Programmering

Innholdsfortegnelse

Oppdatert november 2022

Innledning

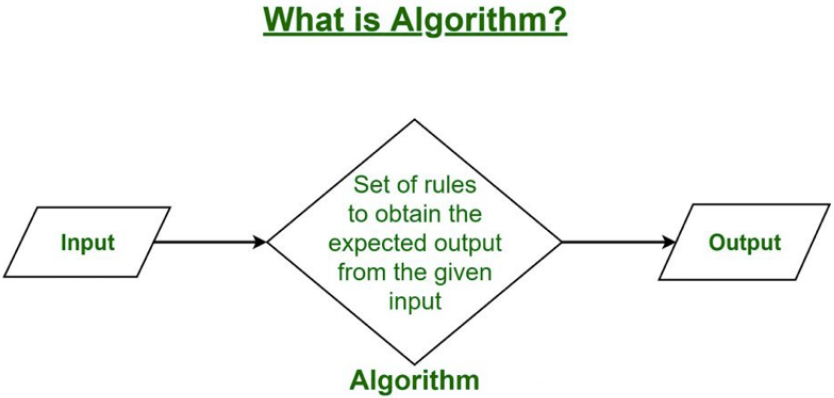
Programmering for trinn 8 til 13

Denne introduksjonen i programmering, er tenkt som grunnlag for å kunne bygge videre med egne oppgaver og prosjekter. Innholdet er tilrettelagt for **Matematikk og Naturfag** og lagt opp så praktisk som mulig. Tekst, øvelser, oppgaver og illustrasjoner er hentet fra en rekke ulike sammarbeidsorganisasjoner og gjennomgangen er i sin helhet bygget opp og strukturert av lærer i den videregående skolen. Det er både spennende å utfordrende å få kunnskap om ny teknologi, så denne kursrekken innenholder både introduksjoner, gjennomgang, oppgaver, fasit og prøv selv oppgaver.

For å gjennomføre alle øvelsene er det lagt vekt på at man ikke trenger store investeringer, men heller målrettet arbeider for et godt læringsutbytte. I faget Naturfag benytter vi et startsett for Micro:Bit eller Arduino for å konkretisere øvelsene mest mulig. Nødvendige enheter oppgis i øvelsene og kan bestilles fra flere leverandører.

Kursmål DEL 1A

- Innføring i Algoritmer
- Algoritmisk tenkning



Algoritmer

Denne delen vil ta for seg grunnleggende områder og begreper som er viktige for å forstå programmering bedre. Vi kommer ikke til å gå dypt i hvert emne, men har du ønske om å lære deg mer, så er det mulig gjennom tilvalg og fordypningsemner som vil legges ut senere på denne hjemmesiden.

Uttrykket Algoritme kommer fra den persiske matematikeren og astronomen Mohamed al-Khwarizmi - som skrev flere bøker, blant annet en som inneholder en oppskrift (en algoritme) for hvordan man løser annengradsligninger.

En algoritme er en **presis beskrivelse** av en serie operasjoner som skal utføres for å løse et problem eller et sett med flere problemer. Utfordringen med algoritmer er at de må skrives og følges nøyaktig. De må være i riktig rekkefølge for at vi skal få det resultatet vi forventer. Å skrive i riktig rekkefølge er derfor helt essensielt, da en datamaskin kun følger de instruksjoner som blir gitt i programmet.

Et eksempel på variasjon i Algoritme er hvordan vi i dag, i mange butikker, kan betjener oss selv og kan i tillegg betale i en kasse. Vi finner varene og går så til en selvbetjeningskasse hvor vi scanner varer og betaler, i stedet for at vi går til en kasse hvor det sitter et menneske og scanner varene for oss.

Her kunne man si at overordnet er algoritmen i seg selv lik, men det er hvem som utfører de forskjellige elementene som vil bli forskjellen. Her kunne lagt inn et flytskjema eller en pseudokode for å illustrere det to hendelsene sammen, basert på valg man må gjøre. Denne måten å uttrykke løsninger på kommer vi tilbake til.

Som et eksempel på at ting må skje i riktig rekkefølge kan vi tenke oss det å handle en vare i en butikk. I grove trekk vil det foregå slik:

- Du går inn i butikken
- Finner varen
- Går til selv-skanning disken
- Skanner varene
- Betaler
- Går ut av butikken

Her forstår man at prosessen må følges. Gjør man det i en annen rekkefølge får man ikke samme ønsket resultat. Slik er det også med programmering.

Hva er egenskapene til en algoritme?

Ikke alle instruksjoner, slik som bakeoppskrifter og liknende er satt opp i henhold til en algoritme. For at instruksjoner skal være en algoritme, må den ha følgende egenskaper:

1. **Klar og entydig**: Algoritmen skal være klar og entydig. Hvert av trinnene skal være tydelige i alle aspekter og må bare føre til en løsning.
2. **Veldefinerte inndata**: Hvis en algoritme skal hente signaler (informasjon)fra innganger, bør det være veldefinerte innganger.
3. **Veldefinerte utganger:** Algoritmen må tydelig definere hvilken utgang som skal benyttes, og være veldefinert på hva som skal ut.
4. **Endelighet:** Algoritmen må være endelig, det vil si at den ikke skal ende opp i uendelige løkker eller lignende.
5. **Mulig:** Algoritmen må være enkel, generisk(ensartet) og praktisk, slik at den kan utføres med de tilgjengelige ressursene. Det må ikke inneholde noen tenkt eller visjonært.
6. **Uavhengig av språk:** Algoritmen som er designet må være språkuavhengig, det vil si at det bare må være enkle instruksjoner som kan implementeres på et hvilket som helst språk, og likevel vil utgangen være den samme, slik som man forventet.

Fordeler med algoritmer:

1. De er normalt lette å forstå.
2. Algoritme er en trinnvis presentasjon av en løsning på et gitt problem.
3. I Algoritmen er problemet delt inn i mindre stykker eller trinn, derfor er det lettere for programmereren å konvertere det til et faktisk program.

Ulemper ved algoritmer:

1. Det tar lang tid å skrive en algoritme, så det er tidkrevende.
2. Forgrenings- og løkkesetninger er vanskelige å vise i algoritmer.

{ Når den kjører
< instruksjon; gå fremover>
} while (< betingelse: så lenge det er sti/veg>);
else (< betingelse: snu mot høyre x grader>);

Hvordan designe en algoritme?

For å skrive en algoritme, er følgende en forutsetning:

1. **Kjenne problemet** som skal løses av denne algoritmen.
2. **Begrensningene** i problemet som må vurderes mens du løser problemet.
3. **Innspillene** som må hentes inn for å løse problemet.
4. **Utdataene** som forventes når man har løst problemet.
5. **Løsningen på** dette problemet, med de angitte begrensningene.

   Deretter er algoritmen skrevet ved hjelp av disse parametere, slik at den løser problemet.

Eksempel: Tenk at du skal legge sammen tre tall og skrive ut summen.

1. **Trinn 1: Oppfylle forutsetningene**
   Som beskrevet ovenfor, for å skrive en algoritme, må noen forutsetninger oppfylles.
   a. **Problemet som skal løses av denne algoritmen**: Legg til 3 tall og skriv ut summen.
   b. **Begrensningene i problemet som må vurderes under løsning av problemet**: Tallene må bare inneholde sifre og ingen andre tegn, skal det bare være heltall eller skal de ha desimaltall også?
   c. **Innspillene som skal til for å løse problemet:** De tre tallene som skal legges inn.
   d. **Utdataene som forventes når problemet er løst:** Summen av de tre tallene som er lagt inn som inndata.
   e. **Løsningen på dette problemet, i de gitte begrensningene:** Løsningen består av å legge til de 3 tallene. Det kan gjøres ved hjelp av en hvilken som helst ønsked metode.

2. **Trinn 2: Designe**

   la oss designe algoritmen ved hjelp av de ovennevnte forutsetninger

   Algoritme for å legge til 3 tall og skrive ut summen:

   a. START
   b. Deklarer 3 heltallsvariabler num1, num2 og num3.
   c. Ta de tre tallene som skal legges sammen, som inndata i henholdsvis variabler num1, num2 og num3.
   d. Deklarer en heltallsvariabelsum for å lagre den resulterende summen av de tre tallene.
   e. Legg til de tre tallene, og lagre resultatet i variabelsummen.
   f. Skriv ut verdien av variabel sum
   g. SLUTT

3. **Trinn 3: Teste algoritmen ved å implementere den.**

   Inorder for å teste algoritmen, la oss implementere den i Python.

Satt opp i språket Python


```
if __name__ == "__main__":

    # Variablene for å hente de 3 tallene
    num1 = num2 = num3 = 0

    # Variablen for å lagre sum
    sum = 0

    # Hent inn de tre tallene fra bruker
    num1 = int(input("Legg Inn 1 nummer: "))

    num2 = int(input("Legg Inn 2 nummer: "))

    num3 = int(input("Legg Inn 3 nummer: "))

    # Kalkuler sum ved hjelp av + operator og lagre den i variabelen sum
    sum = num1 + num2 + num3

    # Skriv ut summen
    print("\nSum av de tre nummerne er:", sum)
```

- **Priori Analyse:** "Priori" betyr "før".
  Priori-analyse betyr å sjekke algoritmen før implementeringen. Kontroll av algoritmen når den er skrevet i teoretiske trinn. Effektiviteten til en algoritme måles ved å anta at alle andre faktorer, for eksempel prosessorhastighet, er konstante og ikke har noen effekt på implementeringen. Dette gjøres vanligvis av algoritmedesigneren. Det er i dette trinnet at algoritmekompleksiteten bestemmes.

- **Bakre analyse:** "Posterior" betyr "etter".
  En posterioranalyse betyr å sjekke algoritmen etter implementeringen. Her kontrolleres algoritmen ved å implementere den i et hvilket som helst programmeringsspråk og utføre oppgaven. Denne analysen bidrar til å få den faktiske og reelle analyserapporten om korrekthet, nødvendig plass, forbruk av tid med mer.

    - **Tidsfaktor**: Tiden måles ved å telle antall nøkkeloperasjoner, for eksempel sammenligninger i sorteringsalgoritmen.
    - **Plassfaktor**: Plass måles ved å telle den maksimale minneplassen som kreves av algoritmen.

- **Plasskompleksitet:** Plasskompleksiteten til en algoritme refererer til mengden minne som denne algoritmen krever for å utføre og få resultatet. Dette kan være for innganger, midlertidige operasjoner eller utdata.

Hvordan beregne plasskompleksitet?
Plasskompleksiteten til en algoritme beregnes ved å bestemme følgende 2 komponenter:

- **Fast del:** Dette refererer til plassen som definitivt kreves av algoritmen. For eksempel inndatavariabler, utdatavariabler, programstørrelse osv.

- **Variabel del:** Dette refererer til plassen som kan være forskjellig basert på implementeringen av algoritmen. For eksempel midlertidige variabler, dynamisk minnetildeling, rekursjonsstakkplass osv.

Tidskompleksitet: Tidskompleksiteten til en algoritme refererer til tiden algoritmen krever for å utføre og få resultatet. Dette kan være for normale operasjoner, betingede if-else-setninger, løkkesetninger osv.

Hvordan beregne tidskompleksitet?
Tidskompleksiteten til en algoritme beregnes også ved å bestemme følgende 2 komponenter:

- **Konstant tidsdel:** Enhver instruksjon som utføres bare en gang, kommer i denne delen. For eksempel inngang, utgang, hvis-ellers, bryter, etc.
- **Variabel tidsdel:** Enhver instruksjon som utføres mer enn en gang kommer i denne delen. For eksempel løkker, rekursjon, etc.

Pseudokode

En pseudokode er en måte å beskrive en algoritme på. Pseudokoden er skrevet etter programmeringsspråkets struktur, men for å kunne leses av mennesker ikke maskiner. Pseudokoden skrives normalt før algoritmen og er en del av planleggingen. Det er ingen spesielle regler for hvordan en Pseudokode utformes, men det er vanlig å bruke innrykk for å vise gjentagelser og beslutninger.

Eksempel hentet fra snl.no

La oss si at vi har et register med alle lønnsmottagere i Norge og ønsker å beskrive en metode som genererer to lister med henholdsvis høyt- og lavtlønnede arbeidstakere. Denne metoden kan beskrives i pseudokode som vist her.

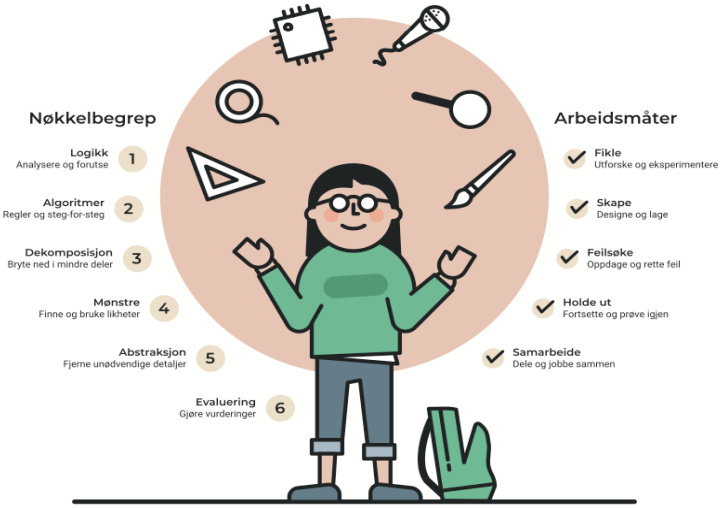
```
Tøm listen med lavtlønnede
Tøm listen med høytlønnede
For hver lønnsmottager:
        Hvis personen tjener under 200.000:
                Legg inn i listen over lavtlønnede
        Hvis personen tjener over 500.000:
                Legg inn i listen over høytlønnede
```

Algoritmisk tankegang

"Algoritmisk tankegang" kommer fra det engelske “computational thinking skills” som enkelt sagt vil si "å tenke som en informatiker".

Algoritmisk tankegang innebærer at eleven lærer å tilnærme seg problemer på en systematisk måte og foreslå løsninger som man kan bruke datamaskiner til å løse (hele eller deler av) dem. Det vil si:

- å tenke på hvilke steg som skal til for å løse et problem.
- å bruke sin teknologiske kompetanse for å få datamaskinen til å løse problemet.



Den algoritmiske tenkeren

Algoritmisk tankegang er ikke det samme som en algoritme. En algoritme beskriver en serie handlinger nøyaktig (oppgaver, operasjoner) som skal til for å løse et problem eller et sett av problemer. Algoritmisk tankegang er en problemløsningsprosess.

En problemløsningsprosess

Algoritmisk tankegang er en problemløsningsprosess som inkluderer å organisere og analysere data på en logisk måte og å lage fremgangsmåter for å løse komplekse problem. Algoritmisk tankegang innebærer å bryte ned store, komplekse problemer til mindre, mer håndterlige del-problemer. Det handler også om å lage abstraksjoner og modeller av den virkelige verden og å generalisere løsninger slik at den kan anvendes til å løse lignende problemer. Denne måten å arbeide på er sentral i programvareutvikling, men kan også brukes som metode i mange andre sammenhenger og fag.

Programmering kan være med å utvikle elevers algoritmiske tankegang ved at de lærer å:

- formulere problemer på en slik måte at vi kan bruke digitale verktøy til å løse dem
- organisere og analysere data på en logisk måte
- bryte ned komplekse problemstillinger til mindre del-problemer
- representere data ved hjelp av abstraksjoner som modeller og simuleringer
- lage automatiserte løsninger ved hjelp av algoritmer (steg-for-steg)
- identifisere, analysere og implementere mulige løsninger på problemet mest mulig effektivt
- prøve, feile og feilsøke
- generalisere løsninger

Fra problem til delproblem

Å kunne bryte ned problemer i mindre enheter/delproblemer er en del av kompetansen elevene trenger å tilegne seg. Typiske kompetansemål innen programmering er:

- Omgjøre problemer til konkrete delproblemer, vurdere hvilke delproblemer som lar seg løse digitalt, og utforme løsninger for disse
- Utvikle og feilsøke programmer som løser definerte problemer, inkludert realfaglige problemstillinger og kontrollering eller simulering av fysiske objekter

Det kan handle om å vurdere ulike løsningsmetoder, og argumentere for valgt løsning. Å kunne argumentere for valgt løsning er også en del av dybdelæringen. For å kunne argumentere må man både ha en forståelse for problemet, finne og vurdere ulike måter å løse problemet på og konkludere med hva man mener fungerer best.

Dersom man finner ut at man skal utvikle en løsning handler det både om utvikling, feilsøking og forbedring - noe som omhandler både algoritmisk tankegang, ferdigheter i programmering og gjerne samarbeid om forbedringer gjennom å motta innspill fra andre, f.eks. kan dette handle om brukergrensesnittet - altså hvordan det oppleves av den som bruker løsningen (eks. er det intuitivt hva man skal gjøre). En løsning kan både handle om å lage et program, skape en app eller kontrollere/simulere fysiske objekter - helt avhengig av hva som er problemet som skal løses.

Arbeidet kan deles inn i fire faser:

1. Forklare hva problemet går ut på, gjerne basert på en reell og kjent problemstilling.
2. Ha en idemyldring, ved hjelp av for eksempel tankekart, for å finne hvilke mindre delproblemer det store problemet består av.
3. Vurder hvilke av delproblemene som kan la seg løse ved hjelp av programmering. Dersom flere delproblemer egner seg, kan de løses av forskjellige grupper.
4. Løsning. Foreslå en løsning på problemet. Løsningen kan innebære nettside, spill, app, kontrollering eller simulering av et fysisk objekt, eller en annen programmeringsbasert løsning.

Algoritme er trinnvis prosedyre for å løse problemet. Flytskjema er et diagram laget av forskjellige former for å vise datastrømmen. ... I algoritme brukes vanlig tekst. I flytskjema brukes symboler / former.

Hva er forskjellen mellom algoritme-pseudokode og flytskjema?

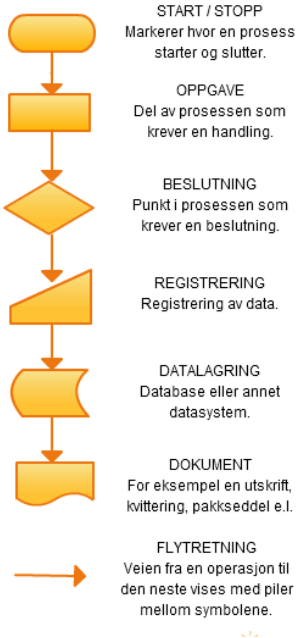
Hovedforskjellen mellom Pseudokode og flytskjema er at pseudokode er en uformell beskrivelse på høyt nivå av en algoritme, mens flytskjema er en illustrasjon av en algoritme. Dermed er pseudokode og flytskjema to metoder for å representere en algoritme.

Flytskjema

Når man skal programmere er det viktig å danne seg et bilde av hva som skal utføres, i hvilken rekkefølge og hvilke inn/output er nødvendige. For å få oversikt over hvordan programmet skal fungere og hvilke grunnlag som gjør at det skal fungere slik vi har tenkt.

Denne oversikten kaller vi for flytskjema. Et flytskjema kan godt tegnes for hånd, men det er som regel bedre å bruke et dataverktøy. Mange vanlige kontorstøtteprogrammer som tekstbehandling, regneark og presentasjonsprogram har innebygget flytskjemasymboler, og det finnes en rekke spesialprogrammer og nettbaserte verktøy (eksempelvis Creately).

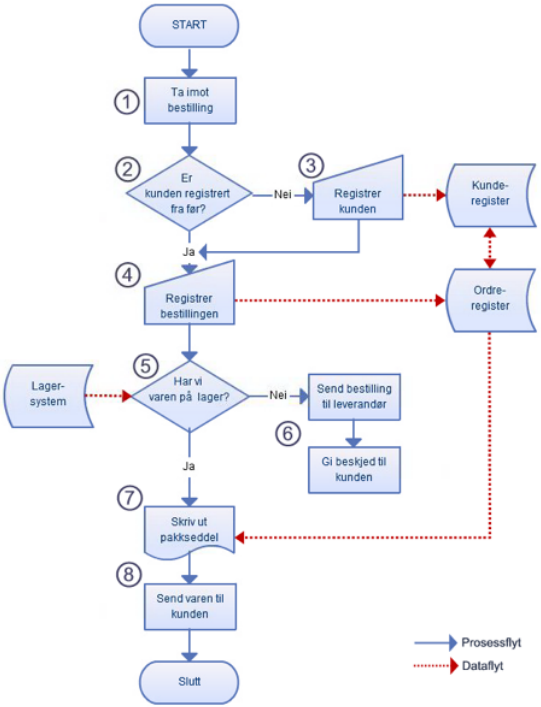
Det lønner seg å lage en kladd før du begynner å tegne flytskjema i et program. En effektiv metode er å bruke Post-It-lapper på glasstavle/tusjtavle- eller annet egnet underlag.



Skriv alle oppgaver, registreringer, beslutninger på hver sin lapp, klistre lappene på tavla og tegn piler mellom dem. Etter hvert som du oppdager nye oppgaver, kan du lage nye lapper og flytte dem rundt på tavla inntil hele prosessen er kartlagt.

I dette tilfellet har vi vist et enkelt flytskjema fra et ordremottak. Vi ser her at det er lagt opp 2 forhold, et hvor man har prosessflyt og et hvor man har dataflyt.

Dette er også en metode som benyttes i programmering, eksempelvis at man har en prosess som handler om input, en om behandling og en om output. Da får man en god oversikt over de ulike kriteriene i programmet og hvordan informasjon samles, skal behandles, og hva man ønsker å bruke det til (funksjon).



Hverdagsalgoritmer

Denne delen er en individuell del hvor du tar utgangspunkt i hverdagsoppgaver og gjøremål og lager en Algoritme ut fra dette.

Eksempler på hverdagsalgoritmer:

- Stå opp
- Spise frokost
- Gå/kjøre/ta buss til skole
- Koke kaffe eller te
- Handle i butikk
- Gå tur med hunnen
- Mate hunnen eller katten
- Lese en bok

Gruppeoppgave og algoritme, tavle eksempel

Denne delen er en gruppeoppgave hvor man tar utgangspunkt i en sekvens, en rekkefølge, som løses på tavlen og ved hjelp av elevene som «Brikker» med tilhørende utstyr for å kunne gjennomføre.

Eksempler på algoritme løst på tavle:
4 elever, elevene heter Per, Pål, Truls, Lise

Når Per reiser seg og står stille, går Pål opp og skrur på tavla og setter seg igjen. Hvis Truls ber Per sette seg igjen, reiser Pål seg igjen og ber Lise skrive Algoritme med blå farge på tavla og sette seg igjen, hvis ikke går Truls opp og skriver Algoritme med rød farge på tavla og setter seg igjen. Hvis det står Algoritme med rød farge på tavla og Truls har satt seg, går Lise opp og skrur av tavlen og setter seg igjen. Hvis det står Algoritme med blå farge på tavla, går Per opp og skrur av tavlen og setter seg igjen.

Del nå inn i 4 grupper med 3-5 elever på gruppa(avhenger av klassestørrelse). Lag så hver deres algoritme, presenter denne, og så tester vi funksjonen av dem.

Kursmål DEL 1B

- Logiske kretser
- Pseudokode
- Flytskjema

Logiske kretser, boolsk

Vi går gjennom noen enkle Boolske utrykk og funksjoner for at du skal forstå litt av denne funksjonen og hvordan den brukes i de praktiske forholdene knyttet til funksjonsbeskrivelse av programmer og styringer i elektronikken.

Boolske uttrykk brukes for å beskrive funksjoner i digitale kretser. De tre grunnleggende funksjonene er:

AND, OR og NOT

Andre funksjoner er såkalte funksjoner som avledes (kommer fra) fra AND, OR og NOT. Disse funksjonene er:

NAND, NOR, XOR og XNOR

En Boolsk funksjon kan beskrives enten ved hjelp av Sannhetsverditabell eller ved hjelp av Funksjonsuttrykk.

To funksjoner er ekvivalente hvis de for alle input-kombinasjoner gir samme output.

Regne regler:

+ = Or, eller port
x = And, OG port

$a + a' = 1$	$a + a = a$	$a + 0 = a$	$a + 1 = 1$
$a \times a' = 0$	$a \times a = a$	$a \times 0 = 0$	$a \times 1 = a$

Sannhetstabell

AND

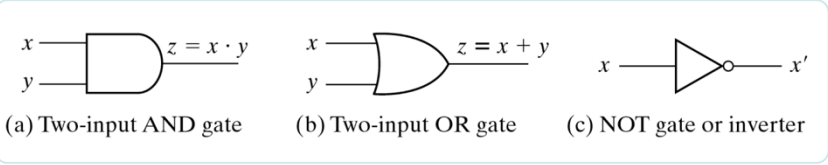
X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

OR

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

NOT

X	Y
0	1
1	0

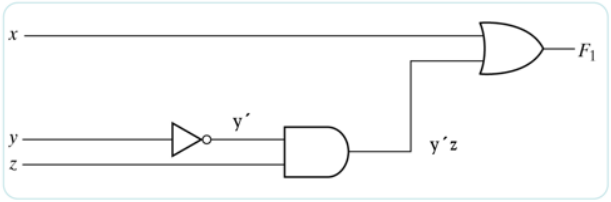


(a) Two-input AND gate (b) Two-input OR gate (c) NOT gate or inverter

Eksempel:

F = x + y´z

Direkte port-implementasjon:



x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

MERK: Invertert betyr «endret», dvs. får du et høyt signal, vil det inverterte være lavt.

Øvelser i Boolsk logikk

Tenk gjennom oppgaven før du starter! Er det noe du må vite for å kunne løse denne, dvs. begrensninger man må ha svar på før man løser den?

Forenklet design av heisstyring med sannhetstabell

- Kontroller den oppsatte sannhetstabellen
- Hvis heisen står stille, dvs. dør lukket, ingen overvekt, ingen knapp trykket, hva viser sannhetstabellen da i KVD?
- Knappen er trykket inn, det er overvekt, døren er lukket, hva viser KVD

Inngangssignaler fra sensor

K = Knapp trykket inn: 0/1

V = Overvekt: 0/1

D = Dør lukket: 0/1

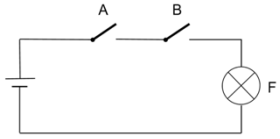
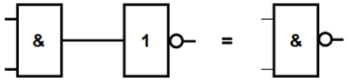
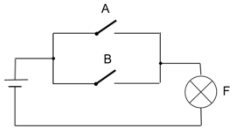
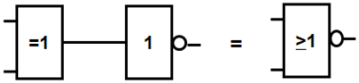
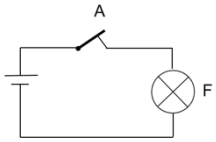
Utgangssignal til aktuator:

M = Motor på: 0/1

K	V	D	M
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Oppgaver logiske kretser

Hvilke porter er illustrert her?

	
	
	Tegn en og eller funksjon

Logiske kretser med Micro:Bit

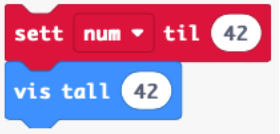
Boolsk

En verdi som bare er sant eller usant.

En boolsk verdi har én av to mulige verdier: sann eller usann. De boolske (logiske) operatorene (*og eller ikke*) tar boolske inndata og lager en annen boolsk verdi. Sammenligning av andre typer (tall, strenger) med logiske operatorer oppretter boolske verdier.
Disse blokkene representerer de sanne og usanne boolske verdiene, som kan kobles til hvor som helst en boolsk verdi forventes:

Det finnes flere forskjellige datatyper som brukes i dataprogrammering. Vi har tidligere set på to av disse typene:

1. Streng (for tekst)



2. Heltall (for tall)

Boolsk er en annen type data. En boolsk datatype har bare to verdier: sann eller usann. På binær måte kan disse to verdiene representeres av tallene 1 = sann, og 0 = usann.

Boolske funksjoner er nyttige i programmering for beslutningstaking, og bestemmer ofte når bestemte funksjoner og deler av programmer skal starte eller slutte å kjøre, og brukes også i databasesøk.

Tenk en funksjon i dagliglivet som består av Boolsk funksjon, eksempelvis:

1.	Lys:	På eller Av?
2.	Tid:	Morgen eller kveld(AM eller PM)?
3.	Du!:	Sove eller våkne?
4.	Vær:	Regn eller ikke regn?
5.	Matematikk:	Lik eller Ikke lik?
6.	Brus:	Cola eller Solo?
7.	I butikken:	Papir eller plast?
8.	Betaling:	Kontanter eller kreditt?

Merk: Argumenter kan gjøres at noen av disse kan ha mer enn to verdier. For eksempel: I butikken kan du ha tatt med egne gjenbrukbare poser eller betale med sjekk.

Hvis du har arbeidet med betingelser eller løkker i programmering, har du allerede arbeidet med denne typen logikk:

1. Hvis en bestemt betingelse er sann, gjør du dette, ellers (hvis betingelsen er usann), gjør noe annet.
2. Mens en bestemt betingelse er sann, gjør du dette

Boolske operatorer: OG, OR og IKKE Hvis du vil gjøre det nyttig å arbeide med boolske for å løse mer komplekse beslutninger og søk, kan vi koble to eller flere boolske boolere til én beslutningserklæring. For å gjøre dette bruker vi det som kalles boolske operatorer. De tre vanligste og de vi vil bruke med micro:bit er And, Or, og Not.

Disse operatorene kan brukes i betingede og løkker på denne måten:

1. Hvis betingelse A er sann OG betingelse B er sann
2. Hvis betingelse A er sann ELLER betingelse B er sann
3. Mens hendelse A IKKE har skjedd

La oss se på hvordan hver av disse fungerer.

OG

(Betingelse A OG betingelse B) For at dette uttrykket skal betegnes som sant, må begge betingelsene i uttrykket være sanne. Hvis både Betingelse A **OG** betingelse B er sann, betegnes uttrykket som sant.

ELLER

(Betingelse A **ELLER** Betingelse B) For at dette uttrykket skal betegnes som sant, trenger bare en av betingelsene i uttrykket å være sann. Hvis Betingelse A er sann, er uttrykket sant uavhengig av om Betingelse B er sann eller usann. Hvis Betingelse B er sann, er uttrykket sant uavhengig av om Betingelse A er sann eller usann.

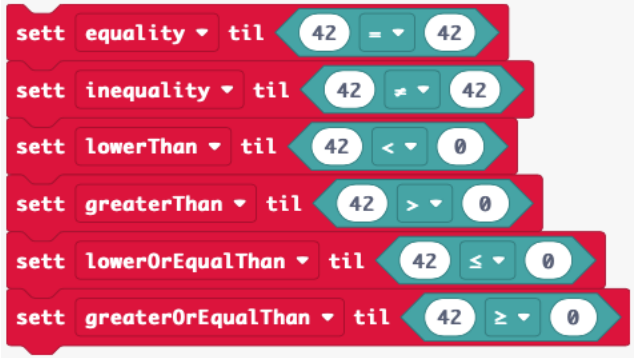
IKKE

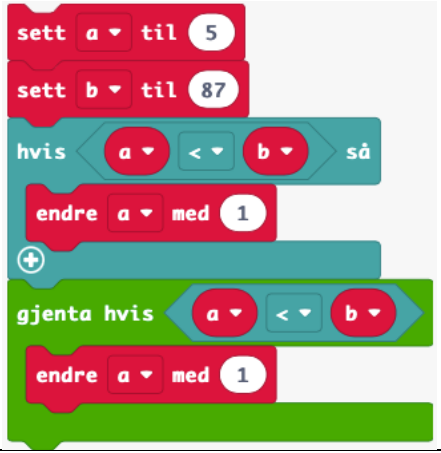
NOT kan brukes når du kontrollerer at en betingelse er usann (eller ikke sann). For eksempel:

1. (**NOT** Condition A og Betingelse B) betegnes som sann bare hvis Betingelse A er usann og Betingelse B er sann.
2. (Betingelse A og **IKKE** betingelse B) betegnes som sann bare hvis Betingelse A er sann og Betingelse B er usann.
3. (**NOT** Condition A og NOT Condition B) betegnes som sann bare hvis både Betingelse A og Betingelse B er sanne. **IKKE** er også nyttig når du bruker en løkke. For eksempel kan du bruke en **IKKE** å sjekke Men knappen A er **IKKE** trykket, fortsett å kjøre denne koden ...

Merk: 'Usann' kan betraktes som tilsvarende 'IKKE sant'.

George Boole (født 2. november 1815, født 8. desember 1864) var en engelsk matematiker, pedagog, filosof og **logikker.** Han arbeidet med differensialligninger og algebraisk **logikk**, og er best kjent som forfatteren av The Laws of Thought (1854) som inneholder **boolsk** algebra.

	Disse blokkene representerer de boolske verdiene **sann** og **usann** som kan brukes når man trenger en boolsk verdi:
	De tre Boolske logiske operatorene
	Disse seks blokkene representerer sammenligningsoperatorer som gir en boolsk verdi. De fleste sammenligninger vi gjøre involverer tallverdier.

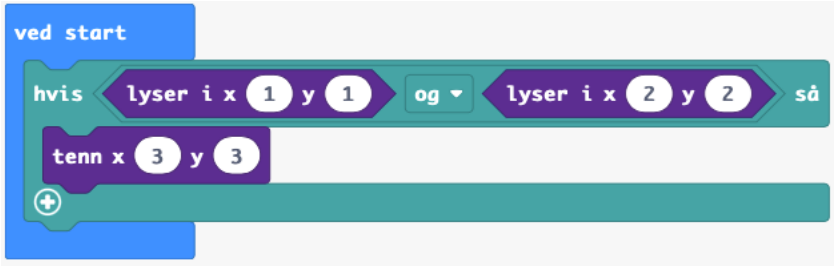


Boolske verdier og operatorer brukes ofte sammen med en if - eller while -setning for å bestemme hvilken kode som skal kjøres videre.

Noen funksjoner returnerer en boolsk verdi, som du kan lagre i en boolsk variabel. Koden nedenfor får for eksempel av/på-statusen for punkt (1, 2) og lagrer dette i den boolske variabelen med navnet på. Deretter tømmer koden skjermen hvis den er sann:

A og B settes til sann hvis, og bare hvis både A og B er sanne:

Dette eksemplet aktiverer LED 3 , 3, hvis lysdioder 1 , 1 og 2 , 2 begge er på:

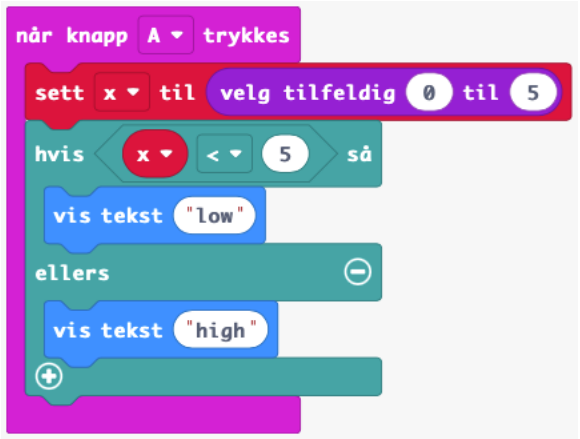


Java Script

```
if (led.point(1,1) && led.point(2,2)) { led.plot(3,3) }
```

Eksempel på sammenlikning av numeriske verdier og string (tekst)

Når du sammenligner to tall, får du en boolsk verdi, for eksempel sammenligningen x < 5 i koden nedenfor:



Java Script

```
input.onButtonPressed(Button.A, () => { let x = randint(0, 5) if(x < 5) { basic.showString("low"); } else { basic.showString("high"); } })
```

Øvelse

Hva skal vi gjøre?

Gjør micro:bit til en bryterfunksjon som du kan slå av og på ved å klappe eller lage høy lyd.

1. Slik bytter du utganger som svar på inngangssignal
2. Slik bruker du boolsk logikk til å lage en bryter som bruker lyd som styring

Slik fungerer det

1. Programmet bruker en variabel kalt **lightsOn** for å holde oversikt over lysets status: enten det er slått på eller av. Vi bruker den som en spesiell type variabel, en boolsk variabel. Boolske variabler kan bare ha to verdier: sann (på) eller usann (av).
2. Når mikrofonsensoren oppdager en høy lyd, slår verdien av **lysOn** ved å stille den til å være **ikke lysOn**.
3. Dette betyr at når du klapper, hvis **lightsOn** er usann (og lysene er av), blir det sant og programmet tenner lysdiodene.
4. Hvis **lightsOn** var sant (og lysene var på), blir det falskt og koden slår av lysdiodene.

Det du trenger

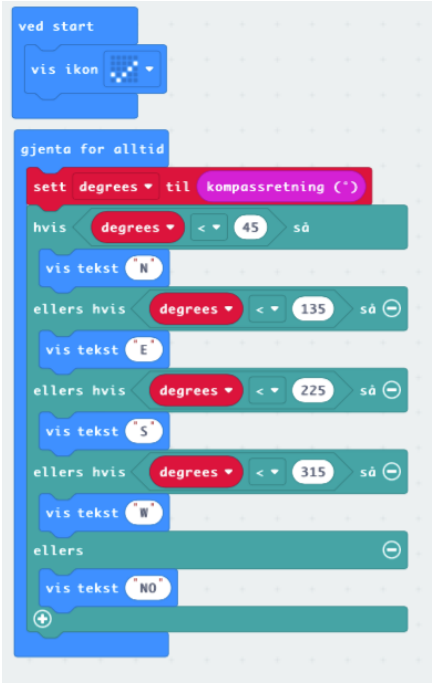
1. En mikro:bit
2. MakeCode IDE
3. Batteripakke til microbiten din

Legg inn koden og test.

Forsøk så å utvikle den ved å legge til andre sann, usanne funksjoner, eksempelvis lyd, andre lys eller funksjoner basert på inngangssignalet klapp/høy lyd. Eksempelvis kan man bestemme hva som er høy lyd og hva som er lav?

Vi skal nå lage et kompass, først i MicroBit IDE, laste ned programmet på Bitén og se om den fungerer.

Vi bruker Basis, Inndata, logikk og variabler.
NB! Noen av variablene må skapes



Referanse:

https://hiof.instructure.com/courses/712/pages/algoritmer
https://dataelektroniker.fagbokforlaget.no/131-digitale-kretser
https://makecode.microbit.org/courses/csintro/booleans/overview (oversatt)
https://no.wikipedia.org/wiki/George_Boole
https://www.plymouth.ac.uk/uploads/production/document/path/3/3760/PlymouthUniversity_MathsandStats_Boolean_algebra_and_logic_gates.pdf
https://microbit.org/projects/make-it-code-it/clap-lights/?editor=makecode#step-3:-improve-it