NTNU
Kunnskap for en bedre verden

Algoritmer

Teoribok, kapittel ”5. Algorithms”

TDT 4105 Grunnkurs

Professor Guttorm Sindre

Læringsmål og pensum

- Mål
  - Lære om
    - Algoritme som konsept
    - Representasjon av algoritmer
    - Oppdagelse av algoritmer
    - Iterative strukturer
    - Rekursive strukturer
    - Effektivitet og korrekthet
- Pensum
  - Kapittel 5 i *Computer Science, an Overview*: Algorithms

Hvorfor vite noe om dette?

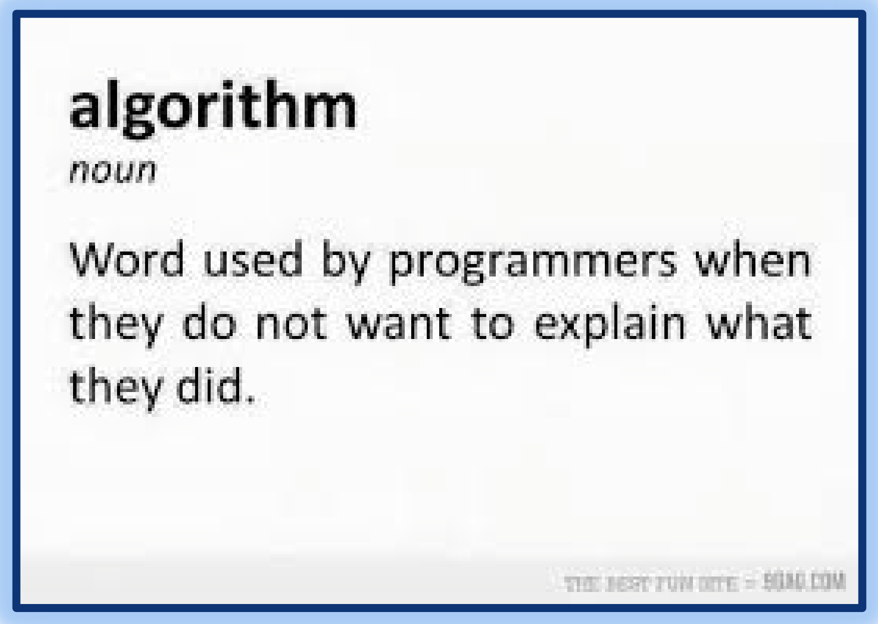
- Innsikt i algoritmer gjør det mulig å lage
  - Programmer som virker
  - Programmer som er bedre enn å bare virke
- Noen kvalitetskriterier for programmer
  - Korrekthet (programmet virker)
  - Enkelhet, koden lett å forstå
  - Brukervennlighet
  - Sikkerhet
  - Ytelse (f.eks rask responstid)
- For noen programmer er ytelse viktig
  - Da er det ikke nok å finne en mulig algoritme
  - Man må finne en <u>rask</u> algoritme

Konseptet algoritme

Kapittel 5.1

Definisjon på algoritme

En algoritme er et **ordnet** sett av **entydige**, **utførbare** skritt som definerer en **terminerende** prosess.



Representasjon av algoritme

- Fordrer veldefinerte primitiver
- En samling primitiver utgjør et programmeringsspråk

- Det fins også algoritmer som mennesker skal utføre
  - Matoppskrifter
  - Bruksanvisninger for teknisk utstyr
  - Monteringsanvisninger for møbler
  - Origamioppskrifter, strikkeoppskrifter, knuter
  - Vinstteknikker i forskjellige spill
  - Metoder for å løse forskjellige matematiske problemer…

Evne til algoritmisk tenkning

- Er viktig for programmere
- Også nyttig i de fleste andre typer jobber
  - Produktutvikling
  - Prosjektgjennomføring
  - Forhandlinger
  - Investeringer
  - Markedsføring
  - Etterforskning
  - Oljeleting
  - …
- Generelt verdifullt å kunne
  - Legge planer som er ambisiøse men likevel gjennomførbare
  - Vurdere betingelser og beste rekkefølge for aksjoner

Primitiver i *pseudokode*

- Tilordning:

  variabelnavn ← uttrykk

- Betinget valg

  hvis betingelse **så** handling

- Gjentatt utføring:

  så lenge betingelse **gjør** aktivitet

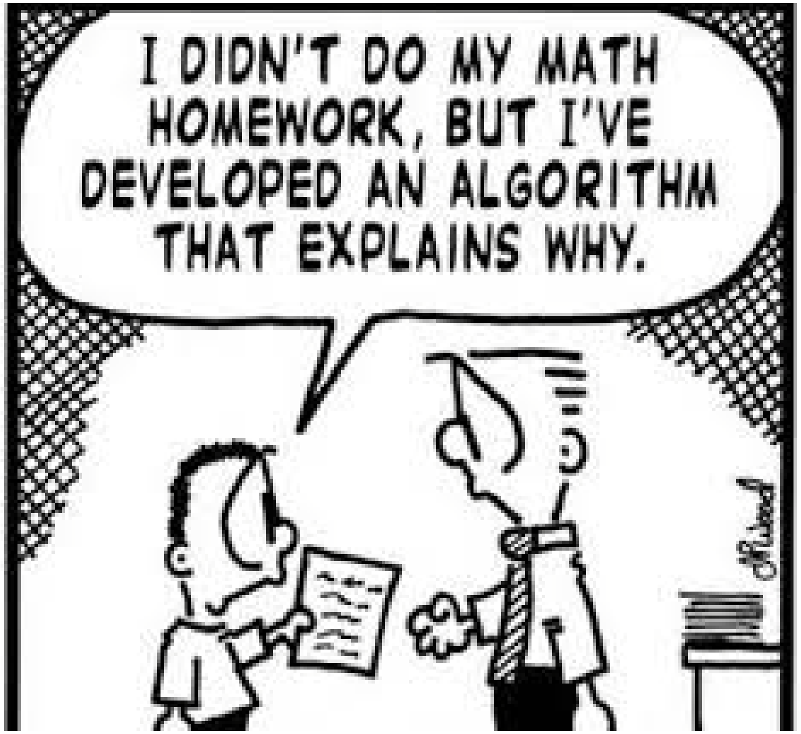
  for alle element **i** sekvens **gjør** aktivitet

- Prosedyre

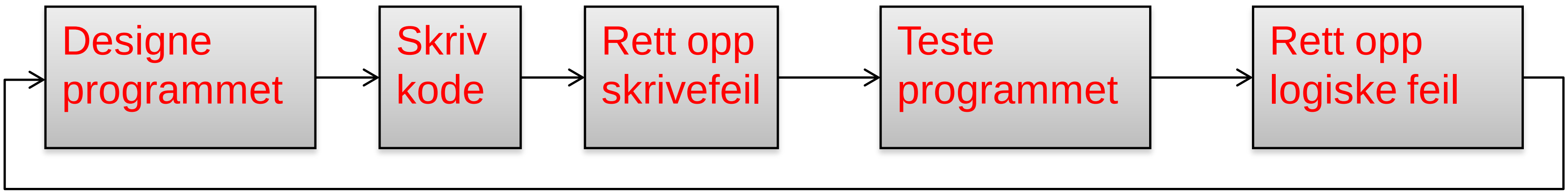
  prosedyre navn (parametre)

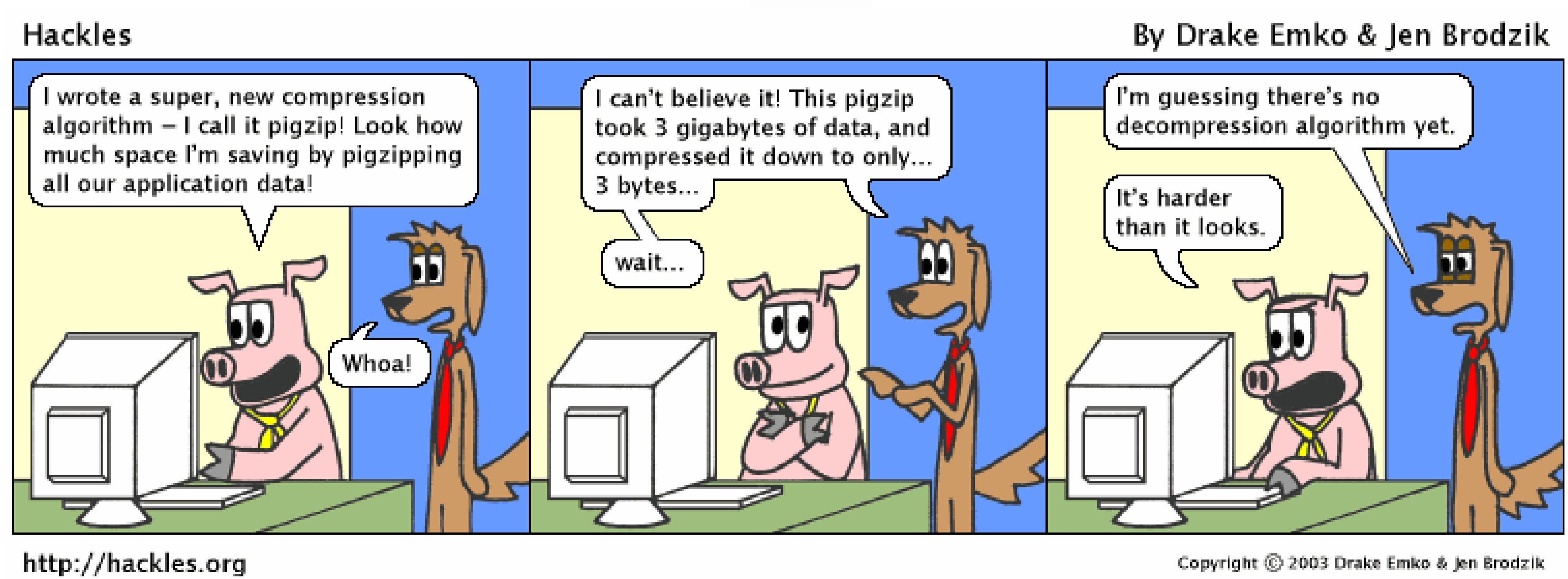
Algoritmeoppdagelse

Kapittel 5.3



Programutviklingssyklus





Polyas problemløsningsfaser (t.v.)

i programmeringskontekst (t.h.)

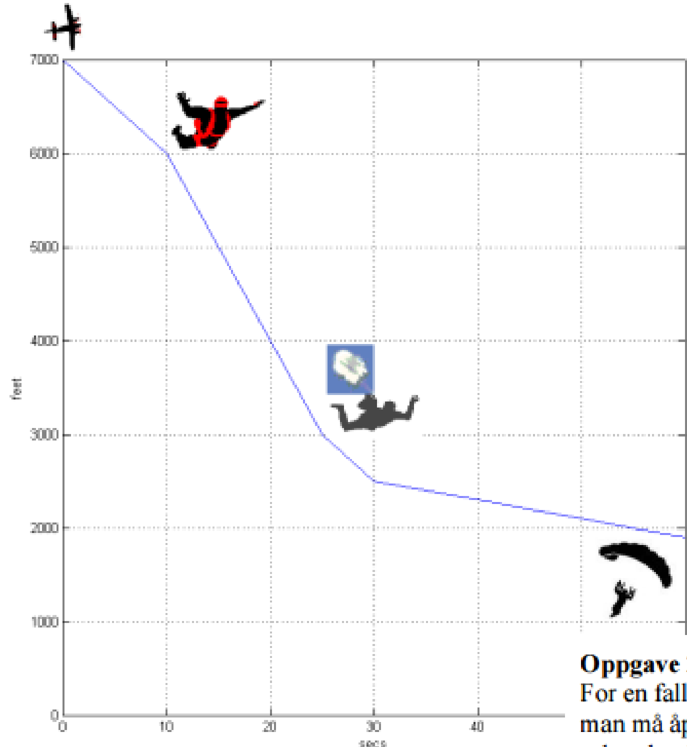
1. Forstå problemet
2. Pønsk ut en plan for å løse problemet
3. Utfør planen
4. Evaluer løsning mhp. nøyaktighet potensial for å løse andre problemer

1. Forstå problemet
2. Få en ide om hvordan en algoritme kan løse problemet
3. Formuler algoritmen og representer den som et program
4. Evaluer programmet mhp nøyaktiget og potensial for å løse andre problemer

Forstå problemet – Eksempel

Eksamen desember 2014, oppgave 2e

Vi benytter en litt forenklet versjon av jordens fysiske lover:



Figur 1. Hopp fra 7000 fot.

En fallskjermhopper faller (med konstant/gjennomsnittlig hastighet) 100 fot pr. sekund, de 10 første sekundene, og deretter med konstant topphastighet på 200 fot pr. sekund til skjermen må åpnes i 3000 fots høyde (se figur 1). Hvis man mot normalt hopper ut under 3000 fot må skjermen utløses umiddelbart (etter 0 sekunder).

Oppgave 2e (5%)

For en fallskjermhopper er det veldig viktig å være klar over hvor mange sekunder man kan vente før man må åpne fallskjermen (Se figur 1). Lag en funksjon `feet2seconds` som regner ut hvor mange sekunder det tar å falle fra et oppgitt antall fot ned til 3000 fot (inn-parameter `feet`, og retur-verdi `seconds`). Bruk informasjon gitt i starten av oppgave 2 (forklaringen til figur 1) til å beregne riktig tid. Hvis antall fot er under 3000 skal funksjonen returnere 0.

Eksempler på bruk:

```
>> feet2seconds( 12500 )
ans =   52.5000
>> feet2seconds( 7000 )
ans =    25
>> feet2seconds( 2000 )
ans =     0
>>
```

Eksempel (forts.)

- Mange ineffektive og unødig kompliserte løsninger
- Typisk tabbe:
  - "Oversette" teksten direkte til kode:

```
funksjon feet2seconds(feet)
  sec ← 0
  så  lenge feet > 3000
    sec ← sec + 1
    hvis sec < 10
         feet ← feet – 100
    ellers
         feet ← feet – 200
  returner sec
```

- Ulemper:
  - Avrunding (hele sek.)
  - Unødig lang kode
  - Unødig treg kode
    - tidsbruk øker med fallhøyde
- Enklere og bedre:

```
funksjon feet2seconds(feet)
    sec ← 0
    hvis feet > 4000
        sec ← 10 + (feet-4000)/200
    ellers hvis feet > 3000
        sec ← (feet-3000)/100
    returner sec
```

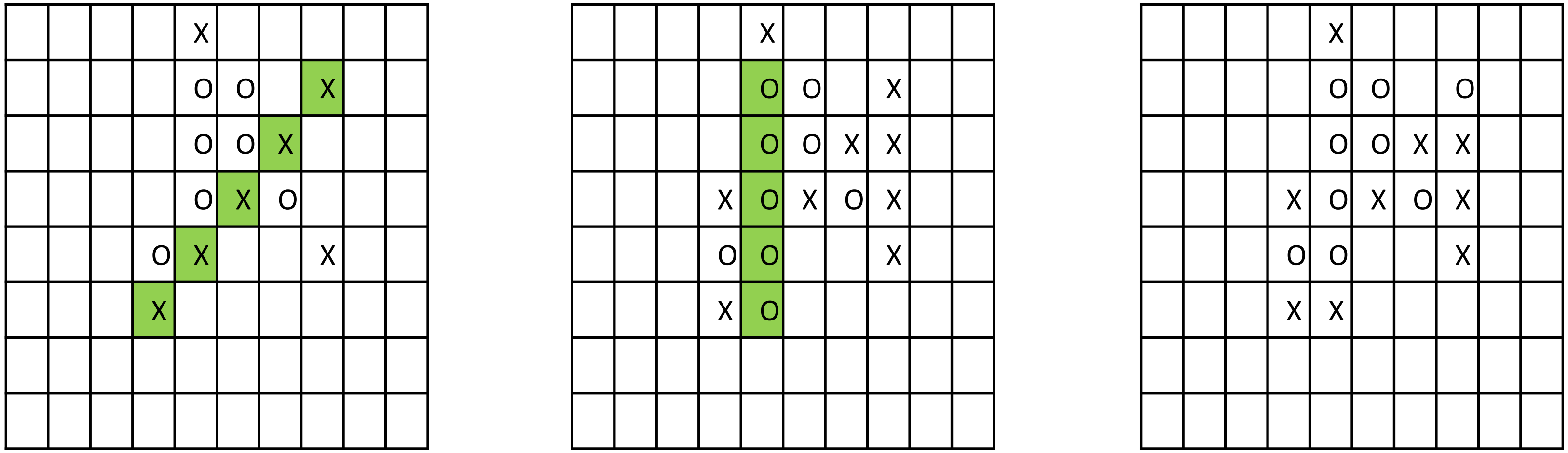
Få foten innenfor

- Forsøk å løse problemet baklengs
- Løs et enklere, relatert problem
  - Løsne på noen av problembeskrankningene
  - Løs deler av problemet først (**bottom-up**-metoden)
- Skrittvis forbedring: del opp problemet i mindre problemer (**top-down**-metoden)

"The boss wants me to create a computer algorithm that can convert hindsight into foresight."

Eksempel på et problem

- Vi holder på med et program for å spille bondesjakk
- Etter hvert trekk trengs en funksjon for å sjekke om noen har vunnet spillet
- Kriterium for seier:
  - En spiller (X eller O) må ha 5 på rad
  - De 5 på rad kan være vannrett, loddrett eller diagonalt



Få foten innenfor

- Forsøk å løse problemet baklengs

```
    hvis ant_X >= 5 så:
        returner X som vinner
    hvis ant_O >= 5 så:
        returner O som vinner
```

(og så tenke: hvordan komme hit?)

- Løs et enklere, relatert problem
  - Eksempel: først prøve å løse bare vannrett?
- Skrittvis forbedring: del opp problemet i mindre problemer (**top-down**-metoden)
  - En funksjon vannrett, en loddrett, en diagonalt…

Forsøk på naiv algoritme

```
funksjon seier ( )
    teller_X ← 0
    teller_O ← 0
    for alle rader i spillebrett:  # Sjekker vannrett
      for alle ruter i rad:
        hvis rute == X:
                teller_O ← 0
                teller_X ← teller_X + 1
                hvis teller_X >= 5:
                        returner X som vinner
        hvis rute == O:
                teller_O ← teller_O + 1
                hvis teller_O >= 5:
                        returner O som vinner
                teller_X ← 0
        hvis rute == ' ':
                teller_O ← 0
                teller_X ← 0
# TO BE DONE tilsvarende doble for-løkker loddrett og for 2 diagonaler…
```

Algoritma vil virke, men

- Er langt fra den mest effektive…
- Går igjennom alle ruter i hele brettet 4 ganger
  - Med brettstørrelse **N x N** betyr dette at vi sjekker $4N^2$ ruter
  - Kan spare litt på dette:
    - For diagonalene droppe 10 første og 10 siste ruter
      - umulig å få 5 på rad der
    - Stoppe hvis for kort igjen til kanten til å nå 5 på rad
    - Sjekker da omtrent $4N^2 - 4N - 40$ ruter
  - Men dette er en <u>ubetydelig</u> besparelse
    - Jo større brett, jo mer vil det kvadratiske leddet dominere
    - Tidsforbruk på å sjekke seier vil øke kvadratisk
- Det fins en essensiell egenskap ved problemet som vi ikke har utnyttet. Hvilken?

Mer effektiv variant

- En seier er alltid resultat av siste trekk
  - Kun sjekke spilleren som trakk, og kun deler av brettet
  - Kun en liten del av brettet, utover fra siste trekk

```
funksjon tell_like(x, y, dx, dy, symbol)
    teller ← 0
    så lenge rute[x,y] == symbol
        teller ← teller + 1
        x ← x + dx
        y ← y + dy
    returner teller
```

```
funksjon seier(x, y, symbol)
    returner (tell_like(x-1, y, -1, 0, symbol) + tell_like(x+1, y, 1, 0, symbol) >= 4  eller
              tell_like(x, y-1, 0, -1, symbol) + tell_like(x, y+1, 0, 1, symbol) >= 4 eller
              tell_like (x-1,y-1, -1, -1,symbol) + tell_like(x+1,y+1,1,1,symbol) >= 4 eller
              tell_like (x-1,y+1, -1,1,symbol) + tell_like(x+1,y-1,1, -1,symbol) >= 4 )
```

Forskjell på algoritmene

- **Den naive algoritmen**
  - Sjekker (nesten) alle rutene i brettet 4 ganger
  - Brett med bredde og høyde **n**
    - kjøretiden utvikler seg kvadratisk (som funksjon av $\mathbf{n^2}$)
  - Seierssjekken vil bli tregere jo større brett vi bruker

- **Den smartere algoritmen:**
  - Sjekker maksimalt 20 ruter
  - Seierssjekken går like fort uansett hvor stort brettet er

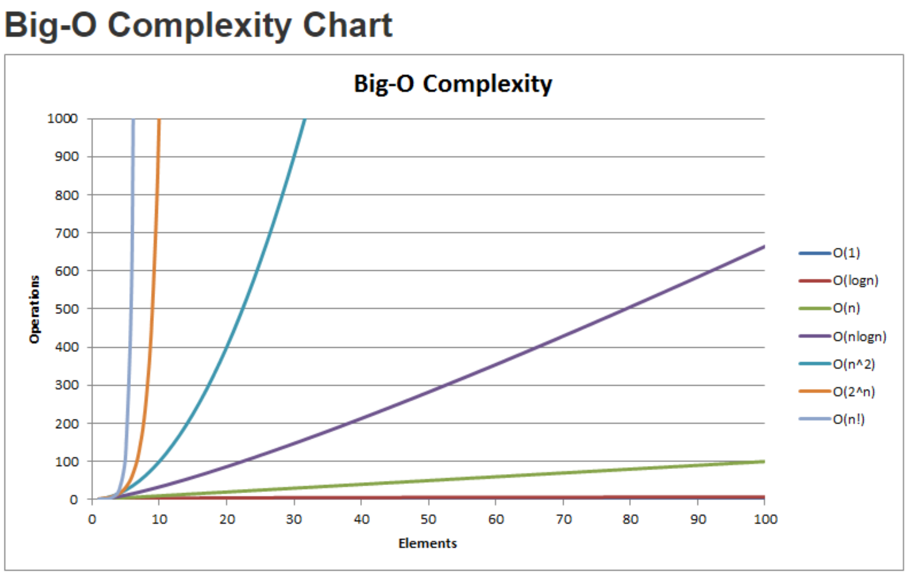
- **Store Θ-notasjon (Big-theta):**
  - Den naive algoritmen er $\Theta(n^2)$
  - Den smartere algoritmen er $\Theta(1)$

Hva betyr Θ ?

- Anta at ei algoritme sitt tidsforbruk er gitt som f(n)
  - Hvor n uttrykker størrelsen på dataene / datamengden
- Tre ulike notasjoner:
  - O (Big O), betyr at
    - f(n) = **O(g(n))**   hvis $|f(n)| <= k \cdot |g(n)|$
    - g(n) angir en øvre grense for f(n) for tilstrekkelig store n
  - Ω (Big Omega)
    - f(n) = **Ω(g(n))**   hvis $|f(n)| >= k \cdot |g(n)|$
    - g(n) angir en nedre grense for f(n) for tilstrekkelig store n
  - Θ (Big Theta)
    - f(n) = **Θ(g(n))**   hvis $k_1 \cdot |g(n)| <= |f(n)| <= k_2 \cdot |g(n)|$
    - G(n) angir både en øvre og nedre grense for f(n) for tilstrekkelig store n
    - f(n) tilhører både mengden av funksjoner som er O(g(n)) og som er Ω(g(n)), dvs. befinner seg i snittet av disse to mengdene

O, Ω og Θ

- Gir omtrentlige uttrykk for tidsforbruk
  - Eksakt uttrykk vanskelig
    - Avhengig av detaljert implementasjon og maskin man kjører på
  - Omtrentlige uttrykk er ofte tilstrekkelig
    - Skille mellom bedre og dårligere algoritmer
    - NB: hvis **n** blir stor nok



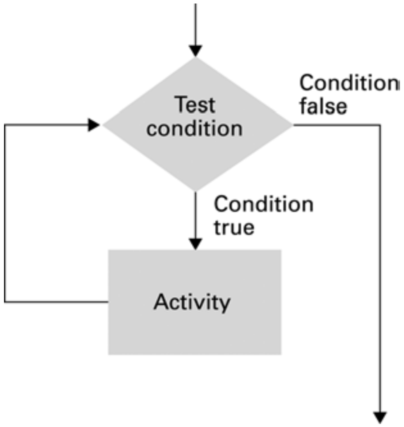
Iterative strukturer

Kapittel 5.4

Iterative strukturer

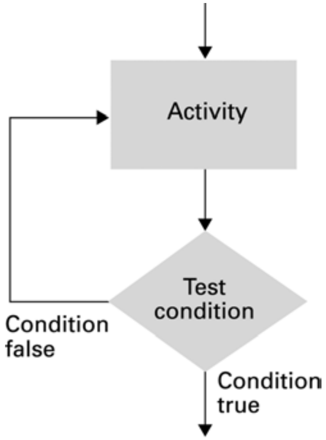
- Pretest løkke:

```
while (betingelse) do
   (loop body)
```



- Posttest løkke:

```
gjenta (loop kropp)
   inntil (betingelse)
```



Komponenter i repetisjonskontroll

- **Initialiser:**
  - Etabler en initiell tilstand
  - som modifiseres mot termineringsbetingelsen
- **Test:**
  - Sammenlikne nåværende tilstand med termineringsbetingelsen
  - terminer repetisjonen hvis den er lik
- **Endre:**
  - Endre tilstanden på en slik måte
  - at den nærmer seg termineringsbetingelsen

Algoritme for sekvensielt søk (i sortert liste)

```
Prosedyre Søk (Liste, MålVerdi)
   hvis (Liste tom) så
      (erklær at søket er mislykket)
   ellers (Velg første enhet i Liste som TestEnhet):
      sålenge (MålVerdi > TestEnhet og det er
               flere enheter igjen å undersøke)
      gjør (Velg neste enhet i Listen som TestEnhet):
         hvis (MålVerdi == TestEnhet)
            så (erklær søket vellykket)
         ellers (erklær søket mislykket)
```

Tidsforbruk for sekvensielt søk?

- Noen ganger er vi heldige, søkt element er tidlig i lista
- Andre ganger uheldige, sent i lista
  - Dvs. det vi leter etter kommer sent i alfabetet
- Gjennomsnittlig må vi se på N/2 elementer, hvor N er antall elementer i lista

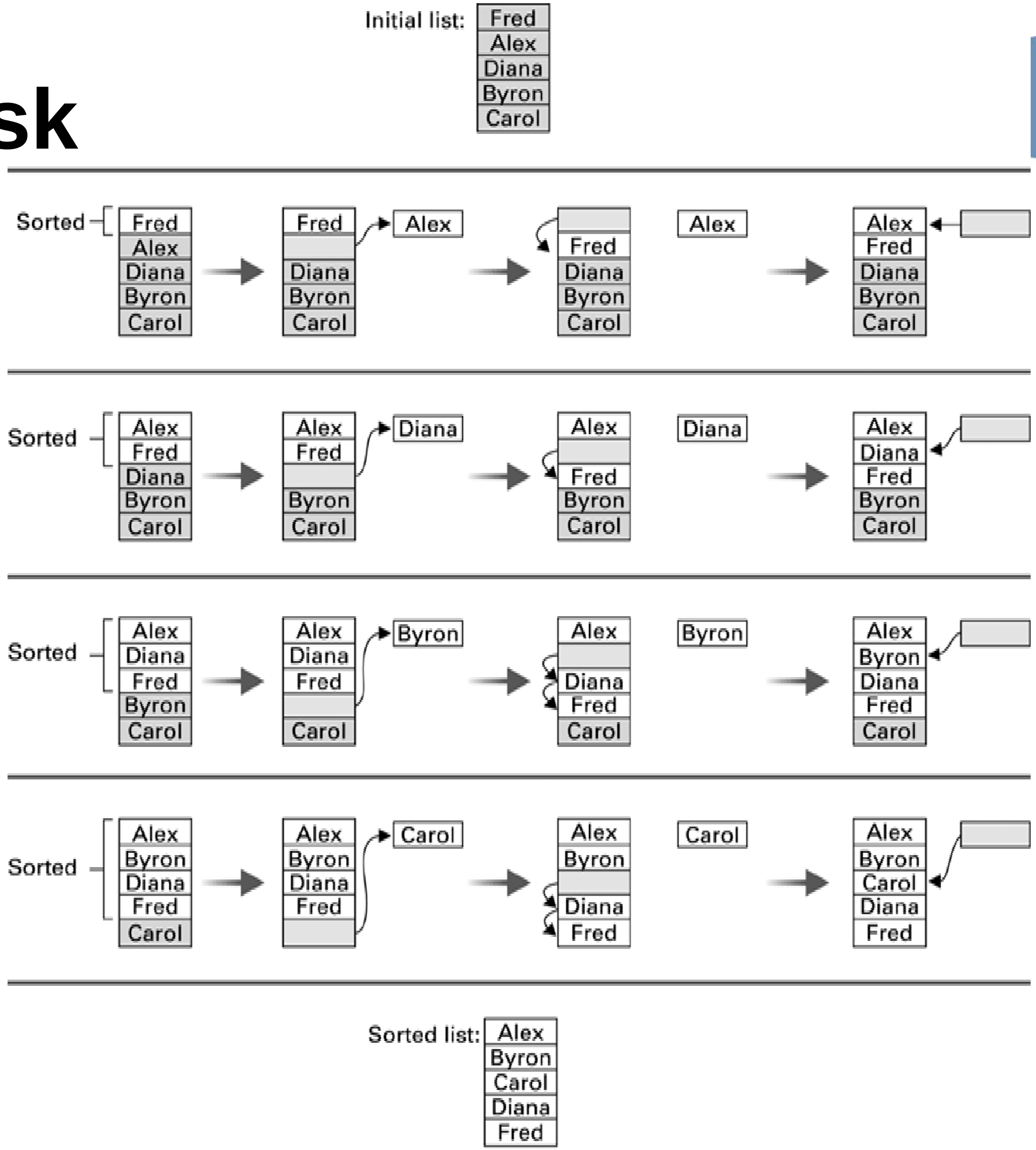
- ”Big theta notation” – uttrykk for hva algoritmens tidsforbruk som en funksjon av n er proporsjonal med
- <u>Sekvensielt søk: Θ(n)</u>
  - Dvs. tidsforbruk proporsjonal med n
  - Hvis lista blir 5 ganger så lang, vil søkene også ta 5 ganger så lang tid

Sorteringsproblemet

- **Input**:
  - Sekvens <$a_1, a_2, \ldots. a_n$> av elementer som kan rangeres ordinalt (tall, bokstaver, ord)
- **Output**:
  - En permutasjon av disse elementene slik at:

$$a_1 \leq a_2 .. \leq a_n$$

Sortere alfabetisk lista Fred, Alex, Diana, Byron og Carol



«Insertion sort» i pseudokode

```
procedyre Sort (List)
N ← 2;
while (value of N not exceed length of List) do
  (   Pivot ← Nth element in List
     (Move Pivot to temporary location
           leaving a hole in List)
  while (name above hole is greater than Pivot) do
     hole element ← element above
     prev pos element ← empty
     empty place ← Pivot;
     N ← N +1
    )
```

Jfr. Fig 5.11

Tidsforbruk for insertion sort?

- Vi har to løkker inni hverandre
- Hver av løkkene kan måtte se på N elementer

- <u>Insertion sort: $\Theta(n^2)$</u>
- Dvs. tidsforbruk proporsjonal med kvadratet av n
- Hvis lista blir 5 ganger så lang, vil sortering ta 25 ganger så lang tid

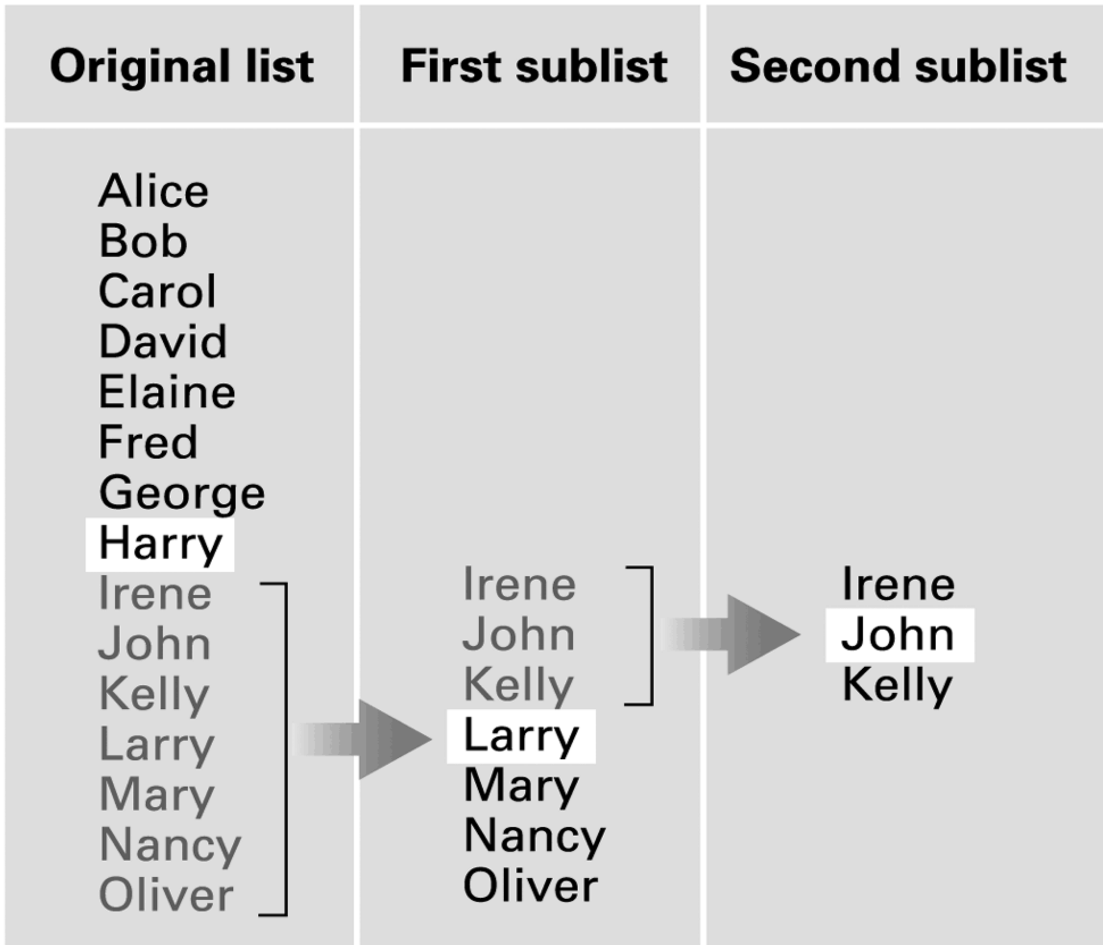
Rekursive strukturer

Kapittel 5.5

Rekursjon

- En funksjon kaller seg selv
  - Men den nye ”inkarnasjonen” av funksjonen jobber på et mindre delproblem enn den første inkarnasjonen,
    - osv. med påfølgende inkarnasjoner …
  - Mange inkarnasjoner av en funksjon kan eksistere samtidig
  - Den sist kalte blir ferdig først
  - Alle andre venter til sine kall returnerer
- Kan gi elegante og kompakte løsninger på innfløkte problemer
  - Dvs. oppnå mye med lite kode
  - Men kan være vanskelig å forstå

Bruke strategien til å søke etter John i en liste



Første utkast til binært søk

hvis (Liste tom) **så**
  (meld at søket mislyktes)
ellers
  TestElement ← midt-elementet i Liste ;
    Utfør blokka med instruksjoner under som tilhører det passende tilfellet.
        tilfelle 1: MålVerdi = TestElement
            (erklær søket vellykket)
        tilfelle 2: MålVerdi < TestElement
            (Søk i delen av Liste som kommer før TestElement etter
                MålVerdi og rapporter resultat av det søket)
        tilfelle 3:
            (Søk i delen av Lista som kommer etter TestElement etter
                MålVerdi, og rapporter resultat av det søket)

Binært søk i pseudokode

prosedyre Søk (Liste, MålVerdi)
hvis (Liste tom) **så**
    (erklær søket mislykket)
ellers
    TestElement ← midt-elementet i Liste ;
    Utfør instruksjonsblokka under som er tilhører det passende tilfellet.
        tilfelle 1: MålVerdi = =TestElement
            (meld at søket lyktes)
        tilfelle 2: MålVerdi < TestElement
            (Søk i delen av Liste som kommer før TestElement og
            rapporter resultat av det søket.)
        tilfelle 3:
            (Søk i delen av Lista som kommer etter TestElement og
            rapporter resultat av det søket).

Tidsforbruk for binærsøk?

- For hvert oppslag vi gjør, halveres gjenværende liste som vi må søke i
- Tidsforbruket blir da en logaritmisk funksjon

- Binærsøk: $\Theta(\log_2 n)$

- Dvs. tidsforbruk proporsjonalt med toerlogaritmen til n
- Hvis lista blir 4 ganger så lang, får vi bare to ekstra oppslag
- Binærsøk er dermed lurere enn sekvensielt søk som er $\Theta(n)$
  - Men forutsetter sortert liste, sekvensielt søk også mulig på usortert

Algoritmeytelse (effektivtet) og -korrekthet

Kapittel 5.6

Analyse av algoritmer

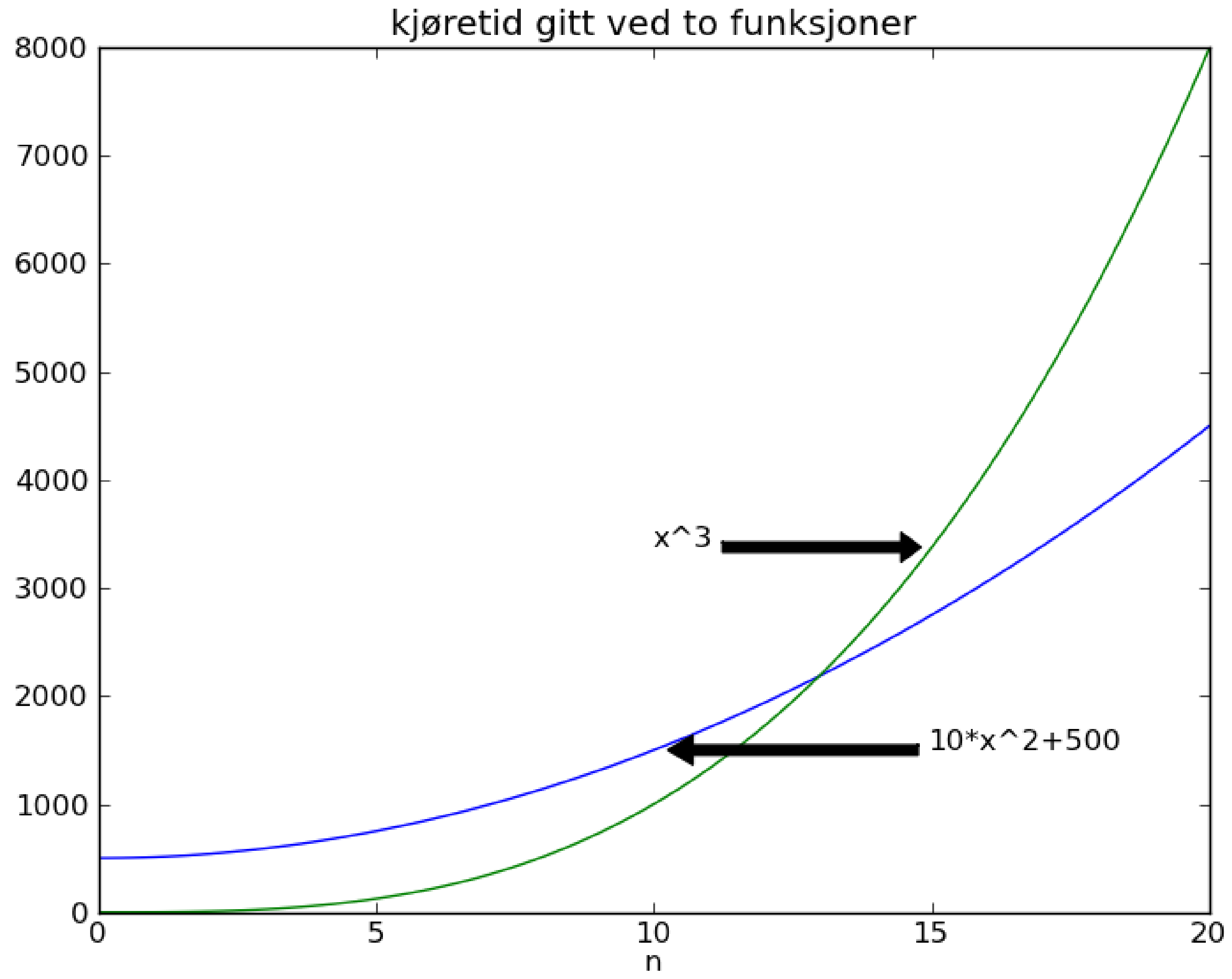
- Mange kvalitetskrav rettes til dataprogrammer
  - Korrekthet (programmet virker, gir det svaret det skal…)
  - Brukervennlighet
  - Sikkerhet
  - Ytelse (f.eks. rask responstid)
- Analyse av algoritmer fokuserer på <u>ytelse</u>
  - Ofte ikke nok med korrekt svar, må også få det innen rimelig tid
    - Lite hjelp om ABS-bremsene først slår inn når bilen allerede er i grøfta
    - Hvis et program bruker for lang tid, får vi info når den er blitt irrelevant
    - Brukeren blir lei av å vente, velger en konkurrent
  - I det siste har det også blitt fokus på redusert strømforbruk

Analyse av algoritmer (2)

- Mye fokus på iterative strukturer
  - Løkker og rekursjon
    - Tidsforbruk i kode som gjentas ved multiplisere seg opp
  - Prøve å flytte mest mulig kode utenfor løkke / rekursjon
- Merk at:
  - Kall av innebygde funksjoner ser ut som enkle instruksjoner
    - MEN inneholder ofte skjulte løkker
    - Gjelder f.eks
      - Funksjoner på vektorer, arrays, matriser (Søking, sum, max, sortering, …)
      - Funksjoner på strenger (for eksempel søke etter delstrenger)
  - I tillegg til valg av lur algoritme vil også valg av <u>lur datastruktur</u> være viktig

Stor theta notasjon, $\Theta(g(n))$

- Når datamengden blir stor
  - vil leddene med størst potens av n dominere
  - Konstanter og ledd med lavere potens blir relativt sett uvesentlige
- Uttrykker omtrentlig hvordan kjøretiden utvikler seg
  - som funksjon av n når n blir stor
- Ved store datamengder vil noen algoritmer lurere enn andre, for eksempel
  - Binærsøk $\Theta(\log_2 n)$ lurere enn sekvensielt søk $\Theta(n)$
  - Quicksort eller mergesort, $\Theta(n \log_2 n)$, lurere enn insertion sort $\Theta(n^2)$
- Eksponetielle algoritmer, $\Theta(c^n)$, hvor c er en konstant
  - Vil bruke veldig lang tid for store n, for eksempel n er lengde på passord:
    - $\Theta(10^n)$, knekke en numerisk pin ved å prøve alle mulige kombinasjoner
    - $\Theta(256^n)$, knekke et ASCII-basert passord, alle mulige kombinasjoner


kjøretid gitt ved to funksjoner
8000
7000
6000
5000
4000
3000
2000
1000
0
0
5
10
15
20
n
x^3
10*x^2+500

Oppsummering

- Algoritmer er ordnede sett av entydige, utførbare skritt som definerer en terminerende prosess
- Algoritmer kan representeres med pseudokode
- Algoritmer er problemløsning
- *Iterative* og *rekursive* algoritmer er alternative problemløsningsmåter
- Analyse av algoritmer kan hjelpe oss til å vurdere løsninger mhp. effektivitet og korrekthet