen börjar i starttillståndet, och uppdaterar sitt interna tillstånd allteftersom den läser nya tecken i ordet.

När ordet, det vill säga sekvensen av kommandon, är slut, kommer tillståndsmaskinen drivits till något tillstånd. Ifall det tillståndet är ett accepterande, säger man att ordet accepteras av maskinen.

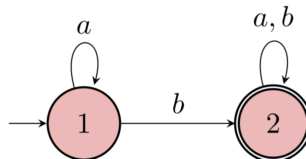
Tillståndsmaskiner ritas oftast som riktade grafer. Tillstånden ritas som cirklar. Starttillståndet markeras genom att man ritar en pil som pekar på den. De accepterande tillstånden markeras genom att rita dem som dubbla cirklar.

Övergångsfunktionen ritas som pilar mellan tillstånden, markerade med tecken ur alfabetet. En pil från ett tillstånd S_1 till ett annat tillstånd S_2 som är markerad med ett tecken σ betyder att övergångsfunktionen avbildar (S_1, σ) på S_2.

Ifall flera tecken i alfabetet går från ett tillstånd till ett annat, så skrivs samtliga till samma pil, så att det endast går en pil från ett tillstånd till ett annat.

Ett ord driver maskinen genom att vandra genom dess tillstånd. Det börjar i starttillståndet med hela ordet. Sedan konsumerar den tecknet längst till vänster i ordet, och flyttar längs pilen märkt med detta tecken. Sedan gör den om proceduren, tills att det inte finns några tecken kvar.

Exempel 6.1.1. Betrakta den deterministiska tillståndsmaskinen i Figur 6.1. Maskinen är över alfabetet $\{a, b\}$ och figuren består av två tillstånd, 1 och 2,



Figur 6.1: En deterministisk tillståndsmaskin.

ritade som cirklar. Starttillståndet 1 är markerat med en pil som pekar på den, medan tillstånd 2 är accepterande, vilket markeras genom att den ritas med två cirklar.

Figuren innehåller tre pilar. En av dem utgår från tillstånd 1, och loopar runt och pekar på tillstånd 1. Den pilen är markerad med a. En annan pil går från tillstånd 1 till tillstånd 2, och är märkt med b. Den sista pilar loopar från

tillstånd 2 till sig självt, och är markerad med a och b.

Låt nu maskinen drivas av ordet *aababa*. Då kommer den stanna i tillstånd 1 i de två första stegen. Därefter kommer det aktiva tillståndet ändras från tillstånd 1 till tillstånd 2, varefter den stannar i detta tillstånd resten av körningen. Eftersom detta är ett accepterande tillstånd, så accepterar maskinen i Figur 6.1 ordet *aababa*.

I allmänhet gäller att maskinen accepterar alla ord över $\{a, b\}$ som innehåller minst ett b. Alla ord som någon gång drivs till tillstånd 2 kommer nämligen stanna det. De ord som inte drivs till tillstånd 2 är de som stannar i tillstånd 1 hela tiden, vilket bara kan hända ifall b ej förekommer i ordet. ▲

Enskilda tillståndsmaskiner studeras vanligtvis genom sin visuella representation, men för att studera dem i allmänhet krävs en formell definition.

Definition 6.1.2. En *deterministisk tillståndsmaskin* över ett alfabet Σ består av

(i) en ändlig mängd tillstånd Ω.

(ii) en övergångsfunktion δ från $\Omega \times \Sigma$ till Ω.

(iii) ett starttillstånd $S \in \Omega$.

(iv) en mängd accepterande tillstånd $F \subset \Omega$

Ifall $\delta(S_1, \sigma) = S_2$ säger man att det finns en σ-*övergång* från S_1 till S_2. △

Exempel 6.1.3. Tillståndsmaskinen i Figur 6.1 ges av $\Omega = \{1, 2\}$, $S = 1$ och $F = \{2\}$, och dess övergångsfunktion är ▲

$\Omega \times \Sigma$	$(1, a)$	$(1, b)$	$(2, a)$	$(2, b)$
$\delta(S, s)$	1	2	2	2

Tabell 6.2: Övergångsfunktionen för maskinen i Figur 6.1

Anmärkning 6.1.4. Ett annat sätt att beskriva övergångsfunktionen är i termer av tripplar (S_1, σ, S_2), där S_1 och S_2 är tillstånd i maskinen och σ är ett tecken i alfabetet. Trippeln (S_1, σ, S_2) tolkas som *ifall maskinen är i tillstånd S_1 och nästa tecken är σ, så är nästa tillstånd S_2*. Övergångsfunktionen i Exempel 6.1.3 ges av följande triplar.

Ω	1	1	2	2
Σ	a	b	a	b
Ω	1	2	2	2

Tabell 6.3: Övergångstripplar för maskinen i Exempel 6.1.1

Man kan således beskriva övergångsfunktionen som en delmängd Δ av $\Omega \times \Sigma \times \Omega$, med egenskapen att för varje par (S_1, σ) av tillstånd S_1 och tecken σ förekommer exakt en trippel på formen (S_1, σ, S_2) i Δ.

Notera att mängden av övergångstripplar är grafen av funktionen δ, enligt Definition 1.2.1.

Övergångsfunktionen är bara definierad på enskilda tecken. Vad händer när man låter ett ord driva maskinen, snarare än ett enskilt tecken? Det fångas av den utvidgade övergångsfunktionen.

Definition 6.1.5. Den *utvidgade övergångsfunktionen* δ^* från $\Omega \times \Sigma^*$ till Ω definieras rekursivt enligt följande:

$$\begin{cases} \delta^*(S, \varepsilon) = S \\ \delta^*(S, \sigma w) = \delta^*(\delta(S, \sigma), w). \end{cases}$$

Ordet w *driver* en maskin från S_1 till S_2 ifall $\delta^*(S_1, w) = S_2$. △

Tanken med definition är som följer. Låt w vara ett ord i Σ^* och S det tillstånd som maskinen startar i. Ifall w inte innehåller något tecken, är det tomt och maskinen stannar därför kvar i S.

Ifall w innehåller ett eller fler tecken så konsumerar maskinen av tecknet längst till vänster (σ i den rekursiva definitionen) och uppdaterar sitt nuvarande tillstånd till $\delta(S, \sigma)$. Därefter gör den om proceduren på det förkortade ordet.

Eftersom alla ord har ändlig längd, kommer maskinen efter ett ändligt antal steg konsumerat alla tecken, och då stannar den, eftersom $\delta^*(S, \varepsilon) = S$.

Exempel 6.1.6. Ifall $w = aabab$ och $S = 1$, så beräknas $\delta^*(S, w)$ för maskinen i Figur 6.1 enligt

$$\begin{aligned} \delta^*(S, w) &= \delta^*(1, aabab) = \delta^*(\delta(1, a), abab) \\ &= \delta^*(1, abab) = \delta^*(\delta(1, a), bab) \\ &= \delta^*(1, bab) = \delta^*(\delta(1, b), ab) \\ &= \delta^*(2, ab) = \delta^*(\delta(2, a), b) \\ &= \delta^*(2, b) = \delta^*(\delta(2, b), \varepsilon) \\ &= \delta^*(2, \varepsilon) = 2. \end{aligned}$$

Notera hur det vänstra argumentet i körningen uppdateras från 1 till 2 när man för första gången når b. ▲

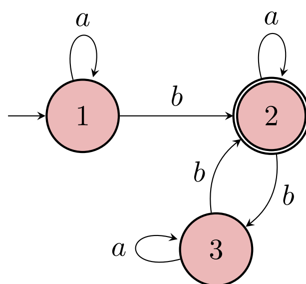
Definition 6.1.7. *Språket* för en deterministisk tillståndsmaskin M med starttillstånd s och accepterande tillstånd F är

$$L(M) = \{w \in \Sigma^* \mid \delta^*(s, w) \in F\}.$$

En deterministisk tillståndsmaskin *avgör* ett språk L ifall dess språk sammanfaller med L. △

Exempel 6.1.8. Maskinen i Exempel 6.1.1 avgör det reguljära språket $a^*b(a \cup b)^*$. ▲

Eftersom δ^* är en funktion, så driver varje ord maskinen till ett unikt tillstånd. Maskinens beteende är alltså förutbestämt, givet ordet som driver det och vilket tillstånd den startar i. Det är därför de kallas för deterministiska.



Figur 6.4: En deterministisk tillståndsmaskin med tre tillstånd.

Exempel 6.1.9. Tillståndsmaskinen i Figur 6.4 består av tre tillstånd, numrerade 1 till 3. Tillstånd 1 är ett starttillstånd och tillstånd 2 är accepterande. Från tillstånd 1 utgår två övergångar: en a-övergång som loopar tillbaka och en b-övergång till tillstånd 2. Från tillstånd 2 utgår en a-övergång som loopar tillbaka, och en b-övergång som går till tillstånd 3. Slutligen utgår två övergångar från tillstånd 3: en a-övergång som även den loopar tillbaka till tillstånd 3 och en b-övergång som går tillbaka till tillstånd 2.

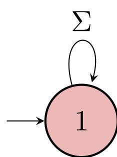
Låt maskinen drivas av ordet *ababaa*. Den kommer stanna i starttillståndet en tur och konsumera ett a. Sedan flyttar maskinen över till tillstånd 2 när den konsumerar ett b, där den stannar i en tur och konsumerar ett a. Därefter

Ord	*ababaa*	*babaa*	*abaa*	*baa*	*aa*	*a*	ε
Tillstånd	1	1	2	2	3	3	3

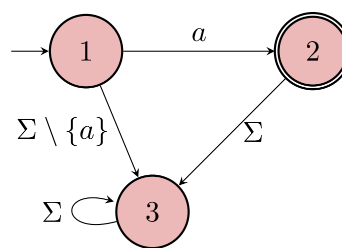
Tabell 6.5: Övergångar i Figur 6.4

flyttar den över till tillstånd 3 och konsumerar ett b. Där stannar den i två turer, och konsumerar de två sista a:na. Således driver ordet *ababaa* maskinen till tillstånd 3. Eftersom tillstånd 3 inte är accepterande, så ingår *ababaa* inte i maskinens språk. ▲

Exempel 6.1.10. Låt Σ vara ett alfabet och a ett tecken i Σ och betrakta tillståndsmaskinerna nedan. Den vänstra, Figur 6.6, består av ett starttillstånd



Figur 6.6: Tillståndsmaskin för $\emptyset$.



Figur 6.7: Tillståndsmaskin för $\{a\}$.

som är inte är accepterande, och där alla tecken loopar tillbaka starttillståndet. Denna maskin accepterar inget ord, eftersom den saknar accepterande tillstånd. Man kan uttrycka det som att den avgör det tomma språket.

Den högra, Figur 6.7, består av tre tillstånd. I övre vänstra hörnet av figuren finns tillstånd 1, som är ett icke-accepterande starttillstånd.

Två pilar utgår ifrån detta tillstånd, en märkt med a som går till ett accepterande tillstånd 2 till höger om tillstånd 1. Den andra pilen går snett ner till höger, till ett icke-accepterande tillstånd 3, och är markerade med $\Sigma \setminus \{a\}$. Slutligen går det en pil från tillstånd 2 till tillstånd 3, markerad med Σ.

Maskinen i Figur 6.7 avgör språket $\{a\}$. Ett sätt att se det är att notera att tillstånd 3 är ett icke-accepterande tillstånd ur vilket det saknas en pil som pekar på ett annat tillstånd. Inget ord som driver maskinen till det tillståndet ingår i maskinens språk. Tillstånd av denna typ kallas för *skräptillstånd.* ▲

För att bevisa egenskaper hos tillståndsmaskiner används ofta *induktion över ordens längd.* Tanken är att eftersom alla ord är ändligt långa, så är satsen

P gäller för alla ord

ekvivalent med satsen

för alla naturliga tal n, så gäller P för ord av längd n.

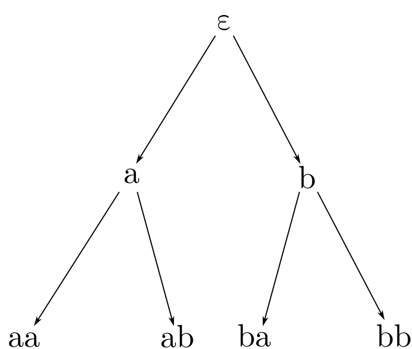
Rent konkret har induktionen två delar.

(i) Visa att påståendet gäller för ordet med längd 0, det vill säga det tomma ordet.

(ii) Visa att om påståendet gäller för alla ord av längd p, så gäller det för alla ord av längd $p + 1$.

I ett typiskt bevis är det första påståendet enkelt att bevisa, medan det andra kräver mer arbete. Tricket är att varje ord w av längd $p + 1$ kan skrivas som σu, där u är ett ord av längd p och σ är ett tecken i alfabetet.

Enligt induktionsantagandet gäller påståendet för u, eftersom det har längd p. För att visa att påståendet gäller allmänt, räcker det därför att visa att påståendet fortsätter vara sant, även när man lägger till ett tecken. Ett sätt



Figur 6.8: Korta ord över $\{a, b\}$.

att tänka är att orden över ett alfabet kan ordnas i nivåer, baserat på hur långa de är. Då säger implikationen i induktionssteget att om alla ord i en nivå uppfyller en viss egenskap, så ska även alla ord på nästa nivå göra det.

I Figur 6.8 så innebär det att ifall påståendet gäller för andra nivån (uppifrån), så ska det även gälla för alla ord på nivån längst ner. Rent konkret: ifall det gäller för a och b, måste det även gäller för aa, ab, ba och bb.

Följande sats illustrerar hur tekniken fungerar, och används dessutom senare i kompendiet.

Sats 6.1.11. *Ifall w och u är ord och δ^* är den utvidgade övergångsfunktionen för en tillståndsmaskin, så gäller att*

$$\delta^*(S, wu) = \delta^*(\delta^*(S, w), u)$$

för alla tillstånd S.

Bevis. Ifall w har längd 0, det vill säga $w = \varepsilon$, så gäller att

$$\delta^*(S, wu) = \delta^*(S, u) = \delta^*(\delta^*(S, \varepsilon), u) = \delta^*(\delta^*(S, w), u).$$

Alltså stämmer satsen i basfallet. Antag nu att för alla ord v av längd p gäller att

$$\delta^*(S, vu) = \delta^*(\delta^*(S, v), u).$$

Låt w vara ett ord av längd $p+1$. Då kan man skriva $w = \sigma v$, där σ är ett tecken i alfabetet och v är ett ord av längd p. Per definition och induktionsantagande får man att

$$\begin{aligned}
\delta^*(S, wu) &= \delta^*(S, \sigma v u) && \text{(definition av } w\text{)} \\
&= \delta^*(\delta^*(S, \sigma), vu) && \text{(definition av } \delta^*\text{)} \\
&= \delta^*(\delta^*(\delta^*(S, \sigma), v), u) && \text{(induktionsantagande)} \\
&= \delta^*(\delta^*(S, \sigma v), u) && \text{(definition av } \delta^*\text{)} \\
&= \delta^*(\delta^*(S, w), u) && \text{(definition av } w\text{)}
\end{aligned}$$

□

Notera att den rekursiva definitionen av δ^* gör det möjligt att använda induktionsantagandet på ett enkelt sätt.

6.2 Icke-deterministiska tillståndsmaskiner

De tillståndsmaskiner som behandlas i föregående kapitel var deterministiska. Givet ett ord och ett starttillstånd är det helt bestämt hur maskinen drivs av ordet. I detta kapitel presenteras en generalisering av deterministiska tillståndsmaskiner, så kallade *icke-deterministiska tillståndsmaskiner*. Mer specifik införs följande möjligheter.

(i) Maskinen kan byta tillstånd utan att konsumera tecken. Man säger att den konsumerar det tomma ordet, eller att det finns en *ε-övergång*.

(ii) Maskinen kan det finnas flera övergångar för samma tecken ur samma tillstånd.

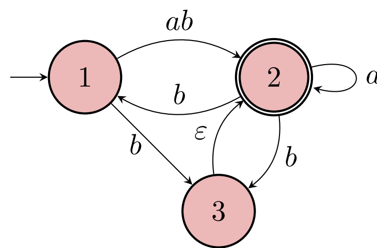
Dessa öppnar upp för icke-determinism, det vill säga att maskinens beteende inte är förutbestämt. Utöver detta införs även två ytterligare egenskaper.

(i) Maskinen kan konsumera flera tecken åt gången i en övergång. Detta kallas att övergången är *glupsk*.

(ii) Maskinen kan *hänga sig* på ett ord, vilket innebär att ett ord inte driver maskinen till något tillstånd alls.

Till skillnad från deterministiska tillståndsmaskiner så finns det inte ett unikt sätt som ett ord kan driva en icke-deterministisk tillståndsmaskin på. Man kan istället tänka att övergångarna representerar *möjligheter*. Precis som en slantsingling kan leda till antingen krona eller klave, kan ett ord driva en icke-deterministisk tillståndsmaskin till noll, en eller flera tillstånd.

Exempel 6.2.1. Maskinen i Figur 6.9 består av tre tillstånd: 1, 2 och 3. Starttillstånd är 1 och det accepterande tillståndet är 2. Från tillstånd 1 utgår två övergångar: en som går till tillstånd 2 och är märkt med ab, och en går till tillstånd 3 och är märkt med b. Det finns alltså ingen övergång från tillstånd 1



Figur 6.9: En icke-deterministisk tillståndsmaskin.

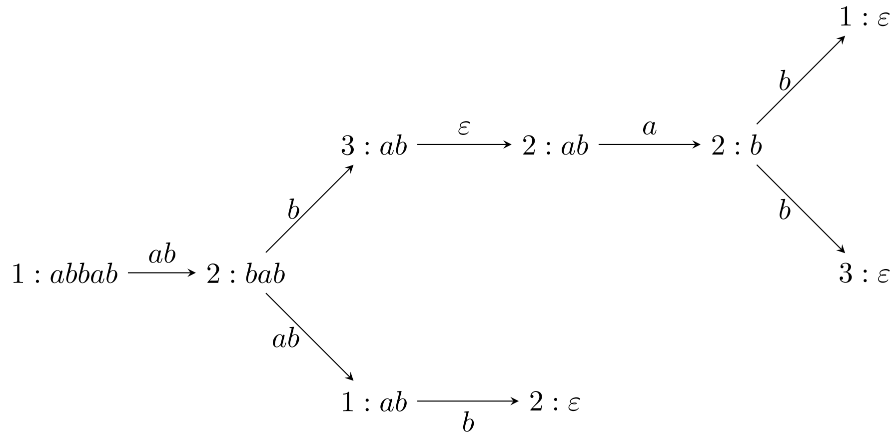
som konsumerar endast a, utan enbart en märkt med b och en glupsk som är märkt med ab. Konsekvensen är att maskinen kommer hänga sig i tillstånd 1, om ordet börjar på aa.

Från tillstånd 2 utgår två övergångar märkt med b: en till tillstånd 1 och en till tillstånd 3. Det går dessutom en a-övergång från tillstånd 2 till sig själv. Från tillstånd 3 så utgår endast en ε-övergång till tillstånd 2.

Observera att maskinen kan inte hänga sig i tillstånd 3, eftersom maskinen alltid kan använda den tomma övergången och gå till tillstånd 2, och därefter konsumera antingen a eller b.

Betrakta ordet $abbab$. I starttillståndet så måste maskinen konsumera ab och flytta till tillstånd 2. Det ord som är kvar är bab, och då finns det två alternativ. Antingen går maskinen tillbaka till tillstånd 1, eller så går den vidare till tillstånd 3.

I det förra fallet, återstår ab och det finns bara en möjlighet: att återgå till tillstånd 2 och konsumera ab. Detta avslutar körningen. I den senare fallet går maskinen vidare till tillstånd 3 och återigen återstår ab. Här måste maskinen använda den tomma övergången till tillstånd 2 och där konsumera ett a. Då står maskinen i tillstånd 2 med ett b, och har återigen två val: antingen gå



Figur 6.10: Möjliga körningar av *abbab* i maskinen i Figur 6.9.

till tillstånd 1 eller tillstånd 3. Oavsett vad maskinen väljer, så konsumerar övergången den sista det sista tecknet, vilket avslutar körningen. ▲

Formellt definieras icke-deterministiska tillståndsmaskiner som följer.

Definition 6.2.2. En *icke-deterministisk tillståndsmaskin* över ett alfabet Σ består av

(i) en ändlig mängd tillstånd Ω.

(ii) en ändlig delmängd Δ ur $\Omega \times \Sigma^* \times \Omega$, som kallas för *övergångsrelationen.*

(iii) en icketom delmängd starttillstånd $S \subset \Omega$.

(iv) en mängd accepterande tillstånd $F \subset \Omega$.

△

Övergångsrelationen tolkas på följande sätt: ifall en trippel (S_1, w, S_2) ingår i övergångsrelationen så finns det en w-övergång från S_1 till S_2.

Exempel 6.2.3. Tillståndsmaskinen i Exempel 6.2.1 fås genom att sätta Ω lika med $\{1, 2, 3\}$, $S = \{1\}$ och $F = \{2\}$. Övergångsrelationen Δ ges i Tabell 6.11.

Ω	Σ^*	Ω
1	ab	2
1	b	3
2	a	2
2	b	1
2	b	3
3	ε	2

Tabell 6.11: Övergångsrelationen för tillståndsmaskinen i Exempel 6.2.1.

▲

Deterministiska tillståndsmaskiner är ett specialfall av icke-deterministiska, där alla ord i övergångstripplarna är tecken och där det för varje par (S_1, w) av tillstånd S_1 och tecken σ finns precis ett tillstånd S_2 så att (S_1, w, S_2) ligger i övergångsrelationen.

Definition 6.2.4. Den *utvidgade övergångsrelationen* Δ^* är en delmängd av $\Omega \times \Sigma^* \times \Omega$ som definieras genom följande rekursion.

$$\begin{cases} (S, \varepsilon, S) \in \Delta^* & \text{för varje } S \in \Omega \\ (S_1, uw, S_3) \in \Delta^* & \text{ifall } (S_1, u, S_2) \in \Delta \text{ och } (S_2, w, S_3) \in \Delta^*. \end{cases}$$

Ett ord w *driver* en icke-deterministisk tillståndsmaskin från S_1 till S_2 om (S_1, w, S_2) ingår i maskinens utvidgade övergångsrelation. △

Notera likheten mellan ovanstående definition och definitionen av den utvidgade övergångsfunktionen för en deterministisk tillståndsmaskin.

Givet ett tillstånd S_1 och ett ord w så är det inte säkert att det finns ett tillstånd S_2 så att (S_1, w, S_2) ingår i den utvidgade övergångsrelationen. Detta är innebörden av att maskinen hänger sig på ordet w.

Det kan även vara så att det finns flera tillstånd S_2 så att (S_1, w, S_2) ingår i den utvidgade övergångsrelationen. Detta tolkas som att beräkningen inte är förutbestämd.

Definition 6.2.5. *Språket* för en icke-deterministisk tillståndsmaskin M med starttillstånd S och accepterande tillstånd F är

$$L(M) = \{w \in \Sigma^* \mid \text{det finns } S_1 \in S \text{ och } S_2 \in F \text{ så att } (S_1, w, S_2) \in \Delta^*\},$$

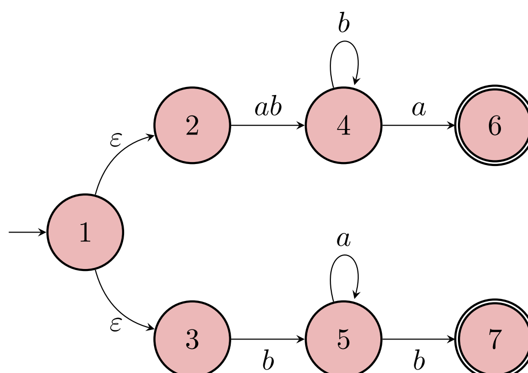
det vill säga alla ord som kan driva maskinen från ett starttillstånd till ett accepterande tillstånd. Man säger att maskinen *avgör* språket $L(M)$. △

Poängen med icke-deterministiska tillståndmaskiner är att de är enklare att konstruera än deterministiska, eftersom de bättre speglar det sätt på vilket människor beskriver språk och mönster.

Exempel 6.2.6. I Figur 6.12 ges en icke-deterministisk tillståndsmaskin som avgör $abb^*a \cup ba^*b$. Det består av sju tillstånd, numrerade från 1 till 7. Tillstånd 1 är längst till vänster och är ett starttillstånd. Ur detta leder två pilar till höger, den ena snett upp och den andra snett ner, till två tillstånd 2 och 3. Båda pilarna är märkta med ε. Från tillstånd 2 går en pil ut till höger till ett nytt tillstånd 4. Pilen är märkt med ab. Från tillstånd 4 utgår två pilar. Den ena är märkt med b och loopar tillbaka till sig själv. Den andra leder till ett accepterande tillstånd 6, som ligger till höger om tillstånd 4.

Från tillstånd 3 går en pil till ett nytt tillstånd 5 till höger om tillstånd 3. Pilen är märkt med b. Från tillstånd 5 utgår, liksom från tillstånd 4 två pilar. Den ena är märkt med a och loopar tillbaka till tillstånd 5. Den andra är märkt med b och går ut till ett accepterande tillstånd 7, som är beläget till höger om tillstånd 5.

Det går inte några pilar ur de accepterande tillstånden. Detta innebär att ett ord måste vara tomt när det kommer dit för att accepteras.



Figur 6.12: En tillståndsmaskin som avgör $abb^*a \cup ba^*b$.

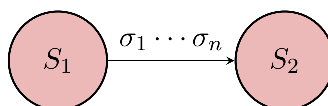
Ett ord driver maskinen till ett accepterande sluttillstånd på två sätt. Antingen börjar ordet på ab följt av ett godtyckligt antal b:n och ett avslutande a. Eller så börjar ordet på b, följt av ett godtyckligt antal a:n innan ett avslutande b. Med andra ord kommer maskinen acceptera orden i språket $abb^*a \cup ba^*b$. ▲

6.3 Delmängdskonstruktionen

I förstone kunde man tro att icke-deterministiska tillståndsmaskiner är kraftfullare sina deterministiska motsvarigheter, i bemärkelsen att de skulle kunna avgöra språk som inte deterministiska maskiner kan.

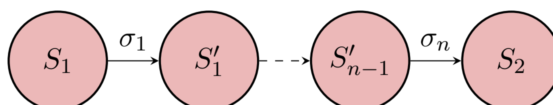
Så är dock inte fallet. För varje icke-deterministiska tillståndsmaskin finns en deterministisk dito som avgör samma språk. Det finns även en generell metod för att konstruera den, som kallas för *delmängskonstruktionen*. Detta avsnitt handlar om att beskriva den och bevisa att den fungerar.

Metoden tillämpas på icke-deterministiska tillståndsmaskine utan glupska övergångar. Därför eliminerar man glupska övergångar genom att införa mellanliggande tillstånd med en ingående och en utgående pil motsvarande tecknen i ordet. Rent formellt: Ifall (S_1, w, S_2) ligger i övergångsrelationen och $w =$



Figur 6.13: Glupsk övergång.

$\sigma_1 \cdots \sigma_n$ med är ett ord med minst två tecken, så inför man nya tillstånd S'_1, $\ldots$, S'_{n-1} samt övergångstripplar (S_1, σ_1, S'_1), (S'_1, σ_2, S'_2), $\ldots$, $(S'_{n-1}, \sigma_n, S_2)$ i övergångsrelationen. Poängen är att ett ord enbart kan driva maskinen genom



Figur 6.14: Oglupsk övergång motsvarande den i Figur 6.13.

samtliga mellanliggande tillstånd ifall den är i tillstånd S_1 och har ordet w som

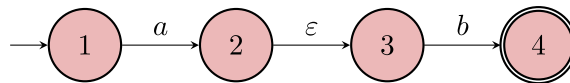
prefix.

Definition 6.3.1. Låt w vara ett godtyckligt ord över alfabetet och X en mängd av tillstånd i en icke-deterministisk tillståndsmaskin. Då består *w-tillslutningen* $w(X)$ av samtliga tillstånd som kan nås från ett tillstånd i X via konsumption av w. Rent formellt:

$$w(X) = \{S_2 \mid \text{ det finns } S_1 \text{ i } X \text{ så att } (S_1, w, S_2) \in \Delta^*\}.$$

△

Exempel 6.3.2. Maskinen i Figur 6.15 består av fyra tillstånd, numrerade 1 till 4 från vänster till höger. Tillstånd 1 är ett starttillstånd och 4 är ett accepterande tillstånd. Från tillstånd 1 går en pil till tillstånd 2, märkt med a. Från tillstånd 2 går en ε-övergång till tillstånd 3. Slutligen går det en b-övergång från tillstånd 3 till tillstånd 4. Här är a-tillslutningen av $\{1\}$ lika



Figur 6.15: En liten tillståndsmaskin.

med $\{2, 3\}$, medan b-tillslutningen av samma mängd blir den tomma mängden och ab-tillslutningen $\{4\}$. Mängden $\{2, 3\}$ har b-tillslutningen $\{4\}$, medan dess a-tillslutning är $\emptyset$. ▲

Anmärkning 6.3.3. Ifall S är ett tillstånd, så ligger (S, ε, S) ligger i Δ^* per definition. Därför är X en delmängd av $\varepsilon(X)$.

En konsekvens av detta är att $\varepsilon(w(X)) = w(X)$ för alla ord w. Varför? Enligt ovanstående resonemang är $w(X) \subseteq \varepsilon(w(X))$. För att visa att $\varepsilon(w(X)) \subseteq w(X)$, låt S_1 ligga i $\varepsilon(w(X))$. Då finns ett tillstånd S_2 i $w(X)$ så att $(S_2, \varepsilon, S_1) \in \Delta^*$.

Men ifall S_2 ligger i $w(X)$, så finns det S_3 i X så att (S_3, w, S_2) ligger Δ^*. Enligt den rekursiva definitionen av Δ^* gäller då att

$$(S_3, \varepsilon w, S_1) = (S_3, w, S_1) \in \Delta^*.$$

Alltså ligger S_3 i $w(X)$, vilket bevisar att $\varepsilon(w(X))$ är en delmängd av $w(X)$ och alltså att $\varepsilon(w(X)) = w(X)$.

Delmängdskonstruktionen är en deterministisk tillståndsmaskin som konstrueras utifrån en given icke-deterministisk maskin. Namnet kommer av att tillstånden är delmängder av tillstånd från den icke-deterministiska maskinen.

Algoritm 6.3.4. Delmängdskonstruktionen beräknas genom följande algoritm.

(i) Starttillståndet i den nya maskinen är mängden

$$s = \varepsilon(S),$$

det vill säga alla tillstånd som det tomma ordet kan driva maskinen till.

(ii) Välj ett tillstånd X i den nya maskinen från vilket det inte går några övergångar.

(iii) För varje σ i alfabetet, beräkna σ-tillslutningen av X. Om $\sigma(X)$ inte finns i maskinen, lägg till det.

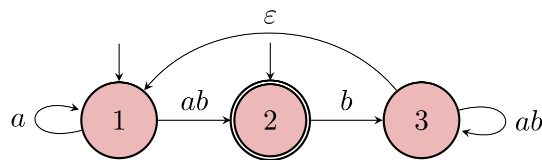
(iv) Lägg till en σ-övergång från X till $\sigma(X)$.

(v) Upprepa (ii)-(iv) tills det inte finns några tillstånd utan övergångar.

De accepterande tillstånden i den nya maskinen är alla tillstånd som innehåller minst ett accepterande tillstånd från den gamla maskinen.

Eftersom antalet tillstånd i en icke-deterministisk tillståndsmaskin är ändligt, så är antalet möjliga delmängder av tillstånd ändliga. Därför kommer Algoritm 6.3.4 efter ett tag sluta generera nya tillstånd. Att maskinen som konstrueras i algoritmen är deterministisk följer av konstruktionen: för varje tillstånd och tecken så lägger man till precis en övergång till en ny mängd.

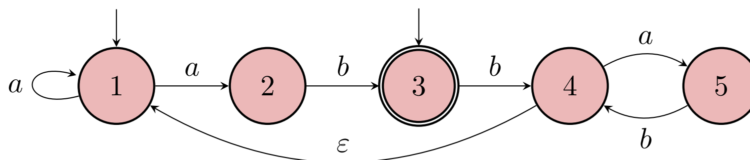
Exempel 6.3.5. Maskinen i Figur 6.16 består av tre tillstånd på rad, numrerade 1 till tillstånd 3 från höger till vänster. Tillstånd 1 och tillstånd 2 är starttillstånd, och tillstånd 2 är accepterande. Från tillstånd 1 utgår två över-



Figur 6.16: En glupsk icke-deterministisk tillståndsmaskin

gångar: en a-övergång som loopar tillbaka till tillstånd 1, och en ab-övergång som går till tillstånd 2. Från tillstånd 2 går en b-övergång till tillstånd 3. Från tillstånd 3 går en ε-övergång tillbaka till tillstånd 1, och en ab-övergång som loopar tillbaka till tillstånd 3.

Eftersom maskinen är glupsk, så börjar man med att lägga till två tillstånd så att maskinen inte längre är glupsk. Tillstånden döps om för att reflektera detta. Det lättaste sättet att beräkna delmängdskonstruktionen är att göra



Figur 6.17: En oglupsk icke-deterministisk tillståndsmaskin.

en tabell. Kolonnen längst till vänster innehåller delmängder av tillstånd, och sedan är det en kolonn per tecken i alfabetet.

Tanken är att rad för rad beräkna σ-utvidningen av delmängderna längst till vänster. Varje gång man får en delmängd som inte står i kolonnen längst till

vänster, lägger man till den längst ner. Denna process upprepas tills dess att det inte finns några tomma rader.

Beräkningen för den icke-deterministiska maskinen i Figur 6.17 är som följer. Resultatet återfinns i Tabell 6.18.

(i) Starttillståndet är $\{1, 3\}$, då dessa är starttillstånd i den ursprungliga tillståndsmaskinen och det saknas ε-övergångar från dem.

(ii) Från starttillståndet så beräknar man

$$a(\{1, 3\}) = \{1, 2\}$$

eftersom den gamla maskinen saknar a-övergångar ur 3 och har två a-övergångar ur 1.

På liknande sätt får man att $b(\{1, 3\}) = \{1, 4\}$, genom att man antingen går från 3 till 4, eller från 3 till 4 till 1 via ε-övergången.

Båda dessa delmängder är nya läggs som nya rader under $\{1, 3\}$.

(iii) Nästa steg är att beräkna a- och b-tillslutningarna av $\{1, 2\}$, vilka är $\{1, 2\}$ och $\{3\}$. Den senare är ny och läggs till längst ner i tabellen.

(iv) När man beräknar tillslutningarna av $\{1, 4\}$ får man återigen två nya delmängder, $\{1, 2, 5\}$ och $\emptyset$, som läggs till i tabellen. Den tomma mängden uppkommer eftersom det inte finns några b-övergångar ur varken tillstånd 1 eller tillstånd 4.

(v) Tillslutningarna av $\{3\}$ är $\emptyset$ och $\{1, 4\}$. Ingen av dessa är nya.

(vi) I nästa steg beräknas tillslutningarna av $\{1, 2, 5\}$, vilka är $\{1, 2\}$ och $\{1, 3, 4\}$. Den senare är ny, och läggs till längst ner i tabellen.

(vii) Den tomma mängden har tomma tillslutningar per definition.

(viii) Slutligen så är tillslutningarna av $\{1, 3, 4\}$ mängderna $\{1, 2, 5\}$ och $\{1, 4\}$. Ingen av dessa är ny.

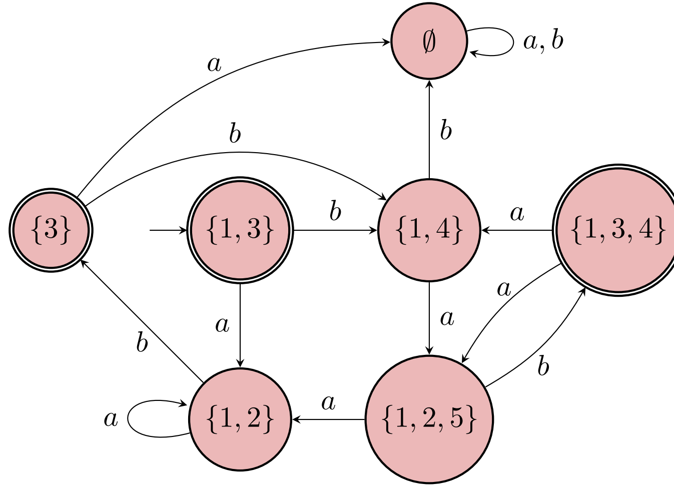
(ix) Inga tomma rader återstår, och algoritmen är klar.

De accepterande tillstånden i den nya maskinen är de som innehåller ett accepterande tillstånd från den gamla maskinen: i detta fall delmängderna $\{1, 3\}$, $\{3\}$ och $\{3, 5\}$. Den nya maskinen har alltså tre accepterande tillstånd.

Delmängder	a	b
$\{1, 3\}$	$\{1, 2\}$	$\{1, 4\}$
$\{1, 2\}$	$\{1, 2\}$	$\{3\}$
$\{1, 4\}$	$\{1, 2, 5\}$	$\emptyset$
$\{3\}$	$\emptyset$	$\{1, 4\}$
$\{1, 2, 5\}$	$\{1, 2\}$	$\{1, 3, 4\}$
$\emptyset$	$\emptyset$	$\emptyset$
$\{1, 3, 4\}$	$\{1, 2, 5\}$	$\{1, 4\}$

Tabell 6.18: Delmängdskonstruktionen för tillståndsmaskinen i Figur 6.17.

I Tabell 6.18 återfinns resultatet. Figur 6.19 är en visualisering av tillståndsmaskinen. ▲



Figur 6.19: Den deterministiska tillståndsmaskin som ges i Tabell 6.18.

Anmärkning 6.3.6. Låt σ vara ett tecken i alfabetet. Eftersom $\sigma(\emptyset) = \emptyset$, uppfyller den utvidgade övergångsfunktionen i delmängdskonstruktionen att

$$\delta^*(\emptyset, w) = \emptyset$$

för alla ord w. Den tomma mängden utgör alltså ett skräptillstånd i delmängdskonstruktionen. Dessutom gäller att

$$\delta(X, \sigma) = \sigma(X)$$

för alla tillstånd X i maskinen. Som brukligt definieras den utvidgade övergångsfunktionen genom rekursionen

$$\begin{cases} \delta^*(X, \varepsilon) = X \\ \delta^*(X, \sigma w) = \delta^*(\delta(X, \sigma), w) = \delta^*(\sigma(X), w). \end{cases}$$

Detta har följande konsekvens.

Sats 6.3.7. *För alla tillstånd X i delmängdskonstruktionen och ord w gäller att $\delta^*(X, w) = w(X)$*

Bevis. Induktion över längden på w. När $w = \varepsilon$, gäller att $w(X) = \varepsilon(X)$ och $\delta^*(X, w) = X$. Det som behöver visas är alltså att $\varepsilon(X) = X$.

Eftersom alla tillstånd i delmängdskonstruktionen är på formen $u(X_1)$ där X_1 är en mängd av tillstånd i den gamla maskinen och u antingen är det tomma ordet eller ett enskilt tecken, så gäller enligt Anmärkning 6.3.3 att

$$X = w(X_1) = \varepsilon(w(X_1)) = \varepsilon(X).$$

Alltså är $\delta^*(X, w) = X = \varepsilon(X) = w(X)$.

Antag nu att för alla ord w av längd p, så gäller $\delta^*(X, w) = w(X)$ för alla tillstånd X i delmängdskonstruktionen. Låt σw vara ett ord av längd $p + 1$. Då gäller att

$$\delta^*(X, \sigma w) = \delta^*(\delta^*(X, \sigma), w) = w(\delta^*(X, \sigma)) = w(\sigma(X))$$

enligt Anmärkning 6.3.6.

Det återstår att bevisa att $w(\sigma(X)) = \sigma w(X)$. Vad innebär detta? De är båda mängder av tillstånd, så för att visa att de är lika behöver man visa att

$$\sigma w(X) \subseteq w(\sigma(X)) \quad \text{och} \quad w(\sigma(X)) \subseteq \sigma w(X).$$

Låt T_1 ligga i mängden $\sigma w(X)$. Då finns det ett tillstånd T_2 i mängden X så att $(T_2, \sigma w, T_1)$ ligger i den utvidgade övergångsrelationen Δ^*. Men enligt Definition 6.2.4 av den utvidgade övergångsrelationen finns det då ett tillstånd T_3 så att

$$(T_2, \sigma, T_3) \in \Delta^* \text{ och } (T_3, w, T_1) \in \Delta^*.$$

Per Definition 6.3.1 ligger T_3 i $\sigma(X)$, eftersom T_2 ligger i X, och således ligger T_1 i $w(\sigma(X))$. Detta bevisar att $\sigma w(X)$ är en delmängd av $w(\sigma(X))$.

Antag nu att S_1 ligger i $w(\sigma(X))$. Då finns det ett tillstånd S_2 i $\sigma(X)$ så att (S_2, w, S_1) ligger i den utvidgade övergångsrelationen Δ^*. Men eftersom S_2 ligger i $\sigma(X)$, så finns det ett tillstånd S_3 i X så att (S_3, σ, S_2) ligger i Δ^*.

Sammantaget gäller alltså att

$$(S_3, \sigma, S_2) \in \Delta^* \text{ och } (S_2, w, S_1) \in \Delta^*.$$

Men enligt Definition 6.2.4 av Δ^* så innebär detta att $(S_3, \sigma w, S_1)$ ligger i Δ^*. Detta innebär att S_1 ligger i $\sigma w(X)$, och alltså är det bevisat att $w(\sigma(X))$ är en delmängd av $\sigma w(X)$.

Sammantaget gäller alltså att

$$\sigma w(X) \subseteq w(\sigma(X)) \quad \text{och} \quad w(\sigma(X)) \subseteq \sigma w(X),$$

vilket bevisar att $\sigma w(X) = w(\sigma(X))$, och alltså att

$$\delta^*(X, \sigma w) = w(\sigma(X)) = \sigma w(X),$$

vilket avslutar induktionen. □

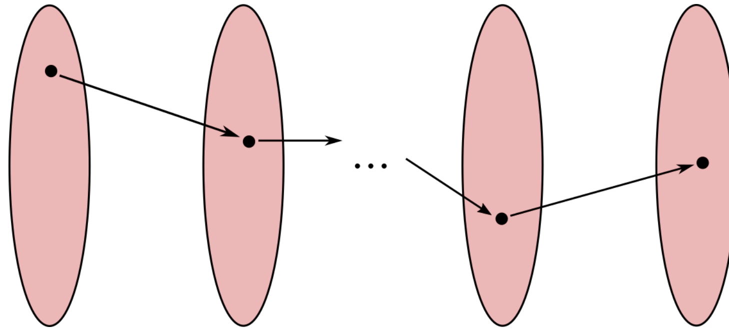
Ovanstående sats innebär att ett ord w driver delmängdskonstruktionen av en tillståndsmaskin till mängden av alla tillstånd som är w kan driva den underliggande tillståndsmaskinen till. Det har följande konsekvens.

Sats 6.3.8. *Delmängdskonstruktionen av en tillståndsmaskin avgör samma språk som den ursprungliga tillståndmaskinen.*

Bevis. Ett ord accepteras av delmängdskonstruktionen av en tillståndsmaskin om och endast om den drivs till ett tillstånd som innehåller ett accepterande tillstånd.

Enligt Sats 6.3.7 är detta ekvivalent med att ordet kan driva den underliggande maskinen till ett accepterande tillstånd, det vill säga att den underliggande maskinen accepterar ordet. □

Figur 6.20 illustrerar Sats 6.3.7. Ellipserna är tillstånd i delmängdskonstruktionen, medan punkterna inuti ellipserna är tillstånd i den underliggande maskinen. Pilarna mellan punkterna är övergångar i den underliggande maskinen.



Figur 6.20: Illustration av Sats 6.3.7.

Övningar

Övning 6.1. Beskriv hur ordet *aaabba* driver maskinen i Figur 6.1 i Exempel 6.1.1.

Övning 6.2. Beskriv hur ordet *baaaba* driver maskinen i Figur 6.4 i Exempel 6.1.9.

Övning 6.3. Konstruera en deterministisk tillståndsmaskin för språket *bab*.

Ange antingen tillståndsmängden, övergångsfunktionen, de accepterande tillstånden och starttillstånden, eller en grafisk representation.

Övning 6.4. Konstruera en deterministisk tillståndsmaskin för språket som består av ett jämnt antal a:n följt av ett udda antal b:n.

Ange antingen tillståndsmängden, övergångsfunktionen, de accepterande tillstånden och starttillstånden, eller en grafisk representation.

Övning 6.5. Konstruera en deterministisk tillståndsmaskin för det reguljära språket $(aa \cup bb)^*$.

Ange antingen tillståndsmängden, övergångsfunktionen, de accepterande tillstånden och starttillstånden, eller en grafisk representation.

Övning 6.6. Konstruera en deterministisk tillståndsmaskin för språket över $\{0, 1\}$ som består av alla ord som representerar binära tal som är jämnt delbara med 4.

Ange antingen tillståndsmängden, övergångsfunktionen, de accepterande tillstånden och starttillstånden, eller en grafisk representation.

Övning 6.7. Konstruera en deterministisk tillståndsmaskin för det reguljära språket $(a \cup b)^*ab$ över $\{a, b\}$

Ange antingen tillståndsmängden, övergångsfunktionen, de accepterande tillstånden och starttillstånden, eller en grafisk representation.

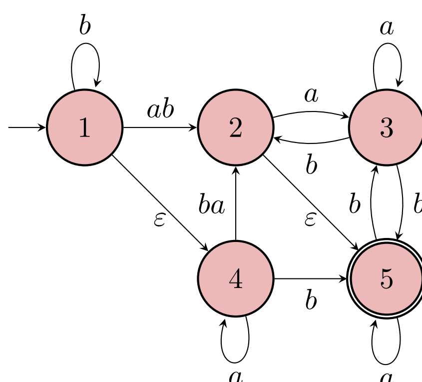
Övning 6.8. Konstruera en deterministisk tillståndsmaskin för det reguljära språket $(aa)^*b(a \cup b)^*a$ över $\{a, b\}$

Ange antingen tillståndsmängden, övergångsfunktionen, de accepterande tillstånden och starttillstånden, eller en grafisk representation.

Övning 6.9. Betrakta den icke-deterministiska maskinen som har tillståndsmängd $\{1,2,3,4,5\}$ och övergångsrelation.

Ω	1	1	1	2	2	3	3	3	4	4	4	5	5
Σ^*	b	ab	ε	ε	a	a	b	b	a	b	ab	a	b
Ω	1	2	4	5	3	3	2	5	4	5	2	5	3

Starttillståndet är 1 och det accepterande tillståndet är 5. En grafisk representation ges nedan



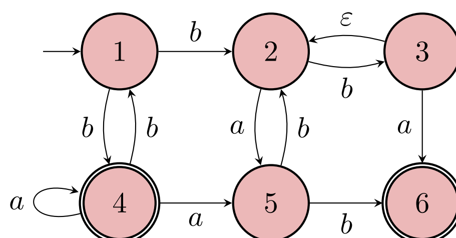
Figur 6.21: En icke-deterministisk tillståndsmaskin.

Beräkna samtliga sätt som ordet *abab* kan driva maskinen på. Ingår *abab* i maskinens språk?

Övning 6.10. Betrakta den icke-deterministiska maskinen som har tillståndsmängd $\{1,2,3,4,5,6\}$ och övergångsrelation

Ω	1	1	2	2	3	3	4	4	4	5	5
Σ^*	b	b	b	a	ε	a	b	a	a	b	b
Ω	2	4	3	5	2	6	1	4	5	5	6

Starttillstånden är 1 och 3, och de accepterande tillstånden är 4 och 6. En grafisk representation ges nedan



Figur 6.22: En icke-deterministisk tillståndsmaskin.

Beräkna samtliga sätt som ordet *bab* kan driva maskinen på. Ingår *bab* i maskinens språk?

Övning 6.11. Konstruera en icke-deterministisk tillståndsmaskin med glupska övergångar för språket $(ba)^*a \cup b^*a$.

Ange antingen tillståndsmängden, övergångsfunktionen, de accepterande tillstånden och starttillstånden, eller en grafisk representation.

Övning 6.12. Konstruera en icke-deterministisk tillståndsmaskin med glupska övergångar för språket $(aa \cup bb)^*ab$.

Ange antingen tillståndsmängden, övergångsfunktionen, de accepterande tillstånden och starttillstånden, eller en grafisk representation.

Övning 6.13. Betrakta den icke-deterministiska maskinen i Figur 6.21 i Övning 6.9. Beräkna

(i) ε-tillslutningen av $\{1, 2, 3\}$.

(ii) aab-tillslutningen av $\{3, 5\}$.

(iii) ba-tillslutningen av $\{1, 3, 4\}$.

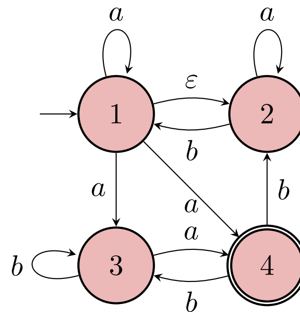
Övning 6.14. Betrakta den icke-deterministiska maskinen i Figur 6.22 i Övning 6.10. Beräkna

(i) ab-tillslutningen av $\{2, 6\}$.

(ii) aba-tillslutningen av $\{3\}$.

(iii) aa-tillslutningen av $\{2, 4\}$.

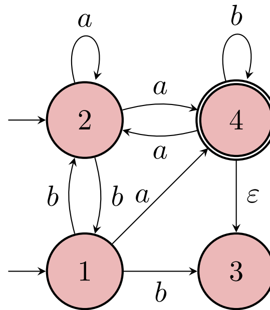
Övning 6.15. Använd delmängdskonstruktionen för att finna en deterministisk tillståndsmaskin som avgör samma språk som nedanstående icke-deterministiska tillståndsmaskin.



Tillståndsmängden är $\{1, 2, 3, 4\}$, starttillståndet är 1, det accepterande tillståndet är 4 och övergångsrelationen ges av

Ω	1	1	1	1	2	2	3	3	4	4
Σ^*	a	a	a	ε	a	b	a	b	b	b
Ω	1	3	4	2	2	1	4	3	2	3

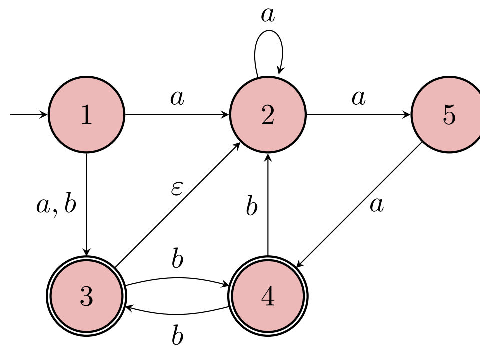
Övning 6.16. Använd delmängdskonstruktionen för att finna en deterministisk tillståndsmaskin som avgör samma språk som nedanstående icke-deterministiska tillståndsmaskin.



Tillståndsmängden är $\{1, 2, 3, 4\}$, starttillståndet är 1, det accepterande tillstånden är 4 och övergångsrelationen ges av

Ω	1	1	1	2	2	2	4	4	4
Σ^*	a	b	b	a	a	b	a	b	ε
Ω	4	2	3	2	4	1	2	4	3

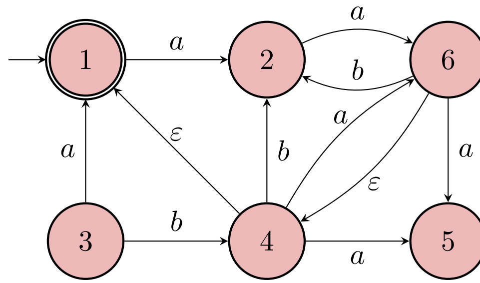
Övning 6.17. Använd delmängdskonstruktionen för att finna en deterministisk tillståndsmaskin som avgör samma språk som nedanstående icke-deterministiska tillståndsmaskin.



Tillståndsmängden är $\{1, 2, 3, 4, 5\}$, starttillståndet är 1, de accepterande tillstånden är 3 och 4 och övergångsrelationen ges av

Ω	1	1	1	2	2	3	3	4	4	5
Σ^*	a	a	b	a	a	ε	b	b	b	a
Ω	2	3	3	2	5	2	4	3	2	4

Övning 6.18. Använd delmängdskonstruktionen för att finna en deterministisk tillståndsmaskin som avgör samma språk som nedanstående icke-deterministiska tillståndsmaskin.



Tillståndsmängden är $\{1, 2, 3, 4, 5, 6\}$, starttillståndet är 1, det accepterande tillståndet är 1 och övergångsrelationen ges av

Ω	1	2	3	4	4	3	3	4	6	6
Σ^*	a	a	a	b	ε	b	a	a	a	ε
Ω	2	6	1	2	1	4	1	6	5	4

7 Tillståndsmaskinernas språk

I början av föregående kapitel nämndes att tillståndsmaskiner räcker för att avgöra de reguljära språken. Men faktum är att även det omvända gäller: alla språk som kan avgöras av en tillståndsmaskin kan beskrivas genom ett reguljärt uttryck.

De första två avsnitten i detta kapitel ägnas åt att bevisa denna ekvivalens. Det sista avsnittet ägnas åt ett annat sätt att betrakta reguljära språk, genom särskiljning av ord. Detta ger en tredje karakterisering av de reguljära språken.

7.1 Reguljära språk avgörs av tillståndsmaskiner

Syftet med detta avsnitt är att bevisa följande sats.

Sats 7.1.1. *Alla reguljära språk kan avgöras av en deterministisk tillståndsmaskin.*

Satsen bevisas återigen genom induktion. Men den här gången används ett mer avancerat argument än tidigare, som kräver lite förklaring.

Alla reguljära språk skapas genom att börja från ett eller flera basfall (det tomma språket eller ett språk som består av ett tecken), och sedan tillämpa union, sammansättning och Kleenetillslutning ändligt många gånger. Beroende på vilken ordning man tillämpar operationerna i så får man olika språk.

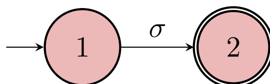
Konsekvensen av detta är att ifall man vill visa att alla reguljära språk har en viss egenskap, så räcker det att visa att basfallen har egenskapen, samt att om två reguljära språk L_1 och L_2 har egenskapen så har även språken L_1L_2, $L_1 \cup L_2$ och L_1^* den aktuella egenskapen.

Detta är en mer generell version av induktionsbevis, där man har flera basfall och tillåter flera olika induktionssteg. Detta gör bevisen längre, men i princip fungerar det likadant som ett vanligt induktionsbevis.

Tekniken liknar induktion över ordens längd. Faktum är att om man definierar *komplexiteten* av ett reguljärt uttryck som antalet språkoperationer som används för att bygga upp det, så är bevistekniken ekvivalent med att göra induktion över uttryckens komplexitet.

Bevis. Enligt Sats 6.3.8 kan alla språk som kan avgöras av en icke-deterministisk tillståndsmaskin avgöras av en deterministisk dito. Alltså räcker det att konstruera icke-deterministiska tillståndsmaskiner.

Låt L vara ett reguljärt språk. Ifall L är tomt, avgörs det av tillståndmaskinen i Figur 6.6. Ifall det består av ett tecken σ, avgörs det av den icke-deterministiska maskinen i Figur 7.1. Denna maskin består av två tillstånd, 1 och 2. Det förra

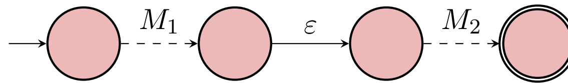


Figur 7.1: En maskin som avgör $\{\sigma\}$.

är starttillstånden, och det senare är ett accepterande tillstånd. Det finns exakt en övergång från 1 till 2, märkt med σ.

Antag istället att L är sammansättningen av två språk L_1 och L_2, som kan avgöras av två deterministiska tillståndsmaskiner M_1 och M_2.

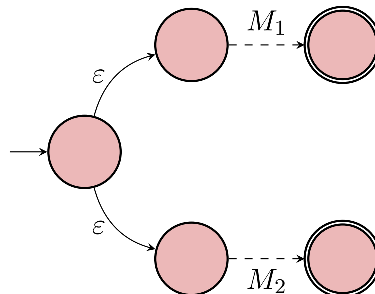
Låt M vara den icke-deterministiska tillståndsmaskin som erhålls genom att dra en ε-övergång från varje accepterande tillstånd i M_1 till varje starttillstånd M_2. Starttillstånden i M är starttillståndet i M_1, och de accepterande tillstånden i M är de accepterande tillstånden i M_2.



Figur 7.2: Sammansättningen av två maskiner via seriekoppling.

Ett ord kommer drivas från ett starttillstånd i M_1 till ett sluttillstånd i M_2 om och endast om det är en sammansättning av två ord x och y. Ordet x driver maskinen till ett accepterande tillstånd i M_1 och ligger således i L_1. Ordet y driver maskinen, via någon ε-övergång, till starttillståndet i M_2, och därefter till ett accepterande tillstånd i M_2.

För att avgöra språket $L_1 \cup L_2$, betrakta maskinen som konstrueras genom att införa ett nytt starttillstånd som kopplas med ε-övergångar till starttillstånden i M_1 och M_2. De accepterande tillstånden i den nya maskinen är de i M_1 och M_2. Den nya maskinen kommer avgöra unionen av de två språken, eftersom

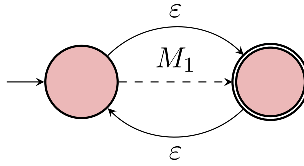


Figur 7.3: Unionen av två maskiner via parallellkoppling.

ifall ett ord ska accepteras av den nya maskinen, måste gå från exakt ett av M_1:s eller M_2:s starttillstånd för att nå ett accepterande tillstånd i samma maskin. Det vill säga, ordet måste antingen ligga i L_1 eller i L_2.

Antag slutligen att L är Kleenetillslutningen av ett språk som avgörs av deterministisk maskin M_1. Sätt ε-övergångar från starttillståndet i M_1 till samtliga accepterande tillstånd, och ε-övergångar från de accepterande tillstånden tillbaka till starttillståndet. Den resulterande maskinen kommer avgöra Kleenetillslutningen av språket som avgörs av M_1, eftersom det enda sättet att nå ett accepterande tillstånd är att antingen gå direkt dit från starttillståndet via en ε-övergång, eller gå igenom M_1.

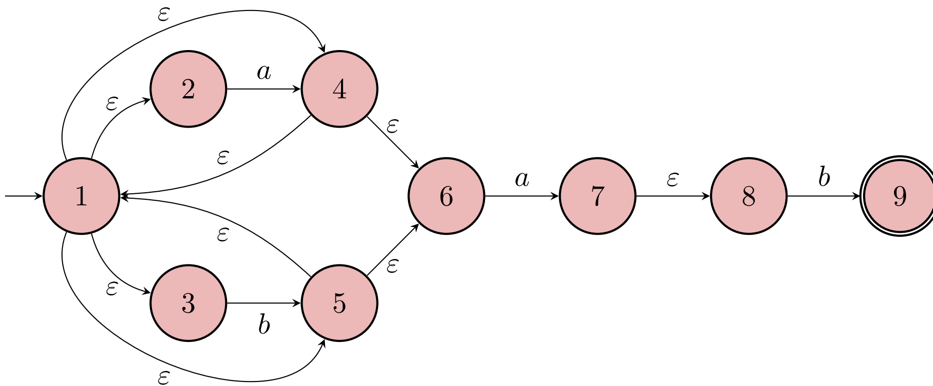
När man väl nått ett accepterande tillstånd, kan man antingen gå tillbaka till starttillståndet och gå igenom M_1 igen, eller så kan man avsluta. De ord som



Figur 7.4: Kleenetillslutningen av en maskin via återkoppling.

accepteras kommer alltså bestå av ett antal kopior av ord från språket som avgörs av M_1, vilket exakt är de orden som ingår i Kleenetillslutningen av språket. □

Exempel 7.1.2. Betrakta språket $(a \cup b)^*ab$, som består av alla ord som slutar på ab. Genom att använda serie-, parallell- och återkopplingar så som i beviset av Sats 7.1.1, kan man konstruera den icke-deterministiska maskinen nedan. Maskinen i Figur 7.5 är tämligen överdimensionerad. Faktum är att det finns



Figur 7.5: En maskin som avgör $(a \cup b)^*ab$.

en deterministisk tillståndsmaskin med endast tre tillstånd som avgör språket $(a \cup b)^*ab$ (se Övning 6.7).

Poängen med Sats 7.1.1 är dock inte att konstruera effektiva tillståndmaskiner, utan att visa att man alltid kan konstruera en maskin som avgör språket. För att göra det krävs en metod som alltid fungerar, och som speglar hur de reguljära språken är konstruerade.

Frågan om hur många tillstånd en tillståndsmaskin behöver ha för att avgöra ett språk återkommer i det sista avsnittet i detta kapitel. ▲

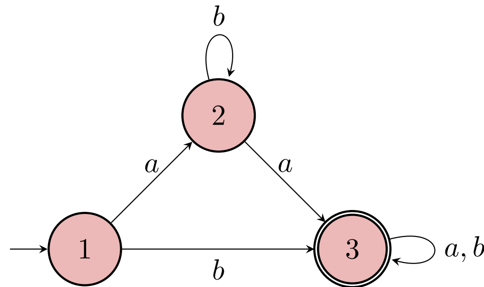
7.2 Tillståndsmaskinernas språk är reguljära

Givet ett reguljärt språk så kan man med hjälp av metoderna i beviset av Sats 7.1.1 och maskinerna i Exempel 6.1.10 konstruera en icke-deterministisk tillståndsmaskin som känner igen språket i fråga. Denna maskin kan man i

sin tur konvertera till en deterministisk maskin som avgör samma språk, med hjälp av delmängdskonstruktionen.

I föreliggande avsnitt behandlas den omvända frågeställningen: kan man givet en deterministisk tillståndsmaskin ta fram ett reguljärt uttryck för det språk som den avgör? Svaret är ja, och metoden kallas för Kleenes algoritm.

Exempel 7.2.1. Maskinen i Figur 7.6 består av tre tillstånd, numrerade 1 till 3. De är arrangerade i en liksidig triangel, med tillstånd 1 och 3 i basen och 2 mittemellan dem. Tillstånd 1 är starttillstånd, och tillstånd 3 är det accepterande tillståndet. Från tillstånd 1 går en a-övergång till 2 och en b-

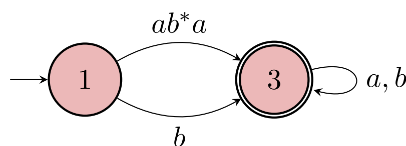


Figur 7.6: En tillståndsmaskin med tre tillstånd.

övergång till 3. Från 2 går en a-övergång till 3, och en b-övergång som loopar tillbaka till 2. Från 3 utgår endast en övergång, märkt med a och b, och denna loopar tillbaka till 3.

Ett ord kan driva maskinen från starttillståndet till det accepterande tillståndet på två sätt. Antingen så börjar ordet på a, följs av ett godtyckligt antal b:n och sedan ytterligare ett a, eller så börjar det på b.

Ett annat sätt att säga detta är att antingen har ordet har ett prefix i det reguljära språket ab^*a, eller så börjar på b. Detta kan man illustrera genom att ta bort tillståndet 2, och införa en övergång från 1 till 3 med märkt med ab^*a. Nu finns det två övergångar från 1 till 3: en för alla ord i språket ab^*a och en

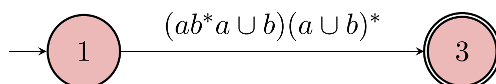


Figur 7.7: En tillståndsmaskin med två tillstånd.

för ordet b. Ett ord accepteras om det ingår i endera av dessa språk. Detta kan visas genom att man ersätter övergångarna med en övergång, märkt med $ab^*a \cup b$.

Slutligen så kan man ta bort loopen från 3 till sig själv genom att lägga till $(a \cup b)^*$ i slutet på uttrycket som markerar övergången mellan 1 och 3. Detta eftersom loopen innebär att när ett ord hamnat i 3, så kan man ha vilket ord som helst kvar och det kommer stanna där.

Resultatet blir följande figur. Alltså är tillståndmaskinens språk $(ab^*a \cup b)(a \cup$



Figur 7.8: En tillståndsmaskin med två tillstånd.

$b)^*$. ▲

Poängen med ovanstående exempel är att (i) seriekoppling motsvarar sammanfogning. (ii) parallellkoppling motsvarar union. (iii) loopar motsvarar Kleenetillslutningar. Detta mönster återfinns redan i beviset av Sats 7.1.1. Men i Kleenes algoritm vänder man på det: istället för att införa seriekopplingar, parallellkopplingar och loopar, så tar man bort dem.

Problemet är att det kan finnas många olika sätt som en maskin kan drivas från ett tillstånd till ett annat, och att en generell metod måste ta hänsyn till alla möjliga sätt.

Antag att tillstånden i en deterministisk tillståndsmaskin är numrerade från 1 till n. För varje par i och j av tillstånd, så definierar man en serie $\alpha_{ij}^0, \ldots, \alpha_{ij}^n$ av reguljära uttryck.

Idén är att ifall k är ett tal mellan 0 och n, så är betecknar α_{ij}^k de ord som driver maskinen från tillstånd i till tillstånd j, utan att passera något tillstånd som är numrerat högre än k. Detta innebär att när k är lika med antalet tillstånd i maskinen, så är α_{ij}^n alla ord som driver maskinen från i till j.

Algoritm 7.2.2. Låt M vara en deterministisk tillståndsmaskin och låt δ vara dess övergångsfunktion. Kleenes algoritm beräknar uttrycken α_{ij}^k genom följande rekursion.

(i) För varje par i och j av tillstånd i maskinen, så är

$$\alpha_{ij}^0 = \sigma_1 \cup \cdots \cup \sigma_m$$

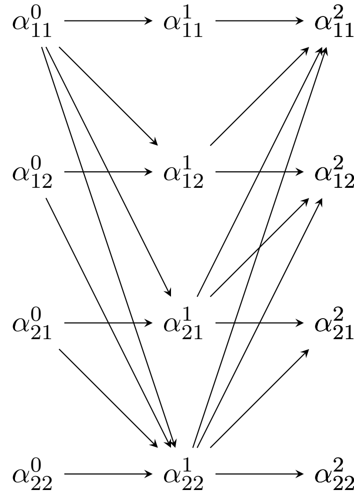
där $\sigma_1, \ldots, \sigma_m$ är de tecken som uppfyller $\delta(i, \sigma_l) = j$.

(ii) Antag att för något k så är α_{ij}^k är definierade för alla par av tillstånd i och j. Då är

$$\alpha_{ij}^{k+1} = \alpha_{ij}^k \cup \alpha_{i(k+1)}^k \left(\alpha_{(k+1)(k+1)}^k\right)^* \alpha_{(k+1)j}^k.$$

Man kan tänka på Kleenes algoritm som en funktion α, som givet i, j och k returnerar det reguljära uttrycket α_{ij}^k. Själva algoritmen är då inget annat än en rekursiv definition av funktionen. Det som gör det hela komplicerat är att man istället för ett basfall har ett basfall för varje par av tillstånd. Dessutom gör rekursionssteget att man behöver använda flera av dessa för att beräkna ett enskilt värde av algoritmen.

I Figur 7.9 illustreras hur de olika värdena på α_{ij}^k hänger ihop i fallet med två tillstånd. En pil från ett reguljärt uttryck indikerar att värdet används i nästa



Figur 7.9: Kleenes algoritm på två tillstånd

steg av beräkningen. Genom att gå tillbaka genom trädet, kan man se att alla basfall används för att beräkna värdena längst ut till höger.

Detta gör Kleenes algoritm jobbig att beräkna för hand. Om tillståndsmaskinen har n stycken tillstånd, så har Kleenes algoritm $n \cdot n = n^2$ basfall. Eftersom k går från 0 till n, så finns det $n^2(n+1)$ stycken reguljära uttryck. Ifall $n = 6$, så innebär detta $6^2 = 36$ basfall och $6^2 \cdot 7 = 252$ reguljära uttryck.

Exempel 7.2.3. Genom att tillämpa algoritmen på maskinen i Figur 7.6, får man resultatet i Tabell 7.10. Den första kolonnen ges genom direkt inspektion

k	0	1	2	3
α_{11}^k	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
α_{12}^k	a	a	$a \cup ab^*b$	$a \cup ab^*b$
α_{13}^k	b	b	$b \cup ab^*a$	$(b \cup ab^*a) \cup (b \cup ab^*a)(a \cup b)^*(a \cup b)$
α_{21}^k	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
α_{22}^k	b	b	$b \cup bb^*b$	$b \cup bb^*b$
α_{23}^k	a	a	$a \cup bb^*a$	$(a \cup bb^*a) \cup (a \cup bb^*a)(a \cup b)^*(a \cup b)$
α_{31}^k	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
α_{32}^k	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
α_{33}^k	$a \cup b$	$a \cup b$	$a \cup b$	$(a \cup b) \cup (a \cup b)(a \cup b)^*(a \cup b)$

Tabell 7.10: Kleenes algoritm tillämpad på Figur 7.6

av maskinen. De andra beräknas utifrån detta, till exempel så har man att

$$\alpha_{23}^1 = \alpha_{23}^0 \cup \alpha_{21}^0 \left(\alpha_{11}^0\right)^* \alpha_{12}^0 = b \cup (\emptyset\emptyset^* a) = b \cup (\emptyset\{\varepsilon\}a) = b$$

eftersom $\emptyset L = \emptyset$ för alla språk L. Liknande beräkningar utförs för varje uttryck i varje kolonn. Rent konkret så är det enklast att räkna varje kolonn för sig. ▲

Följande sats visar hur de reguljära uttrycken som beräknas med hjälp av Kleenes algoritm relaterar till maskinens språk.

Sats 7.2.4. *Orden som ingår i det reguljära språket* α_{ij}^k *är de ord som driver maskinen från tillstånd* i *till* j*, utan att gå igenom något tillstånd* $x > k$*.*

Bevis. Beviset är induktivt över k. När $k = 0$, så säger satsen att α_{ij}^k ska innehålla de ord som driver maskinen från i till j utan att gå igenom några andra tillstånd. Detta gäller per definition.

Antag nu att för något k så består α_{ij}^k är de ord som driver maskinen från tillstånd i till j, utan att gå igenom något tillstånd $x > k$.

Låt w vara ett ord som driver maskinen från i till j utan att gå igenom något tillstånd $x > k + 1$. Då finns det två alternativ.

(i) Ordet w driver inte maskinen genom $k + 1$. Då ingår w i α_{ij}^k, varför det även ingår i

$$\alpha_{ij}^{k+1} = \alpha_{ij}^k \cup \alpha_{i(k+1)}^k \left(\alpha_{(k+1)(k+1)}^k\right)^* \alpha_{(k+1)j}^k.$$

(ii) Ordet w driver maskinen att passera $k + 1$ en eller flera gånger. Det innebär att $w = w_1 w_2 w_3$, där w_1 driver maskinen till $k + 1$ från i utan att passera $k + 1$, w_3 driver maskinen från $k + 1$ till j utan att passera $k + 1$, och w_2 är en mellanbit som driver maskinen från $k + 1$ till sig självt ett antal gånger.

Enligt induktionsantagandet ingår w_1 i α_{ik}^k, och w_3 i $\alpha_{(k+1)j}^k$. Ifall ordet w_2 är tomt driver w maskinen att passera $k + 1$ exakt en gång. Annars kan man skriva $w_2 = u_1 u_2 \cdots u_m$, där varje delord u_k driver maskinen från $k + 1$ till $k + 1$ utan att passera varken $k + 1$ eller något tillstånd med högre siffra.

Med andra ord ingår varje u_k i $\alpha_{(k+1)(k+1)}^k$, och w_2 ingår i det reguljära språket

$$\left(\alpha_{(k+1)(k+1)}^k\right)^*.$$

Alltså ligger $w = w_1 w_2 w_3$ i sammansättningen

$$\alpha_{i(k+1)}^k \left(\alpha_{(k+1)(k+1)}^k\right)^* \alpha_{(k+1)j}^k.$$

Således är det bevisat att alla ord som driver maskinen från i till j utan att passera något tillstånd $x > k + 1$ ligger i α_{ij}^{k+1}.

Det återstår att visa att inga andra ord ligger i α_{ij}^{k+1}. Även detta visas med hjälp av induktion. Basfallet är α_{ij}^0, vilket per definition enbart består av ord som inte går genom några andra tillstånd än i och j.

Antag att för något k så består α_{ij}^k inte av något ord som går genom $x > k$. Då kommer

$$\alpha_{ij}^{k+1} = \alpha_{ij}^k \cup \alpha_{i(k+1)}^k \left(\alpha_{(k+1)(k+1)}^k\right)^* \alpha_{(k+1)j}^k.$$

inte att innehålla några ord som driver maskinen genom att tillstånd $x > k+1$, eftersom det antingen ingår i α_{ij}^k eller är en sammansättning av tre ord som inte går genom tillstånd $x > k + 1$.

Detta bevisar att α_{ij}^{k+1} består exakt av de ord som driver maskinen från i till j utan att passera ett tillstånd $x > k + 1$. □

Ifall n är antalet tillstånd i tillståndsmaskinen, så består α_{ij}^n av alla ord som driver maskinen från i till j. Detta har följande konsekvens.

Sats 7.2.5. *Alla språk som avgörs av en tillståndsmaskin är reguljära.*

Bevis. Antag att maskinens starttillstånd är i och dess accepterande tillstånd är $\{n_1, \ldots, n_m\}$. Antag att maskinen har n stycken tillstånd. Maskinens språk består då av alla ord som driver maskinen från i till någon av n_1, n_2, och så vidare till n_m, det vill säga

$$\alpha_{in_1}^n \cup \alpha_{in_2}^n \cup \cdots \cup \alpha_{in_m}^n.$$

Alltså ges maskinens språk av ett reguljärt uttryck. □

Exempel 7.2.6. Maskinen i Exempel 7.2.3 har starttillstånd 1, accepterande tillstånd 3 och består av 3 tillstånd. Alltså är dess språk

$$L = \alpha_{13}^3 = (b \cup ab^*a) \cup (b \cup ab^*a)(a \cup b)^*(a \cup b)$$

Att detta är lika med $(ab^*a \cup b)(a \cup b)^*$ (svaret som gavs i Exempel 7.2.1) ses på följande sätt.

Språket L är unionen av två språk: $b \cup ab^*a$ och $(b \cup ab^*a)(a \cup b)^*(a \cup b)$. Språket $(a \cup b)^*(a \cup b)$ består av alla ord av längd 1 eller mer, det vill säga alla ord utom ε.

Därmed består språket $(b \cup ab^*a)(a \cup b)^*(a \cup b)$ av alla ord har ett äkta prefix i $b \cup ab^*a$.

Språket L består därför av alla ord som antingen ingår i eller har ett äkta prefix i $b \cup ab^*a$. Detta är ekvivalent med att ligga i språket $(ab^*a \cup b)(a \cup b)^*$.

Ett mer kompakt sätt att skriva detta är via förenklingen

$$\begin{aligned} L &= (b \cup ab^*a) \cup (b \cup ab^*a)(a \cup b)^*(a \cup b) \\ &= (b \cup ab^*a)(\{\varepsilon\} \cup (a \cup b)^*(a \cup b)) \\ &= (b \cup ab^*a)(a \cup b)^* \end{aligned}$$

där man utnyttjar att $\alpha\beta \cup \alpha\gamma = \alpha(\beta \cup \gamma)$ för alla reguljära uttryck α, β och γ (se Övning 5.15). ▲

Anmärkning 7.2.7. Kleenes algoritm tenderar att ge upphov till mycket långa reguljära uttryck. Ifall $k = 2$, $i = 1$ och $j = 2$, får man att

$$\alpha_{ij}^k = \alpha_{12}^2 = \underbrace{\alpha_{12}^1}_{\beta} \cup \alpha_{12}^1 \underbrace{(\alpha_{22}^1)^*}_{\gamma})\alpha_{22}^1 = \beta \cup \beta\gamma^*\gamma.$$

Dessa kan i sin tur beräknas till

$$\beta = \alpha_{12}^1 = \alpha_{12}^0 \cup \alpha_{11}^0 \left(\alpha_{11}^0\right)^* \alpha_{12}^0$$

och

$$\gamma = \alpha_{22}^1 = \alpha_{22}^0 \cup \alpha_{21}^0 \left(\alpha_{11}^0\right)^* \alpha_{12}^0.$$

Genom att sätta in detta i definitionen får man att

$$\alpha_{12}^2 = \beta \cup \beta\gamma^*\gamma$$
$$= \left(\alpha_{12}^0 \cup \alpha_{11}^0 \left(\alpha_{11}^0\right)^* \alpha_{12}^0\right) \cup \left(\alpha_{12}^0 \cup \alpha_{11}^0 \left(\alpha_{11}^0\right)^* \alpha_{12}^0\right)$$
$$\left(\alpha_{22}^0 \cup \alpha_{21}^0 \left(\alpha_{11}^0\right)^* \alpha_{12}^0\right)^* \left(\alpha_{22}^0 \cup \alpha_{21}^0 \left(\alpha_{11}^0\right)^* \alpha_{12}^0\right).$$

I föreliggande fall kan man dock förenkla resultatet något genom att använda resultat i Övning 5.15:

$$\beta \cup \beta\gamma^*\gamma = \beta(\{\varepsilon\} \cup \gamma^*\gamma) = \beta\gamma^*$$
$$= \left(\alpha_{12}^0 \cup \alpha_{11}^0 \left(\alpha_{11}^0\right)^* \alpha_{12}^0\right) \left(\alpha_{22}^0 \cup \alpha_{21}^0 \left(\alpha_{11}^0\right)^* \alpha_{12}^0\right)^*.$$

Notera att algoritmen är beroende av vilken numrering av tillstånden som används. Olika numreringar kommer ge upphov till olika reguljära uttryck. Men maskinens språk kommer fortfarande vara samma, oavsett vilken numrering som används, även om de reguljära uttrycken som fås genom Sats 7.2.5 är olika för olika numreringar.

7.3 Särskiljning av ord

De två föregående avsnitten ägnades åt att bevisa att tillståndsmaskinernas språk och de reguljära språken är samma klass av språk. I detta avsnitt ges en tredje karakterisering.

Definition 7.3.1. En deterministisk tillståndsmaskin *särskiljer* en mängd ord ifall den driver alla ord i mängden till olika tillstånd. △

Alla tillståndsmaskiner särskiljer mellan ord som de accepterar och ord som de inte accepterar, men även ord som både accepteras och inte accepteras kan särskiljas.

Exempel 7.3.2. Tillståndsmaskinen i Figur 6.1 i Exempel 6.1.1 särskiljer orden b och a, eftersom de drivs till olika tillstånd. Den särskiljer inte b och ab, eftersom maskinen drivs till samma tillstånd av båda orden. ▲

Sats 7.3.3. *Låt z vara ett godtyckligt ord. Om en tillståndsmaskin inte särskiljer två ord w och u, särskiljer maskinen inte heller orden wz och uz.*

Bevis. Ifall en tillståndsmaskin med utvidgad övergångsfunktion δ^* inte särskiljer två ord w och u, så är $\delta^*(S, w)$ lika med $\delta^*(S, u)$ och därmed gäller

$$\delta^*(S, wz) = \delta^*(\delta^*(S, w), z)) = \delta^*(\delta^*(S, u), z)) = \delta^*(S, uz),$$

enligt Sats 6.1.11. □

Ovanstående sats fångar intuitionen att tillståndsmaskiner saknar arbetsminne. Det spelar ingen roll hur en maskin hamnat i ett visst tillstånd, utan dess framtida beteende beror enbart på dess nuvarande tillstånd.

Särskiljning kan även definieras för allmänna språk, oavsett om de är reguljära eller inte.

Definition 7.3.4. Låt L vara ett språk och M en mängd ord över samma alfabet. Språket L *särskiljer* M ifall det för varje par av ord w och u i M finns ett ord z med egenskapen att wz tillhör L men inte uz, eller vice versa. △

Ett annat sätt att uttrycka det är att ett språk L *inte* särskiljer w och u ifall

$$wz \in L \iff uz \in L$$

gäller för alla ord z.

Alla språk särskiljer mellan de ord som ingår i det och de som inte ingår i det, genom att sätta $z = \varepsilon$.

Exempel 7.3.5. Språket

$$L = \{a^n b^n \mid n \in \mathbb{N}\}$$

särskiljer alla ord på formen a^n och a^m där n är skilt från m, eftersom valet $z = b^n$ gör att $a^n z = a^n b^n$ tillhör L, medan $a^m z = a^m b^n$ inte gör det. Språket L särskiljer alltså en oändlig mängd ord. ▲

Exempel 7.3.6. Det reguljära språket $L = ab^*a$ särskiljer inte orden a och ab. För att om z är sådant az tillhör L, måste z vara på formen $b^n a$ för något naturligt tal n. Men då gäller att $abz = ab^{n+1}a$ tillhör L.

På samma sätt gäller att ifall abz ligger i L, så måste $z = b^n a$, varvid $az = ab^n a$ tillhör L. Sammantaget får man att

$$az \in L \text{ om och endast om } abz \in L$$

för alla ord z, och vilket innebär att a och ab inte särskiljs av L. ▲

Exempel 7.3.7. Språket L över $\{a, b\}$ som består av ett enda tecken a särskiljer tre ord: ε, a och b. Ordet a särskiljer ε och a samt ε och b, då $\varepsilon a = a$ ingår i L men inte aa eller ba. Likaledes särskiljs a och b av ordet $z = \varepsilon$, eftersom $az = a$ ligger i L men inte $bz = b$. ▲

De två särskiljningsbegreppen är relaterade till varandra.

Sats 7.3.8. *Om en deterministisk tillståndsmaskins språk särskiljer en mängd ord så särskiljer även maskinen dessa ord.*

Bevis. Antag att två ord w och u inte särskiljs av en tillståndsmaskin och låt z vara ett godtyckligt ord. Enligt Sats 7.3.3 kan inte maskinen särskilja wz och uz.

Då finns det två möjligheter. Antingen driver wz och uz maskinen till ett accepterande tillstånd, och då ingår båda i $L(M)$. Eller så driver maskinen orden till icke-accepterande tillstånd, varvid inget av dem ingår i $L(M)$. Alltså gäller att

$$wz \in L(M) \text{ om och endast om } uz \in L(M).$$

för alla ord z, och $L(M)$ kan inte särskilja w och u. □

Sats 7.3.9. *En deterministisk tillståndsmaskin har minst lika många tillstånd som antalet ord som dess språk kan särskilja.*

Bevis. Alla ord som särskiljs av en maskins språk särskiljs av maskinen själv, enligt Sats 7.3.8. Således kan maskinens språk inte särskilja fler ord än maskinen har tillstånd, och en tillståndsmaskin har minst lika många tillstånd som dess språk kan särskilja. □

Exempel 7.3.10. Språket som består av enbart ett tecken a särskiljer minst tre strängar, enligt Exempel 7.3.7. Alltså måste en deterministisk maskin som avgör språket ha minst tre tillstånd. En sådan ges i Figur 6.7 i Exempel 6.1.10. ▲

Kopplingen mellan antalet tillstånd och särskiljning ger ett sätt att bevisa att ett givet språk inte är reguljärt.

Sats 7.3.11. *Ett språk som särskiljer oändligt många ord är inte reguljärt.*

Bevis. Varje reguljärt språk är språket för någon deterministisk tillståndsmaskin. Eftersom dessa endast har ändligt många tillstånd, kan reguljära språk endast särskilja ändligt många olika strängar. □

Exempel 7.3.12. Enligt Exempel 7.3.5 så särskiljer språket $\{a^n b^n \mid n \in \mathbb{N}\}$ oändligt många strängar, och är således inte reguljärt. ▲

Om ett språk som mest särskiljer ändligt många strängar, måste det då vara reguljärt? Svaret är ja! Beviset är dock för avancerat för detta kompendium och utelämnas.

Sats 7.3.13. *Varje språk som endast kan särskilja ändligt många ord är reguljärt.*

Resultaten i detta kapitel kan sammanfattas i följande sats.

Sats 7.3.14. *För ett formellt språk L är följande villkor ekvivalenta.*

(i) *L är reguljärt.*

(ii) *L kan avgöras av en deterministisk tillståndsmaskin.*

(iii) *L kan endast särskilja ändligt många ord.*

Bevis. Sats 7.1.1 och Sats 7.2.5 visar att L är reguljärt om och endast om det kan avgöras av en deterministisk tillståndsmaskin. Sats 7.3.13 visar att alla språk som endast kan särskilja ändligt många ord är reguljärt, medan Sats 7.3.11 implicerar att reguljära språk endast kan särskilja ändligt många ord. □

Sats 7.3.14 säger är att tre sätt att beskriva formella språk (reguljära uttryck, tillståndsmaskiner och särskiljning) i grund och botten ger samma familj.

Man kan likna det vid att man har tre fotografier. Inget av dem ger en fullständig bild av det som de föreställer, utan visar vissa delar och döljer andra. Men satsen och dess bevis säger oss två saker.

För det första så garanterar satsen att *fotografierna föreställer samma föremål.* För det andra så beskriver bevisen *hur de olika bilderna relaterar till varandra.* Det gör att man kan byta mellan de olika bilderna, för att få bättre förståelse för föremålet som studeras. Den här typen av samband är mycket användbara i tillämpningar.

Säg att person vill välja ut en viss mängd data ur en stor mängd data. För att göra det skriver hon ett reguljärt uttryck som matchar den data hon vill ha. Detta går snabbt, eftersom de reguljära uttrycken passar hur människor resonerar.

Reguljära uttryck kan dock inte användas av en dator. En dator konverterar därför det givna uttrycket till en tillståndsmaskin som avgör det reguljära uttryck. Genom att köra all data genom tillståndsmaskinen och spara den data som tillståndsmaskinen accepterar, så kan datorn på effektivt sätt returnera all data som användaren önskar.

Särskiljning ger ett mer abstrakt perspektiv på reguljära språk. Det beskriver språket i termer av vilka ord som det ska särskilja. Det fina är att beskrivningen är neutral, eftersom den enbart refererar till språket och dess ord. Det beskriver dessutom skillnaden mellan språk som är reguljära och de som inte är det, och ger ett sätt att bevisa att språk inte är reguljära.

Övningar

Övning 7.1. Använd metoderna i beviset av Sats 7.1.1 för att konstruera en icke-deterministisk tillståndsmaskin för språket aba. Ange antingen tillståndsmängd, starttillstånd, accepterande tillstånd och övergångsrelation, eller en grafisk representation.

Övning 7.2. Använd metoderna i beviset av Sats 7.1.1 för att konstruera en icke-deterministisk tillståndsmaskin för språket $(aa)^*$. Ange antingen tillståndsmängd, starttillstånd, accepterande tillstånd och övergångsrelation, eller en grafisk representation.

Övning 7.3. Använd metoderna i beviset av Sats 7.1.1 för att konstruera en icke-deterministisk tillståndsmaskin för språket $(a \cup b)b^*$. Ange antingen tillståndsmängd, starttillstånd, accepterande tillstånd och övergångsrelation, eller en grafisk representation.

Övning 7.4. Använd metoderna i beviset av Sats 7.1.1 för att konstruera en icke-deterministisk tillståndsmaskin för språket $ab^* \cup ba^*$. Ange antingen tillståndsmängd, starttillstånd, accepterande tillstånd och övergångsrelation, eller en grafisk representation.

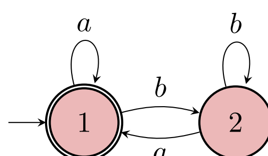
Övning 7.5. Använd metoderna i beviset av Sats 7.1.1 för att konstruera en icke-deterministisk tillståndsmaskin för språket $a^*(b^* \cup ab^*)$. Ange antingen tillståndsmängd, starttillstånd, accepterande tillstånd och övergångsrelation, eller en grafisk representation.

Övning 7.6. Använd metoderna i beviset av Sats 7.1.1 för att konstruera en icke-deterministisk tillståndsmaskin för språket $(ba \cup b^*a)^*$. Ange antingen tillståndsmängd, starttillstånd, accepterande tillstånd och övergångsrelation, eller en grafisk representation.

Övning 7.7. Finn ett reguljärt uttryck för språket till den deterministiska tillståndsmaskinen med tillståndsmängd $\{1, 2\}$, starttillstånd 1, accepterande tillstånd 1 och övergångsfunktion

Ω	a	b
1	1	2
2	1	2

Nedan ges en grafisk representation.

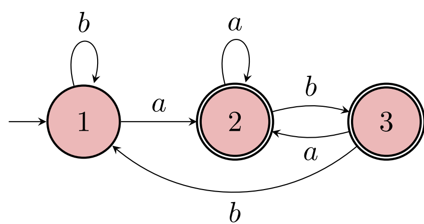


Du behöver inte använda Kleenes algoritm.

Övning 7.8. Finn ett reguljärt uttryck för det till den deterministiska tillståndsmaskinen med tillståndsmängd $\{1, 2, 3\}$, starttillstånd 1, accepterande tillstånd 2 och 3 och övergångsfunktion

Ω	a	b
1	2	1
2	2	3
3	2	3

Nedan ges en grafisk representation.

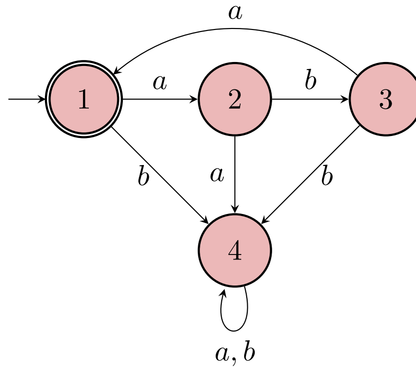


Du behöver inte använda Kleenes algoritm.

Övning 7.9. Finn ett reguljärt uttryck för språket till den deterministiska tillståndsmaskinen med tillståndsmängd $\{1, 2, 3, 4\}$, starttillstånd 1, accepterande tillstånd 1 och övergångsfunktion

Ω	a	b
1	2	4
2	4	3
3	1	4
4	4	4

Nedan ges en grafisk representation.

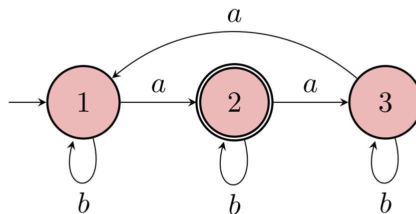


Du behöver inte använda Kleenes algoritm.

Övning 7.10. Finn ett reguljärt uttryck för språket till den deterministiska tillståndsmaskinen med tillståndsmängd $\{1, 2, 3\}$, starttillstånd 1, accepterande tillstånd 2 och övergångsfunktion

Ω	a	b
1	2	1
2	3	2
3	1	3

Nedan ges en grafisk representation.



Du behöver inte använda Kleenes algoritm.

Övning 7.11. Ge ett reguljärt uttryck för alla ord som driver maskinen i Övning 7.9 från tillstånd 1 till tillstånd 2.

Övning 7.12. Ge ett reguljärt uttryck för alla ord som driver maskinen i Övning 7.10 från tillstånd 1 till tillstånd 2 utan att passera tillstånd 3.

Övning 7.13 ($\star$)**.** Tillämpa Kleenes algoritm på tillståndsmaskinen i Övning 7.7.

Övning 7.14 (⋆)**.** Tillämpa Kleenes algoritm på tillståndsmaskinen i Övning 7.8.

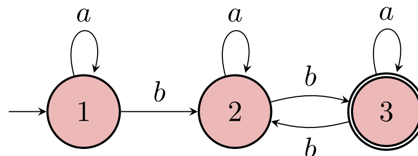
Övning 7.15. Särskiljer tillståndsmaskinen i Övning 7.9 mängden $\{\varepsilon, a, b, ab\}$

Övning 7.16. Särskiljer tillståndsmaskinen i Övning 7.8 mängden $\{aba, aab, baa\}$?

Övning 7.17. Särskiljer språket $(a \cup b)ab^*$ mängden $\{a, bb, bab\}$?

Övning 7.18. Särskiljer språket $L = (ab \cup ba)^*aa^*$ mängden $\{ba, ab, aba\}$?

Övning 7.19. Betrakta tillståndsmaskinen nedan.

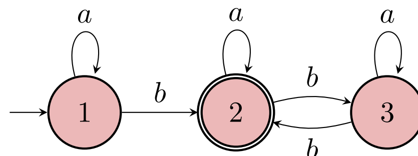


Den har tillståndsmängd $\{1, 2, 3\}$, starttillstånd 1, accepterande tillstånd 3 och övergångsfunktion

Ω	a	b
1	1	2
2	2	3
3	3	2

Maskinen avgör språket $a^*ba^*b(a^*ba^*b)^*$. Är den minimal? Om så, ge en mängd med särskiljande ord. Annars, konstruera en maskin med färre tillstånd än den givna som avgör samma språk.

Övning 7.20. Betrakta tillståndsmaskinen nedan.

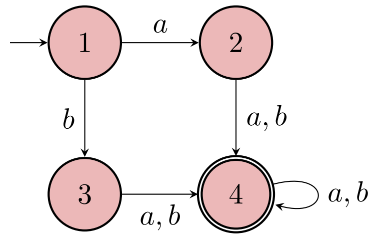


Den har tillståndsmängd $\{1, 2, 3\}$, starttillstånd 1, accepterande tillstånd 2 och övergångsfunktion

Ω	a	b
1	1	2
2	2	3
3	3	2

Maskinen avgör språket $a^*b(a^*ba^*b)^*$. Är maskinen minimal? Om så, ange en mängd med särskiljande ord. Annars, konstruera en maskin med färre tillstånd än den givna som avgör samma språk. Notera att den enda skillnaden från maskinen i Övning 7.19 är det accepterande tillståndet.

Övning 7.21. Betrakta tillståndsmaskinen nedan.

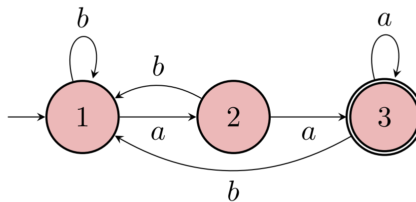


Den har tillståndsmängd $\{1, 2, 3, 4\}$, starttillstånd 1, accepterande tillstånd 4 och övergångsfunktion

Ω	a	b
1	2	3
2	4	4
3	4	4
4	4	4

Maskinen avgör språket $(a \cup b)(a \cup b)(a \cup b)^*$. Är maskinen minimal? Om så, ange en mängd med särskiljande ord. Annars, konstruera en maskin med färre tillstånd än den givna som avgör samma språk.

Övning 7.22. Betrakta tillståndsmaskinen nedan.



Den har tillståndsmängd $\{1, 2, 3\}$, starttillstånd 1, accepterande tillstånd 3 och övergångsfunktion

Ω	a	b
1	2	1
2	3	1
3	3	1

Maskinen avgör språket $(a \cup b)^*$. Är den minimal? Om så, ge en mängd med särskiljande ord. Annars, konstruera en maskin med färre tillstånd än den givna som avgör samma språk.

Övning 7.23 ($\star$). Är språket som består av alla palindrom över alfabetet $\{a, b\}$ reguljärt (se Övning 5.17)? Hitta en oändlig mängd med ord som språket särskiljer, eller konstruera ett reguljärt uttryck eller maskin som avgör ordet.

Övning 7.24 ($\star$)**.** Är språket som består av alla ord över $\{a, b\}$ där antalet a:n i ordet är kongruent med 1 eller 2 modulo 3. Hitta en oändlig mängd med ord som språket särskiljer, eller konstruera ett reguljärt uttryck eller maskin som avgör språket.

Övning 7.25 ($\star$)**.** Är språket som består av alla ord över $\{a, b\}$ som inte innehåller exakt ett b reguljärt? Hitta en oändlig mängd med ord som språket särskiljer, eller konstruera ett reguljärt uttryck eller maskin som avgör språket.

Lösningar till udda övningsuppgifter

Kapitel 1

Grundläggande uppgifter

Övning 1.1. Elementen i mängderna är:

(i) $\{-1, 0, 1, 2\}$

(ii) $\{-2, 2, 3\}$

(iii) $\{1, 9, 11\}$

(iv) $\{1\}$

(v) $\{-2, -1, 0, 2, 3\}$

Övning 1.3.

(i) Nej, den första mängden innehåller talen 0 och 1 som element, men de finns inte som element i den andra mängden (som bara innehåller elementet $\{0, 1\}$).

(ii) Nej, den första mängden innehåller -2, men det gör inte den andra mängden.

(iii) Ja, båda mängderna är $\{0, 1\} = \mathbb{B}$.

(iv) Ja, båda mängderna är mängden av udda heltal. För varje heltal x har vi $2x - 1 = 2(x - 1) + 1$, där ju även $x - 1$ är ett heltal.

(v) Ja, båda mängderna är mängden av ickenegativa heltal. Om $x \geqslant 0$ så är ju $\sqrt{x^2} = x$, medan om $x < 0$ så är $\sqrt{x^2}$ det positiva talet $-x$.

Övning 1.5. Antag motsatsen, det vill säga att det finns ett $x \in \mathbb{Z}$ som är det största heltalet. Men $x + 1 > x$ för alla $x \in \mathbb{Z}$, så x kan inte vara det största heltalet. Alltså kan det inte finnas något största heltal.

Övning 1.7. De regler som beskriver funktioner är (ii), (iii), (v), (vi) och (vii). Regel (i) beskriver inte en funktion eftersom dess utdata är slumpmässigt, så den kommer inte ge samma resultat varje gång för samma indata. Regel (iv) beskriver inte en funktion eftersom dess utdata beror på vilken dag det är.

Övning 1.9.

(i) Funktionerna är *inte* lika med varandra, för de har olika definitionsmängder.

(ii) Funktionerna är *inte* lika med varandra, för de har olika definitionsmängder (och olika målmängder).

(iii) Funktionerna är *inte* lika med varandra, för till exempel är $f(-1) = 1$ men $g(-1) = -1$.

(iv) Funktionerna är lika med varandra.

Övning 1.11. Enligt Definition 1.1.3 så är C en delmängd av A om alla element i C också är element i A. Låt oss ta ett godtyckligt element $x \in C$. Eftersom $C \subseteq B$ så måste x också vara ett element i B; vi vet alltså att $x \in B$. Eftersom $B \subseteq A$ så måste x också vara ett element i A. Då x var ett godtyckligt element i C så visar detta att alla element i C också är element i A.

Övning 1.13. Vi har redan visat att alla positiva heltal är udda eller jämna tal. Om vi visar följande så är vi klara.

(i) Om $x > 0$ är ett jämnt tal så är $-x$ också jämnt.

(ii) Om $x > 0$ är ett udda tal så är $-x$ också udda.

Påståendena (i) och (ii) tillsammans innebär att även alla negativa heltal är jämna eller udda, eftersom alla negativa heltal kan skrivas som $-x$ för ett positivt heltal x.

Bevis av (i). Antag att $x = 2k$, där k är ett heltal. Då är $-x = -2k = 2 \cdot (-k)$ ett jämnt tal, eftersom även $-k$ är ett heltal.

Bevis av (ii). Antag att $x = 2k + 1$, där k är ett heltal. Då är $-x = -2k - 1 = 2(-k - 1) + 1$ ett udda tal, eftersom även $-k - 1$ är ett heltal.

Mer avancerade uppgifter

Övning 1.15. Antag motsatsen, det vill säga att det bara finns ett ändligt antal primtal, säg n stycken. Då kan vi lista alla primtal: $p_1, p_2, p_3, \ldots, p_n$. Låt N vara produkten av alla dessa primtal, plus ett. Alltså är $N = (p_1 p_2 p_3 \cdots p_n) + 1$. Talet N är större än alla primtal i vår lista. Talet N kan inte vara ett primtal, för då skulle vår lista inte innehålla alla primtal, vilket vi ju sa att den skulle göra. Enligt Sats 1.4.7 måste N då vara en produkt av flera primtal, säg $N = q_1 q_2 \cdots q_m$ där $q_1, q_2, \ldots, q_m$ är primtal. Eftersom q_1 är ett primtal så måste det finnas i vår lista med alla primtal. Vi kan sortera listan så att q_1 är det första primtalet i listan, alltså $q_1 = p_1$. Detta betyder att

$$p_1 q_2 \cdots q_m = N = (p_1 p_2 p_3 \cdots p_n) + 1.$$

Alltså är

$$(p_1 q_2 \cdots q_m) - (p_1 p_2 p_3 \cdots p_n) = 1$$

och vi kan bryta ut p_1 och få

$$p_1 \underbrace{(q_2 \cdots q_m - p_2 p_3 \cdots p_n)}_{=k} = 1.$$

Talet som vi har kallat k är ett heltal. Vi har alltså fått $p_1 k = 1$ där p_1 är ett primtal och k ett heltal. Om två heltal uppfyller $p_1 k = 1$ måste $p_1 = 1$ och $k = 1$. Men detta är omöjligt, för $p_1 \geqslant 2$ eftersom det är ett primtal. Alltså måste vårt grundantagande om att det finns ändligt många primtal vara falskt. Därmed måste det finnas oändligt många primtal.

Övning 1.17. Enligt Sats 1.4.2 så är antalet funktioner från X till Y lika med $|Y|^{|X|}$ om X och Y är ändliga mängder. Eftersom $\mathbb{B} = \{0, 1\}$ så är $|\mathbb{B}| = 2$, och enligt Övning 1.16 så är $|\mathbb{B} \times \mathbb{B}| = |\mathbb{B}| \cdot |\mathbb{B}| = 2 \cdot 2 = 4$. Alltså är antalet olika funktioner från $\mathbb{B} \times \mathbb{B}$ till $\mathbb{B}$ lika med $|\mathbb{B}|^{|\mathbb{B}\times\mathbb{B}|} = 2^4 = 16$.

Övning 1.19.

(i) Om vi utgår från definitionen $P \uparrow Q = \neg(P \wedge Q)$ och tar negation av båda leden så får vi

$$\neg(P \uparrow Q) = \neg\neg(P \wedge Q) = P \wedge Q,$$

eftersom $\neg\neg B = B$ för alla $B \in \mathbb{B}$. Vi vet att $\neg R = R \uparrow R$, så med $R = P \uparrow Q$ fås

$$\neg(P \uparrow Q) = (P \uparrow Q) \uparrow (P \uparrow Q).$$

Detta visar att $P \wedge Q = (P \uparrow Q) \uparrow (P \uparrow Q)$.

(ii) Från De Morgans lagar (Övning 1.10) vet vi att $\neg(P \vee Q) = (\neg P) \wedge (\neg Q)$. Negation av båda leden ger

$$P \vee Q = \neg((\neg P) \wedge (\neg Q)).$$

Vi har redan visat att alla funktioner i högerledet ($\neg$ och $\wedge$) kan uttryckas som kombinationer av $\uparrow$, så därför kan även $P \vee Q$ uttryckas som kombinationer av $\uparrow$.

(iii) Enligt Övning 1.17 finns det endast 16 olika funktioner från $\mathbb{B} \times \mathbb{B}$ till $\mathbb{B}$, så vi skulle kunna gå igenom alla dessa funktioner – kanske enklast genom att skriva upp alla möjliga sanningstabeller – och visa att var och en av dem kan skrivas som kombinationer av $\uparrow$ (eller av de funktioner som vi redan har visat kan uttryckas med hjälp av $\uparrow$).

Övning 1.21.

(i) Uppgiften kan lösas på olika sätt, och vi anger tre olika möjliga lösningsförslag nedan.

Förslag 1. I vänsterledet $(A \cup B) \setminus C$ ingår precis de element som finns i A eller B men inte i C. Det är förstås samma element som finns i $A \setminus C$ eller $B \setminus C$, det spelar ingen roll om vi tar bort elementen som finns i C innan eller efter unionen.

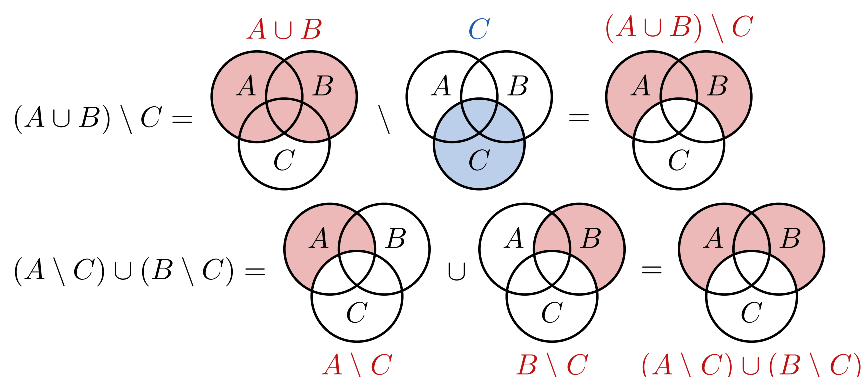
Förslag 2. Vill vi vara mer rigorösa kan vi använda Övning 1.20 och visa att vänsterledet är en delmängd av högerledet och vice versa.

- $(A \cup B) \setminus C \subseteq (A \setminus C) \cup (B \setminus C)$: Tag ett godtyckligt $x \in (A \cup B) \setminus C$. Elementet x måste ligga i den del av $A \cup B$ som inte överlappar med C; alltså är x i A eller B, men inte i C. Om $x \in A$ så är därmed $x \in A \setminus C$, och denna mängd är en delmängd till högerledet $(A \setminus C) \cup (B \setminus C)$, så x tillhör högerledet. Om istället $x \in B$ så är $x \in B \setminus C$, vilket också är en delmängd till högerledet, så x tillhör högerledet. Detta visar att vänsterledet är en delmängd av högerledet.

- $(A \setminus C) \cup (B \setminus C) \subseteq (A \cup B) \setminus C$: Tag ett godtyckligt $x \in (A \setminus C) \cup (B \setminus C)$. Då finns x antingen i $A \setminus C$ eller $B \setminus C$. Om $x \in A \setminus C$ så tillhör x även $(A \cup B) \setminus C$, för A är en delmängd av $A \cup B$. Om istället $x \in B \setminus C$ så tillhör x också $(A \cup B) \setminus C$ eftersom även B är en delmängd av $A \cup B$. Detta visar att högerledet är en delmängd av vänsterledet.

Enligt Övning 1.20 är därmed $(A \cup B) \setminus C = (A \setminus C) \cup (B \setminus C)$.

Förslag 3. Ett mindre formellt sätt att visa likheten är att rita upp venndiagrammen för vänsterledet och högerledet, och konstatera att det blir samma diagram. Till exempel så här:



(ii) De tre varianterna från del (i) fungerar här också. Vi ger ett lösningsförslag i linje med förslag 1.

Elementen i $C \setminus (A \cup B)$ är precis de element som finns i C, men *inte* i $A \cup B$; eller med andra ord, element som finns i C, men varken i A eller B. I $C \setminus A$ finns element som tillhör C men inte A, och i $C \setminus B$ finns element som tillhör C men inte B. Snittet av dessa mängder innehåller alltså element som tillhör C men varken A eller B, vilket visar att $C \setminus (A \cup B) = (C \setminus A) \cap (C \setminus B)$.

Övning 1.23. Kom ihåg att funkionen F är definierad rekursivt genom

$$F(n) = \begin{cases} 0 & \text{om } n = 0, \\ 1 & \text{om } n = 1, \\ F(n-1) + F(n-2) & \text{om } n \geqslant 2. \end{cases} \tag{A}$$

Låt oss visa, med hjälp av induktion, att

$$F(n) = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}} \tag{B}$$

för $n = 0, 1, 2, \ldots$

Basfallen blir att visa att ekvation (B) gäller för $n = 0$ och $n = 1$. Detta kan vi lätt kontrollera:

$$\frac{\phi^0 - (1-\phi)^0}{\sqrt{5}} = \frac{1-1}{\sqrt{5}} = 0 = F(0),$$

$$\frac{\phi^1 - (1-\phi)^1}{\sqrt{5}} = \frac{2\phi - 1}{\sqrt{5}} = \frac{(1+\sqrt{5}) - 1}{\sqrt{5}} = 1 = F(1),$$

där vi för $n = 1$ använde att $\phi = (1+\sqrt{5})/2$. Nu är vi klara med basfallen.

För induktionssteget visar vi att om ekvation (B) gäller upp till och med $n = k$ så gäller den även för $n = k+1$ (där $k \geqslant 1$). Från ekvation (A) vet vi att

$$F(k+1) = F(k) + F(k-1).$$

Ekvation (B) gäller för $F(k)$ och $F(k-1)$, så vi byter ut dem i ovanstående ekvation och får

$$\begin{aligned}
F(k+1) &= \frac{\phi^k - (1-\phi)^k}{\sqrt{5}} + \frac{\phi^{k-1} - (1-\phi)^{k-1}}{\sqrt{5}} \\
&= \frac{\phi^k + \phi^{k-1} - (1-\phi)^k - (1-\phi)^{k-1}}{\sqrt{5}} \\
&= \frac{\dfrac{\phi^{k+1}}{\phi} + \dfrac{\phi^{k+1}}{\phi^2} - \dfrac{(1-\phi)^{k+1}}{1-\phi} - \dfrac{(1-\phi)^{k+1}}{(1-\phi)^2}}{\sqrt{5}} \\
&= \frac{\left(\dfrac{1}{\phi} + \dfrac{1}{\phi^2}\right)\phi^{k+1} - \left(\dfrac{1}{1-\phi} + \dfrac{1}{(1-\phi)^2}\right)(1-\phi)^{k+1}}{\sqrt{5}}. \qquad \text{(C)}
\end{aligned}$$

Genom att skriva om på gemensamt bråkstreck och utnyttja att $\phi^2 = \phi + 1$ kan vi visa att

$$\frac{1}{\phi} + \frac{1}{\phi^2} = \frac{\phi+1}{\phi^2} = \frac{\phi+1}{\phi+1} = 1.$$

På liknande sätt blir

$$\begin{aligned}
\frac{1}{1-\phi} + \frac{1}{(1-\phi)^2} &= \frac{2-\phi}{(1-\phi)^2} = \frac{2-\phi}{1-2\phi+\phi^2} = \frac{2-\phi}{1-2\phi+(\phi+1)} \\
&= \frac{2-\phi}{2-\phi} = 1.
\end{aligned}$$

Insättning i ekvation (C) ger nu

$$F(k+1) = \frac{\phi^{k+1} - (1-\phi)^{k+1}}{\sqrt{5}},$$

vilket visar ekvation (B) för $n = k+1$. Därmed är induktionssteget, och hela beviset, klart.

Programmeringsuppgifter

Övning 1.P1. Förslag på programkod (Python):

```
n = int(input('Skriv in n: '))

F = []
for k in range(0, n+1):
   if k == 0:
```

```
        F.append(0)
      elif k == 1:
        F.append(1)
      else:
        F.append(F[-1] + F[-2])

  print(f'F({n}) är', F[-1])
```

Exempel på körning av programmet upprepade gånger:

```
Skriv in n: 5
F(5) är 5
Skriv in n: 10
F(10) är 55
Skriv in n: 20
F(20) är 6765
Skriv in n: 40
F(40) är 102334155
```

Kapitel 2

Övning 2.1. Talet kan skrivas som

$$42 = 32 + 10 = 32 + 8 + 2 = 1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^1.$$

Alltså är $42 = (101010)_2$.

Övning 2.3. Enligt Sats 2.1.3 kan ett N-bitars osignerat heltal anta 2^N värden. I detta fall är $N = 5$, så antalet möjliga värden är $2^5 = 32$.

Övning 2.5. (i) $32 = 2 \cdot 16^1 = (20)_{16}$.

(ii) $2B1 = 2 \cdot 16^2 + 11 \cdot 16^1 + 1 \cdot 16^0 = 2 \cdot 4096 + 11 \cdot 16 + 1 = 8369$

(iii) $60 = 48 + 12 = 3 \cdot 16^1 + 12 \cdot 16^0 = (3C)_{16}$

(iv) En siffra i bas 16 kan anta 16 olika värden. För att lagra så många värden krävs 4 bitar, eftersom 4-bitars osignerat heltal kan lagra $2^4 = 16$ bitar enligt Sats 2.1.3.

Övning 2.7. Först bevisar man att (i) $\implies$ (ii). Antag att heltalen p, q och r är sådana att $a = pn + r$ och $b = qn + r$ och $0 \leqslant r \leqslant n - 1$.

Vi ska visa att det finns ett heltal k så att $a = b + kn$. Flyttar man över b till vänsterledet får man $a - b = kn$. Enligt antagandet är $a = pn + r$ och $b = qn + r$, och sätter man in detta i vänsterledet får man

$$a - b = pn + r - (qn + r) = pn - qn + r - r = (p - q)n.$$

Sätter man $k = p - q$ får man att $a - b = kn$, det vill säga $a = b + kn$.

För att bevisa att (ii) $\implies$ (i), antag att det finns ett heltal k så att $a = b+kn$, det vill säga $a - b = kn$. Beviset består av två steg.

(i) Först visar vi att för alla heltal a så finns det ett heltal p

(ii) Sedan visar vi att ifall a och b är kongruenta modulo n, så måste r och s vara lika.

För att visa det första påståenden, sätt p till det största heltalet som uppfyller $pn \leqslant a$ och låt sedan $r = a - pn$. Enligt antagande gäller då att $pn + r = pn + a - pn = a$.

Det återstår att visa att $0 \leqslant r \leqslant n-1$. Antag att så inte är fallet, det vill säga $n \leqslant r$. Då gäller att

$$(p+1)n = pn + n \leqslant pn + r = a.$$

Men vi valde ju p till att vara det största heltalet så att $pn \leqslant a$. Detta är en motsägelse, och alltså måste $0 \leqslant r \leqslant n-1$.

Nu när vi bevisat att alla heltal a och b kan skrivas på formen $pn + r$ och $qn + s$, där $0 \leqslant r, s \leqslant n-1$, återstår att visa att om a och b är kongruenta modulo n, så är $r = s$.

Enligt definitionen av kongruens, så finns det ett heltal k så att $a = b + kn$. Då gäller att

$$pn + r = qn + s + kn \iff r - s = kn - pn - qn = (k - p - q)n.$$

Notera att $-(n-1) \leqslant r - s \leqslant n-1$, eftersom $0 \leqslant r, s \leqslant n-1$. Därmed måste $(k-p-q)n = 0$, och alltså är $r - s = 0$, det vill säga $r = s$.

Övning 2.9. Om a delar b, så finns det ett heltal k så att $ak = b$. Om b delar c, så finns det heltal l så att $bl = c$. Då gäller att

$$akl = bl = c$$

Alltså delar a talet c med kvoten kl.

Övning 2.11. Enligt konjugatregeln gäller $x^2 - 1 = (x-1)(x+1)$. Så genom att sätta $P = x - 1$, $K = x + 1$ och $Q = x^2 - 1$ får vi att

$$PK = (x-1)(x+1) = x^2 - 1 = Q.$$

Alltså delar polynomet $x - 1$ polynomet $x^2 - 1$.

Övning 2.13. För att maskinen ska returnera 1 måste A_i vara lika med 1, och alla andra bitar vara lika med 0. Detta kan man uttrycka som att

$$R_i = (\neg A_1) \wedge (\neg A_2) \wedge \cdots \wedge (\neg A_{i-1}) \wedge (A_i) \wedge (\neg A_{i+1}) \wedge \cdots \wedge (\neg A_n).$$

Övning 2.15. För att maskinen ska returnera 1 måste C vara lika med 0, och någon av A eller B vara 1 (eller båda). Detta kan man uttrycka som att

$$R = (A \vee B) \wedge (\neg C).$$

Kapitel 3

Övning 3.1. Per definition får vi

(i) $(1010.011)_2 = 2^3 + 2^1 + 2^{-3} + 2^{-3} = 8 + 2 + 0.25 + 0.125 = 10.375$

(ii) $(11011.01)_2 = 2^4 + 2^3 + 2^1 + 2^0 + 2^{-2} = 16 + 8 + 2 + 1 + 0.25 = 27.25$

Övning 3.3. Genom att använda metoderna i Exempel 3.1.1 så man att

$$(0.3)_{10} = (0.0110011001100110\ldots)_2 = 0.\overline{01100110}.$$

Övning 3.5.

(i) Nej, enligt Exempel 3.1.1 är binärutvecklingen av 0.1 oändlig.

(ii) Ja, då $15.25 = (1111.01)_2$

(iii) Nej, eftersom $3.03125 = 2^1 + 2^0 + 2^{-5} = (11.00001)_2$ så saknas det tillräckligt med bitar efter decimalpunkten.

Övning 3.7. (i) Decimalformen av binärbråket 1.1011_2 är 1.6875, så flyttalet motsvarar

$$1 \cdot 1.6875 \cdot 2^5 = 54.$$

(ii) Decimalformen av binärbråket 1.10011_2 är 1.59375, så flyttalet motsvarar

$$-1 \cdot 1.59375 \cdot 2^3 = -12.75.$$

(iii) Decimalformen av binärbråket 1.111_2 är 1.875, så flyttalet motsvarar

$$1 \cdot 1.875 \cdot 2^1 = 3.75.$$

Övning 3.9. Varje flyttal är på formen $x = s \cdot m \cdot 2^e$, där s är -1 eller 1, m är ett rationellt tal som uppfyller $1 \leqslant m < 2$ och e är ett heltal. Eftersom m är rationellt, så finns det heltal a och b som uppfyller att $m = a/b$.

Ifall e inte är negativt är 2^e ett heltal, och då är

$$x = s \cdot \frac{a}{b} \cdot 2^e = \frac{sa2^e}{b}$$

en kvot av två heltal, och således rationellt. Om e är negativt är $-e$ positivt, och 2^{-e} ett heltal. Då kan vi skriva x som

$$x = s \cdot \frac{a}{b} \cdot 2^e = s \cdot \frac{a}{b} \cdot 2^{-(-e)} = \frac{sa}{b2^{-e}}.$$

Detta är också en kvot mellan två heltal, och således rationellt.

Eftersom $\sqrt{2}$ inte är rationellt, finns det inget flyttal som motsvarar det, och därför kan det inte lagras i en dator.

Övning 3.11. Det finns fyra fall, beroende på om x och y är ickenegativa eller ej.

(i) När x och y båda är ickenegativa så är $x + y$ ickenegativt, och således gäller

$$|x + y| = x + y = |x| + |y|.$$

(ii) När x och y båda är negativa så är $-x$ och $-y$ positiva. Därför är $-x-y$ positivt. Dessutom är $|x| = -x$ och $|y| = -y$ och således

$$|x + y| = |-(-x - y)| = -x - y = |x| + |y|.$$

(iii) När x är negativt och y är ickenegativt, så gäller att $|x| = -x$ och $|y| = y$. Dessutom gäller att $-x > x$ (talet x är ju negativt.) Nu finns två delfall.

(a) Om $x + y$ är negativt får man att

$$|x + y| = -(x + y) = -x - y < -x + y = |x| + |y|.$$

(b) Om $x + y$ är ickenegativt får man att

$$|x + y| = x + y = -x + y = |x| + |y|.$$

I vilket fall gäller att $|x + y| \leqslant |x| + |y|$.

(iv) När x är ickenegativt och y negativt, kan vi använda exakt samma resonemang som i fall (iii), genom att byta namn på x och y.

Övning 3.13. För att visa att funktionen är kontinuerlig måste vi visa att den är kontinuerlig i varje punkt. Vi tar därför en godtycklig punkt, som vi kallar x_0.

Det som ska bevisas är följande: för alla reella tal ε som är positiva, så finns det ett δ så att

$$|x_0 - y| < \delta \implies |f(x_0) - f(y)| < \varepsilon.$$

för alla reella tal y. Kärnan i uppgiften är alltså att välja δ på ett sådant sätt att den här implikationen alltid gäller. I allmänhet så beror valet av δ på ε och punkten x_0. Det som söks är alltså ett uttryck som innehåller x_0 och ε, som gör implikationen håller.

Vi börjar med att arbeta med högerledet i implikationen.

$$|f(x_0)-f(y)| = |x_0^2-2-(y^2-2)| = |x_0^2-y^2| = |(x_0-y)(x_0+y)| = |x_0-y||x_0+y|.$$

Genom att addera och subtrahera $2x_0$ i det högra absolutbeloppet får vi att

$$|x_0 - y||x_0 + y| = |x_0 - y|| - x_0 + y + 2x_0|.$$

Nu utnyttjar vi att $|-x| = |x|$ och triangelolikheten och får

$$|x_0 - y|| - x_0 + y + 2x_0| = |y - x_0||y - x_0 + 2x_0| \leqslant |y - x_0| \left(|y - x_0| + |2x_0|\right)$$

Alltså har vi visat att

$$|f(x_0) - f(y)| \leqslant |y - x_0| \left(|y - x_0| + |2x_0|\right).$$

Men ifall $|x_0 - y| = |y - x_0| < \delta$ så gäller att

$$|f(x_0) - f(y)| \leqslant |y - x_0| \left(|y - x_0| + |2x_0|\right) = \delta(\delta + 2|x_0|).$$

Det sista steget är att sätta δ till ett värde som beror av x_0 och ε. Tanken är att detta värde ska vara sådant att ifall $|x_0 - y| < \delta$, så ska $\delta(\delta + 2|x_0|) < \varepsilon$.

Hur ska vi välja δ? Det är lite knepigt. Men vi noterar att om $\delta < 1$, så gäller att

$$|f(x_0) - f(y)| \leqslant \delta(\delta + 2|x_0|) < \delta(1 + 2|x_0|).$$

Om det dessutom gäller att $\delta < \varepsilon/(1 + 2|x_0|)$, så får vi att

$$\delta(1 + 2|x_0|) < \frac{\varepsilon(1 + 2|x_0|)}{1 + 2|x_0|} = \varepsilon.$$

Därför ser vi att om δ uppfyller både att $\delta < 1$, och $\delta < \varepsilon/(1+2|x_0|)$, så gäller att

$$|f(x_0) - f(y)| < \delta(1 + 2|x_0|) < \varepsilon.$$

Så om vi kan visa att oavsett vad ε och x_0 så finns det ett positivt δ som uppfyller dessa krav, så vore vi klara.

Men detta är enkelt att åstadkomma, sätt bara δ till

$$\delta = \frac{1}{2} \min \left\{ 1, \frac{\varepsilon}{1 + 2|x_0|} \right\},$$

det vill säga hälften av det minsta av talen 1 och $\varepsilon/(1+2|x_0|)$. Detta kommer dels uppfylla att $\delta < 1$, och att $\delta < \varepsilon/(1 + 2|x_0|)$.

Kapitel 4

Övning 4.1. Resultatet blir

$$\begin{aligned}
x_1 &= \mathrm{Mod}_{10}(3 \cdot 0 + 2) = \mathrm{Mod}_{10}(2) = \mathbf{2},\\
x_2 &= \mathrm{Mod}_{10}(3 \cdot 2 + 2) = \mathrm{Mod}_{10}(8) = \mathbf{8},\\
x_3 &= \mathrm{Mod}_{10}(3 \cdot 8 + 2) = \mathrm{Mod}_{10}(26) = \mathbf{6},\\
x_4 &= \mathrm{Mod}_{10}(3 \cdot 6 + 2) = \mathrm{Mod}_{10}(20) = \mathbf{0},\\
x_5 &= \mathrm{Mod}_{10}(3 \cdot 0 + 2) = \mathrm{Mod}_{10}(2) = \mathbf{2}.
\end{aligned}$$

Perioden är 4.

Övning 4.3.

(i) Om $a = 0$ så blir

$$x_{n+1} = \mathrm{Mod}_m(0 \cdot x_n + b) = \mathrm{Mod}_m(b) = b.$$

Alla pseudoslumptal blir alltså lika med b.

(ii) Om $a = 1$ så blir

$$x_{n+1} = \mathrm{Mod}_m(x_n + b).$$

Låt oss se vad som händer om $m = 10$ och $x_0 = 0$ för olika värden på b.

b	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
0	0	0	0	0	0	0	0	0	0	0
1	1	2	3	4	5	6	7	8	9	0
2	2	4	6	8	0	2	4	6	8	0
3	3	6	9	2	5	8	1	4	7	0
4	4	8	2	6	0	4	8	2	6	0
5	5	0	5	0	5	0	5	0	5	0
6	6	2	8	4	0	6	2	8	4	0
7	7	4	1	8	5	2	9	6	3	0
8	8	6	4	2	0	8	6	4	2	0
9	9	8	7	6	5	4	3	2	1	0

De flesta av dessa talföljder är väldigt förutsägbara, inte minst de som motsvarar $b = 1$ och $b = 9$.

Övning 4.5.

(i) Ja, för $2^1 = 2$ och $2^2 = 4 \equiv 1 \pmod{3}$.

(ii) Nej, för $2^1 = 2$, $\quad 2^2 = 4 \equiv 0 \pmod{4}$ och om $k > 2$ gäller $2^k = 4 \cdot 2^{k-2}$, vilket är delbart med 4 och därför är kongruent med 0 modulo 4 enligt Övning 2.10. Det går alltså inte att skriva 1 eller 3 som 2^k modulo 4.

(iii) Vi gör en tabell och skriver upp värdena på a^i modulo 7, för olika värden på a. (Vi kan använda Hjälpsats 4.1.8 för att räkna ut a^{i+1} från a^i.)

a	a^1	a^2	a^3	a^4	a^5	a^6	a^7
1	1	1	1	1	1	1	1
2	2	4	1	2	4	1	2
3	3	2	6	4	5	1	3
4	4	2	1	4	2	1	4
5	5	4	6	2	3	1	5
6	6	1	6	1	6	1	6

Detta visar att 3 och 5 är primitiva element modulo 7.

Övning 4.7. Välj till exempel $m = 10$. Ett konkret exempel är $x = 7$, $y = 2$, $z = 2$. Då gäller nämligen

$$7 \cdot 2 = 14 \equiv 4 = 2 \cdot 2 \pmod{10},$$

men vid division med 2 fås

$$7 \not\equiv 2 \pmod{10}.$$

(Faktum är att om $2x = 2y + 10$ så ger division att $x = y + 5$, så x och y kan då inte vara kongruenta modulo 10.)

Övning 4.9. Vi börjar med att lista alla positiva tal på formen $2^k - 1$ som är mindre än 500:

$$1, \quad 3, \quad 7, \quad 15, \quad 31, \quad 63, \quad 127, \quad 255.$$

Vi går igenom denna lista och kontrollerar om varje tal är ett primtal eller ej. 1 är förstås inget primtal, för alla primtal är heltal större än 1. Vi vet att 3 och 7 är primtal, medan 15 inte är ett primtal eftersom $15 = 3 \cdot 5$.

Talet 31 är varken delbart med 2, 3 eller 5. Då kan 31 inte heller vara delbart med någon multipel av de talen. Eftersom $7 \cdot 7 = 49 > 31$ så kan 31 inte vara delbart med något tal större än eller lika med 7 heller. Alltså är 31 ett primtal.

Talet 63 är delbart med 3 (för $63 = 21 \cdot 3$), så det är inte ett primtal.

Talet 127 är inte delbart med 2, 3 eller 5. Testdivision med 7 ger att $127 = 18 \cdot 7 + 1$, så det är inte delbart med 7 heller. Nästa kandidat som 127 kan vara delbart med är talet 11, men $11 \cdot 11 = 121$, vilket betyder att $127 = 11 \cdot 11 + 6$, så det är inte delbart med 11. Eftersom $13 \cdot 13 = 169 > 127$ så kan 127 inte vara delbart med något annat tal. Alltså är 127 ett primtal.

Talet 255 är delbart med 5 och alltså inget primtal.

Sammanfattningsvis är primtalen 3, 7, 31 och 127. Det finns alltså 4 Mersenneprimtal som är mindre än 500.

Övning 4.11. Meddelandet är krypterat med ett Caesarkrypto. Genom att prova sig fram kan man komma fram till att nyckeln är 1, så det avkrypterade meddelandet är ”Stockholms matematiska cirkel”.

Programmeringsuppgifter

Övning 4.P1. Förslag på programkod (Python):

```
m = 2147483647
a = 16807
b = 0

x0 = 1

x = x0
for n in range(20):
    x = (a*x + b) % m
    print(x)
```

Exempel på körning av programmet:

```
16807
282475249
1622650073
984943658
1144108930
```

```
470211272
101027544
1457850878
1458777923
2007237709
823564440
1115438165
1784484492
74243042
114807987
1137522503
1441282327
16531729
823378840
143542612
```

Övning 4.P3. Förslag på programkod (Python):

```
m = 2147483647
a = 16807
b = 0

x0 = 1

x = x0
for n in range(20):
    x = (a*x + b) % m
    u = x / m
    print(u)
```

Exempel på körning av programmet:

```
7.826369259425611e-06
0.13153778814316625
0.7556053221950332
0.4586501319234493
0.5327672374121692
0.21895918632809036
0.04704461621448613
0.678864716868319
0.6792964058366122
0.9346928959408276
0.3835020774898595
0.5194163720679545
0.8309653461123655
0.034572110527461446
0.05346163504452521
0.5297001933351626
0.6711493840772423
```

```
0.007698186211147432
0.3834156507548949
0.06684223751856118
```

Övning 4.P5. Förslag på programkod (Python):

```
m = 5
a = 2
# Lista som visar om talen från 0 till m-1 har genererats.
lista = [False] * m
# Låt x = a^i (mod m), bocka av de tal som har genererats.
# Det räcker att gå upp till i=m-1.
x = a
for i in range(1, m):
   lista[x] = True
   x = (x*a) % m
# Gå nu igenom listan och se om alla tal har bockats av.
okej = True
for i in range(1, m):
   if lista[i] == False:
      okej = False
      break
# Skriv ut resultat.
if okej:
   print(f'a är ett primitivt element modulo m')
else:
   print(f'a är INTE ett primitivt element modulo m')
```

Programmet ska visa att 2 och 3 är primitiva element modulo 5, medan 1 och 4 inte är det. Programmet ska också visa att 2 är ett primitivt element modulo 547, medan 3 inte är det.

Kapitel 5

Övning 5.1. Låt $s_1, \ldots, s_n$ vara orden i ett ändligt språk. Då är

$$s_1 \cup s_2 \cup \cdots \cup s_n$$

ett reguljärt uttryck för språket.

Övning 5.3. Ett förslag är $(aa)^*b(bb)^*$.

Övning 5.5. Ett förslag är $0 \cup 1(0 \cup 1)^*00$.

Övning 5.7. Ett exempel är

$$\{ab, aab, aaab, \ldots, \} = \{a^n b \mid n \geqslant 1\}.$$

Övning 5.9. Ifall $n = 0$, är $L^n = \{\varepsilon\}$, så antalet ord i L^0 är 1, vilket är lika med k^0.

Om n är positivt, kan man resonera som följer. Varje ord i L^n konstrueras genom att välja n stycken ord ur L och sammansätta dem. Varje gång man ska välja ord har man k alternativ. Sammantaget har man då

$$\underbrace{k \cdots k}_{n \text{ stycken}} = k^n$$

alternativ.

Sammantaget innehåller $L^0 \cup L \cup \cdots \cup L^n$

$$1 + k + k^2 + \cdots + k^n$$

ord. Detta är en geometrisk serie med startvärde 1 och $n+1$ termer, och summa

$$\frac{k^{n+1} - 1}{k - 1}.$$

Fotnot: den sista likheten bevisas genom att sätta

$$s = 1 + k + k^2 + \cdots + k^n.$$

Då gäller

$$ks = k + k^2 + \cdots + k^{n+1}$$

och således

$$ks - s = k^{n+1} + k^n - k^n + \cdots + k - k - 1 = k^{n+1} - 1.$$

Genom att faktorisera ut s och dividera bort en faktor kan man isolera summan s i vänsterledet:

$$(k-1)s = k^{n+1} - 1 \iff s = \frac{k^{n+1} - 1}{k - 1}.$$

Övning 5.11. (i) Intuitivt kan man tänka man börjar med det tomma ordet och sedan tar man ett tecken i taget. Vill man göra ett induktionsbevis, kan man använda resultatet i deluppgift (ii).

När $n = 0$ är ordet tomt, och dess suffixspråk är ε. Antag att ekvationen gäller för något p. Då ger ekvationen i deluppgift (ii) omedelbart att

$$\mathrm{Suf}(\sigma_1 \cdots \sigma_{p+1}) = (\mathrm{Suf}(\sigma_1 \cdots \sigma_p)\sigma_{p+1}) \cup \mathrm{Suf}(\sigma_{p+1})$$

Per antagande gäller att

$$\begin{aligned}\mathrm{Suf}(\sigma_1 \cdots \sigma_p)\sigma_{p+1} &= \{\sigma_1\sigma_2 \cdots \sigma_p, \sigma_2 \cdots \sigma_p, \ldots, \sigma_p, \varepsilon\}\sigma_{p+1} \\ &= \{\sigma_1 \cdots \sigma_p\sigma_{p+1}, \sigma_2 \cdots \sigma_{p+1}, \ldots, \sigma_p\sigma_{p+1}, \sigma_{p+1}\}.\end{aligned}$$

Eftersom $\mathrm{Suf}(\sigma_{p+1}) = \{\varepsilon, \sigma_{p+1}\}$, finner man att

$$\mathrm{Suf}(\sigma_1 \cdots \sigma_{p+1}) = \{\sigma_1\sigma_2 \cdots \sigma_{p+1}, \sigma_2 \cdots \sigma_{p+1}, \ldots, \sigma_{p+1}, \varepsilon\},$$

vilket avslutar induktionssteget.

(ii) Låt wu vara ett ord i L_1L_2, där w ligger i L_1 och u ligger i L_2. Om v är ett suffix till wu, så finns det ett ord z så att $zv = wu$. Det finns då två alternativ.

Antingen är z längre än eller lika långt som w. Då måste v vara ett suffix till u, och ligger alltså i $\mathrm{Suf}(L_2)$.

Eller så är z kortare än w. Då måste $v = w_1u$, där w_1 är ett suffix till w. Med andra ord ligger v i sammansättningen $\mathrm{Suf}(L_1)L_2$.

Sammantaget gäller att v ligger i unionen $\mathrm{Suf}(L_1)L_2 \cup \mathrm{Suf}(L_2)$.

(iii) Ett ord w ligger i $L_1 \cup L_2$ om och endast om det antingen ligger i L_1 eller L_2. Ett suffix till w kommer därför antingen ligga i $\mathrm{Suf}(L_1)$ eller i $\mathrm{Suf}(L_2)$, varför suffixspråket av $L_1 \cup L_2$ är $\mathrm{Suf}(L_1) \cup \mathrm{Suf}(L_2)$.

(iv) Ett ord ligger i L^* om och endast om det har formen

$$w = w_1w_2 \cdots w_n$$

där w_1, ..., w_n ligger i L. Ett suffix till detta ord har formen

$$uw_{k+1}w_{k+2} \cdots w_n$$

där u är ett suffix till ordet w_k. Eftersom w_k är ett ord i L, och

$$w_{k+1}w_{k+2} \cdots w_n$$

ligger i L^*, så ligger suffixet i språket $\mathrm{Suf}(L)L^*$.

Övning 5.13. Ekvationerna är

(i) $\mathrm{Pre}(\sigma_1\sigma_2 \cdots \sigma_n) = \{\sigma_1\sigma_2 \cdots \sigma_n, \sigma_1 \cdots \sigma_{n-1}, \ldots, \sigma_1, \varepsilon\}$

(ii) $\mathrm{Pre}(L_1L_2) = \mathrm{Pre}(L_1) \cup L_1 \mathrm{Pre}(L_2)$,

(iii) $\mathrm{Pre}(L_1 \cup L_2) = \mathrm{Pre}(L_1) \cup \mathrm{Pre}(L_2)$,

(iv) $\mathrm{Pre}(L^*) = L^* \mathrm{Pre}(L)$.

för alla språk L_1 och L_2 och tecken σ_1, ..., σ_n. Bevisen för dem är exakt likadana som i Övning 5.11, fast man byter ordning på alla sammansättningar.

Prefixspråket av $ab^*(ab)^*$ ges då av

$$\begin{aligned}
\operatorname{Pre}(ab^*(ab)^*) &= \operatorname{Pre}(ab^*) \cup ab^* \operatorname{Pre}((ab)^*) \\
&= (\operatorname{Pre}(a) \cup a \operatorname{Pre}(b^*)) \cup ab^* \operatorname{Pre}((ab)^*) \\
&= (\varepsilon \cup a \cup a(b^* \operatorname{Pre}(b))) \cup ab^* \operatorname{Pre}((ab)^*) \\
&= (\varepsilon \cup \underbrace{a \cup a(b^*(\varepsilon \cup b)}_{ab^*}) \cup ab^* \operatorname{Pre}((ab)^*) \\
&= \varepsilon \cup ab^* \cup ab^*(ab)^* \operatorname{Pre}(ab) \\
&= \varepsilon \cup ab^* \cup ab^* \underbrace{(ab)^*(\varepsilon \cup a \cup ab)}_{(ab)^* \cup (ab)^* a} \\
&= \varepsilon \cup ab^* \cup ab^*((ab)^* \cup (ab)^* a) \\
&= \varepsilon \cup ab^*(\underbrace{\varepsilon \cup (ab)^*}_{(ab)^*} \cup (ab)^* a) \\
&= \varepsilon \cup ab^*((ab)^* \cup (ab)^* a) \\
&= \varepsilon \cup ab^*(ab)^* \cup ab^*(ab)^* a.
\end{aligned}$$

Observera att förenklingarna använder resultatet i Övning 5.15.

Övning 5.15. Ett ord w ligger $\alpha(\beta \cup \gamma)$ om och endast om $w = w_1 w_2$, där w_1 ligger i α och w_2 ligger antingen i β eller i γ.

Det förra fallet är ekvivalent med att w ligger i $\alpha\beta$, det senare med att w ligger i $\alpha\gamma$. Alltså måste w ligga i unionen $\alpha\beta \cup \alpha\gamma$. Således är det bevisat att

$$w \in \alpha(\beta \cup \gamma) \iff w \in \alpha\beta \cup \alpha\gamma,$$

vilket innebär att $\alpha(\beta \cup \gamma) = \alpha\beta \cup \alpha\gamma$.

Övning 5.17. (i) De palindrom över $\{a, b\}$ som har fyra tecken eller färre är

$$\varepsilon, a, b, aa, ab, aaa, aba, bab, bbb, aaaa, abba, baab, bbbb.$$

(ii) En rekursiv definition av ett språk sker i två steg. Först anger man ett eller flera basfall: ord som per definition ingår i mängden. Därefter ges en eller flera rekursionssteg, av typen

om w ingår i mängden, så ingår u i mängden

där u definieras i termer av w. För att hitta rätt rekursionssteg, låt w vara en palindrom. Då finns det två sätt att bilda nya palindrom: antingen lägger man till ett a på båda sidor, eller så lägger man till ett b på båda sidor. Med andra ord

om w är en palindrom, så är awa och bwb palindrom.

Basfallen är de som ej kan beskrivas på detta sätt. Det finns tre stycken: ε, a och b. Den kompletta rekursiva definition är således.

(a) ε, a, b ingår i $P(\{a,b\})$.

(b) om w ligger i $P(\{a,b\}$, så ligger awa och bwb i $P(\{a,b\})$.

(iii) Samma resonemang som ovan kan tillämpas på ett godtyckligt alfabet, med resultatet att $P(\Sigma)$ defineras genom

(a) ε och σ ligger i $P(\Sigma)$, där σ är ett godtyckligt tecken i Σ.

(b) om w ligger i $P(\Sigma)$, ligger $\sigma w \sigma$ i $P(\Sigma)$, där σ är ett godtyckligt tecken i Σ.

Kapitel 6

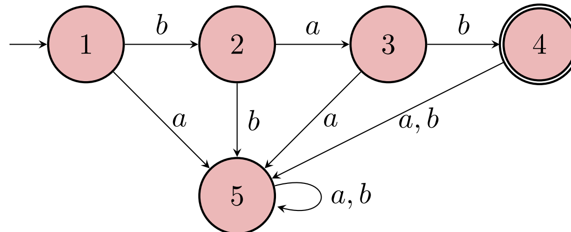
Övning 6.1. Maskinen drivs enligt följande.

Ord	$aaabba$	$aabba$	$abba$	bba	ba	a	ε
Tillstånd	1	1	1	2	2	2	2

Övning 6.3. Tillståndsmängden är $\{1,2,3,4,5\}$. Starttillståndet är 1, det accepterande tillståndet 4 och övergångsfunktionen ges nedan.

Ω	a	b
1	5	2
2	3	5
3	5	4
4	5	5
5	5	5

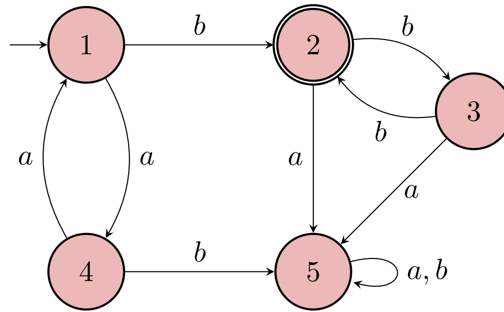
Nedan följer en grafisk representation.



Övning 6.5. Tillståndsmängden är $\{1,2,3,4,5\}$. Starttillståndet är 1, det accepterande tillståndet är 3 och övergångsfunktionen ges nedan.

Ω	a	b
1	2	3
2	1	5
3	5	4
4	5	3
5	5	5

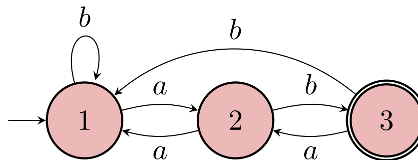
Nedan följer en grafisk representation.



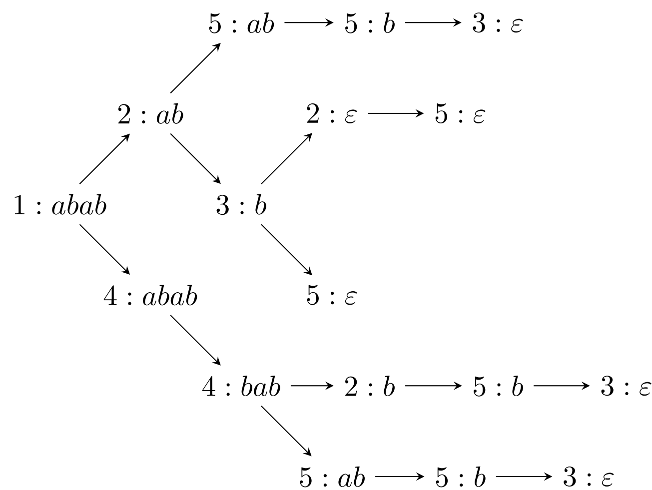
Övning 6.7. Tillståndsmängden är $\{1, 2, 3\}$. Starttillståndet är 1, det accepterande tillståndet är 3 och övergångsfunktionen ges nedan.

Ω	a	b
1	2	1
2	1	3
3	2	1

Nedan följer en grafisk representation.



Övning 6.9. Beräkningen sammanfattas i figuren nedan.

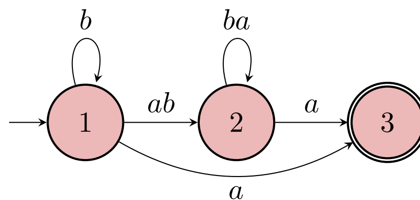


Ordet *abab* kan alltså driva maskinen till tillstånd 2, 3 och 5. Eftersom 5 är ett accepterande tillstånd, så ingår *abab* i maskinens språk.

Övning 6.11. Tillståndsmängden är $\{1, 2, 3\}$. Starttillståndet är 1, det accepterande tillståndet är 3 och övergångsrelation ges nedan.

Ω	Σ^*	Ω
1	b	1
1	ba	2
1	a	3
2	ba	2
2	a	3

En grafisk representation kan se ut som följer.



Övning 6.13. (i) $\{1, 2, 3, 4, 5\}$.

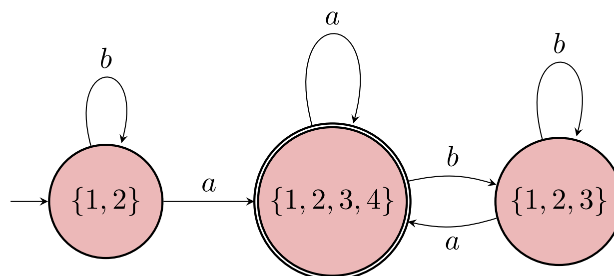
(ii) $\{2, 3, 5\}$

(iii) $\{2, 3, 4, 5\}$

Övning 6.15. Starttillståndet är ε-tillslutningen av $\{1\}$, det vill säga $\{1, 2\}$.

Ω	a	b
$\{1, 2\}$	$\{1, 2, 3, 4\}$	$\{1, 2\}$
$\{1, 2, 3, 4\}$	$\{1, 2, 3, 4\}$	$\{1, 2, 3\}$
$\{1, 2, 3\}$	$\{1, 2, 3, 4\}$	$\{1, 2, 3\}$

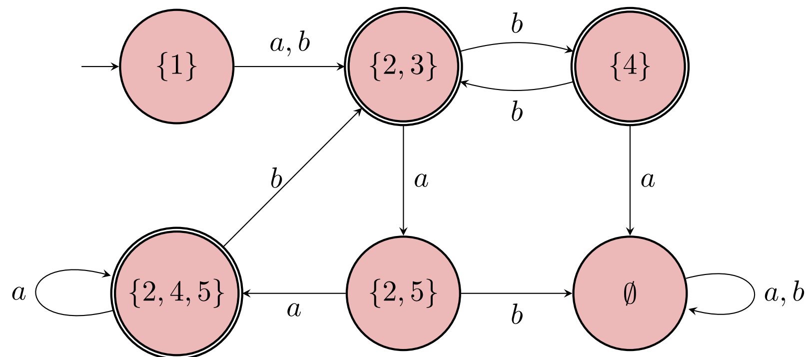
De accepterande tillstånden är $\{1, 2, 3, 4\}$ och $\{1, 2, 3\}$. Nedan är en grafisk representation.



Övning 6.17. Starttillståndet ges av ε-tillslutningen av $\{1\}$, vilket är $\{1\}$. Därefter beräknas delmängdskonstruktionen successivt.

Ω	a	b
$\{1\}$	$\{2,3\}$	$\{2,3\}$
$\{2,3\}$	$\{2,5\}$	$\{4\}$
$\{2,5\}$	$\{2,4,5\}$	$\emptyset$
$\{4\}$	$\emptyset$	$\{2,3\}$
$\{2,4,5\}$	$\{2,4,5\}$	$\{2,3\}$
$\emptyset$	$\emptyset$	$\emptyset$

De accepterande tillstånden är $\{2,3\}$, $\{4\}$ och $\{2,4,5\}$. Nedan är en grafisk representation.

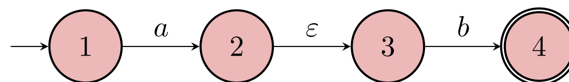


Kapitel 7

Övning 7.1. Tillståndsmängden är $\{1,2,3,4\}$, starttillstånd 1, accepterande tillstånd 4 och övergångsrelationen är

Ω	1	2	3
Σ^*	a	ε	b
Ω	2	3	4

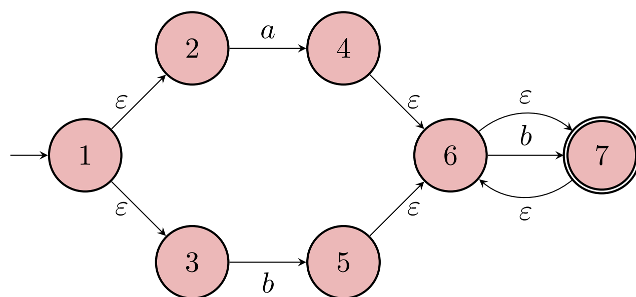
En grafisk representation ges nedan.



Övning 7.3. Tillståndsmängden är $\{1,2,3,4,5,6,7\}$, starttillstånd 1, accepterande tillstånd 7 och övergångsrelationen är

Ω	1	1	2	3	4	5	6	6	7
Σ^*	ε	ε	a	b	ε	ε	b	ε	ε
Ω	2	3	4	5	6	6	7	7	6

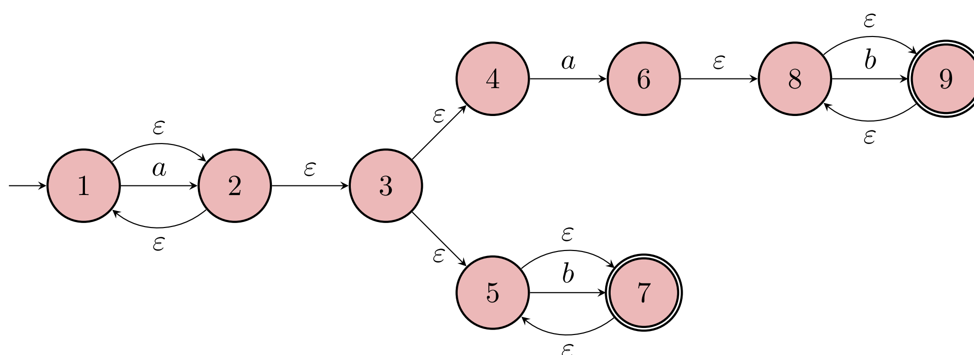
En grafisk representation ges nedan.



Övning 7.5. Tillståndsmängden är $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, starttillstånd 1, accepterande tillstånd 8 och 9 och övergångsrelationen är

Ω	1	1	2	2	3	3	5	5	7	4	6	8	8	9
Σ^*	a	ε	ε	ε	ε	ε	b	ε	ε	a	ε	b	ε	ε
Ω	2	2	1	3	4	5	7	7	5	6	8	9	9	8

En grafisk representation ges nedan.



Övning 7.7. Ett ord accepteras av maskinen om och endast om det ingår i det reguljära språket

$$(b^*a)^*,$$

det vill säga om det antingen är tomt eller slutar på a.

Övning 7.9. Ett ord accepteras av maskinen om och endast om det ingår i det reguljära språket

$$(aba)^*$$

det vill säga om det består av ett antal upprepningar av aba.

Övning 7.11. Ett ord driver maskinen från tillstånd 1 till tillstånd 2 ifall det ingår i språket $a(ba)^*$.

Övning 7.13. Nedan ges en tabell med korrekta svar. Observera att uttrycken är förenklade.

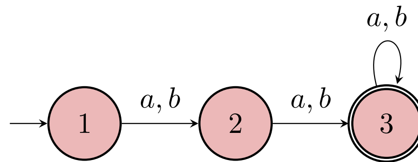
	0	1	2
α_{11}^k	a	a^*a	$(a^*b)^*a^*a$
α_{12}^k	b	a^*b	$(a^*b)^*a^*b$
α_{21}^k	a	a^*a	$(a^*b)^*a^*a$
α_{22}^k	b	a^*b	$(a^*b)^*a^*b$

Övning 7.15. Ja. Mängden driver maskinen till tillstånd 1, 2, 4 och 3 (i den ordningen).

Övning 7.17. Ja. Orden a och bb särskiljs av $z = a$ och orden a och bab särskiljs av $z = b$, medan orden bb och bab särskiljs av $z = \varepsilon$.

Övning 7.19. Maskinen är minimal. En särskiljande mängd ord ges av $\{\varepsilon, b, bb\}$. Orden ε och b särskiljs av $z = b$, medan ε och bb samt b och bb särskiljs av $z = \varepsilon$.

Övning 7.21. Maskinen är inte minimal. En tillståndmaskin med tillståndsmängd $\{1, 2, 3\}$ som avgör samma språk ges nedan.



Dess starttillstånd är 1, accepterande tillstånd 3 och övergångsfunktionen ges av

Ω	a	b
1	2	2
2	3	3
3	3	3

Övning 7.23. Palindromspråket över $\{a, b\}$ är *inte* reguljärt. Det särskiljer till exempel språket

$$\{a^n : n \text{ icke-negativt heltal}\}$$

eftersom om m är skilt från n, så är $a^m b^m$ en palindrom, men inte $a^n b^m$ (se Exempel 7.3.5 för detaljer.)

Övning 7.25. Språket är reguljärt: ett uttryck för det är

$$a^* \cup a^* b a^* b (a \cup b)^*$$

Förslag till vidare läsning

Max Tegmark, *Liv 3.0: att vara människa i den artificiella intelligensens tid*
Volante, 2017

Timothy Sauer, *Numerical Analysis*
Pearson, 2014, Andra upplagan

Donald Knuth, *The Art of Computer Programming (vol. 2: Seminumerical Algorithms)*
Addison-Wesley, 1998, Tredje upplagan

Lennart Salling, *Formella språk, automater och beräkningar*
Egen utgivning, 2001, Andra upplagan

Shyamalendu Kandar, *Introduction to automata theory, formal languages and computation*
Pearson, 2013

Sakregister

Symboler

$\{\cdot\}$, mängdparenteser . . . **2**, 3
$|\cdot|$, antal element . . . 2
$|\cdot|$, absolutbelopp . . . 42
$(\cdot,\cdot)$, ordnat par . . . 5
$[\cdot,\cdot]$, intervall . . . 41
$\emptyset$, tomma mängden . . . 2
$\in$, tillhör . . . 2
$\subseteq$, delmängd . . . 3
$\cup$, union . . . 4
$\cap$, snitt . . . 4
$\setminus$, mängddifferens . . . 4
$\times$, kartesisk produkt . . . 5
$\vee$, eller, OR . . . 8
$\veebar$, XOR . . . 17
$\wedge$, och, AND . . . 8
$\neg$, negation, NOT . . . 9
$\Longrightarrow$, implikation . . . 10
$\uparrow$, NAND . . . 17
$\equiv$, kongruens . . . 23

A

absolutbelopp, $|\cdot|$ . . . 42
alfabet . . . 60
algoritm . . . 43, 69
AND, $\wedge$ . . . 8
avrundningsfel . . . 37

B

$\mathbb{B} = \{0, 1\}$ . . . 8
bas 10 . . . 19
bas 2 . . . 19
basfall
  i induktionsbevis . . . 12
  i rekursion . . . 7
bevis . . . 1
  direkt bevis . . . 11
  induktionsbevis . . . 12
  motsägelsebevis . . . 11
binary32, binary64 . . . 38
binära talsystemet . . . 19
binärbråk . . . 34
binärpunkt . . . 33
bisektion . . . *se* intervallhalvering
bit . . . 21
Boolesk funktion . . . 9
Boolesk variabel, $\mathbb{B}$ . . . 8
byte . . . 22

C

Church-Turings tes . . . 69

D

De Morgans lagar . . . 16
decimala talsystemet . . . 19
decimalpunkt . . . 33
definition . . . 1
definitionsmängd . . . 5
delbarhet, a delar b . . . 25
delmängd, $\subseteq$ . . . 3
delmängdskonstruktionen . . . 81
deterministisk tillståndsmaskin . . 70
  formell definition . . . 71
  kleenes algoritm . . . 93, 94
  språk . . . 72
  särskilja ord . . . 98
differens av mängder, $\setminus$ . . . 4
distributiva lagen . . . 68
double-precision . . . 38

E

ekvation . . . 40
element . . . 2
eller, $\vee$ . . . 8
exponent i flyttal . . . 36

F

$f : X \to Y$, funktion . . . 5
Fibonaccis talföljd . . . 7
fixtal . . . 36
flyttal . . . 36
funktion . . . 5
  kontinuerlig funktion . . . 42
  likhet mellan funktioner . . . 7

G

graf till funktion . . . 5
grind . . . *se* logisk grind
gräns
  övre gräns . . . 45

H

heltal, $\mathbb{Z}$ . . . 2

osignerat heltal . . . 21
signerat heltal . . . 28

I
icke, $\neg$ . . . 9
icke-deterministisk
delmängdskonstruktion . . . 81
formell definition . . . 77
språk . . . 78
tillståndsmaskin . . . 75, 77
implikation, $\Longrightarrow$ . . . 10
indata till funktion . . . 5
induktionsbevis . . . 12
över ordens längd . . . 74
över språks komplexitet . . . 90
induktionssteg . . . 12
intervall, $[\cdot, \cdot]$ . . . 41
intervallhalvering . . . 43

J
jämnt tal . . . 1

K
kartesisk produkt, $\times$ . . . 5
kleenes algoritm . . . 93, 94
kleenetillslutning
alfabet . . . 63
språk . . . 64
kongruens, $\equiv$ . . . 23
kontinuerlig funktion . . . 42
krypteringsnyckel . . . 55

L
linjär kongruensgenerator . . . 49
logisk grind . . . 25
lösning till ekvation . . . 40

M
mantissa . . . 36
maskin . . . *se* tillståndsmaskin
maskinepsilon . . . 38
maskinprecision . . . *se* maskinepsilon
mellanliggande värden . . . 42
minsta övre gräns . . . 45
Mod_n . . . 24
modulo, $a \equiv b \pmod{n}$ . . . 23
motsägelsebevis . . . 11
multiplikationsprincipen . . . 11
mängd . . . 2
målmängd . . . 5

N
NAND, $\uparrow$ . . . 17
negation, $\neg$ . . . 9
nollställe . . . 40
NOT, $\neg$ . . . 9
nyckel, kryptering . . . 55

O
och, $\wedge$ . . . 8
oktett . . . 22
om P så Q, $\Longrightarrow$ . . . 10
OR, $\vee$ . . . 8
ord . . . 60
driva
tillståndsmaskin . 70, 72, 78
längd . . . 60
n-faldig upprepning . . . 62
palindrom . . . 68
prefix . . . 61
reversion . . . 62
sammansättning . . . 61
suffix . . . 61
tomt . . . 60
ordnat par, $(\cdot, \cdot)$ . . . 5
osignerat heltal . . . 21

P
period hos slumptalsgenerator . . . 50
precision hos flyttal . . . 37
double-precision . . . 38
single-precision . . . 38
prefixegenskapen . . . 67
primitivt element . . . 50
primtal . . . 13
pseudoslumptal . . . 48
punkt
binärpunkt & decimalpunkt . 33
påstående . . . 7

Q
$\mathbb{Q}$, mängden av rationella tal . . . 2

R
$\mathbb{R}$, mängden av reella tal . . . 2
rationella tal, $\mathbb{Q}$ . . . 2
reella tal, $\mathbb{R}$ . . . 2
reguljärt

- språk 65
- uttryck 66

rekursivt definierad funktion . . . 6, 7

S

sanningstabell 9
sanningsvärde (0 eller 1) 8
sats 1
satsen om mellanliggande värden 42
signerat heltal 28
signifikand . . . . . *se* mantissa
signifikant
- mest signifikant bit 29
- minst signifikanta siffror 23

single-precision 38
slump 48
slumpfrö 48
snitt, $\cap$ 4
språk 62
- deterministisk
  - tillståndsmaskin 72
- icke-deterministisk
  - tillståndsmaskin 78
- kleenetillslutning 64
- n-faldig upprepning 63
- palindrom- 68, 105
- prefix- 65
- reguljärt 65
- sammansättning 63
- suffix- 65
- särskilja ord 99

stark induktion 13
särskiljning
- deterministisk
  - tillståndsmaskin 98
- språk 99

T

talsystem
- binära 19
- decimala 19

tecken i flyttal 36
teckenbit 29
tillslutning
- ε- 80
- w- 80

tillstånd
- accepterande 70, 77
- start- 70, 77

tillstånd hos slumptalsgenerator . 48
tillståndsmaskin 69
- deterministisk 70
- driva 70, 72, 78
- formell definition 71, 77
- grafisk representation av 70
- hänga sig 76
- icke-deterministisk 75
- kleenes algoritm 93, 94
- språk 72, 78
- särskilja ord 98
- tillstånd 70
- övergångsfunktion 70
- övergångsrelation 77

tomma mängden, $\emptyset$ 2
tomma ordet, ε 60
turingmaskin 69
tvåkomplementsform 29

U

udda tal 1
union, $\cup$ 4
utdata från funktion 5

V

venndiagram 3

X

XOR, $\veebar$ 17

Z

$\mathbb{Z}$, mängden av heltal 2
$\mathbb{Z}_{\geqslant 0}$, ickenegativa heltal 7

Ä

äkta slump 48

Ö

överflöde 22
övergång
- ε- 75
- glupsk 76
- tillståndsmaskin 70

övergångsfunktion 70
- utvidgad 72

övergångsrelation 77
- utvidgad 78

övergångstriplar 71
övre gräns 45