Kapittel 0 – Introduksjon

0.1 ALGORITMENS ROLLE

Algoritme: Oppskrift på hvordan man løser en bestemt oppgave (eks. matoppskrift). Før en maskin kan utføre en oppgave, må en algoritme for hvordan oppgaven kan utføres bli oppdaget og representert på en måte datamaskinen forstår. Altså er en algoritme et sett med instruksjoner datamaskinen kan utføre.

Program: En representasjon av en algoritme.

Programmering: Prosessen med å utvikle et program på et språk maskinen forstår.

Softvare: Programvare – programmene på en datamaskin.

Hardware: Maskinvare – de fysiske delene maskinen består av.

Data: Representasjon av informasjon.

Metadata: Sier noe om hva slags type data det er snakk om.

Hei.

Har ordren din på oppfølging for å gi deg nærmere informasjon om når den er ventet inn på vårt lager.

For øyeblikket er siste informasjon vi har fra produsent, forventet inn 27/12

0.2 DATAMASKINENS OPPRINNELSE

Abacus: regneapparat fra tidlig gresk og romersk sivilisasjon (brett med kuler).

Pascal og Leibniz: blant oppfinnerne av regnemaskinene/datamaskinene basert på tannhjul.

Augusta Ada Byron: kjent som verdens første programmerer.

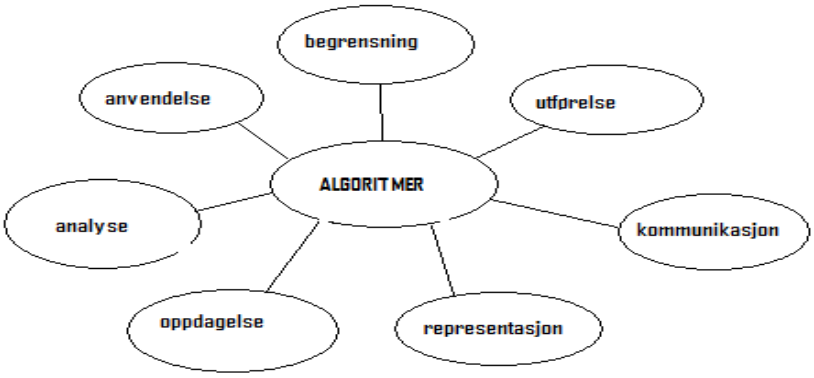
Herman Hollenrith: representasjon av informasjon som hull i et papirkort på slutten av 1800-tallet. Arbeidet førte til at IBM ble opprettet.

George Stibitz: 1940 – elektromekanisk maskin.

Steve Jobs og Stephen Wozniak: grunnla Apple Inc i 1976, etter å ha bygd den første hjemme PC-en.

1981: IBM introduserte den første desktop maskinen (tastatur og skrivebord basert) kalt Personal Computer. Software var utarbeidet av Microsoft.

0.3 VITENSKAPEN OM ALGORITMEN



Algoritmens rolle i informasjonsteknologi:

0.4 ABSTRAKSJON

Skille mellom de eksterne egenskapene til en komponent og de interne detaljene til en komponents konstruksjon. Vi trenger ikke forstå alt. Dette gjør det mulig å konstruere, analysere og håndtere store komplekse datasystemer.

Eksempler på abstraksjon i informatikk:

- Vi kan utvikle en algoritme uten å forstå hvordan den kan programmeres.
- Vi kan programmere et program uten å vite hvordan maskinen i detalj utfører programmet.
- Vi kan lagre data på en harddisk uten å forstå hvordan harddisken fungerer.

0.5 KONTUREN I STUDIET

(Her står det bare litt om de forskjellige kapitlene i boka)

0.6 SOSIALE ETTERVIRKNINGER

Konsekvensbasert etikk: Analyserer problemet basert på konsekvensene av forskjellige utfall.

Pliktbasert etikk: eks – medlemmene i et samfunn har forskjellige plikter som skal utføres.

Lovbasert etikk: Lover bestemmer den etiske atferden.

Karakterbasert etikk: God atferd er et resultat av god karakter.

Eksempler på it i bruk:

- Informasjonssystemer
- Vitenskapelig anvendelse
- Visualisering
- Bildebehandling

- Kommunikasjon
- Informasjon

Kapittel 1 – Datalagring

1.1 LAGRING AV INFORMASJON VED HJELP AV BITMØNSTRE

Bits: kombinasjoner av 0 og 1, som informasjon omkodes til i dagens datamaskiner.

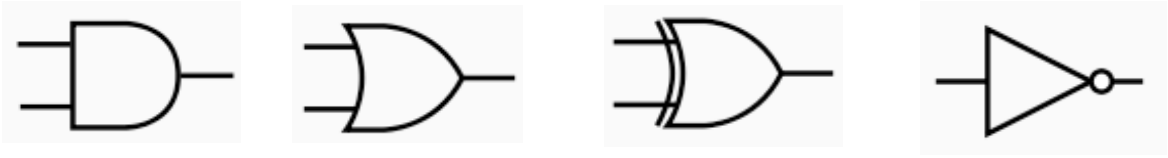
Boolsk algebra: en matematisk notasjon for å uttrykke logiske funksjoner ved hjelp av sanne og usanne variabler. Vi tenker oss at 0 representerer usann og 1 representerer sann.

Boolske operasjoner: Operasjoner som manipulerer sann/usann variabler. Oppkalt etter matematikeren George Boole.

Porter/gates: en fysisk innretning som tar en eller flere elektroniske signaler inn, og produserer et elektronisk signal ut.

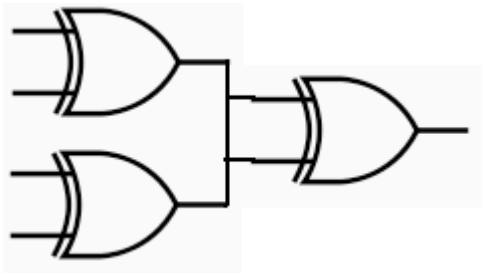
Ulike porter og deres symboler:

AND OR XOR(exclusive) NOT



Input	**Output**	**Input**	**Output**	**Input**	**Output**	**Input**	**Output**
00	0	00	0	00	0	0	1
01	0	01	1	01	1	1	0
10	0	10	1	10	1		
11	1	11	1	11	0		

I dagens datamaskiner er portene stort sett implementert som små elektroniske kretsløp, hvor 0 og 1 representerer strøm og ikke strøm.



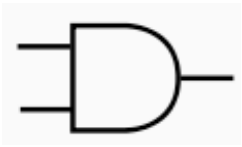
Flip-flop: et kretsløp som produserer 0 eller 1. Kretsløpet forblir stabilt inntil en puls fra en annen krets forårsaker en endring. Output vil flippe eller floppe mellom 0 og 1 avhengig av input. Modellen viser hvordan en utstyrsenhet kan kontrolleres ved hjelp av porter (digitalt kretsløpsdesign). Flip-flop konseptet er også et eksempel på bruk av abstraksjon, og av abstrakte verktøy. Når en datamaskin konstrueres, brukes det nederste laget med komponenter som abstrakt verktøy for de neste lagene. En flop-flop er også viktig når det gjelder lagring av en bit.

EKSEMPEL

Boolsk algebra: A AND B = C

Kretsdiagram: A

B



C

Sannhetstabell:

A	**B**	**C**
0	0	0
0	1	0
1	0	0
1	1	1

VLSI – Very large scale integration: en teknologi som tillater oss å konstruere millioner av elektriske komponenter på en wafer eller chip. Dette kan brukes til å lage miniatyr utstyr, som består av millioner av flip-flopper samt deres kontrollerende kretser. VLSI kan i noen tilfeller brukes til å lagre et helt datasystem på en eneste chip.

Hexadesimal notasjon: En stream er en lang streng med bits. Disse kan være vanskelige for menneskene å forholde seg til. Hexadesimal notasjon er en forkortnings notasjon, der hvert sett med fire bits representeres av et tegn. Vi får et 16-talls system fordi 2^4=16.

Bit pattern	Hexadecimal representation
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Eksempel på bruk av hexadesimal notasjon i HTML:

- Rødt = 2 hex siffer

- Grønt = 2 hex siffer
- Blått = 2 hex siffer

<body bgcolor =”#r r g g b b#”>

<body bgcolor =”#990099#”>

Gir lilla bakgrunnsfarge

1.2 HOVEDMINNET

For å lagre data har en datamaskin en stor samling kretsløp (som flip-flops) som alle kan lagre en bit. Dette bit-reservoaret er kjent som maskinens hovedminne.

Organisering av en bit i en lagringscelle:

High order end 0 1 0 1 1 0 1 0 low order end

Most significant bit↑ ↑ least significant bit

Dersom innholdet i en minnecelle skulle tolkes som en numerisk Verdi vil den første biten være den viktigste.

Alle celler har en unik adresse (celle0, celle1, celle2….). Hele kolleksjonen av bits i hovedminnet er ordnet i en rekke. Hvis du skal lagre en string på 16 bits vil det lagres i to celler etter hverandre.

Read operasjon: kretsløpet som inneholder bitsene er forbundet med henting og lagring av data. Kretsløpet henter data fra minnet ved å spørre elektronisk etter en celles adresse.

Write operasjon: Når informasjon lagres i en celle ved at et bestemt bitmønstre plasseres på en bestemt adresse.

RAM - Random access memory: ettersom at hovedminnet består av uavhengige celler, kan cellene behandles individuelt. En datamaskins hovedminne er derfor ofte kalt RAM.

DRAM – dynamic memory og SDRAM – synchronous dram: er teknikker for å forminske tiden det tar å hente noe fra en minnecelle.

Måling av minnekapasitet:

2^10 celleenheter = 1024 = 1 kiloByte (KB)

2^20 celleenteter = 1 048 576 = megaByte (MB)

2^30 celleenheter = 1 073 741 824 = gigaByte (GB)

Denne navngivingen har i stor grad vært et problem ettersom at 1000 er forbundet med kilo, ikke 1024 osv.

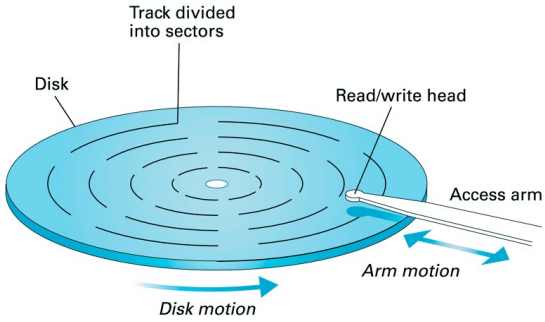
1.3 MASSE LAGRING

On-line: informasjon som er tilgjengelig på maskinen uten menneskelig innblanding.

Off-line: menneskelig innblanding er nødvendig for at maskinen skal kunne nå informasjonen. Enheten må for eksempel slås på eller settes inn i maskinen.

Bakdelen med masselagringssystemer er at de i hovedsak avhenger av mekanisk bevegelse, og derfor bruker mye lengre tid på å lagre og hente data enn hovedminnet, der all aktivitet foregår elektronisk.

MAGNETISKE SYSTEMER



Harddisk: en tynn magnetisk disk som holder på dataene. Et lese/skrive hode gir tilgang til de forskjellige sporene på plata. I mange tilfeller består et platelagringssystem av flere disker plassert over hverandre. Lese/skrive hodene mellom platene beveger seg i så tilfelle unisont med hverandre. Fordi et spor inneholder mer informasjon enn vi vanligvis ønsker å manipulere på en gang, er sporene delt opp i mindre sektorer. Informasjonen på en sektor består av en kontinuerlig streng av bits. Fordi sektorene i ytterkant av disken kan holde mer informasjon en sektorene innerst på disken, benyttes en teknikk kalt Zoned-disk recording. Flere spor regnes som soner. En typisk disk inneholde da ti soner. Alle sporene innengor en sone, inneholder samme antall sektorer. Hver sone vil ha flere sektorer per spor, enn sonen som ligger innenfor. Dette er mer effektivt da hver sektor kan behandles som et individuelt sett med bits. Spor og sektorer blir markert magnetisk på disken gjennom en formateringsprosess. De fleste datasystemer kan foreta denne initialiseringen. En disk kan reformeres dersom innholder er skadet, men dette vil ødelegge informasjonen som var på disken fra før. Kapasiteten til et disk lagringssystem avhenger av antall plater og tettheten mellom spor og sektorer.

Diskett: et lavkapasitetssystem bestående av en enkelt plastik disk. Fleksible disketter kalles ofte floppy disks. Dette er det vi kaller en Off-line lagringsenhet da disketten må settes inn i datamskinens lese/skrive enhet. Ettersom at disketten kun har plass til 1.44 MB, er den stort sett erstattet av andre lagringsenheter.

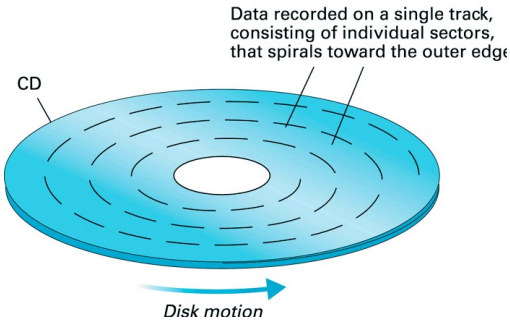
Et disksystems ytelse avgjøres av flere faktorer:

- Søke tid
- Forsiknkelse på grunn av rotasjon
- Tiden det tar å få tilgang
- Overføringsraten

Et harddisksystem har vanligvis mye bedre ytelse en floppysystemer, da lese/skrive-hode ikke er nær diskens overflate. Rotasjonstiden blir betydelig mindre. På grunn av den mekaniske bevegelsen vil et system basert på elektronisk kretsløp, alltid være mest effektivt.

Magnetiske bånd (magnetic tape): en eldre for for lagring basert på magnetisk teknologi. Informasjonen ligger på et magnetisk bånd rundt en spole. For å få tilgang til dataen plasseres tapen i et magnetbånd drev (tape drive), som kan lese, skrive og spole tilbake tapen under komtroll av datamaskinen. Størrelsen på tapen er varierende. Ettersom at båndet må flyttes mellom et lese/skrive-hode og over på en annen spole, er dette ikke et effektivt lagringssystem. Fordelen er derimot den høye kapasiteten.

OPTISKE SYSTEMER



Compact disk (CD): Informasjon tillegges en cd ved å skape variasjoner i cdens klare overflaten. Informasjonen kan leses av en laserstråle som registrerer regelmessighetene i overflaten mens cden spinner. Cd teknologi er basert på audio recording (lyd innspilling) kalt CD-DA (compact disk-digital audio). Cdene vi bruker i dag til lagring av data, har så godt som samme format. Det som skiller dem er at sporene på cdene i dag er formet som en spirral fra midten og ut. Sporene er delt opp i sektorer, men egne identifiserings merker og kapasitet på 2KB med data. For å maksimere kapasiteten er informasjonen lagret med en uniform, lineær tetthet over hele sporet. For å bevare en konstant hastighet er CD-DA-spillere designet til å variere rotasjonsfarten avhengig av hvor laserstrålen befinner seg. Systemer for lagring av data spinner derimot raskere og med konstant fart. De må derfor tilpasses variasjonen i overføringsraten. Lagring på cd egner seg best til lagring av lange kontinuerlige strenger med data. Dersom vi ønsker tilgang til data i en tilfeldig rekkefølge, er magnetiske systemer beste egnet. Tradisjonelle cder har en kapasitet på mellom 600 og 700MB.

Digital versatile disk (DVD): består av sammensatte, halvgjennemsiktige lag, som tjener som en klar overflate sett fra en presist fokusert laser. En dvd har dermed en kapasitet på flere gigabyte og egner seg til lagring av mulitmedia.

FLASH DRIVERS/MINNEPENNER

I et flashminnesystem lagres bits ved å sende elektroniske signaler direkte til lagringsenheten. Der de gjør at elektroner fanges i små kammer av silikondioksid hvor de fører til forandringer i små elektroniske kretser. Ettersom at det ikke benyttes mekaniske bevegelse i prosessen, er dette et off-line system som kan konkurrere med andre elektroniske systemer. Til tross for at flashminne kan benyttes slik som RAM, vil ikke dette fungere i lengden. Etter hvert som data sletter fra kammerne for så å lagre noe nytt om og om igjen, vil silikondeoxidkammerne sakte skades. I datamaskinens hovedminnet forandres data flere genger i sekundet. Når det gjelder digital kameraer, mobiltelefoner og lignende, er derimot flash minne godt egnet. I motsetning til magnetiske- og obtiske systemer, er ikke flashminne sensitivt til fysist støt.

FILLAGRING OG GJENFINNING

Physical record (fysiske poster): Informasjon lagret i masselagringssystemer er gruppert i filer. En datablokk som samsvarer med en lagringsenhets karakteristikk kalles en fysisk post. En stor fil lagret på en masselagringsenhet, vil typiskt bestå av mange fysiske poster.

Logical record (logiske poster): I tillegg til denne fysiske inndelingen, har filer ofte en naturlig deling avhengig av informasjon. Eksempelvis vil en fil bestående av et tekstdokument bestå av avsnitt og sider. Dette kalles logiske poster.

Fields (felt): logiske poster består igjen av mindre enheter kalt fields. En logisk post som består av en informasjon om en student, vil inneholde felt som navn, adresse osv.

Key field (nøkkelfelt): er det feltet i den logiske posten som kan brukes til å identifisere posten. For eksempel et studentnummer.

En fysisk post kan bestå av flere logiske poster. En logisk post kan også være delt få flere forskjellige fysiske poster. Når data skal hentes ut fra en masselagringsenhet må altså informasjonen først samles. En vanlig løsning på problemet er å holde av et område i masselagringsenheten til samling av data.

Buffer: et lagringsområde til å holde på data midlertidig. Vanligvis mens data transporteres fra en enhet til en annen.

1.4 INFORMASJON REPRESENTERT SOM BITMØNSTRE

REPRESENTASJON AV TEKST

Tegnene i en tekst består alle av et unikt bitmønster.

ASCII: tegn representeres ved hjelp av en streng på 8 bit, slik at det passer i en typisk minnecelle på en byte. 128 tegn kan fremstilles ved hjelp av ASCII.

Unicode: 16 bit representerer et tegn. 65 536 forskjellige symboler kan fremstilles. Unicode ble utviklet slik at alle språk skulle kunne representeres.

Det er også utviklet standarder som tar i bruk 32 bit for å representere et symbol. Vi skiller mellom to typer tekst. En tekstfil består av en lang sekvens med symboler kodet ved hjelp av ASCII eller unicode og er skrevet i en teksteditor. Tekster produsert av en word prosessor, som microsoft sin word, består av ord.

REPRESENTASJON AV NUMERISKE VERDIER

Binær notasjon: Vi fremstiller tall ved hjelp av null og en, ikke 1 2 3 osv. Vi har flere typer notasjoner. Toers komplement notasjon er egnet til å representere både negative og positive tall, mens flyttalls notasjon er egnet til å representere desimaltall.

REPRESENTASJON AV BILDER

Bilder kan representeres som en samling av ”dotter” eller pixler (picture elements). En slik samling kalles et bitmap. Det finnes flere måter å kode et bilde på. I et sort/hvitt-bilde kan hver pixel representeres ved hjelp av en eneste bit. Når det gjelder fargebilder kodes de gjerne på to forskjellige måter.

1. RGB-koding: hver pixel representerer en trefargede komponenter, en rød, en blå og en grønn. Normalt brukes en bit til å bestemme fargens intensitet. Tre byte representerer en pixel i det orginale bildet.

2. Lys-komponenter: kalles også pixel luminanse og består av en sum med røde, blå og grønne komponenter. To andre komponenter, kalt rød og blå krominans, bestemmer forholdet mellom pixel luminans og rødt og blått lys i pixelen. Sammen består de tre komponentene av nok informasjon til å reprodusere pixelen. Tv har bidratt til å gjøre dette til en populær dekodings metode for koding av bilder.

Fremstillingen av bilder som bitmaps er fordelaktig ettersom at pixel-formatet er lett viselig på en skjerm. Ulempen er at denne representasjonen vanskelig lar seg forstørre. Den eneste måten å gjøre dette på er å forstørre hver pixel. Bildet vil da få et kornete utseende.

En alternativ måte er å fremstille bilder som en samling av geometriske strukturer. Denne representasjonen tillater utstyrsenheten å bestemme hvordan de geometriske strukturene skal fremstilles og brukes i forskjellige tegne programmer.

Computer-aided design(CDA): systemer hvor tredimensjonale objekter fremstilles og manipuleres på dataskjermer.

REPRESENTASJON AV LYD

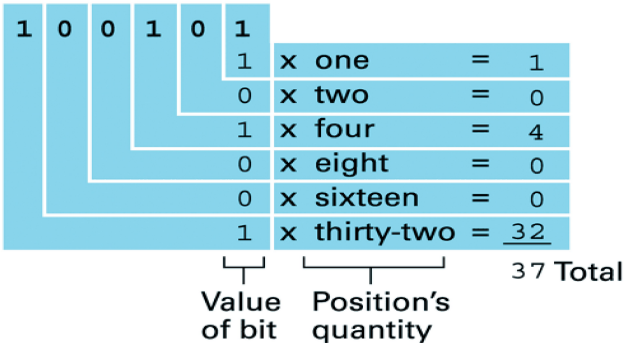
Sampling: Amplituden i en lydbølge samples ved at verdiene i intervaller tar vare på. (eks 0, 1.5, 2.0, 1.5, 3.0, 4.0, 3.0 viser en stigende kurve som faller litt, for så å stige igjen før den går tilbake til 0) Hver eneste sampel representeres av 16 bits for audio og 32 for stereo.

MIDI – Musikal Instrument Digital Interface: brukes i stor grad i elektroniske musikk instrumenter (Synthesizers) vi finner i, keyboard, i videospill og på websider. Ved å kode retningslinjene for produksjonen av musikk på et elektronisk instrument, unngår denne teknikken den store lagringskapasiteten som trengs ved samling. MIDI koder hvilket instrument som skal spille hvilken note samt hvor lenge. Det betyr at en klarinett som spiller en D i to sekunder, kan kodes i tre byte, isteden for to millioner bits ved en samplingsrate på 44 100 sample per sekund. Bakdelen er at lyden kan høres forskjellig ut avhengig av de elektroniske instrumentene.

1.5 DET BINÆRE SYSTEMET

BINÆR NOTASJON

Konvertering fra titallssystemet til totallssystemet:



I titallssystemet ville det på høyre side stått 10, 100, 1000 osv, mens i to tallssytemet gjelder kvantiteten 1, 2, 4, 6, 8 osv (2^0, 2^1, 2^2, 2^3).

Konvertering fra titallssystemet til totallssystemet:

Tall	Deler på 2	Kvotient	Rest
35	35/2	17	1
17	17/2	8	1
8	8/2	4	0
4	4/2	2	0
2	2/2	1	0
1	½	0	1

$35_{10} = 100011$

BINÆR ADDISJON

Regler:

	0		0		1		1
+	0	+	1	+	0	+	1
=	0	=	1	=	1	=	10

Eks. 111010
\+ 11011
= 1010101

BINÆRE BRØKER

101.101 – tallene til venstre er heltallet, mens tallene til høyre er desimalene. Tallene etter det vi kaller radix point har kvantitet 1/2, ¼ og 1/8 (2^-1, 2^-2, 2^-3 osv). Ellers regner vi på samme måte som tidligere dersom vi skal konvertere til tall i titallsystemet. Addisjon fungerer som vanlig.

1.6 LAGRING AV HELTALL

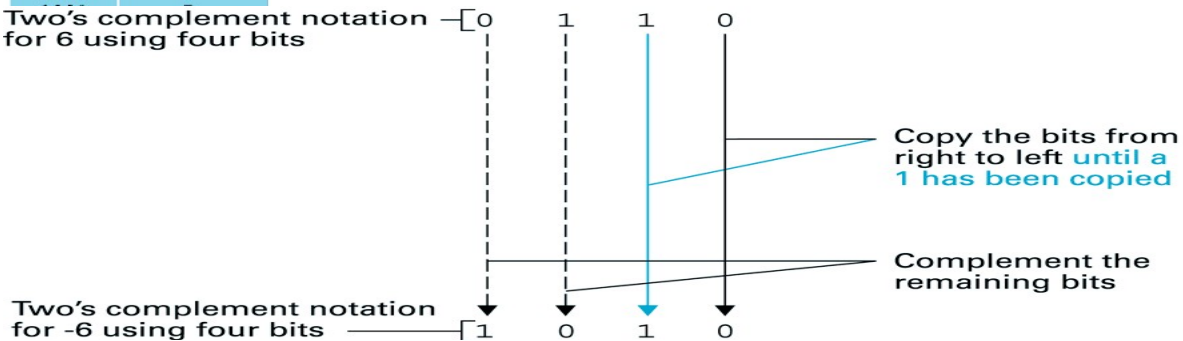
Vi skal se på to forskjellige notasjonssystemer, toers komplement notasjon og eksess notasjon. Begge brukes for lagring av heltall, både negative og positive. De er basert på det binære systemet.

TOERS KOMPLEMENT NOTASJON

Bit pattern	Value represented
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6

Dette er det mest populære systemet i dagens datamaskiner. Systemet bruker ulike antall bits til å representere de forskjellige verdiene. Det vanligste er å bruke 32 bits. Mange nummer kan da representeres. Bitten til venstre angir om tallet er negativt eller positivt og kalles for sign bit. De negative tallene starter med 1, mens de positive tallene starter med 0.

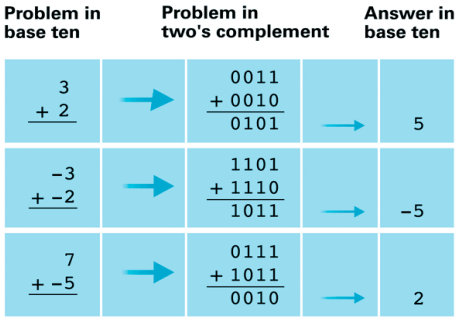
Dersom vi skal finne toers komplement notasjonen til et positivt heltall, konverterer vi tallet på vanlig måte. Når man derimot skal finne



toers komplement notasjon til et negativt tall, ser vi først bort fra minus og konverterer på vanlig måte. Deretter kopierer vi det positive tallet, fra høyre mot venstre, til vi kommer til den første 1eren. Deretter snur vi 0 til 1 og en til 0 som vist i figuren nedenfor.

ADDISJON I TOERS KOMPLEMENT NOTASJON

Addisjon foregår på samme måte som tidligere, men vi må ta hensyn til sign biten. En fordel med toers komplement er at maskinen bare trenger å kunne addere, ikke trekke fra. Det ser du i figuren. Det trengs altså bare en krets til å utføre begge deler.



OVERFLOW

I toers komplement er det en grense for hvor mange bit vi kan ta i bruk i hvert tilfelle. Dersom vi for eksempel bruker toers komplement med 4 bit, er 7 den største verdien vi kan representere og -8 er den minste. Dersom vi skal legge sammen 5 og 4 vil svare bli -7. Overflow oppstår når svaret faller utenfor verdiområde vi har til rådighet. Vi kan sjekke dette ved å se på sign biten til svaret vi får. Dersom addisjon av to positive verdier resulterer i et negativt svar, eller to negative verdier resulterer i et positivt svar, har vi overflow. I dagens datamaskin er som nevn toers komplement ofte representert ved hjelp av 32 bits. Den høyeste verdien vil være 2 147 483 647. Overflow vil sjelden inntreffe.

EKSESS NOTASJON

Forskjellen på toers komplement notasjon og eksess notasjon er at sign biten er omvendt. I eksess notasjon står 1 foran de positive tallene, mens 0 står foran de negative tallene. Når vi skal finne eksess notasjon starter vi nedenfra og teller binært oppover. Tabellen figuren viser, er en eksess åtte konverteringstabell. 1100 representerer 12 i vanlig binær notasjon. Vi ser at 1100 representerer 4 i eksess notasjon. Altså 12-8. 0000 representerer vanligvis 0, men her representerer det -8. Bitmønstre med 5 tegn kalles eksess 16 notasjon. 10000 representerer 0 og ikke 16 som det vanligvis ville ha gjort. Et trebits eksess system vil være en eksess fire notasjon.

Bit pattern	Value represented
1111	7
1110	6
1101	5
1100	4
1011	3
1010	2
1001	1
1000	0
0111	-1
0110	-2
0101	-3
0100	-4
0011	-5
0010	-6
0001	-7
0000	-8

ANALOG VS. DIGITAL

Digitalt: en verdi kodes som en serie med siffer, og deretter lagret ved hjelp av forskjellig utstyrsenheter, som hver og en representerer et av sifrene.

Analogt: hver verdi lagres på en eneste utstyrsenhet, som kan representere en hvilken som helst verdi innenfor en rekkevidde.

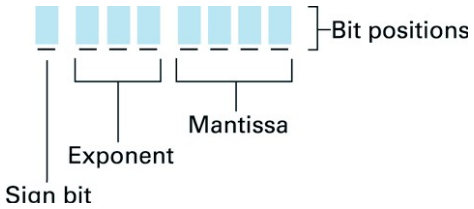
Fordi et digitalt system består av flere enheter, vil dette være mer robust enn et analogt system.

1.7 LAGRING AV BRØKER

I tillegg til å lagre de binære sifrene, må vi lagre posisjonen til kommaet (radix point).

FLYTTALLSNOTASJON (Floating-Point Notation)

High order end eller sign bit tar vare på fortegnet. 0 er positivt og 1 er negativt. Deretter deler vi inn i et eksponent felt og en mantissa.



Eksempel

Vi tar for oss bitmønsteret 01101011 der 0 er sign bit, 110 er eksponent og 1011 er mantissa. Vi finner eksponenten ut fra tre-bits eksess notasjon. 110 tilsvarer 2. Det forteller oss at vi skal flytte radix punktet to plasser til høyre (en negativ eksponent ville ført til at vi skulle flytte mot venstre). Vi har .1011 (mantissa) og får 10.11. Vi har funnet plassen til komma og funnet tallet i titallssystemet, 2 ¾.

Dersom vi skal lagre et flyttall gjør vi det omvendte av eksempelet. Dersom vi skal kode 1 1/8 gjør vi først om til binært, 1.001. Deretter kopierer vi råtallet inn i mantissa fra venstre mot høyre. Vi starter med det første ett tallet fra venstre. Mantissa vår blir seende slik ut: 1001. Når vi skal finne eksponenten tar vi for oss mantissa .1001. for å få 1.001 ser vi at radix punktet må flyttes en plass til høyre, altså er eksponenten positiv 1, som er 101 i eksess notasjon. Til slutt fyller vi inn sign bit, som i dette tilfellet er 0. Vi får 0 101 1001.

Dersom vi skal sette 3/8, som er .011, inn i mantissa vil vi få 1100. Dette er fordi vi skal starte med det første ett tallet til venstre. Vi sier at representasjonen er på normalisert form. Dette hindrer ulike representasjoner av samme verdi. Alle verdier forskjellig fra null vil ha en mantissa som starter med 1.

TRUNCATION ERROR (Avrundingsfeil)

Dersom vi skal kode 2 5/8, 10.101, vil vi ikke få plass i mantissa. Vi får avrundingsfeil. Vi mister en del av verdien ettersom at mantissa ikke er stor nok. I dagens datamaskiner brukes 32 bits til å lagre flyttall i. Avrundingsfeil vil dermed ikke oppstå så ofte. En annen form for avrundingsfeil får vi dersom vi skal kode uendelige uttrykk, som 1/3. Flere verdier er uendelige i totallssystemet enn i titallssystemet. Det finnes forskjellige løsninger på problemet.

Eksempel: 2¼ +1/8 +1/8 dersom vi begynner med å legge sammen 2 ¼ og 1/8, får vi allerede her avrundingsfeil. Vi ender opp med verdien 2 ½ istedenfor 2 5/8. Neste steg vi dermed også få en avrundingsfeil og vi blir sittende igjen med feil svar. Dersom vi derimot begynner med å legge sammen 1/8 og 1/8 får vi ¼ og unngår avrundingsfeil. Når vi deretter legger til 2 ¼ vil

vi også få et korrekt svar. Den generelle regelen vil være å legge sammen de minste tallene først.

1.8 DATAKOMPRESJON

Det kan ofte være nyttig å redusere størrelsen på data når det kommer til lagring og overføring. Det finnes forskjellige teknikker for dette.

Kompresjonssystemer faller inn under to kategorier:

Lossless: informasjon går ikke tapt i prosessen.

Lossy: kan føre til tap av informasjon underveis, men reduserer størrelsen mer enn ved lossless systemer. Systemene er derfor populære til komprimering av lyd og bilder hvor små tap ikke spiller noen rolle.

Run-length encoding: er populært i tilfeller der komprimert data består av lange sekvenser med samme verdier. Dette er et lossless system. Det går ut på å erstatte en sekvens, med identisk data, med en kode som indikerer elementet som repeteres og antall ganger. Det benyttes mindre plass å indikere at et bitmønster består av 253 enere, 118 nuller og så 87 enere, enn å liste opp alle 458 bitsene.

Frequency-dependent encoding: dette er også en lossless teknikk. Lengden på bitmønstre avhenger av hvor ofte en verdi forekommer. En Q forekommer ikke like ofte som en E, og vil derfor ha lengre bitmønster enn E. David Huffman utviklet en algoritme for å finne slike koder. Vi kaller dem derfor Huffman koder.

Relativ encoding/differential encoding: brukes når data kan deles opp i nesten like blokker, som kommer etter hverandre. Dette kan både være en lossy eller en lossless teknikk avhenging av hvordan man koder. Vi sammenlikner en blokk med blokken den foregående blokken. Hver blokk kodes ved at forholdet til blokken foran beskrives. En blokk kan for eksempel være et bilde i en videosekvens. Det vil være liten forskjell fra bilde til bilde.

Dictionary encoding: vi ser på gjentagelse av data i en bitstrøm eller tekst. Isteden for å gjenta data, henviser vi til hvor vi finner dem. Vi tenker vanligvis på denne teknikken som lossless, men vi skal se at den i noen tilfeller må betraktes som lossy. Denne typen teknikk kan brukes i ord prosessorer (eks Word og lignende). Slike programmer har gjerne en ordbok som refereres til istedenfor å lagre hele hver eneste verdi i ordet.

Lempel-Ziv-Welsh (LZW): er et eksempel på adaptive dictionary encoding, hvor ordboken forandres gjennom komprimerings prosessen.

Eksempel: xyx xyx xyx xyx

Ordbok:
1: x Vi legger til tegn og ord I ordboken etter hvert som de forekommer I teksten.
2: y
3: space
4: xyx

Svar: 121343434

BILDE KOMPRIMERING

GIF (Graphic Interchange Format): er en ordbokbasert komprimeringsteknikk. Antall farger som kan tilordnes en pixel reduseres til 256. Kombinasjonen av rødt, grønt og blått til hver farge, kodes ved at tre bytes tas i bruk. Kodene til alle 256 fargene lagres i en tabell kalt paletten. Hver pixel kan nå representeres ved hjelp av en byte, som indikerer hvor i paletten fargen befinner seg. Dette er en lossy teknikk ettersom at fargene ikke nødvendigvis er identiske med fargene i det originale bildet. Man kan også benytte et adaptivt system ved hjelp av LZW teknikker. En av fargene i en GIF-palett er vanligvis tilordnet verdien ”transparent”. Det vil si at bakgrunnen kan skinne gjennom på områder med denne verdien. Komprimeringsmetoden egner seg best til enkle animasjoner. De 256 fargene vil ikke kunne gjengi et fotografi av høyere presisjon.

JPEG (utviklet av Joint Photographic Experts Group): er egnet til komprimering av fotografier og brukes på de fleste digitalkameraer. JPEG omfatter flere forskjellige metoder. De tilbyr blant annet en lossless metode, men denne er ikke den beste. Derimot er JPEG sin baseline standard, den mest brukte. Denne metoden utnytter at menneskeøyet er mer sensitivt overfor lys enn for farge. Dersom vi tar for oss et bilde kodet i luminance og chrominance-komponenter, finner vi først gjennomsnittsverdien for chrominance over 2*2 pixler. Vi reduserer dermed størrelsen med 4, samtidig som vi bevarer lyset. Neste steg er å dele bildet inn i 8*8 pixelblokker, for deretter å komprimere hver blokk som en enhet. Nå benyttes tradisjonelle komprimeringsteknikker, som run-length og relativ koding.

TIFF (Tagged Image File Format): brukes ofte som standard format for lagring av foto med informasjon, som tid, dato osv. Bildene lagres normalt som røde, grønne og blå komponenter uten komprimering. De komprimeringsteknikkene, som er inkludert i TIFF, er som oftest designet til å komprimere bilder av tekstdokumenter. Disse bruker variasjoner av run-length encoding, fordi et slikt bilde vil bestå av mye hvitt. Fargebilde komprimeringen ligner på den som er benyttet i GIF.

KOMPRIMERING AV AUDIO OG VIDEO

MPEG (utviklet av Motion Picture Experts Group): standarder for komprimering av lyd og video.

Videokomprimering: videoen deles opp i sekvenser med bilder. Bare noen av bildene kodes fult og helt. Disse kalles for I-frames. Bildene mellom I-frames kodes ved hjelp av relative kodingsteknikker. I-frames komprimeres ved hjelp av teknikker som liknes JPEG.

Lydkomprimering: den mest kjente teknikken en MP3komprimering. Detaljene, som menneskeøret ikke klarer å oppfatte, fjernes.

- Temporal masking: øret oppfatter ikke lavere lyder i en kort periode etter en høy lyd.
- Frequency masking: lyder med enkelte frekvenser kamuflerer lavere lyder i nærliggende frekvenser.

Komprimering av lyd og video er også viktig når det kommer til overføring. Videopresentasjoner avhenger av en overføringsrate på 40Mbps, mens MP3 trenger 64Kbps.

1.9 KOMMUNIKASJONSFEIL

Informasjon kan gå tapt når den sendes fra et sted til et annet. Dette kan blant annet komme av funksjonsfeil i en krets. Ulike teknikker er utviklet for å oppdage (feildetekterende koder) og rette opp (feilkorrigerende koder) feil.

PARITY BITS

Vi legger til 0 eller 1 foran et bitmønster slik at det inneholder et odde antall enere. Vi kaller denne biten for parity bit. Dersom et manipulert bitmønster består av par antall enere har en error oppstått. Dette er odd parity, mens even parity vil fungere motsatt. Minnecellene i dagens datamaskiner har, i tillegg til en byte, plass til en parity bit. Når en byte lagres i en minnecelle, legger 0 eller 1 til avhengig av hvor mange enere som finnes fra før. Når mønstre kommer tilbake igjen, sjekkes antall enere på nytt. Dersom en error har oppstått returnerer minnet en advarsel. Parity systemet vil ikke kunne oppdage det dersom et bitmønster er blitt utsatt for et partall med errorer. En løsning er å benytte en checkbyte. Den første biten sammenlignes med de åtte neste, og den andre biten sammenlignes med de åtte deretter.

FEILKORRIGERENDE KODER

Hamming distance: dersom et gyldig bitmønster skal forandres til et annet gyldig bitmønster, må minst tre bits forandres. Vi kan finne det riktige bitmønsteret ved å sammenlikne differansen mellom alle gyldige bitmønster og mønsteret vi mottok. Mønsteret med lavest differanse vil sannsynligvis være mønsteret som ble sendt. Vi kan oppdage opp til to feil.

De gyldige: Eksempel (010100):

Symbol	Code
A	000000
B	001111
C	010011
D	011100
E	100110
F	101001
G	110101
H	111010

Mønsteret vi fikk tilsendt må ha vært en D!

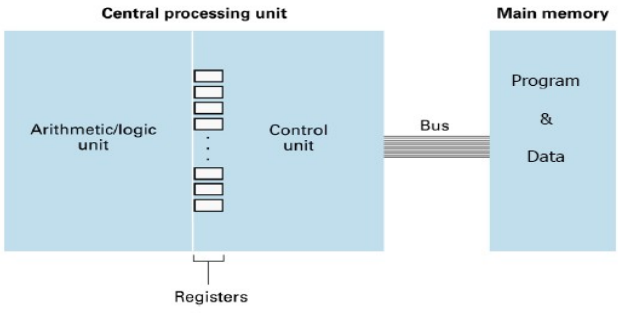
Mottatt bits	Distanse
010100	2
010100	4
010100	3
010100	1
010100	3
010100	5
010100	2
010100	4

Kapittel 2 – Datamanipulasjon

Kap 2.1 MASKINARKITETKTUR:

CPUens tilkoblingspinner kobles til maskinens hovedkort (**motherboard).**

Kretssystemet i maskinen som kontrollerer manipulasjon av data kalles **central processing unit (CPU)**. Prosessoren CPU er den som behandler data! Før var prosessorene kjempestore, nå har de blitt veldig små og kalles derfor mikroprosessorer (**microprocessors).**



CPU har tre deler: **arithmetic/logic unit, control unit og register unit.**

1. Arithmetic/logic unit (ALU) er en samling elektroniske kretser som kan manipulere data som ligger i registre (feks legge sammen tall). Data manipulasjon!
2. Control unit (CU) koordinerer maskinens aktiviteter.

3. Register unit inneholder datalagringsceller, kalt **registers,** som blir brukt som midlertidig informasjonslagringsplass i CPU.

I register'ene i registers unit har vi **general-purpose registers** og **special-purpose registers**. General-purpose registers er midlertidig lagring av data som manipuleres i CPU. Disse holder informasjon for arithmetic/login enheten og skaffer lagringsplass for resultatene til enheten.

For å utføre en operasjon på data som er lagret i main memory sendes informasjon fra memory til general-purpose registers. Så blir arithmetic/login enheten informert hvilket register som inneholder data og hvem som skal motta den.

En maskins CPU og main memory henger sammen med masse ledinger kalt en **BUS.** Gjennom denne BUSen «leser» CPU informasjon fra main memory ved å gi adressen til den aktuelle minnecellen (memory cell) sammen med et ekektrisk signal som sier at den skal hente data på den aktuelle cellen. På en lik måte «skriver» CPUen data inn i minnet, ved å gi adressen til minnecellen og et elektrisk signal som sier at den skal lagre dataen som blir sent til den.

Stored-program koseptet går ut på å lagre en maskins program i main memory.

2.2 MASKINSPRÅK:

maskinspråk (**machine language**) er et språk maskinen forstår. En instruksjon i dette språket blir kalt en machine instruction (eller en machine-level instruction).

To CPU arkitektur filosofier:

En er at CPU skal være laget for å utføre et minimalt sett av maskininstruksjoner. Denne tolkningen leder til **reduced instruction set computer (RISC)**.

På den andre siden mente mange at CPUen skulle kunne utføre mange kompliserte instrusjoner. Resultatet ble **complex instruction set computer (CISC)**. Positivt med RISC er at den er kjapp. Positivt med CISC er at det er mye lettere å programmere, pga at en enkel instruksjon kan brukes til å utføre en oppgave som ville trengt en flerinstruksjon i RISC. Både RISC og CISC er vanlige.

Instruksjonssettet kan klassifiseres i tre kategorier: data transfer, arithmetic/login og control.

- **Data transfer** består av instruksjoner som krever forskyvning (kopiering av data) fra

et sted til et annet. En forespørsel om å fylle et universal register med innholdet til en minnecelle er en LOAD instruksjon. Og når noe skal fraktes til minnecellen blir det kalt STORE. En viktig gruppe instruksjoner i Data transfer er å kommunisere med apparater utenfor CPU-main memory kontekst (printer osv). Disse blir kalt **I/O Instructions** siden de tar seg av input og output aktivitetene.

- **Arithmetic/login** består av de instruksjonene som sier til kontrollenheten at den skal be om en aktivitet innenfor ALUen. ALUen er også kapabel til å utføre operasjoner som ligger utenfor aritmetiske orerasjoner, f.eks XOR, AND, OR. En annen samling av operasjoner som er tilgjengelig i de fleste ALUer er å flyte register til høyre eller venstre inni registeret. Disse operasjonene blir kalt SHIFT og ROTATE operasjoner.
- **Control** består av de instruksjonene som dirigerer utførelsen av et program istedenfor manipulasjon av data. Denne gruppen består av de fleste interessante intruksjonene i en maskins, f.eks JUMP instruksjoner som blir brukt til å dirigere CPUen til å utføre en instruksjon utenom den neste i listen. Disse JUMP-instruksjonene kommer i to former.

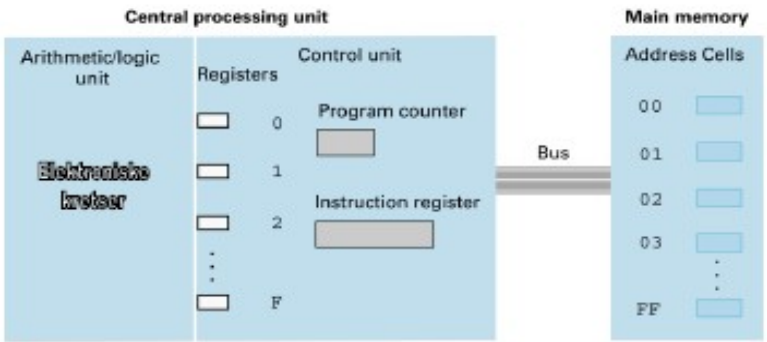
  - **unconditional jums**
  - **conditional jumps**

Et eksempel på den første er «hopp til trinn 5», et eksempel på den andre er «hopp til trinn 5 hvis verdien er 0».

Forklarende maskinspråk:

Kontrollenheten inneholder 16 registre og 256 hoved minneceller, hver med en kapasitet på 8 bits. Den kodete versjonen av maskin instruksjoner består av to deler: **op-code** (operating code) felt og **operand** felt. Bitkombinasjonen i op-code feltet indikerer hvilken operasjon som STORE, JUMP, SHIFT og XOR kreves av instruksjonen. Bitkombinasjonen som blir funnet i operandfeltet skaffer mer detaljert informasjon om operasjoner med op-code.

det komplette maskinspråkene til vår maskin (Appendix C) inneholder bare 12 basic instruksjoner. Hver av disse er i kodeform ved bruk av 16-bits, representert av 4 hexadesimale siffer. Op-code for hver av instruksjonene består av de første 4 bits (eller første hexadecimal siffer). Disse op-codene er representert av de hexadesimale tallene fra 1 til C.



«operand» feltene til hver instruksjon består av 3 heksadesimale tall, og gir den generelle instruksjonen gitt av op-koden. f.eks instruksjonen 35A7:

- Her er 3 op-kode for å lagre innhold i et register.
- 5 er tallet på registeret som skal lagres.
- A7 er minnecellen som skal motta data
-

cache memory er en del av høyhastighetsminne inni CPUen. I dette spesielle minneområdet prøver maskinen å holde en kopi av den delen av hovedminne som er av interesse.

2.3 PROGRAM EXECUTION

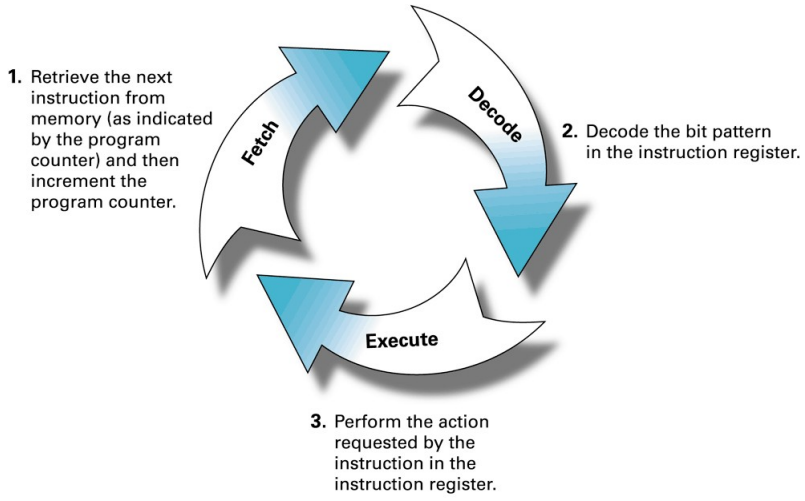
- En datamaskin følger et program lagret i minne ved å kopiere instruksjonene fra minne til CPU. I CPU blir instruksjonene dekodet og fulgt. I CPU har vi blant annet instruks register og program register. Instruks register blir brukt til å holde instruksen som blir utført. Og program register inneholder adresse neste instruksjon som skal bli utført.

- CPU utfører jobben ved å repetere en algoritme gjennom en tre-steg prosess, kalt maskin syklus:

1. **Fetch:** CPU spør hovedminne om instruksjonene (2 celler fra hovedminne) som er lagret på adressen indikert av program telleren. Program telleren holder kontroll på hvilke instruksjoner som er de neste. CPU plasserer den mottatte instruksjonen i sitt instruksjons register, og øker program telleren med to, slik at den inneholder adressen for neste instruks i hovedminne. (Unntak: Ved en JUMP instruksjon skal man endre verdien i program telleren hvis innholdet i det indikerte registeret er det samme som innholdet i register 0.)

2. **Decode:** CPU dekoder instruksjonene. Finner ut hva operand field og op-code ”forteller”).

3. **Execute:** CPU utfører instruksjonen ved å aktivere riktig krets for å utføre oppgaven.



2.4 ARITMETISKE/LOGISKE INSTRUKSJONER

Logiske operasjoner

De logiske operasjonene AND, OR og XOR er operasjoner som kombinerer to inputbits til en enkelt outputbit. Disse operasjonene kan utvides til operasjoner som kombinerer to bit-strenger til en enkelt output-streng. Dette kan gjøres ved å bruke basisoperasjonen på de individuelle kolonnene, se følgende eksempler:

```
        10011010        10011010        10011010
AND     11001001    OR  11001001    XOR 11001001
        10001000        11011011        01010011
```

Et vanlig bruksområde for AND-operasjonen er dersom man skal plassere 0-er i en del av et bit-mønster, uten at det forstyrrer den andre delen. For eksempel, hva vil skje dersom 00001111 er den første operanden i en AND-operasjon? Uten å vite hva den andre operanden er, kan vi allikevel konkludere med at de fire viktigste bitsene i resultatet vil være 0-er. De fire minst betydningsfulle bitsene i resultatet vil være en kopi av den delen fra den andre operanden:

```
        00001111
AND     10101010
        00001010
```

Denne bruken av AND-operasjonen er et eksempel på en prosess som kalles *maskering*. Her bestemmer en operand, kalt en maske, hvilken del av den andre operanden som vil påvirke resultatet. Når det gjelder AND-operasjonen vil maskering gi et resultat som er delvis en kopi av en av operandene, der 0-er opptar de ikke-kopierte plassene. Slike operasjoner er nyttige når man skal manipulere et bit-map, strenger med bits der hver bit representerer tilstedeværelsen/fraværet av et bestemt objekt. (For eksempel bilderepresentasjon.) Et eksempel: En streng på 52 bits, der hver bit er assosiert med et bestemt kort, kan brukes til å representere en hånd i poker ved å tildele 1-ere til de 5 bitsene som representerer kortene man har på hånden, og 0 til alle de andre. Sett at de 8 bitsene som er i en minnecelle brukes som et bit-map, og vi vil finne ut om det objektet som assosieres med den tredje biten fra venstre er tilstede. Da bruker vi AND-operasjonen på hele byten, med masken 00100000, som kun gir en byte med bare 0 dersom den tredje biten fra venstre selv er null.

Der hvor AND-operasjonen kan brukes til å kopiere en del av en streng, og plassere 0-ere i den delen som ikke er kopiert, kan OR-operasjonen brukes til å kopiere en del av en streng ved å sette 1-ere i den ikke kopierte delen. Eksempel følger:

```
        11110000
OR      10101010
        11111010
```

XOR-operasjonen brukes ofte til å danne komplementet (”tilhørende del”) til en bit-streng. Ved å utføre en XOR-operasjon på en hvilken som helst byte som har en maske av 1’ere, vil

gi dette komplementet. Se på forholdet mellom den andre operanden og resultatet i dette eksempelet:

```
        11111111
XOR     10101010
        01010101
```

I maskinspråket i Appendix C brukes op-kodene 7, 8, 9 forholdsvis for de logiske operasjonene OR, AND, XOR. Eks: 7ABC → Utfør en OR-operasjon på innholdet i register B og C, og plasser resultatet i register A.

Rotasjon og shift operasjoner

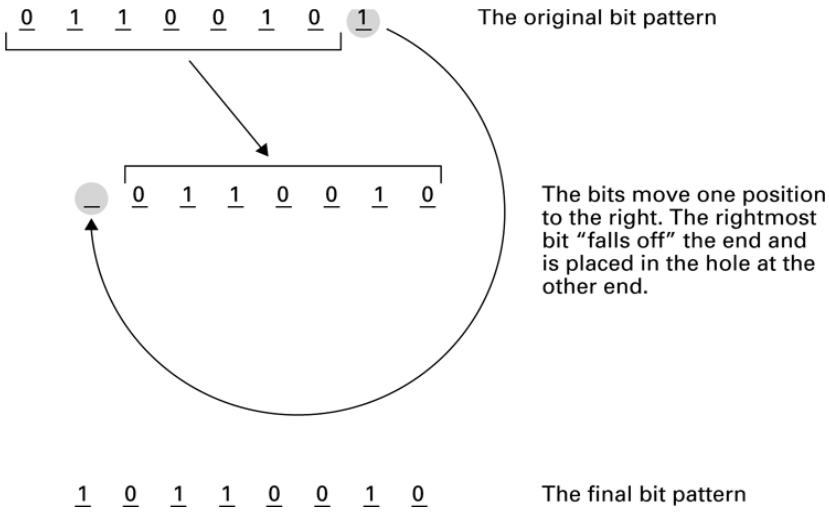
Rotasjon og shift-operasjoner gjør det mulig å flytte bits innenfor et register, og brukes ofte til å løse problemer angående plassering. Disse operasjonene klassifiseres av retningen på bevegelsen (høyre/venstre), og om prosessen er sirkulær. Innenfor disse retningslinjene for klassifisering er det mange varianter med blandet terminologi. Vi tar en kjapp titt på dette:

Et register som inneholder en byte med bits. Hvis vi skyver innholdet en bit til høyre, kan vi se for oss at biten lengst til høyre faller over kanten, og at det blir en åpen plass helt til venstre i registeret. Det som skjer med denne ekstra biten og den åpne plassen er et av de fremste kjennetegnene for ulike shift-operasjoner. En teknikk er å plassere den biten som falt av høyrekanten, på den ledige plassen på venstresiden. Dette kalles *circular shift*, eller *rotation*. Så hvis vi foretar et *right circular shift*, en forskyvning mot høyre på godt norsk, på et 8-bits mønster 8 ganger, har vi det sammen mønsteret vi startet med.

En annen teknikk er å se bort i fra den biten som faller over kanten, og fylle den ledige plassen med 0. Dette refereres ofte til som en *logical shift*, logisk forskyvning, og en slik forskyvning til venstre kan brukes til å multiplisere toers komplementnotasjon med 2. Å forskyve binærtall til venstre er jo det samme som å gange med 10. Divisjon på to kan oppnås ved å skyve binærstrengen til høyre. Men uansett hva slags forskyvning vi gjør, må vi passe på at sign-biten blir bevart når vi bruker visse notasjonssystemer. Vi finner ofte forskyvninger mot høyre hvor den ledige plassen alltid fylles ut (noe som skjer ved sign-bit posisjonen) med dens opprinnelige verdi. Forskyvninger hvor sign-biten er uendret kalles noen ganger *arithmetic shifts*, arimetiske forskyvninger.

Maskinspråket i Appendix C inneholder kun forskyvning mot høyre (sirkulær), utpekt av op-koden A. I dette tilfellet er det første heksadesimale sifferet i operanden som spesifiserer hvilket register som skal roteres, og resten av operanden spesifiserer antallet bits som skal roteres. Så instruksjonen A501 betyr da: ”Roter innholdet i register 5 til høyre med en bit.”Så hvis register 5 opprinnelig inneholdt bit-mønsteret 65 (heksadesimal), ville den inneholdt B2 etter denne instruksjonen er utført (figur 2.12).

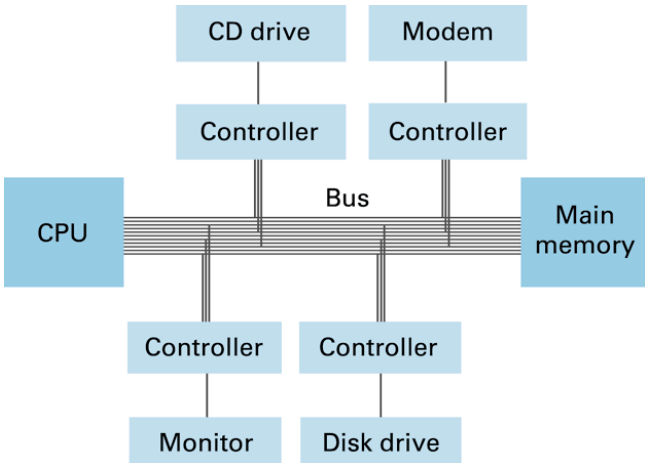
Figur 2.12 Rotere bit-mønsteret 65 (heksadesimal) en bit til høyre


0 1 1 0 0 1 0 1
The original bit pattern
_ 0 1 1 0 0 1 0
The bits move one position to the right. The rightmost bit “falls off” the end and is placed in the hole at the other end.
1 0 1 1 0 0 1 0
The final bit pattern

Aritmetiske operasjoner

Vi har sett at subtraksjon kan simuleres som addisjon av to negative tall. Multiplikasjon er egentlig bare repetert addisjon, og divisjon er repetert subtraksjon. (6/2 = 3 fordi **tre** 2-ere kan trekkes fra 6). Av denne grunn er noen små CPU-er designet med kun addisjonsinstruksjonen, eller kun addisjon- og subtraksjonsinstruksjonen. Det finnes flere varianter for hver aritmetisk operasjon. (Se adderingsoperasjoner i Appendix C.) Dersom verdiene som skal adderes er lagret i toers komplementnotasjon, må adderingen utføres som en enkel kolonne-for-kolonne addisjon. Men hvis verdiene er lagret som flytpunktsnotasjon må addisjonsprosessen trekke ut mantissa fra hver enkelt bit-mønster, skyve dem til høyre/venstre (avhengig av eksponentfeltet), sjekke sign-bitsene, utføre addisjonen, og oversette resultatet til flytpunktsnotasjon. Til tross for at begge operasjonene betraktes som addisjon, blir de utført forskjellig av maskinen.

2.5 KOMMUNIKASJON MED ANDRE UTSTYRSENHETER



Kontroller: mellomledd mellom datamaskinen (CPU og hovedminnet) og ytre enheter, som mus, tastatur, lagringssystemer, skrivere osv. I PC-er kan en kontroller enten være montert fast på hovedkretskortet, eller den kan være en fleksibel krets, som kan settes i hovedkretskortet. En kontroller kan være som en liten datamaskin, med egen minnekrets og enkel CPU. Den oversetter meldinger mellom datamaskinen og de perifere enhetene. Slik trenger ikke prosessoren forholde seg til alle slags periferenheter, men til en ”oversetter” som snakker ”prosessorens språk”. De kan også de kan jobbe i ulike hastigheter. Kontroller er tilknyttet maskinens buss.

Input-instruksjoner: tilsvarende ”LOAD”. Hent melding/data fra periferenhet via en kontroller (feks mus).

Output-instruksjoner: tilsvarende ”STORE”. Send melding/data til periferenhet via en kontroller (f.eks. skjerm eller printer).

Memory-mapped I/O: Hver kontroller svarer bare på referanser til et unikt sett med adresser, mens hovedminnet ignorerer disse. Når CPU sender instruksjon om å lagre et bitmønster i hovedminnet, som har en adresse som tilhører en kontroll, lagres det i kontrollen og ikke i hovedminnet. det samme gjelder LOAD instruksjoner. Et slikt kommunikasjonssystem kalles Memory-mapped I/O.

I/O instruksjoner: et alternativ til Memory-mapped I/O er å utvikle instruksjoner for direkte transport til og fra kontroller.

Port: en kontroller tilknyttes de perifere enhetene ved hjelp av kabler, eller porter. En port sitter bak på PC-en. Her kan eksternt utstyr kobles direkte til.

USB og FireWire: standardiserte kommunikasjonssystemer, som forenkler prosessen med å legge perifere enheter til en PC. Kontrolleren oversetter datamaskinens signaler til standard signaler for UBS eller FireWire, og kan dermed kommunisere dirkete med kontrolleren. Altså trenger man ikke kjøpe en ny kontroller selv om man kjøper et digitalt kamera.

DIREKTE MINNETILGANG

DMA (Direct memory access): i løpet av de nanosekundene CPU ikke bruker bussen, kan kontrollene selv kommunisere med hovedminnet. CPU kan sende bitmønster med forespørsel om å hente informasjon, fra en sektor på en disk, til kontrolleren festet til disken. Kontrolleren kan lese sektoren og plassere innholdet i et spesifikt område i hovedminnet. CPU kan i mellomtiden utføre andre oppgaver. At både kontrollerne og CPU konkurrerer om bussen kan være en hindring. Denne hindringen er kjent som von Neumann bottelneck.

Handshaking: toveis kommunikasjon mellom to datakomponenter. En printer både mottar og sender informasjon til datamaskinen. Et status ord sendes fra den perifere enheten til kontrolleren i form av et bitmønster. For en printer kan for eksempel den siste biten indikere om skriveren er tom for papir, mens den andre kan indikere om skriveren er klar for å motta data.

Parallell kommunikasjon: en form for kommunikasjon mellom enheter. Flere signaler/bits transporteres samtidig, men på forskjellige linjer. Data kan transporteres effektivt på denne måten, men det kreves en kompleks ”vei”. Et eksempel er en buss. Alle PC-er er utstyrt med minst en parallell port som transporterer data til og fra maskinen, åtte bits om gangen.

Seriell kommunikasjon: et og et signal transporteres over en enkelt linje. Eksempler på dette er kommunikasjon over en telefonlinje, et lokalt nettverk, USB og FireWire. Når digital data skal transporteres på denne måten, må bitmønstret først konverteres til lyd ved hjelp av et modem, for deretter å konverteres tilbake når de har kommet frem til destinasjonen.

DSL (Digital Subscriber Line): for rask overføring over lengre avstander. Data overføres med en høyere frekvens. Eksempler på dette er radio sendinger og kabeltv.

OVERFØRINGSRATER

Hastigheten bits overføres i fra en enhet til en annen, måles i bits per sekund (bps).
Kilo-bps (Kbps) = 1000 bits per sek
Mega-bps (Mbps) = 1 mill bits per sek
Giga-bps (Gbps) = 1 milliard bits per sek

Bandwidth/båndbredde: maksimumsraten avhenger av type kommunikasjonsbane og anvendt implementeringteknologi. Høyt båndbredde tilsier en høy overføringsrate og stor kapasitet.

Multiplexing: koding av data slik at en kommunikasjonsbane fungerer som flere kommunikasjonsbaner.

2.6 ANNEN MASKINARKITEKTUR

Pipelining

Elektriske pulser beveger seg gjennom en ledning i lysets hastighet. Siden lysets beveger seg ca. 1 fot på et nanosekund, kreves det minst 2 nanosekunder for CPU å hente en instruksjon fra en minnecelle som er 1 fot unna. (Forespørselen om den bestemte minnecellen må sendes til minnet, minst et nanosekund, og instruksjonen må sendes tilbake til CPU, enda et nanosekund.) Altså, å hente og utføre instruksjoner i en slik maskin krever flere nanosekunder. For å forbedre en datamaskins prestasjoner kan man gjøre den raskere, men det man egentlig vil er å forbedre maskinens *throughput*, som betyr den mengden arbeid maskinen kan utføre i løpet av et gitt tidsrom. Et eksempel på hvordan dette kan gjøres er pipelining. Pipelining kalles også ”samlebåndsprinsippet”. Pipelining tillater enhetene i maskinsyklusen å overlappe, altså at flere instruksjoner kan kjøre samtidig. For eksempel: Mens en instruksjon utføres, blir den neste instruksjonen dekodet, og den tredje blir hentet samtidig, som på et samlebånd. Altså kan flere instruksjoner være i ”røret” (the pipe) samtidig, hver av dem i ulike faser av utførelsen. Maskinens totale *throughput* vil øke, selv om tiden det kreves for å hente og utføre hver enkelt instruksjon er den samme. (Når en JUMP-instruksjon nås, vil ikke det forspranget som oppnås ved forhåndshenting slå til, siden instruksjonene i ”røret” ikke er de som trengs.)

Moderne maskiner kan ofte hente flere instruksjoner samtidig og utføre mer enn en instruksjon av gangen når de gjeldende instruksjonene ikke avhenger av hverandre.

(Multiprosessormaskiner skulle vi ikke ha om)

Kapittel 3 - Operativsystemer

Operativsystemet er programvaren som kontrollerer alle arbeid datamaskinen utfører.

3.1 OPERATIVSYSTEMENES HISTORIE:

I starten var datamaskinene store, og kunne romme et helt rom. Det tok mye forberedelser å kjøre et program, en måtte sette på magnetiske bånd, lage hull i hullkort og bytte på brytere. Etter hver gang et problem oppstod måtte en bytte på alt dette igjen.

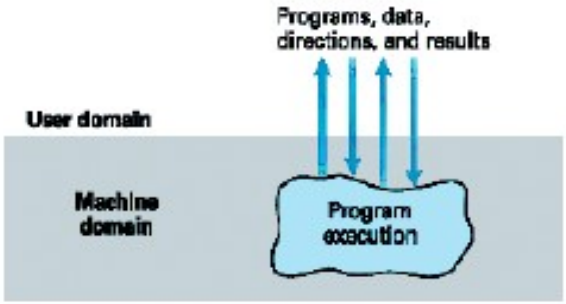
Etter hvert ble det satt **computer operators** til å gjøre denne jobbe. For å kjøre programmene ble det laget en kø ved å legge alle avhengighetene til de forskjellige programmene inn i maskinens hovedlager slik at de kunne utføres en etter en.

Dette var starten på **batchprosesserig**, som innebærte at alle programmene ble samlet i en batch (gruppe). Programmene ble så utført en etter en, uten at brukeren hadde noe med det å gjøre. I batchprosessering ble programmene lagret i en kø (en FIFO kø, First-in, First-out), som funker som en vanlig butikk-kø. Ok med batch var feks lønnskjøring, regnskapsregistrering og lignende programmer hvor data og program var lagret på forhånd.

I tidlige batchprosessering systemer ble hvert program fulgt av en rekke instruksjoner som ble kodet ved hjelp av et system som heter **job control language** (JCL). Når det var et program sin tur til å bli kjørt, ble instruksjonene skrevet ut slik at computer operatoren kunne følge med.

Ved å ha en computer operator mistet brukeren all interaksjon med programmet, som er litt dumt hvis en feks skal skrive i en text-editor.

For å løse dette ble nye operativsystemer utviklet. Disse gjorde det mulig for et program som ble utført å sende en dialog med brukeren gjennom en fjernterminal.



Dette ble kalt for **interactive prosessering**, men i starten foregikk dette bare ved at brukeren skrev inn noe på en terminal og at maskinen skrev ut responsen på papir. Når det kom til interactive prosessering så var det viktig at maskinen var såpass rask at brukeren ikke måtte tilpasse seg maskinen.

Maskinen ble tvunget til å utføre oppgaver etter en deadline, en prosess som ble kalt for **real-time-prosessering**, hvor det som ble utført ble sagt å skje i reell tid.

Real-time-prosessering hadde ikke vært noe problem hvis det bare hadde vært en bruker som trengte å bruke maskinen, men på 60- og 70-tallet var maskinene dyre, så hver maskin måtte ha mange brukere om gangen. Dette førte til problemer fordi maskinene ofte bare utførte en oppgave om gangen, slik at bare en av brukerne fikk oppleve real-time-processing.

Løsningen på dette problemet var et operativsystem som kunne gi plass til mange brukere om gangen. Dette systemet ble kalt **time-sharing**.

En måte å implementere time-sharing på var å bruke **multiprogramming** hvor tiden ble delt i intervaller, og utførelsen for hver oppgave ble begrenet til et intervall av gangen. På slutten av hvert intervall ble den aktive oppgaven satt midlertidig til side, og en annen oppgave fikk kjøre i neste intervall. Ved å bytte på disse intervallene ofte så det ut som at mange oppgaver ble utført samtidig.

I dag finnes multiprogrammering i både enbruker- og flerbrukersystemer. I enbrukersystemer kalles dette **multitasking**, altså at en bruker utfører mange oppgaver samtidig.

Ved utviklingen av multi-bruker og time-sharing maskiner ble det laget store maskiner som ble koblet til mange arbeidsstasjoner, slik at brukerne kunne kommunisere med maskinen utenfor maskinrommet uten å bruke et computer operator.

Operativsystemer har vokst fra enkle programmer som henter og utfører jobber en om gangen, til komplekse systemer som koordinerer time-sharing, opprettholder både programmer og datafiler i maskinens hovedlager og svarer direkte på forespørsel fra brukeren.

Load-balancing vil si å dynamisk tildele oppgaver til forskjellige prosessorer slik at alle prosessorene blir brukt effektivt.

Scaling vil si å dele opp oppgaver i flere deloppgaver som er kompatible med antall prosessorer tilgjengelig.

3.2 ARKITEKTUREN TIL OPERATIVSYSTEMER:

brukerprogramvare (application software) består av programmer for å utføre oppgaver som er spesielle for maskines utnytting.

Systemprogramvare (system software) utfører de oppgavene som er generelle for datasystemer generelt. Systemprogramvare er det som tilfører infrastrukturen som brukerprogramvaren trenger.

Innenfor systemprogramvare finner vi to kategorier, den ene er operativsystemet og den andre består av software kjent som **utility software**.

Utility software består av software som utvider mulighetene til operativsystemet.

Komponentene til et operativsystem:

Delen av et operativsystem som håndterer kommunikasjon kalles ofte et **shell**, gamle shells kommuniserte med brukeren gjennom text ved bruk av tastaturet og skjermen. Mer moderne shells gjør dette ved hjelp av grafisk brukergrensesnitt (graphical user interface, GUI), som vises billedlig på skjermen.

Disse GUI'ene kalles tydeligvis ofte WIMP (Windows, Icons, Menus, Pointers).

Et shell er bare et interface mellom brukeren og kjernen til operativsystemet.

En viktig komponent med et GUI er **Windows Manager**, som setter av plass på skjermen kalt vinduer, og holder styr på hvilke applikasjoner som er tilknyttet hvert vindu.

Kjerne til et operativsystem heter en kjerne (**kernel**). Denne inneholder software komponenter som utgjør basisfunksjoner som maskinens installasjon utgjør. En slik funksjon er «file manager», som koordinerer maskinens hovedlager (holder orden på hvor ting ligger på hovedlager).

For å gjøre det bedre for maskinen's bruker lar de fleste **file managers** (filbehandlere) filene bli samlet i en directory eller folder. Dette gjør at brukeren kan organisere filene sine etter eget formål, ved å samle like filer i en directory.

Ved å la directories inneholde andre directories, kalt subdirectories, blir det dannet hierarki (hierarchical organization).

Hvis du har mange directories inne i en annen directory, dannes det en sti av directories, kalt **directory path**. Path blir som oftest uttrykt ved å liste opp directories etter hverandre, separert med slashes. Feks dyr/forhistorisk/dinosaur. Denne starter i directory dyr, deretter subdirectory forhistorisk,og til slutt sub-subdirecotry dinosaur.

Hvis et annet program har lyst til å bruke en fil, må den spørre file manager om lov, og den gjør det ved å spørre file manager om den får lov til å åpne filen. Hvis ja, gir file manager informansjonen for å finne og manipulere filen.Denne informansjonen er lagret på et sted i main memory og blir kalt en file descriptor.

En annen komponent til kernel består av en samling av **device drivers** (enhetsdrivere), som er software enheten som kommuniserer med **the controllers** (noen ganger også direkte med ytre enheter) for å utføre arbeid på den ytre enheten tilkoblet maskinen.

Forklart på en annen måte: Det betyr at hver enhet i datamaskinen må ha en driver, for å gjenkjenne enheten og drive den.

Alle enhteter må ha unike drivere som er tilpasset for enheten. Driverne håndterer det enheten gjør og skal gjøre, sånn at andre enheter ikke trenger å tenke på det. Resultatet av dette blir et

generisk operativsystem som kan tilpasses som du vil, fordi ingen av driverne er avhengige av hverandre.

En annen komponent til kernel er **memory manager** (minnebehandling), som har som oppgave å koordinere maskinens bruk av hovedminne.

Et komponent i operativsymet kernel er **memory manager**, denne koordinerer maskinens bruk av hovedminne. I et system der datamaskinen bare utfører en oppgave av gangen er det minimalt med dette, men i system som består av flere brukere vil det være stor bruk for en memory manager. I disse tilfellene må mange programmer og blokker av data ligge i minnet, derfor må memory manager finne tildelt minneplass til disse og sørge for at utføringen til hvert program er begrenet til sin tildelte plass.

Oppgavene til en memory manager blir ytterligere komplisert når det totale som trengs overstiger minnet som er tilgjengelig. I dette tilfellet lager memory manager en illusjon av mer minne ved å rotere program og data frem og tilbake mellom hovedminne og hovedlager, dette kaller **paging**.

Illusjonen av ekstra minne som lagres av paging kalles **virtual memory**.

Når et operativsystem starter, gjør det det gjennom en prosedyre som kalles **boot strapping** eller «booting». Dette skjer hver gang en maskin starter, og prosedyren overfører operativsystemet fra hovedlager og inn i hovedminnet.

En CPU er designet slik at programtelleren (program counter) starter med en spesiell forutbestemt adresse hver gang CPUen starter. På dette stedet forventer CPUen å finne begynnelsen på programmet som skal utføres. En maskin sitt hovedminne blir slettet hver gang maskinen skrus av. Det er da ikke mulig å lagre operativsystemet der CPUen leter etter programmer, pga at CPUen ikke finner noe. For å løse dette er en liten del av hovedminne laget til å være **Read only memory** (ROM), som kan bare leses men ikke endres.

I vanlige maskiner er programmet **bootstrap** permanent lagret i maskinens ROM (programmet som lagres i rom heter **Firmware**), dette er da det programmet som kjøres når maskinen skrus på. Instruksjonene i bootstrap forteller CPUen at den skal overføre operativsystemet fra hovedlageret til en plass i minnet. Så forteller bootstrap CPUen at den skal hoppe (utføre et "jump") til denne posisjonen i hovedminne. på dette tidspunktet tar operativsystemet over. Dette kalles **booting**.

Turnkey systemer er maskiner som har all software permanent lagret i hovedminnet. feks en kaffemaskin.

3.3 KOORDINERE MASKINENS AKTIVITETER:

En av de mest fundamentale konseptene av moderne operativsystemer er forskjellen mellom et program og aktiviteten av å utføre et program. Den første er et statisk sett av instruksjoner, mens den siste er en dynamisk aktivitet som endres ettersom tiden går. Denne aktiviteten

kalles en **process** (prosess). Tilhørende en prosess har vi dens status, **process state**.

Oppgavene med å koordinere utføringen av prosessene håndteres av **scheduler** og **dispatcher** inni operativsystemets kernel. «Scheduler» holder styr på alle prosessene som kjører i systemet, legger til prosesser og fjerner utførte prosesser. For å holde styr på alt dette reserverer scheduler en plass i minnet som kalles **process table**. Hver gang et forespørsel om å utføre et program kommer, lager scheduler en ny post i process table.

En prosess er **ready** hvis den har en status slik at prosessen kan fortsette. En prosess er **waiting** hvis prosessen er forsinket inntil noe eksternt skjer (trykke på en knapp feks).

Dispatcher er en komponent av kernelen som overser utføringen av de registrerte prosessene. I time-sharing/multitasking systemer gjør en dette ved multiprogramming, ved å dele tid inn i små segmenter som hver kalles en **time slice** (vanligvis ikke mer enn 50 millisekunder), og så bytte CPU mellom prosessene slik at hver prosess kan utføres i en time slice.

Prosedyren som går ut på å bytte fra en prosess til en annen kalles en **process switch** (eller en **context switch**).

Hver gang dispatcher gir en time slice til en prosess, starter den en timer som vil si ifra når det er slutt på slicen ved å sende et signal som heter **interupt** (avbryt). Når CPUen mottar et avbruddsignal, gjør den ferdig maskinsyklen, lagrer posisjonen i den nåværende prosessen og begynner å utføre et program som kalles **interrupt handler**, som er lagret på en forutbestemt plass i hovedminnet. Denne er en del av **dispatcher** og beskriver hvordan dispatcher skal svare på avbruddsignalet. Etter dette velger dispatcher den prosessen i lista som har høyest prioritet blant de som har status **ready**, restarter **timer circut** og tildeler en **time slice**.

3.4 BEHANDLE KONKURRANSE MELLOM PROSESSER:

En viktig oppgave til et operativsystem er tildelingen av maskinens ressurser til prossessene i systemet. Med ressurs mener vi da både utre og indre enheter i maskinen.

File manager tildeler tilgang til filer, samt harddiskplass for å lage nye filer. Memory manager tildeler minneplass tildeler minneplass. Scheduler (=fordeler) tildeler plass i process table, og dispatcher tildeler time slices (tidsdeling).

Semafor: La oss se for oss et time-sharing/multitasking operativsystem som kontrollerer aktivitetene til en maskin med en separat skriver. Hvis en prosess trenger å printe resultatene sine må den be operativsystemet om tilgang til printerens driveenhet. Da må printeren operativsystemet bestemme om den skal gi tilgang til prosessen, avhenging av om printeren allerede er i bruk av en annen prosess. Hvis den ikke er det skal operativsystemet godkjenne prosessen. Hvis den er i bruk, skal operativsystemet nekte prosessen og kanskje sette den på vent til printeren er ledig.

For å kontrollere tilgangen til printeren må operativsystemet holde orden på om den er opptatt eller ikke. En måte å gjøre dette på er å bruke et flagg, som i denne sammenhengen viser til en bit i memory hvor statusen ofte er *as set (opptatt)* og *clear (klar)*, isedenfor 1 og 0.

Det er et problem med denne flaggmetoden. Oppgaven krever flere maskininstruksjoner. Det er derfor mulig for en oppgave å bli avbrutt etter at et klart-flagg har blitt oppdaget, men før flagget har blitt opptatt. Flagget hentes fra hovedminne og finner ut at skriveren er klar. Men så får prosessen et avbruddssignal, og en ny prosess begynner å kjøre. Den nye prosessen vil også bruke printeren, igjen vil flagget som er hentet fra hovedminne være klart, fordi den forrige ble avbrutt før operativsystemet hadde tid til å sette flagget i hovedminne.

En løsning på dette problemet er interrupt disable og interrupt enable. Instruksjoner som er skrevet i maskinspråk. Disable blokkerer alle avbruddsignal, enable gjør det motsatte.

En annen måte er å bruke **test-and-set** metodene, det de gjør er å si til CPUen at de skal hente verdien til flagget og notete den, og sette verdien til flagget i en instruksjon. Fordelen med det er at CPUen alltid gjør ferdig en instruksjon før den gjenkjenner et avbruddsignal.

Et ordentlig implementert flagg som nettopp ble beskrevet kallet et **semaphore**.

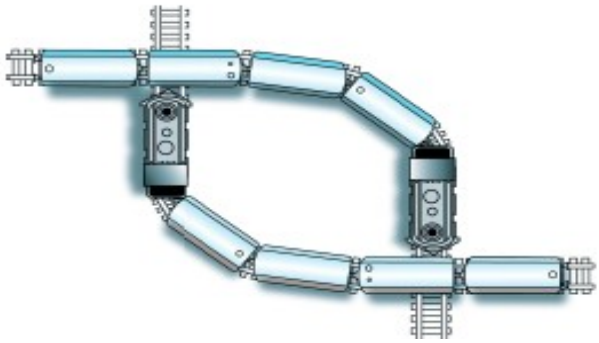
Krav om at bare en prossess av gangen har lov til å utføre en **critical region** (kritisk område)

kalles **mutual exclusion**. Oppsummert betyr dette at en vanlig måte å få mutual exclusion på en critical region er å beskytte en critical region med en semaphore.

> **Vranglås / «deadlock»:** Et annet problem som kan oppstå under ressursfordelingen er vranglås. Da blir to eller fler prosesser stoppet fra å fortsette, for hver av dem venter på aksess til en ressurs som den andre har.

Feks: en prossess har tilgang til printeren, men den venter på tilgang til datamaskinens cd'spiller. En annen prosess har samtidig tilgang til cd'spilleren, men venter på tilgang til printeren.

Et annet eksempel: bildet.



Betingelsene for at vranglås skal oppstå:

- Det må være konkurranse for ressursene som ikke kan deles (nonshareable resources).
- Ressursene tildeles på partiell basis, en ressurs om gangen.
- Når en ressurs har blitt tildelt, kan den ikke kreves tilbake.

Vranglåsen kan fjernes ved å angripe en av disse kriteriene.

Teknikker som angriper den tredje betingelsener i kategorien deadlock detection og correction schemes. I disse tilfellene er forekomsten av vranglås ansett som et så fjerntliggende problem at det ikke er laget noen måte for å unngå den. Istedenfor så finner man det hvis det skjer, og

retter på det ved å tvinge tilbake noen av de tildelte ressursene. Hvis det skjer en deadlock kan operativsystemet eller administrator fjerne (**kill**) noen av prossessene.

Teknikker som angriper en av de to første betingelsene er kjent som deadlock avoidance schemes. En metode er å angripe den andre betingelsen ved å kreve at hver prosess skal be om alle ressursene på en gang. En annen metode er å angriper den første ,betingelsen, ikke ved å fjerne konkurranse direkte, men ved å omgjøre ikke-delbare ressurser til delbare.

Spooling er en teknikk for å gi mange prossesser tilgang til en ressurs.

3.5 SIKKERHET:

Siden operativsystemet overvåker aktivitetene til maskinen, er det naturlig at den har et ansvar med å opprettholde sikkerhet også. I vid betydning viser ansvaret seg i mange former, en er pålitelighet.

Angrep fra utsiden: En viktig oppgav for operativsystemet er å beskytte maskinens ressurser fra uautorisert personell. Når det gjelder maskiner som blir brukt av flere personer, så kan en lage «kontoer» (accounts) som operativsystemet kan bruke under en **login** procedure. Dvs. At brukeren logger inn med sin konto, som operativsystemet gjenkjenner, slik at brukeren kan kontrollere deler av systemet.

Kontoer blir laget av en **super user** eller en **administrator**, dette er en person med høy aksess til operativsystemet etter en login.

Auditing software er software som lagrer og analyserer aktivitetene som tar sted i en maskins system. Feks en rekke forsøk på å logge inn med feil passord, som kan tyde på at noen uten tilgang prøver å logge seg inn. Auditing software kan også identifisere aktiviteter innenfor en brukers konto som ikke stemmer overens med brukerens tidligere oppførsel, noe som kan tyde på at en uautorisert bruker har fått tilgang til denne kontoen.

En annen slyngel som auditing software systemer er laget for å finne er såkalt **sniffing software**, som er software som registrerer aktiviteter og senere sender disse til en mulig ubuden gjest. F.eks kan man bruke dette til å sniffe passord osv.

Ofte er det brukerne selv som har skyld i alle sikkerhetsproblemer, feks når noen lager «Hanne» som passord, som er relativt lett å gjette.

Angrep fra innsiden: CPUer for multiprogrammeringsystemer er laget for å operere i en av to **privilege levels**, som heter «privileged mode» og «unprivileged mode». I privileged mode kan CPUen utføre alle instruksjonene i maskinspråket, mens i unprivileged er det begrenset. Instruksjonene som er tilgjengelig i privileged mode kalles **privileged instructions**.

Typiske slike instruksjoner er de som endrer innholdet i minne register og instruksjoner som kan endre privilege nivået i CPUen.

Kapittel 4 – Nettverk og internett

Med nettverk kan datamaskiner dele data. Et netteverk er ofte klassifisert som:

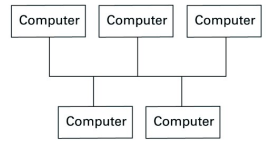
- Local area network (LAN). F.eks Maskiner i en bygning.
- Metropolitan area network (MAN). F.eks maskiner i et samfunn.
- Wide area network (WAN). F.eks. maskiner i nabo byer eller på andre siden av verden.
- Open network. Det offentlige, som Internett.
- Closed network. Innovasjoner eid av individer.

Internett er styrt av TCP/IP protokoll sett.

Topologier

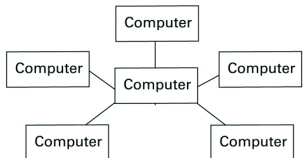
- **Bus** (1990) Implementert under et sett standarder kalt Ethernet(mest populær i dag)

b. Bus



- **Star** (1970)Populær i trådløst nettverk.

c. Star



- Andre toplogier er ring og irregular.

Hub: Sender alle signaler den mottar, ut igjen til alle maskiner koblet til den (fungerer som bus, men ser ut som star)

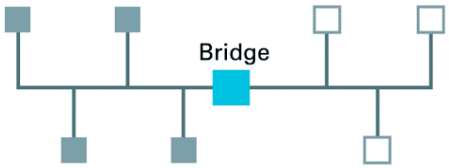
Protokoller- regler

- I et bus nettverk basert på Ethernet standarder, er retten til å sende meldinger kontrollert av Carrier Sense, Multiple Access with Collision Detection(CSMA/CD). Den sender data når busen er stille, hvis to sender samtidig stopper begge opp en vilkårlig stund og prøver igjen.

- For trådløst nett- Carrier Sense, Multiple Access with Collision Avoidance **(CSMA/CA)**. Den venter en kort stund, hvis kanalen har vært stille denne tiden prøver den å sende, og de som har ventet lengst får sende først.
- Hidden terminal Problem kan alle ende systememer kommunisere med AP, men ikke med hverandre. En løsning på dette i noen WiFi nettverk løses ved at maskinene må sende forespørsel til AP og vente på klarsignal.

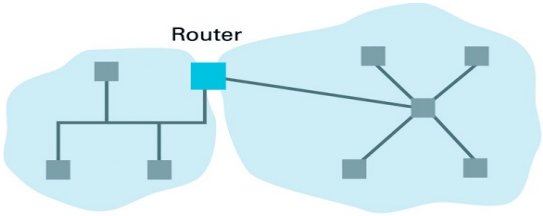
Man kan koble buser sammen ved å bruke:

- **Reapteres**. Sender signaler fram og tilbake mellom to buser.
- **Bridge**. Sender melding over busen bare når adressen er til en maskin på andre siden.



- **Switch.** Kan koble flere buser sammen og fungerer som en bridge.

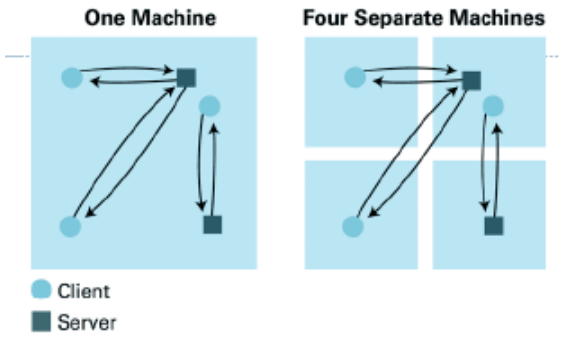
- Hvis WiFi nettverk og Ethernet nettverk skal kobles må det bli koblet på en måte som bygger nettverk av nettverk, dette er kalt **internett** (Internett er noe annet).
- For å lage internett må man bruke **ruter**, som er spesial maskiner brukt til å videresende meldinger.



  - Man bruker adresse system. Hver maskin har en original **lokal adresse** i sitt eget nettverk og en **internett adresse**.
  - Ruteren har en fremsendings tabell (som en adresse bok)
  - **Gateway** er stedet et nettverk er linket til et internett.

Aktiviteter utført på maskiner i et nettverk må kommunisere for å koordinere oppgaver. Denne kommunikasjonen kalles **interprocess communication.** 2 typiske måter er:

- **Kilent/server modell.** Der må en server hjelpe flere klienter samtidig. F.eks kan en printer være en server.



- **Peer to peer modell (P2P).** Prosess der den mottar og gir service. Som når man sender meldinger til hverandre over Internett.

Distributed Systems
Moderen software systemer er designet som Distributed System. De består altså av software enheter som utføres som prosesser på forksjellige datamaskiner.

4.2 INTERNETT

I dag linker Internet en worldwide kombinasjon av LAN, MAN og WANs. En samling av koblete nettverk. Disse er laget av organisasjoner kalt Internet Service Providers **(ISPs)**. System av nettverk betjent av ISP kan klassifiseres i et hierarki:

1. På toppen er **tier-1 ISPs** som består av høy hastighet og høy kapasitet, internasjonal WANs.
2. Koblet til tier-1 ISPs er **tier-2 ISPs.** Den er litt svakere i mulighetene. Men begge to blir sett på som kjernen av Internet.
3. En **access ISP** er under her igjen. Det er et uavhengig internett, noen ganger kalt intranett.
4. Apparatet som individer kobler seg til access ISP med, er **end system eller host(vert).**

Har også andre måter å koble til access ISP som telefonlinjer og kabel. Den fortest voksende er trådløs tilkobling basert på WiFi teknologi. Her er tanken å koble AP til en access ISP. Området til AP'en sin rekkevidde er kalt **hot spot.**

Internett Adressering
- Internett Adressering kalles **IP adresses**(32 bit, snart 128 bit)
- ICANN skal koordinere Internettets operasjoner.
- IP adresse er skrevet i dotted decimal notation.
- Alternativ adressering er at hver maskin identifiseres med **mnemonic navn** (domene konsept). Leselig for mennesker. Hvert domene registreres med ICANN, en prosess kalt registrars (eks. aw.com)

- **.com** er endelse kalt top-level domains (TDLs)
- Har også to- bokstavet endelse som er til country code TLDs (eks. **.no**)
- Navn til venstre er subdomene (**r2r2.compsc.nowhere**.edu)
- Navnservere konverterer mnemonic adresse til IP adresse. Felles systemet heter Domene Navn System(DNS) og prosessen kalles DNS lookup.

Internet Applications

- Elektronisk mail: Simple Mail Transfer Protocol (SMTP) Blir brukt for å sende mail mellom mail serverene og sende ny mail fra personens lokale maskin til personens mail server.

Adresse består av symbol streng etterfulgt av @, etterfulgt av mnemonic streng som gjenkjenner destinasjonens domene. (taroy@stud.ntnu.no)

- Fil Transfer Protokoll(FTP): Client/ server protokoll for å sende filer over the Internet.
- Telnet og Secure Shell: Med telnet kan man kontakte en maskin fra lang avstand og følge dens operativ system innloggings måte for å få adgang til maskinen. Men det er ikke kryptert. Secure Shell er alternativ som er kryptert.
- VoIP (Voice over Internet Protocol): Voice kommunikasjon som telefon system. Skype er et eksempel. Trussel mot tradisjonelt telefon system.
- Internet Radio: Prosess kalt Webcasting. Intern Radio er eksempel på Streaming audio, som vil si overføring av lyd data på sanntid basis. Stasjonen har server som sender programmet til klientene som etterspør, dette kalles N-unicast. Dette har negative sider. Alternativer er P2P eller multicast. Med multicast sender Serveren bare èn kopi og det er ruterens oppgave å kopiere meldingen.

4.3 WORLD WIDE WEB

Multimedia over Internett er basert på hypertext, teks dokumenter som inneholder linker, hyperlinks, til andre dokumenter. I dag har vi også bilder lyd og video og blir derfor ofte kalt hypermedia. Dokumenter er linket til andre dokumenter og et web av lignende informasjon blir dannet. I et data nettverk kan dokumenter i slike web være på forskjellige maskiner og danner et nettverk wide web. Weben på Internett omfatter hele verden og er kalt World Wide Web.

Software pakker som tillater brukere til å ha tilgang til hypertext på Internet er pakker som er klient eller pakker som er server. En klient pakke kalles browser og server pakke kalles Web server. Hypertext dokumenter blir overført mellom browsere og web servere ved å bruke Hypertext Transfer Protocol(HTTP).

For å finne dokumenter på world wide web, har dokumentene en unik adresse Uniform Resource Locator(URL).

http://ssenterprise.aw.com/authors/Shakespear/Julius_Caesar.html

http (protocol man trenger for å få tilgang til dokumentet)

://ssenterprise.aw.com (Mnemonic navn til vert som holder dokumentet)
/authors/Shakespear (Register bane som indikerer plassen til dokumentet inni vertens fil system)

/Julius_Caesar.html(Dokument navn)

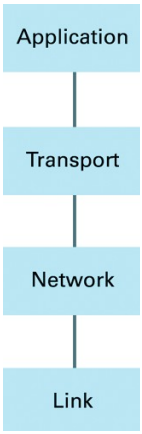
HTML
Hypertext er samme som tekst fil, men hypertext inneholder tags som bestemmer hvordan oppsettet på dokumentet skal se ut. Systemet med tags kalles Hypertext Markup Language

(HTML). (<html> <head> innledning </head> <body> Det som kommer på skjermen </body> </html>)

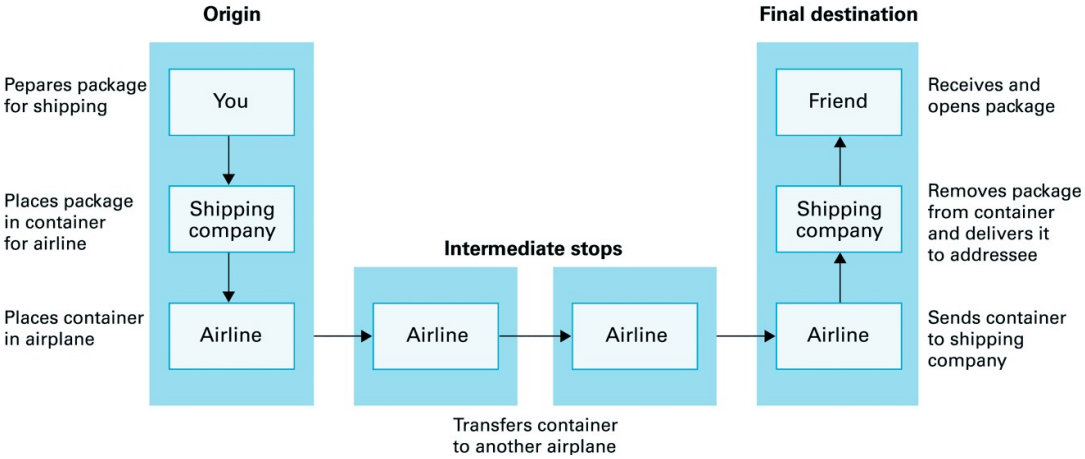
XML(eXtensible Markup Language)
Representere data som tekstfil.(Noter, matematiske uttrykk) Samme stil som HTML.

Klient og server-side aktivitet
I noen tilfeller vil vi ha en web side som inneholder animasjon eller fylle inn info. Hvis aktiviteten blir utført av klienten(browser) kalles det client-side, eller server-side hvis det blir utført av server(Web server).

4.4 INTERNETT PROTOKOLLER



Software som kontrollerer kommunikasjon over Internett har fire lag: **Application lag, transport lag, nettverk lag og link lag.**
En melding starter i application laget som har som oppgave og oversette mnemonic namn til IP adresse og bruker transport laget til å sende meldingen. Tarnsport laget skal godkjenne meldingen, og gjøre den til mindre deler eller pakker. Disse blir sendt over Internett som individuelle enheter. Netteverket skal sørge for at pakkene hele tiden blir sendte rikitge veier. Link laget i ruteren skal ta imot og sende pakkene videre opp igjen i hierarkiet til den er levert i application laget på meldingens destinasjon. Det kan sammenlignes med å sende en pakke med et fly.



TCP/IP Protokoll sett

- Samling protokoll standarder brukt av Internett for å oppfylle det fire nivå-kommunikasjons hierarkiet utført i Internett.
- Applikasjons laget kan velge å sende data via en TCP (Transmission Control Protocol) eller UDP (User Datagram Protocol) versjonen av transport laget. TCP etablerer kontakt før den sender en melding mens UDP bare sender. UDP kalles derfor en kontaktløs protokoll. Flere forskjeller mellom TCP og UDP. UDP er mer effektiv og kjapp, men sjekker ikke alltid kontakt og lignende (UDP bedre for DNC lookup og VoIP, TCP bedre for mail).

- IP standarder behandler med protokoller brukt for kommunikasjon blant nabo nettverk layers, når de utveksler ruting informasjon.

4.5 SIKKERHET

Mange måter en data kan bli angrepet på, bruker onde software, ofte kalt malware.

- Virus er software som smitter en data ved å legge inn seg selv i et program som alt finnes i maskinen. Når vert programmet blir utført blir også viruset utført.
- Worm er et selvstendig program som overfører seg selv gjennom et nettverk, tar opp plass i datamaskiner og videresender kopier av seg selv til andre datamaskiner.
- Trojansk hest er et program som går inn i et datamaskin- system kledd som et ønskelig program
- Spyware er software som samler informasjon om aktiviteter på datamaskinen og rapporterer informasjonen tilbake til opphavsmannen til angrepet.

Phising er en metode som spør om informasjon og utgir seg for å være en annen. F.eks gjennom mail.
- Denial of service (DoS). Overbelaste en datamaskin med meldinger.
- Spam er uønsket junk email.

Beskyttelse og helbredelse

- Firewall ligger et sted i nettverket og filtrerer trafikk (kan ligge i gateway).

Spoofing er å utgi seg for å være en annen.

- Spam filters er firewalls laget for å stoppe uønsket email.
- Proxy server er en software enhet som fungerer som et mellomledd mellom klient og server. Da kan ikke serveren få masse informasjon om klienten som senere kan bli brukt imot klienten.
- Antivirus software oppdager og fjerner virus og infeksjoner.

Kryptering
- Kryptere passord så ”fienden” ikke kan misbruke det.
- Finnes Secure version av systemer. HTTPS er secure version av HTTP.

- Public key kryptering bruker to nøkler. Public key som krypterer meldinger, og private key som avkoder meldinger. Public key brukes av de som skal sende en melding til et bestemt sted og den som skal motta meldingen har private key.
- Certificate authorities sin oppgave er å ha en liste over selskap og deres public keys. (Blir brukt for at man ikke skal kunne utgi seg for å være en annen, og gi ut sin public key.

Kapittel 5 - Alogritmer

For at en datamaskin skal kunne utføre en oppgave trenger den en algoritme som forteller den *nøyaktig* hva den skal gjøre. Den uformelle definisjonen av en algoritme er som følger: En algoritme er en mengde steg som definerer hvordan en oppgave skal utføres.

Først kan det være greit å ha klart for seg forskjellen mellom en algoritme og et program

- En algoritme eksisterer uavhengig av et program.
- Et program blir det først når denne fremgangsmåten er representert på en form som datamaskinen kan kjøre
- En algoritme kan representeres på mange forskjellige måter
  - Tekstlig – på ulike språk
  - Program – i forskjellige programspråk
  - Matematisk – i matematisk notasjon
  - Grafisk – bildeserie, animasjon

5.1 KONSEPTET ALGORITME

Et uformelt overblikk

Algoritmer er ikke bare beregnet på tekniske aktiviteter, det kan brukes til det meste. Man kan lage algoritmer for å skrelle erter, gjøre tryllekunster, osv. Det viser seg maskinsyklusen ikke er noe annet enn en enkel algoritme:

Så lenge stopp-instruksjonen ikke har blitt utført, fortsett å utføre de følgende stegene:

a) *Hent en instruksjon.*

b) *Dekod instruksjonen.*

c) *Utfør instruksjonen.*

Forskere mener at alle aktivitetene i den menneskelige hjernen, inkludert fantasi, kreativitet og avgjørelser, egentlig er et resultat av algoritmer

Den formelle definisjonen av en algoritme

Formell definisjon: En algoritme er en *ordnet mengde* av *utvetydige*, *kjørbare* steg, som definerer en *avsluttende* prosess.

Denne definisjonen krever at stegene i en algoritme må ha en veletablert struktur, med tanke på hvilken rekkefølge de ska utføres i. Dette betyr nødvendigvis ikke at stegene må utføres sekvensiell, hvor det første steget, etterfølges av et annet, og så videre. Noen algoritmer, kalt parallelle algoritmer, inneholder mer enn en sekvens med steg. Hver av disse sekvensene er lagd for å utføres av forskjellige prosessorer i en multiprosessormaskin. I slike tilfeller har ikke den generelle algoritmen et eneste sett med steg som kan sammenlignes med første-steg, andre-steg scenarioet. I stedet for består algoritmen av flere tråder som deler seg og kommer sammen igjen, ettersom ulike prosessorer utfører ulike deler av den helhetlige oppgaven. Det finnes også algoritmer som blir utført av kretser (feks. flip-flop), hvor hver port utfører et enkelt steg av den helhetlige algoritmen. Her blir stegene kontrollert/ordnet etter/av årsak og effekt, etter hvert som portenes handling sprer seg gjennom kretsen.

Det er svært viktig at en algoritme består av kjørbare steg, for ellers ville det ikke vært en algoritme:

Sett opp en liste over alle de positive heltallene.

Dette er ikke en algoritme, ettersom denne oppgaven vil være umulig å utføre, for det finnes jo uendelig mange positive heltall. Datavitenskapen bruker begrepet *effektiv* for å omfange begrepet om å være kjørbar. Altså, å si et steg er effektivt betyr at det er gjørbart. Et annet krav til algoritmer er at de må utvetydige. Det vil si at når en algoritme utføres, må informasjonen i prosessen være tilstrekkelig til å bestemme hvilke handlinger hvert enkelt steg krever. Med andre ord, å utføre stegene i en algoritme krever ingen kreative ferdigheter, det krever kun evnen til å følge instruksjoner. En algoritme må definere en avsluttende prosess, med andre ord må utførelsen av en algoritme må føre til en slutt. Men det finnes likevel områder hvor ikke-avsluttende prosesser er svært nyttige, som for eksempel å overvåke pasienters vitale verdier, opprettholde flyhøyde under en flyvning. Enkelte vil påstå at disse bruksområdene kun involverer repetisjon av algoritmer, hvor hver algoritme når et endepunkt, for og så automatisk gjentas. Andre vil påstå at slike argumenter kun er forsøk på å holde fast ved en overbegrenset formell definisjon. Begrepet algoritme ofte brukt i uformelle sammenhenger, der det refereres til en mengde steg som ikke nødvendigvis definerer en avsluttende prosess. For eksempel lang-divisjon ”algoritmen” som ikke definerer en avsluttende prosess for å dele 1 på 3. Teknisk sett er slike tilfeller misbruk av begrepet algoritme.

Algoritmers abstrakte natur

Det er en vesentlig forskjell mellom en algoritme og dens fremstilling. Denne forskjellen tilsvarer forskjellen mellom en historie og en bok. En historie er abstrakt, mens en bok er en fysisk representasjon av en historie. På samme måte er en algoritme forskjellig fra hvordan den fremstilles. Den samme algoritmen kan representeres på ulike måter:

$F = ((^{9}/_{5}) * {}^{o}C) + 32$

Multipliser temperaturen i Celsius med $^{9}/_{5}$ og legg så til 32 til produktet.

Forskjellen mellom en algoritme og dens fremstilling kan skape problemer når vi skal kommunisere via algoritmer, ofte med tanke på hvor detaljert algoritmen skal være. Dette avhenger av hva slags forkunnskaper den som skal utføre algoritmen har. Algoritmen må være detaljert nok til at den kan forstås av vanlige folk. Er den ikke detaljert nok, vil den

kunne oppfattes som tvetydig. Mens vi er inne på forskjellen mellom algoritmer og hvordan de representeres, ta vi for oss forskjellen mellom to relaterte konsepter; programmer og prosesser.

> En *algoritme* er en mengde entydige, gjennomførbare steg som definerer en prosess som har en begynnelse og en slutt.

> Et *program* er en mengde instruksjoner som grundig forteller hvordan et problem kan løses, en representasjon av en algoritme på en form som kan kjøres av en datamaskin.

> En *prosess* er et program under kjøring; det er en dynamisk aktivitet hvor egenskapene endres etter tid.

5.2 ALGORITMEREPRESENTASJON

Primitives

Algoritmerepresentasjon krever en form for språk, som varierer etter hvem/hva som skal bruke/tolke algoritmen (engelsk, norsk, tall, bilder osv.). I disse ”språkene” kan begreper ha mer enn en betydning, noe som ofte leder til problemer. Kommunikasjonsproblemer oppstår også når språket som brukes til å representere algoritmen ikke er tydelig definert, eller når informasjonen ikke er tilstrekkelig detaljert.

Innen datavitenskapen er det etablert et veldefinert sett med byggeklosser (primitives) som algoritmerepresentasjoner kan konstrueres ut fra. Ved å gi klare definisjoner til disse primitivene fjerner man mange problemer i forhold til tvetydighet, og ved å kreve at algoritmer må være beskrevet innenfor rammene av disse primitivene, gir det en felles standard for detalj. En samling primitiver sammen med en samling regler for hvordan disse primitivene kan kombineres, slik at de kan representere mer komplekse ideer, er grunnlaget for et programmeringsspråk.

Hver primitiv har sin egen syntaks (setningsoppbygging) og semantikk (betydning). For eksempel *air*: Syntaksen til *air* består av tre symboler, hvor semantikken er en gassaktig substans som omgir verden.

For å oppnå en samling av primitiver som skal kunne brukes til å representere algoritmer i forbindelse med datamaskiner, kan vi se på de enkelte instruksjonene maskinen er konstruert for å utføre. Hvis en algoritme er uttrykt *så* detaljert vil vi garantert ha et program som er passende for maskinutføring. Men å uttrykke algoritmer i en slik grad av detalj er langtekkelig, så man bruker vanligvis en samling av ”high-level” primitiver, hvor her enkelt primitiv er et abstrakt redskap som er konstruert av ”lower-level” primitiver som allerede befinner seg i maskinens språk. Resultatet er et formelt programmeringsspråk hvor algoritmer kan utrykkes på et adskillig høyere nivå, enn i maskinspråk.

Pseudokode

En pseudokode er et kompromiss mellom naturlig språk og programmeringsspråk. Når vi skriver i pseudokode konsentrerer vi oss om algoritmen(fremgangsmåten), i stedet for reglene (syntaksen) for det aktuelle programmeringsspråket. Da ender man opp med noe som lett lar seg oversette til et konkret programmeringsspråk.

I denne læreboka vil vi betrakte algoritmeutvikling og algoritmerepresentasjon uten å begrense oss til et konkret programmeringsspråk. Så vår tilnærming til pseudokode er å utvikle en konsistent, konkret skrivemåte for å representere (reccuring semantic structures). Etter hvert vil disse strukturene bli primitivene vi tar utgangspunkt i for å utrykke fremtidige ideer. Her følger noen eksempler på pseudokode:

Midler ← Sparekonto + Brukskonto

Her gir vi navnet Midler summen av Sparekonto og Brukskonto, og Midler kan senere brukes til å referere til den summen.

```
if (ikke regn)
        then  (if (temperatur = varm)
                  then (dra og svøm)
                  else (spill golf)
                 )
        else (se på tv)
```

Vanlig strukturer i pseudokode:

```
if (tilstand) then
(aktivitet)
```

```
while (tilstand) do
```

Hensikten med pseudokoder er å finne en måte å representere algoritmer, på et lesbart, uformelt vis. Vi ønsker et skriftsystem som kan hjelpe oss med å uttrykke våre ideer, hvor vi ikke må følge strenge, formelle regler. Men vi kan for all del endre/utvide pseudokoden når det trengs.

5.3 OPPDAGELSE AV ALGORITMER

Utviklingen av et program består av to aktiviteter: å oppdage den underliggende algoritmen, og å representere den algoritmen som et program.

Å finne algoritmer er ofte den vanskeligste delen av programvareutvikling, siden det innebærer at man må finne en metode for å løse et problem. Så for å forstå problemløsingsprosessen er det viktig å forstå hvordan algoritmer blir oppdaget.

Kunsten å løse problemer

Å finne løsninger på problemer kan være vanskelig, naturligvis. Dersom det fantes en algoritme for å løse problemer ville det gjort verden enklere, men dette har vist seg å være umulig, da ikke alle problemer har algoritmiske løsninger.

I 1946 kom matematikeren G. Polya fram til noen grunnleggende, uformelle faser for problemløsing:
(NB! Disse fasene er ikke steg man må følge når man skal løse et problem, men faser som vil gjennomgås en eller annen gang i løpet av problemløsingsprosessen, ikke nødvendigvis i riktig rekkefølge.)

1. Forstå problemet.
2. Tenk ut en plan for å løse problemet.
3. Utfør planen.
4. Evaluer løsningens med tanke på nøyaktighet, og om den har potensial til å brukes til å løse andre problemer.

Dersom vi oversetter disse punktene til programutvikling, får vi:

1. Forstå problemet.
2. Få et begrep om hvordan en algoritmisk fremgangsmåte kan løse problemet.
3. Formuler algoritmen og representer den som et program.
4. Evaluer programmet med tanke på nøyaktighet, og om den har potensial til å brukes til å løse andre problemer.

Husk at vi diskuterer hvordan problemer blir løst, ikke hvordan vi vil at de skal løses. Når man skal utvikle store software-systemer, kan det å oppdage en feil så sent som i fase 4, bety et enormt tap i ressurser. Å unngå slike katastrofer er et viktig mål for programutviklere, som ville insistert på en grundig forståelse av problemet, før man går i gang med å finne en løsning. Men man kan jo si at en sann forståelse av et problem ikke er oppnådd før man har funnet en løsning. Det faktum at et problem er uløst tyder på mangel på forståelse. Så det å insistere på en grundig forståelse av spørsmålet før man foreslår mulige løsninger, er nokså idealistisk.

Et annet mystisk fenomen er man kan få inspirasjon til å løse et problem man tidligere har jobbet med, uten suksess, mens man jobber med en annen oppgave. Dette fenomenet ble identifisert av H. von Helmholtz i 1896. Fenomenet reflekterer en prosess hvor underbevisstheten fortsetter å jobbe med problemet, og dersom den finner en løsning, vil den ”tvinge” løsningen inn i bevisstheten. I dag er perioden mellom bevisst arbeid med et problem og den plutselige inspirasjonen, kjent som en inkubasjonsperiode.

<u>Få foten i døra</u>

Det er mange ulike fremgangsmåter for å løse en problem på, men en felles faktor for alle disse, er at det gjelder å få foten i døra. Vi bruker et eksempel for å illustrere dette:

A sa at B ville vinne.
B sa at D ville komme på siste plass.
C sa at A ville komme på tredje plass.
D sa at A sitt utsagn ville være korrekt.

Bare et av disse utsagnene var korrekte, og dette var utsagnet vinneren kom med. Hva ble rekkefølgen?

Siden A og D har samme utsagn, og bare et av utsagnene var korrekte, vil begge disse utsagnene være falske. Ergo, vant verken A eller D. (På dette tidspunktet har vi fått foten i døra, og for å komme frem til den fullstedige løsningen må vi bare utvide den kunnskapen vi har fra dette punktet.) Hvis A sitt utsagn var galt, betyr det at heller ikke B vant. Den eneste muligheten er at C vant, og da vil C sitt utsagn være korrekt. Altså kom A på tredje plass. Det betyr at rekkefølgen var enten CBAD eller CDAB. Her kan vi eliminere sistnevnte, siden B sitt utsagn må være galt, og rekkefølgen blir som følger: CDAB.

Å bli fortalt at man må få foten i døra er ikke det samme som å bli fortalt *hvordan* man skal gjøre det. Når man har fått dette fotfestet, samtidig som man innser hvordan man skal utvide den opprinnelige kunnskapen til en løsning på problemet, trenger man kreativ input fra en hva-ville-skje problemløser. Det finnes imidlertid mange generelle tilnærmingsmåter, foreslått av Polya og andre, angående hvordan man skal få foten i døra. En av disse går ut på at man forsøker å løse problemet baklengs. Eks. Dersom man vil finne en algoritme for å brette en papirfugl, kan man begynne med den ferdigbrettede fuglen, og så brette den ut, for å se hvordan den er konstruert.

En annen generell måte å tilnærme seg problemløsing er å se etter et relatert/lignende problem som enten er enklere å løse, eller som har blitt løst tidligere, for å så forsøke å tilføre dets løsning til det nåværende problemet. Denne teknikken har stor verdi i forbindelse med programutvikling. Generelt sett er ikke programutvikling en prosess for å løse et bestemt problem, men å finne en generell algoritme som kan brukes til å løse alle former av problemet. Som for eksempel hvis man skal lage en algoritme for å sortere en liste med navn, vil man lage en algoritme som kan sortere en hvilken som helst liste med navn, ikke bare den listen vi skal sortere i utgangskpunktet.

Enda en tilnærmingsmetode for å få en fot i døra er å bruke *stepwise refinement*. Det er en teknikk som går ut på at man ikke forsøker å løse hele oppgaven, i all dens detalj, på en gang. Ideen går ut på at man deler det opprinnelige problemet inn i flere mindre problemer, helt til man sitter igjen med en mengde lettløselige ”underproblemer”, som naturligvis er enklere å løse en hele det opprinnelige problemet.

Stepwise refinement er en *top-down metode*, ettersom den går fra den generelle til det konkrete. *Bottom-up metoden* går fra det konkrete til det generelle. *Top-down metoden* er essensielt sett et organiserende verktøy, hvor dets problem-løsende egenskaper er konsekvenser av denne organiseringen. Det har lenge vært en viktig design-metode innen dataprosesseringssamfunnet, hvor utvilklingen av store softwaresystemer inkluderer en betydelig organiserende komponent. Som vi vil se i kapittel 7, blir store softwaresystemer ofte designet ved å kombinere ferdiglagde komponenter, en tilnærmingsmetode som tilsvarer *bottom-up*. Så både *top-down* og *bottom-up metodene* er viktige verktøy innen datavitenskap/informatikk.

Grunnen til at det er så viktig å opprettholde et såpass bredt perspektiv, altså at man tar med seg foruttatte ideer og forhåndsvalgte verktøy til problemløsingsprosessen, er at det kan gjøre det vanskelig å se problemets enkelthet.

Vi kan konkludere med at algoritmeoppdagelse forblir en utfordring som må utvikles over lang tid, fremfor å bli lært som et fag, bestående av veldefinerte metoder. Ved å trene opp en potensiell problemløser til å følge bestemte metoder, knuser man de kreative ferdighetene, fremfor å dyrke dem, som man egentlig burde.

5.4 INTERATIVE STRUCTURES (GJENTAGENDE STRUKTURER)

I gjentagende strukturer blir en samling instruksjoner gjentatt i en sløyfe-struktur.

Sekvensiell søkealgoritme

Dersom vi skal søke etter en bestem verdi (ord, tall etc.) i en liste kan det være fordelsmessig å utvikle en algoritme som bestemmer om verdier finnes i lista. Hvis verdien er i lista er søket en suksess, hvis ikke er det en fiasko. Vi går ut ifra at lista er sortert på en eller annen måte (alfabetisk, økende/minkende tallverdi). Slik kan man gå frem:

Figur **

1. Kall det du skal søke etter ”søkestreng”.
2. Kall første element ”testelement”.
3. Hvis testelement == søkestreng (sammenlign) →søket er en suksess
   hvis ikke, gå til neste element (dersom det er et).
   Gjenta dette til det ikke er noen elementer igjen.
4. Hvis man er kommet til siste element og ikke
   funnet element man leter etter
   → søket er en fiasko

Figur 5.6

```
procedure Search (List, TargetValue)
if(List empty)
    then
        (Declare search a failure);
    else
        (Select the first entry in List to be
TestEntry);
         while(TargetValue > TestEntry and
there
                are remain entries to be
         considered)
         do (Select the next entry in the list
as TestEntry.);
         if(TargetValue = TestEntry)
         then (Declare the search a
```

En slik søkealgoritme kalles en sekvensiell søkealgoritme. Den er ganske enkelt, og brukes ofte på kortere lister, da den ikke er særlig effektiv når det kommer til lengre lister.

Loop Control

Det å repetere en instruksjon eller en sekvens av instruksjoner, er et viktig algoritmisk konsept. En måte å gjennomføre en slik repetisjon er den ”gjentagende strukturen” loop, eller sløyfe på godt norsk. En sløyfe består av en samling instruksjoner, dette kalles ”kroppen” til sløyfa. Disse instruksjonene utføres på en gjentagende måte, under veileding av en kontrollprosess. Figur 5.6 illustrerer dette. Her bruker vi en while-løkke for å kontrollere repetisjonen av den bestemte kommandoen **Select the next entry in List as the TestEntry**. Slik jobber while-sløyfen:

*(**while**(betingelse) **do** (innhold))*

sjekk betingelse
utfør innhold
sjekk betingelse
utfør innhold
.
.
.
sjekk betingelse

Helt til betingelsen ikke lenger gjelder.
Ved å bruke sløyfe-strukturen oppnår vi en større grad av fleksibilitet, enn om vi bare hadde skrevet innholdet flere ganger. For eksempel kan vi skrive denne kommandoen flere ganger:

Tilsett en dråpe svovelsyre.
Tilsett en dråpe svovelsyre.
Tilsett en dråpe svovelsyre.

Men det vil ikke gi den samme sekvensen som sløyfe-strukturen gir:

while(pH-verdi > 4)
    do(tilsett en dråpe svovelsyre)

ettersom vi på forhånd ikke vet hvor mange dråper som trengs.

Selv om det er selve innholdet i sløyfa som utfører oppgaven, er kontrollstrukturen minst like viktig, og det viser seg at det er her det oftest oppstår feil.

Figur 5.7 Komponenter i gjentagende kontroll

Initialiser:	Opprett et utgangspunkt som vil forandres i retning av den avsluttende betingelsen.
Test:	Sammenlign den nåværende tilstanden med den avsluttende betingelsen, og avslutt repetisjonen hvis de er like.
Forandre:	Endre tilstanden slik at den går mot den avsluttende tilstanden.

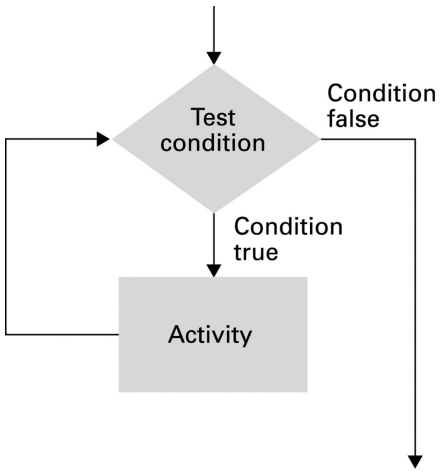
Kontrolleringen av en sløyfe består av tre aktiviteter (se fig. 5.7), hvor hver av disse må være tilstede for en vellykket sløyfe-kontroll. Test-delen har som oppgave å avslutte sløyfa, ved å se etter en betingelse som indikerer at en avslutning skal skje. Denne tilstanden kalles **terminating condition**. Det er på grunnlag av denne testaktiviteten vi setter betingelser inne i hver while-betingelse i pseudokoden. De andre to aktivitetene i sløyfe-strukturen forsikrer at prosessen vil bli avsluttet, før eller siden. Initialiserings-delen etablerer en startbetingelse, og forandrings-delen fører denne betingelsen mot den avsluttende betingelsen. Det er viktig at initialisering- og forandringsstegene må føre til en passende avsluttende tilstand.

Number ← 1;
while(Number!= 6) **do**
(Number ← Number + 2)

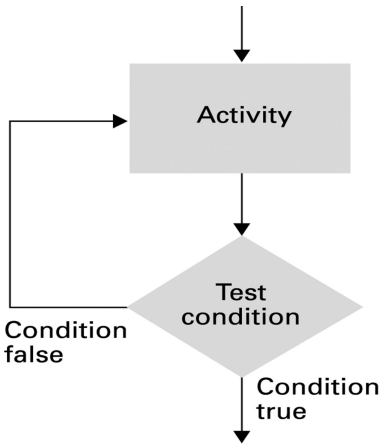
Denne prosessen vil aldri avsluttes fordi Number aldri vil kunne bli 6.

Rekkefølgen for hvordan stegene utføres kan ha konsekvenser. Det finnes to vanlige sløyfestrukturer, som er ganske forskjellige på dette punktet.

Figur 5.8 The while loop structure



Figur 5.9 The repeat loop structure



while (condidtion) **do** (activity)

repeat (activity) **until** (condition)

(Diamantformen indikerer en avgjørelse, et rektangel indikerer en påstand/ sekvens påstander.)

I figur 5.8 blir testen for å avgjøre om prosessen skal avsluttes (i while-løkken) utført før selve innholdet. I figur 5.9 derimot blir sløyfas innhold utført, før det testes om prosessen skal avsluttes. I dette tilfellet blir alltid sløyfas innhold utført, minst en gang, mens i while-strukturen kan det hende innholdet aldri blir utført. (while loop kan også refereres til som pretest loop, og repeat loop kan refereres til som posttest loop)

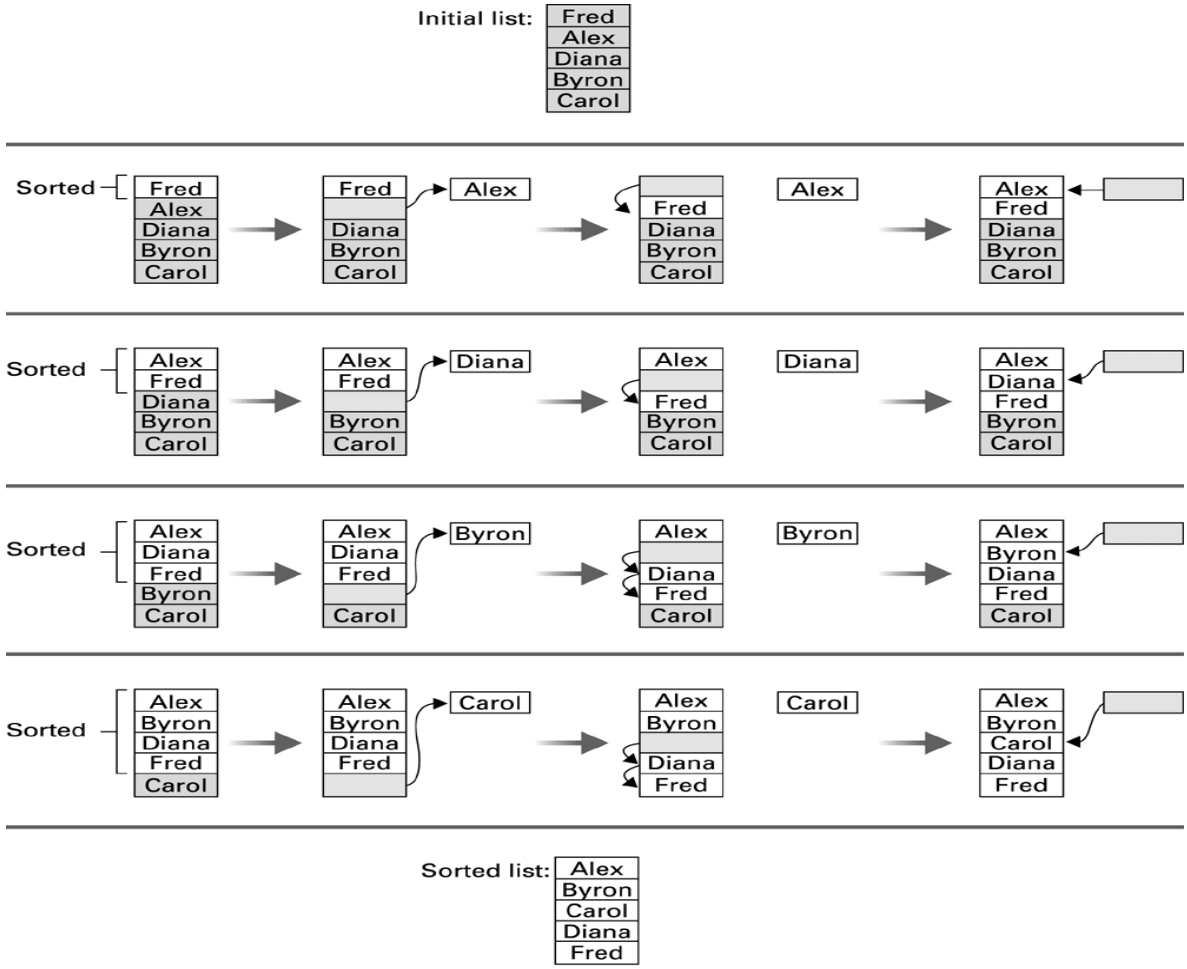
The Insertion Sort Algorithm

Som enda ett eksempel på bruk av gjentagende strukturer, skal vi se på det å sortere en liste med navn i alfabetisk rekkefølge. Listen skal sorteres innenfor seg selv, dvs. at vi skal kun omrokkere på navnene inne i selve lista. Her følger en pseudokode, eksemplifisert i figur 5.10:

Prosedyre Sorter (Liste)
N ← 2;
while (verdien til N ikke overstiger lengden av Liste) **do**
(Velg det n-te elementet i Liste som "vendeelement";
Flytt "vendeelementet" til en midlertidig plass, noe som skaper et hull i Liste;
while (det er et navn over hullet og det navnet er større enn "vendeelementet")
do (flytt navnet som står over hullet, inn i hullet,
slik at det skapes et hull over navnet)
Flytt "vendeelementet" inn i hullet Liste;
N ←N + 1;
)

Denne prosessen vil avsluttes når N overstiger lengden av lista.

Figut 5.10 Sortering av lista Fred, Alex, Diana, Byron og Carol alfabetisk

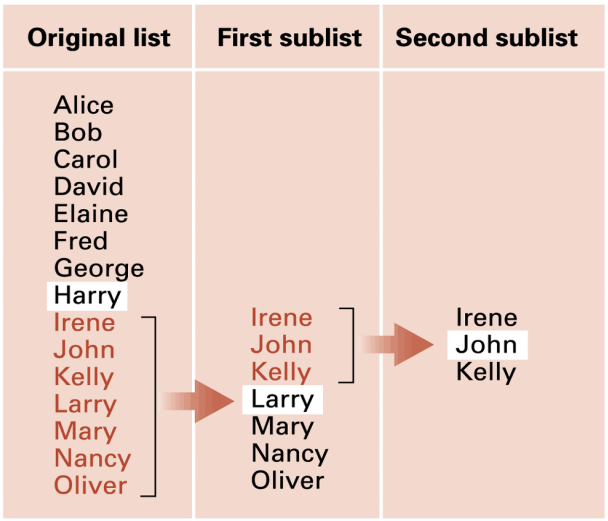


5.5 REKURSIVE STRUKTURER (GJENTAGENDE STRUKTURER)

Rekursive strukturer gir et alternativ til sløyfe-paradigmet når det gjelder å gjenta aktiviteter. Der hvor en sløyfe involverer repetisjon av et sett med instruksjoner, hvor disse instruksjonene først blir fullført og så repetert, involverer rekursjon at settet med instruksjoner blir repetert som en underoppgave i seg selv.

Binær søkealgoritme (Nb! Kan kun brukes på sorterte lister)
Det elementet det søkes etter settes eksempelvis som TargetValue (se figur 5.14), og er utgangspunktet for sammenligning med de andre elementene algoritmen støter på i lista, i sin jakt på det bestemte elementet. Den binære søkealgoritmen deler den gjeldende lista i to, sammenligner midtelementet med søkeelementet, helt til den kommer frem til det elementet den søker. Altså hvis vi ser på eksempelet i figur 5.12, hvor vi skal se om navnet John er i lista, deler algoritmen først lista i to. Der finner den navnet Harry, altså ikke John. Siden John er mindre enn Harry (bokstavkodemessig), kan vi begrense søkeområdet til navnene under Harry. Her deles lista igjen i to, og der finner vi navnet Larry. Larry er mindre enn John, og vi har begrenset søket til de tre navnene over Larry og under Harry. Når lista deles i to igjen er midtelementet John, det navnet vi søker etter, og søket erklæres en suksess. Dersom lista består av et partall elementer, og det da ikke finnes et midtelement, runder man konsekvent opp/ned. Hvis algoritmen har søkt gjennom hele lista uten å finne det elementet vi ser etter, erklæres søket en fiasko, og prosessen avsluttes. (Det er dele-på-to fremgangsmåten som har gitt søkealgoritmen navnen *binær*søk.)

Figur 5.12 Er John i lista?



Figur 5.14 Binær søkealgoritme i pseudokode

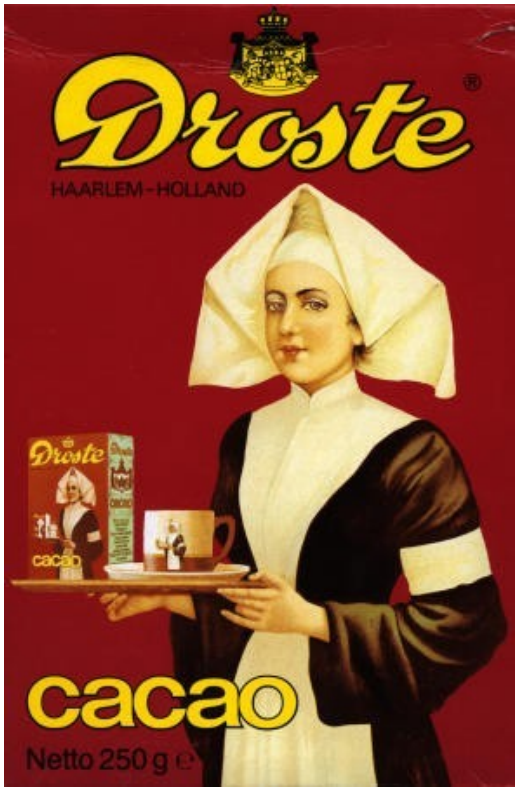
```
procedure Search (List, TargetValue)
if (List empty)
  then
     (Report that the search failed.)
  else
     [Select the "middle" entry in List to be the TestEntry;
      Execute the block of instructions below that is
          associated with the appropriate case.
             case 1: TargetValue = TestEntry
                      (Report that the search succeeded.)
             case 2: TargetValue < TestEntry
                      (Apply the procedure Search to see if TargetValue
                            is in the portion of the List preceding TestEntry,
                            and report the result of that search.)
             case 3: TargetValue > TestEntry
                     (Apply the procedure Search to see if TargetValue
                           is in the portion of List following TestEntry,
                           and report the result of that search.)
     ] end if
```

Rekursiv kontroll

Den binære søkealgoritmen ligner på den sekvensielle ved at hver algoritme ber om at en repeterende prosess skal utføres. Men selve utførelsen av denne repetisjonen er forskjellig. Der hvor den sekvensielle søkealgoritmen involverer en sirkellignende form for repetisjon, utfører den binære søkealgoritmen hvert stadium av repetisjonen som en deloppgave av det forrige stadiet. Denne teknikken, å gjenta seg selv, kalles rekursjon. Som vi har sett, illusjonen skapt av utførelsen av en rekursiv prosedyre er eksistensen til flere kopier av prosedyren, som hver enkelt kalles en aktivering av prosedyren. Disse aktiveringene skapes dynamisk i på et overlappende vis, og forsvinner etter hvert som algoritmen utføres. Av de aktivitetene som

eksisterer på et gitt tidspunkt, er det kun én som har aktiv fremgang. De andre aktivitetene er i en mellomfase, hvor de venter på at en annen aktivitet skal avsluttes før de kan fortsette. Figur *** illustrerer et eksempel på (visuell) rekursjon, hvor damen på pl

Figur * Visuell**



bilde som den opprinnelige plakaten osv.

Ettersom rekursive systemer er gjentagende prosess, er de avhengige av en kontrollstruktur. Akkurat som sløyfe-kontroll, er de rekursive systemene avhengige av at det blir sjekket for en avsluttende tilstand, og et design som sørger for at denne tilstanden vil forekomme. Med andre ord de samme kravene – initialisering, forandring, testing – som for sløyfe-kontroll.

Generelt sett er en rekursiv prosedyre designet for å teste den avsluttende tilstanden (ofte kalt *base case* eller *degenerative case*), før den ber om flere aktiveringer. Hvis den avsluttende tilstanden ikke inntreffer, lager rutinen en annen aktivisering av prosedyren og ber den om å løse et modifisert problem som er nærmere den avsluttende tilstanden, enn det problemet som opprinnelig var tildelt den gjeldende aktiveringen. Dersom den avsluttende tilstanden så ikke inntreffer, vil den gjeldende aktiveringen avslutte uten å lage nye aktiveringer.

Vi ser på initialiserings- og forandringsfasen i binærsøket i figur 5.14. Her stanses produksjonen av flere aktiveringer så fort søkeelementet (TargetValue) er funnet, eller alle elementene i lista er undersøkt. Prosessen initialiseres ved at den gis en konkret liste og søkeelement å gå ut i fra. Så endrer prosedyren sin opprinnelige oppgave til å søke i en mindre liste. Siden den opprinnelige lista har en bestemt lengde og hver endring i forandrings-steg reduserer lista det søkes i, er vi garantert at søkeelementet til slutt blir funnet, eller at prosedyren søker i en tom liste og dermed avsluttes. Denne repeterende prosessen vil garantert stanse.

(Både sløyfe- og rekursive kontrollstrukturer kan brukes til å repetere et sett med instruksjoner, men er de like gode? Kan en algoritme basert på en sløyfe-kontrollstruktur og en basert på rekursiv kontrollstruktur løse de samme oppgavene? Dette er viktige spørsmål innen datavitenskapen, siden svarene vil fortelle oss hvordan vi skap bygge opp programmeringsspråk på en slik måte at vi kan oppnå det beste programmeringssystemet som er mulig.)

5.6 EFFEKTIVITET OG KORREKTHET

Algoritmeeffektivitet

Til tross for at dagens datamaskiner kan utføre millioner av instruksjoner hvert sekund, forblir effektivitet et stort problem når det gjelder utforming av algoritmer. Det er ofte valget mellom

effektive og ueffektive algoritmer som utgjør forskjellen mellom en praktisk løsning på et problem og en upraktisk.

La oss betrakte en sekretær som skal hente og oppdatere studentmapper, fra et arkiv på over 30 000 studenter, hvor disse mappene er sortert etter studentnummer. Det finnes to algoritmer for å søke i en slik liste; sekvensielt søk og binærsøk. Vil valget av algoritme ha noe å si for sekretæren?
Dersom man foretar et sekvensielt søk vil algoritmen starte på begynnelsen av listen, og sammenligne de elementene den finner med det ønskede studentnummeret. Gjennomsnittssøket vil måtte foreta ca 15 000 sammenligninger, og det vil ta omkring 2.5 minutter (dersom det å hente og sjekke hver mappe tok 10 millisekunder).

Dersom man foretar et binærsøk vil algoritmen sammenligne målet for søket med det midterste elementet i lista. Hvis dette ikke er det ønskede elementet, gjenstår kun en halvdel av lista, avhengig om søkeelementet er større eller mindre en midtelementet. Slik fortsetter det, til søkeelementet er funnet. (NB! Dersom lista har et partall elementer, og det ikke er noe midtelement, runder man enten opp eller ned. Da er det viktig å være konsekvent.) Så etter å ha sjekket midtelementet i lista på 30 000 elementer, halveres den til 15 000, så til 7500, 3750 osv. Så i det verst tenkelige tilfellet må vi kun utføre 15 sammenligninger, i en liste med hele 30 000 elementer, noe som kun vil ta 0.15 sekunder. Altså kan vi konkludere med at valg av søkealgoritme har en innvirkning.

Dette eksempelet viser hvor viktig algoritmeanalyse er. (Algoritmeanalyse omfatter læren om tid og lagringsplass, altså de ressursene algoritmer krever.) Algoritmeanalyse tar ofte for seg beste-tilfelle, verste-tilfelle og gjennomsnittlig-tilfelle scenarioer. I eksempelet overfor foretok vi gjennomsnittlig-tilfelle og verste-tilfelle analyse av en konkret liste, men når man evaluerer søkealgoritmer generelt sett, forsøker man heller å finne en formel for algoritmen effektivitet i en tilfeldig liste. Så en liste med n elementer vil i gjennomsnittet måtte foreta n/2 sammenligninger, mens binærsøkalgoritmen kun vil måtte foreta $lg_2(n)$ sammenligninger i verste tilfelle.

Insertion sort

Hvis vi betrakter insertion sort algoritmen (figur 5.11), ser vi at i beste tilfelle vil en liste med n elementer kreve n-1 sammenligninger. (Algoritmen velger et element i lista (pivot element) og sammenligner det med de tidligere elementene, helt til den riktige plassen til dette pivot elementet er funnet. Andre elementet sammenlignes med et element, tredje elementet sammenlignes med et element osv.)

I verste tilfelle må hvert pivot element sammenlignes alle de tidligere elementene, før dets riktige plass blir funnet. Dette forekommer hvis lista er sortert baklengs. Da vil det første pivot elementet (altså det andre elementet i lista) sammenlignes med et element, det andre pivot elementet (tredje elementet i lista) sammenlignes med to elementer osv. Det totale antall sammenligninger i verste tilfelle vil da bli $(1/2)(n^2-n)$.

Gjennomsnittlig vil insertion sort algoritmen kreve $(1/4)(n^2-n)$, i en liste med n elementer. Pivot elementet vil bli sammenlignet med halvparten av elementene før det.

Disse resultatene forteller oss antall sammenligninger som blir gjort i løpet av insertion sort algoritmens kjøretid, som igjen kan fortelle oss hvor lang tid det kreves for å utføre

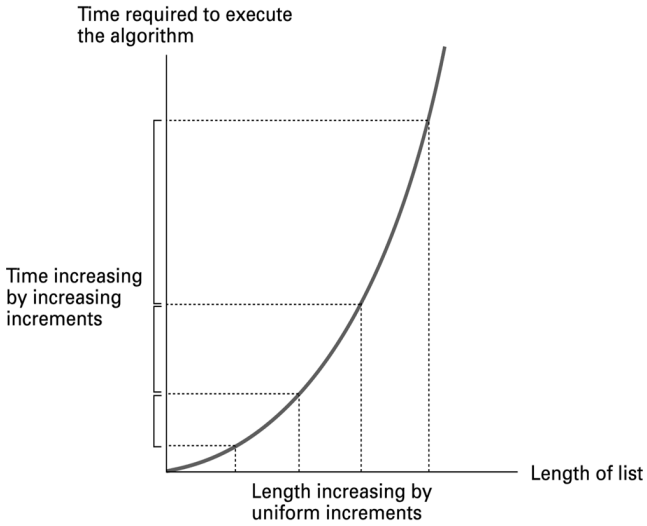
algoritmen. I figur 5.19 ser vi en graf som indikerer hvordan tiden som kreves for å utføre insertion sort-algoritmen øker, samtidig som lengden på lista øker. Denne grafen baserer seg på det verst-tenkelige tilfellet, hvor det å sortere en liste med n elementer kan kreve opptil $(1/2)(n^2-n)$ sammenligninger. Som vi ser av grafen viser det seg at denne algoritmen blir mindre effektiv når lengden på listen øker.
Figur 5.20 viser grafen for det verst tenkelige tilfellet for den binære søkealgoritmen. Som vi ser øker den binære søkealgoritmens effektivitet når lengden på listen øker.

Grafens form, som er fremstilt ved å sammenligne tiden det kreves for en algoritme å utføre oppgaven sin med størrelsen på inputen, reflekterer effektiviteten til en algoritme. Det er vanlig å klassifisere algoritmer ut i fra formene til disse grafene, vanligvis basert på algoritmens verste tilfelle analyse.

Skrivemåten som brukes for å identifisere disse "effektivitets-klassene" kalles noen ganger stor-theta.notasjon. Alle algoritmer som har grafer formet som en parabel, som for eksempel insertion sort, tilhører klassen $\Theta(n^2)$; alle algoritmer som har grafer formet som logaritmiske uttrykk, som for eksempel binærsøk, tilhører klassen $\Theta(\lg_2 n)$. Ved å ha kjennskap til hvilken klasse en bestemt algoritme tilhører kan vi forutsi hvordan den vil opptre, og sammenligne den med andre algoritmer som løser det samme problemet. To algoritmer i $\Theta(n^2)$ vil fremstille lignende forandringer i forhold til mengde tid som kreves, når størrelsen på inputen øker. Med andre ord, tiden som kreves av en algoritme i $\Theta(\lg_2 n)$ vil ikke øke like hurtig som den vil for en algoritme i $\Theta(n^2)$.

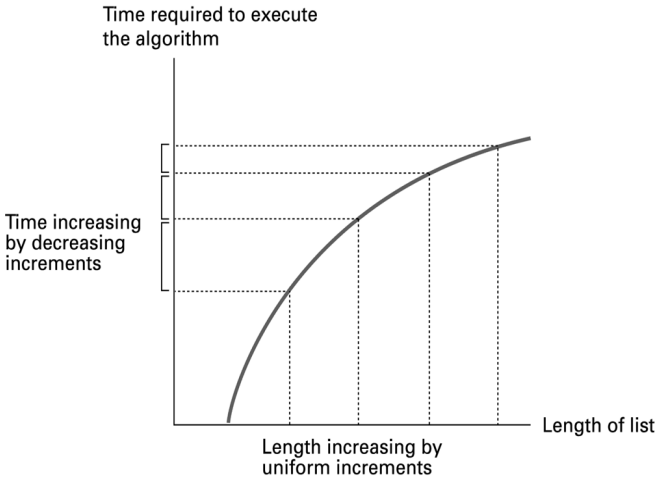
Figur 5.19

Worst case analysis of the insertion sort algorithm



Figur 5.20

Worst case analysis of the binary search algorithm



Software Verification

Vi går tilbake til Polyas fjerde fase i hans grunnleggende, uformelle faser for problemløsing:

> *Evaluer løsningens med tanke på nøyaktighet, og om den har potensial til å brukes til å løse andre problemer.*

Betydningen av denne fasen kan eksemplifiseres av det følgende eksempelet:

En reisende har en kjetting bestående av 7 ringer. Han må bo på et hotell i 7 netter, hvor prisen for et rom i en natt er en ring fra kjettingen. Hva er det færreste antall ringer som må kuttes av for at han skal kunne betale leien, en ring av gangen, uten å betale på forhånd?

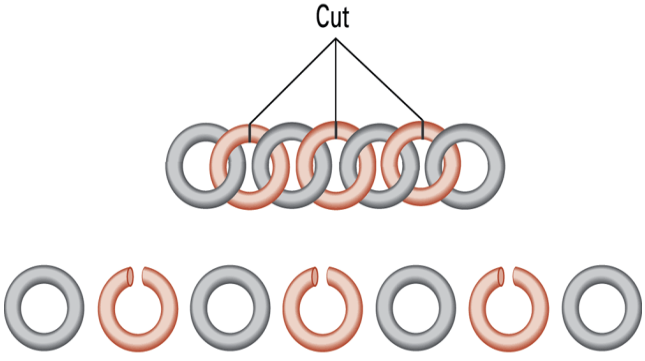
For å løse dette problemet må vi innse at ikke hver enkelt ring i kjettingen må kuttes. Dersom vi bare kutter den andre ringen, blir både den første og tredje ringen separert fra de andre 5. Ved å følge denne innsikten ser vi at løsningen blir å kutte den andre, fjerde og sjette ringen. Se figur 5.21 på neste side.

Dersom vi revurderer problemet ser vi at når kun den tredje ringen i kjettingen er kuttet av, sitter vi igjen med tre mindre kjettinger, med lengde en, to og fire. Med disse kan vi gjøre som følger:
Dag 1: Gi hotellet den ene ringen.
Dag 2: Få tilbake ringen som ble gitt dagen før, og gi hotellet kjettingen bestående av to ringer.
Dag 3: Gi hotellet den ene ringen.
Dag 4: Få tilbake de tre ringene som ble gitt tidligere, og gi hotellet kjettingen bestående av fire ringer.
Dag 5: Gi hotellet den ene ringen.
Dag 6: Få tilbake den ene ringen som ble gitt tidligere, og gi hotellet kjettingen bestående av to ringer.
Dag 7: Gi hotellet den ene ringen.

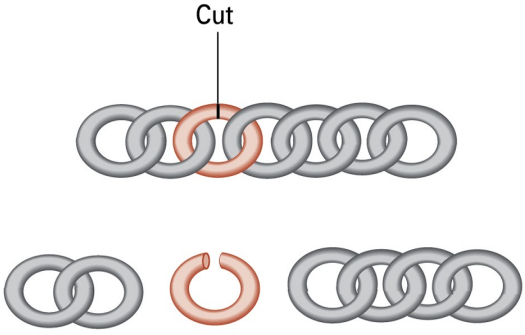
Det første svaret(3 kutt) viser seg å være galt, men hvordan kan vi være sikre på at den nye løsningen er korrekt? Jo, siden hotellet må få en enkelt ring den første dagen, må minst en ring kuttes av. Siden vårt nye svar kun krever et kutt må dette være den optimale løsningen. Se figur 5.22 på neste side.

I programmeringsmiljøet understreker dette eksempelet forskjellen mellom et program som man tror er korrekt og et program som er korrekt. Det finnes mange skrekkeksempler hvor software man *trodde* var korrekt, har feilet på kritiske tidspunkt. Verifisering av software er av den grunn en viktig oppgave, og jakten på effektive verifiseringsteknikker er et aktivt forskningsområde innen datateknikk.

Figur 5.21 Separerer kjettingen med tre kutt



Figur 5.22 Separerer kjettingen med et kutt,



Et stort forskningsområde innen verifisering av software forsøker å bruke formelle, logiske teknikker for å påvise korrektheten til et program. Målet er å bruke ren logikk for å bevise at

algoritmen som representeres av et program gjør det den skal gjøre. Ved å redusere verifiseringsprosessen til en formell prosedyre, slipper man unøyaktige konklusjoner, slik som i eksempelet med kjettingen.

Vi ser nærmere på verifisering av software. På samme måte som et formelt matematisk bevis baserer seg på aksiomer (aksiom = grunnsetning som aksepteres uten bevis, er ofte selvinnlysende), baserer et formet bevis på et programs korrekthet seg på de spesifikasjonene programmet er designet ut i fra. Altså, for å bevise at et program kan sortere en liste med navn riktig, kan vi begynne med å anta at programmets input er en liste med navn. Med andre ord, et bevis på korrekthet begynner med å anta at visse forutsetninger er sanne når programmet starter.

Det neste steget i å bevise korrektheten til et program er å se hvordan konsekvensene av disse forutsetningene sprer seg i programmet. Forskere har analysert ulike programstrukturer for å fastslå hvordan et utsagn, som er sant før programmet blir utført, blir påvirket av utførelsen av programstrukturen. For eksempel, hvis en påstand er sagt om verdien til Y før programmet utføres:

$X \leftarrow Y$

kan den samme påstanden sies om X etter instruksjonen har blitt utført.

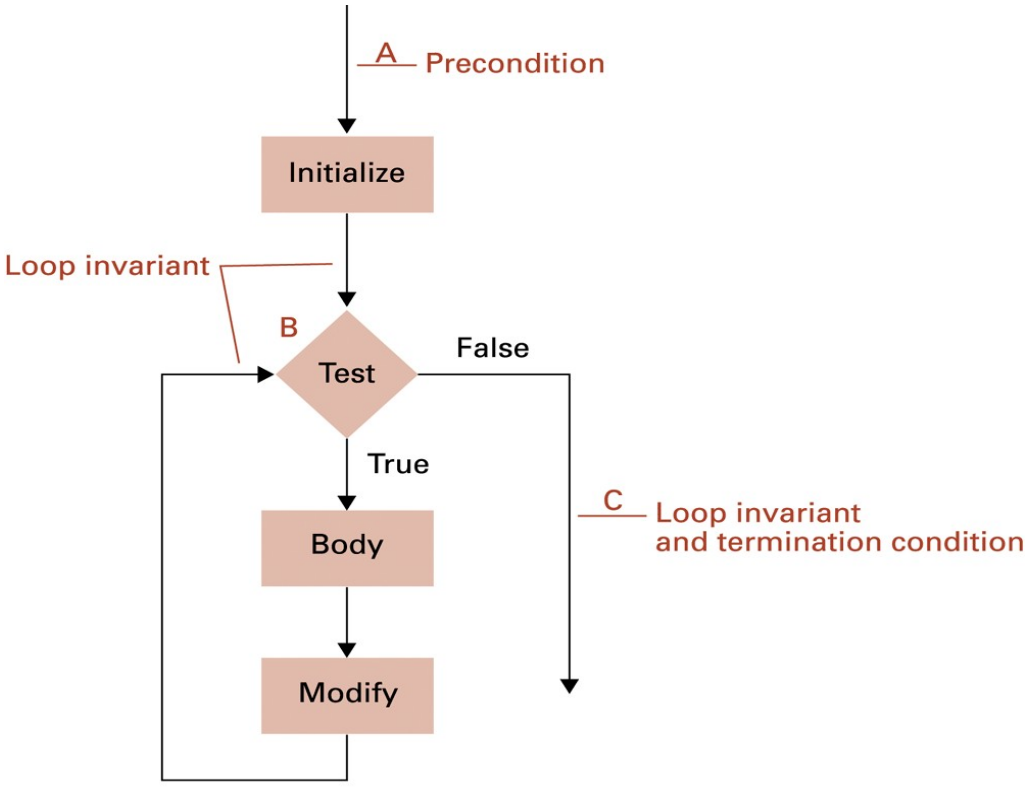
Bevis på korrekthet kan finnes ved å identifisere påstander, kalt *assertions, deklareringer*, som kan etableres i ulike deler av programmet. Resultatet er en samling *deklareringer,* hvor hver enkelt er en konsekvens av programmets forutsetninger og sekvensen av instruksjonene som fører til den delen av programmet hvor deklareringen blir etablert. Hvis deklareringen som etableres på slutten av programmet korresponderer med den ønskede outputen (kalt postcondidtions), kan vi konkludere med at programmet er korrekt.

Som et eksempel kan vi se på en typisk while-sløyfe (se figur 5.23). Gå ut ifra at som en konsekvens av forutsetningen som er gitt ved punkt A, kan vi fastslå at en bestemt *deklarering* er sann, hver gang test for avsluttende tilstand blir gjort (punkt B) i løpet av den repeterende prosessen. (Et deklareringspunkt i en sløyfe som er sann hver gang det punktet i sløyfa nås, kalles en *loop invariant.*) Så hvis denne prosessen noen sinne avsluttes, forflyttes utførelsen av algoritmen til punkt C, hvor vi kan konkludere med at både *loop invariant* og den avsluttende betingelsen fremdeles gjelder.(*Loop invariant* er fremdeles gjeldende siden test for avslutning ikke endrer noen verdier i programmet, og den avsluttende betingelsen gjelder fremdeles siden uten den vil ikke programmet avsluttes.) Hvis disse to utsagnene indikerer de ønskede postcondidtions, kan beviset på korrekthet fullføres ved å vise at initialisering- og forandringskomponentene i sløyfa til slutt vil føre til den avsluttende betingelsen/tilstanden.

Det blir stadig gjort fremskritt innen utviklingen av software verification-teknikker, selv om det er svært utfordrende. Programmeringsspråket SPARK , som ligner på Ada, tillater programmet uttrykt i pseudokode, gir programmerere muligheten til å inkludere deklareringer som for eksempel *preconditions*, *postcondidtions* og *loop invariants*. Så et program som er skrevet i SPARK inneholder ikke bare den algoritmen som skal utføres, men også den informasjonen som trengs for å sjekke korrektheten til programmet. Til tross for at SPARK

har hatt suksess, blir mesteparten av dagens software ”verifisert” ved testing, noe som ikke alltid er pålitelig, da det kan være små detaljfeil som ikke slår ut i testtilstandene, men på et annet, kritisk tidspunkt.

Figur 5.23



Kapittel 9 - Databasesystemer

9.1 DATABASENS GRUNNPRINSIPPER

Database: flerdimensjonal samling av data.

Databaseskjema: en beskrivelse av hele databasestrukturen som brukes av programmet til å opprettholde databasen.

Underskjema: beskrivelse av en dal av databasen. Viktig for tildeling av innsyn i et system. (eks. Studenten kan ikke se hva læreren tjener)

En databaseapplikasjon består av flere programlag. Manipulasjon foregår ved hjelp av database management system (DBMS). Når applikasjonsprogrammet har fastslått handlingen en bruker vil utføre, tas DBMS i bruk, som et abstrakt verktøy, til å ta vare på resultatene. (eks. DBMS går inn i en database og henter ut informasjon)

Distribuert database: en database spredd mellom forskjellige maskiner i et nettverk. Kan føre til problemer i forbindelse med kjøretid.

Fordeler med å skille mellom programvare og DBMS:

- Applikasjonsprogrammet trenger ikke ta hensyn til hvor informasjonen befinner seg.
- DBMS har tilgang til all informasjon, men kan forlange at applikasjonsprogrammet til de forskjellige brukerne har ulike restriksjoner.
- Gjør det mulig å endre organiseringen av databasen, uten å endre applikasjonsprogrammet.

Databasemodell: begrepsmessig beskrivelse av en database.

9.2 RELASJONSMODELL

Relasjoner: rektangulære tabeller som inneholder informasjonen i en database.

Ansatt nr.	**Navn**	**Adresse**
1	Jonas Engeseter	Osloveien 13
2	Hilde Mortensen	Heieveien 4
3	Truls Nilsen	Innegata 34

Tuple: hver rad i relasjonen. I dette tilfellet som inneholder informasjon om den ansatte.

Attributter: hver kolonne i relasjonen. Et attributt sier noe om entiteten/den ansatte. De har alle en datatype (tekst, heltall og lignende).

Det finner tilgjengelige database systemer til PC-er. Et eksempel er Microsoft Access.

Dersom vi nå ønsker å legge til informasjon om hva slags jobber de ansatte har, må vi ta hensyn til dobbelt lagring. Dersom en ansatt har hatt flere forskjellige stillinger i et selskap, vil informasjonen om det ansatte gjentas flere ganger. Dette skjer også dersom flere ansatte har samme stilling. Informasjon om stillingen vil da bli gjentatt flere ganger.

Ansatt nr.	Navn	Adresse	Jobb id	Tittel	Avdeling
1	Jonas Engeseter	Osloveien 13	F2	Leder	Salg
2	Hilde Mortensen	Heieveien 4	S5	Sekretær	Salg
3	Truls Nilsen	Innegata 34	F2	Leder	Salg
1	Jonas Engeseter	Osloveien 13	L8	Sjef	Salg

Vi får også et problem dersom vi ønsker å slette en ansatt. Dersom Hilde Mortensen slutter, og vi sletter denne posten vil også stilling S5 gå tapt, da hun er den eneste med denne stillingen. Problemet løses ved at vi deler inn i flere relasjoner, en for ansatt og en for jobb. Disse kan knyttes sammen til en oppdragsrelasjon.

Ansatt nr.	Navn	Adresse
1	Jonas Engeseter	Osloveien 13
2	Hilde Mortensen	Heieveien 4
3	Truls Nilsen	Innegata 34

Jobb id	Tittel	Avdeling
F2	Leder	Salg
S5	Sekretær	Salg
L8	Sjef	Salg

Ansatt nr	Jobb id	Start dato	Sluttdato
1	F2	03.02.99	4.30.06
2	S5	10.01.07	
3	F2	03.09.01	
1	L8	05.01.06	

Noen ganger kan informasjon gå tapt når vi deler inn i mindre databaser. Vi skiller mellom lossless decomposition og nonloss decomposition.

Primærnøkkel: en eller flere felt som er unike for hver rad i tabellen. Disse er understreket.

Fremmednøkkel: dersom vi vil knytte sammen flere tabeller benytter vi oss av fremmednøkler. For eksempel dersom vi ønsker og vise informasjon fra både ansatt-tabellen og jobb-tabellen, benytter vi jobbId som en fremmednøkkel i den nye tabellen.

DATABASE OPPERASJONER

Vi kan hente ut gitt data fra en tuple og plassere disse i en ny relasjon.

NEW ← SELECT from ANSATT where Ansattnr =”3”

Vi kan hente ut kolonner og plassere disse I en ny relasjon.

NEW1 ← PROJECT Tittel from JOBB

Vi kan slå sammen to tabeller til en.

NEW2 ← JOIN OPPDRAG and JOBB where OPPDRAG.JobbId = JOBB.JobbId

SQL (Structured Query Language)

Dagens databasesystemer inneholder ikke nødvendigvis rutiner som JOIN, SELECT og PROJECT. Isteden tilbyr de rutiner som kan være kombinasjoner av disse. Et eksempel er SQL, som er det underliggende språket i de fleste relasjonsdatabaser.

Eksempel på bruk

Select Ansattnr, Avdeling

From OPPDRAG, JOBB

Where OPPDRAG.JobbId = JOBB.JobbId

And OPPDRAG.Sluttdato = ” ”

Vi henter her ansattnr og avdeling fra tabellene oppdrag og jobb, der jobbId i oppdrag er lik jobbId i jobb-tabellen og Sluttdatoen i oppdrag er tom.

MySQL: relasjonsdatabasesystem.