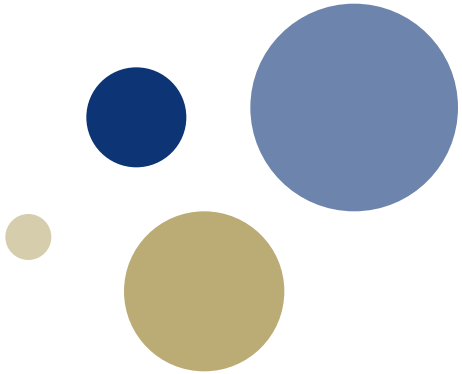
NTNU

Kunnskap for en bedre verden



Algoritmer

Teoribok: ”Algorithms”
Kap 5 fra Brookshear & Brylow:
Computer Science: An Overview

TDT 4110 IT Grunnkurs

Professor Guttorm Sindre

Læringsmål og pensum

- Mål
  - Lære om
    - Algoritme som konsept
    - Representasjon av algoritmer
    - Analyse av tidskompleksitet
    - Iterative og rekursive strukturer

- 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 rask algoritme

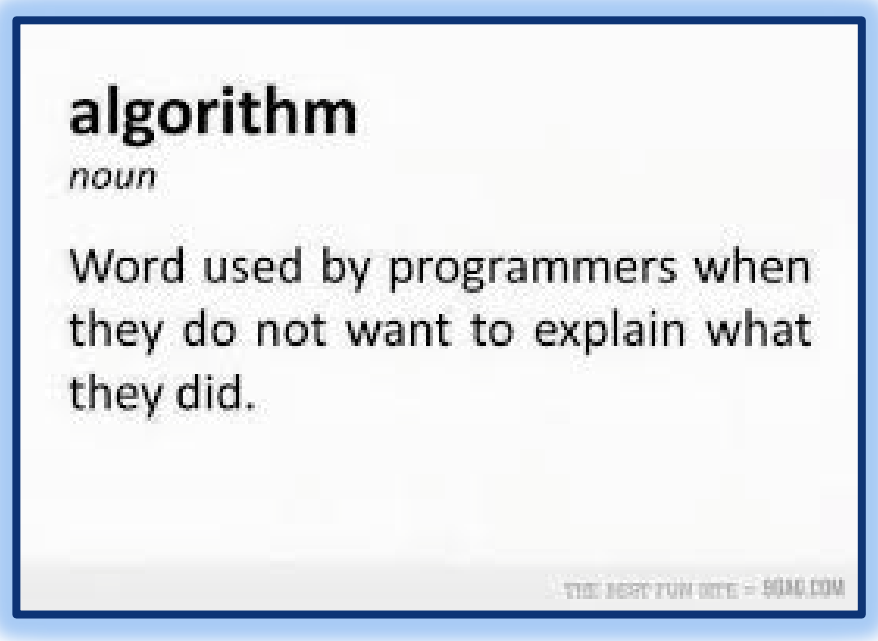
Intro:

Hva er en algoritme?
Representasjon av algoritmer
Hvordan komme på algoritmer?

Kapittel 5.1-5.3

Definisjon på algoritme

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



Representasjon av algoritmer

- Vanlig brukt: *Pseudo*kode
  - ”Naturlig språk” med visse primitiver
  - Uavhengig av spesifikke programmeringsspråk
- Gjerne likevel inspirert av prog.språk
  - Boka bruker en pseudokode som ligner på Python
- Primitiver:

Variabeltilordning
name = expression

Valg
if (condition)
    action
else
    action

Løkker
while (condition)
    activity

Funksjoner
def name(parameters)
    actions

Polyas problemløsningsfaser (t.v.)

i programmeringskontekst (t.h.)

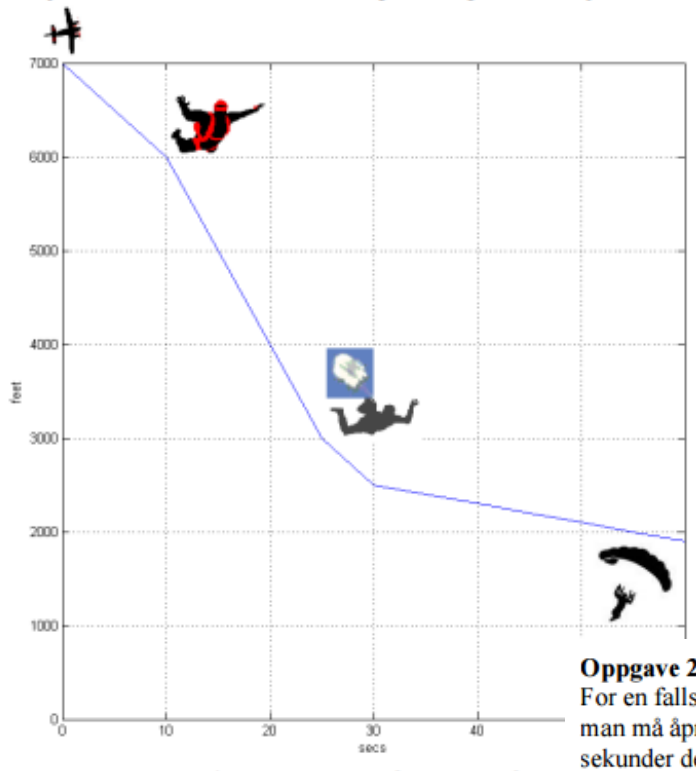
1. Forstå problemet
2. Pønsk ut en plan for å løse problemet
3. Utfør planen
4. Evaluer løsning mhp. nøyaktighet potensial for å løse andre problemer

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 nøyaktighet 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.)

- Naiv løsning
  - ”Oversette” hendelsesforløpet direkte til kode
    - Bruk av løkke per tidsenhet
  - Mange ga bare svar i hele sek.

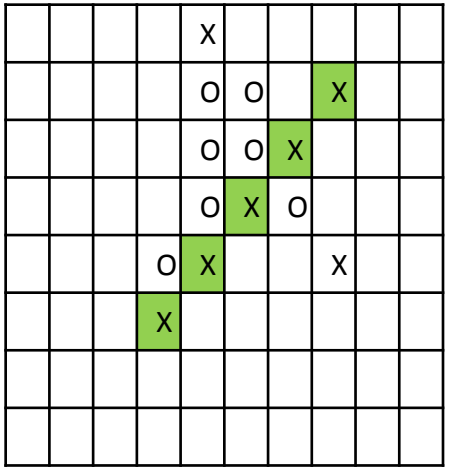
```
def feet2seconds(feet):
    sec = 0
    while feet > 3000:
        sec = sec + 1
        if sec < 10:
            feet = feet – 100
        else:
            feet = feet – 200
    return sec
```

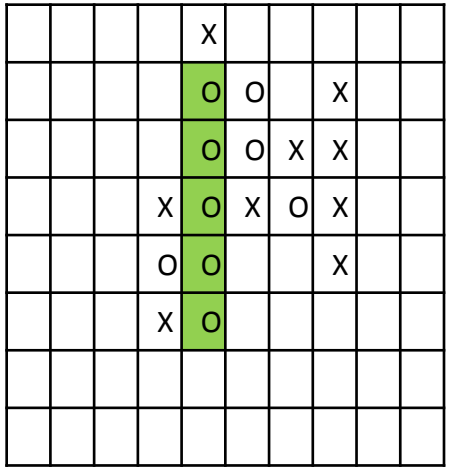
- Naiv løsning ikke best
  - Jfr. graf, funksjon m. et par knekkpunkter, ellers lineær
  - Trenger ikke simulere fallet for å regne ut tiden

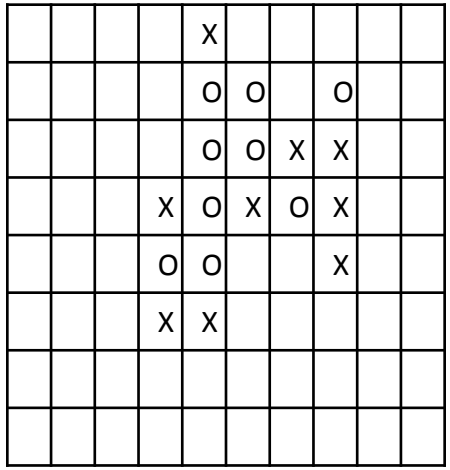
```
def feet2seconds(feet):
    if feet > 4000:
        return 10 + (feet-4000)/200
    elif feet > 3000:
        return (feet-3000)/100
    else:
        return 0
```

Annet eksempel

- Vil lage et program for å spille bondesjakk
- Etter hvert trekk: 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







Forsøk på naiv algoritme

```
count_X = 0
count_O = 0
for all rows in board
    for all cells in row
        if cell is X then
            count_O = 0
            count_X = count_X + 1
            if count_X >= 5 then winner = X
        elif cell is O then
            count_X = 0
            count_O = count_O + 1
            if count_O >= 5 then winner = O
        else
            count_X = count_O = 0
for all columns in board
    % similar inner loop and if-tests
for all diagonals in board
    % similar inner loop and if-tests
```

Dette er en dårlig algoritme, koden er unødig lang og ineffektiv.

På hvilken måte er den ineffektiv, og hvordan kan vi gjøre den bedre?

Mer effektiv variant

- Probleminnsikt: En seier er alltid resultat av siste trekk
  - Kun sjekke spilleren som trakk, og…
  - …kun en liten del av brettet, utover fra ruta for siste trekk

```
def count_equals(x, y, dx, dy, symbol):
   % counts subsequent occurrences of a given symbol (X or O)
   % from pos. x, y, in direction dx, dy, until finding a cell without this symbol
   c = 0
   while cell in pos. x, y is in the board and contains the symbol :
      c = c + 1
      x = x + dx
      y = y + dy


def victory(x, y, symbol):
   % true if player w symbol (X / O) has won the game after the move x,y – otherwise false
   v = (count_equals(x-1, y, -1, 0, symbol) + count_equals (x+1, y, 1, 0, symbol) >= 4  or
        count_equals (x, y-1, 0, -1, symbol) + count_equals (x, y+1, 0, 1, symbol) >= 4 or
        count_equals (x-1,y-1, -1, -1,symbol) + count_equals (x+1,y+1,1,1,symbol) >= 4 or
        count_equals (x-1,y+1, -1,1,symbol) + count_equals (x+1,y-1,1, -1,symbol) >= 4 )
   end
```

Forskjell på algoritmene

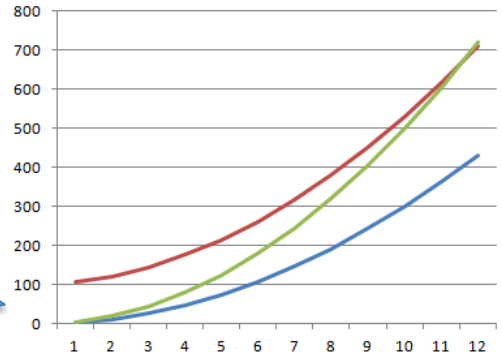
- **Den naive bondesjakkalgoritmen**
  - Sjekker (nesten) alle rutene i brettet 4 ganger
  - Brett med bredde og høyde **n**
    - kjøretiden utvikler seg kvadratisk (som funksjon av $\mathbf{n^2}$)
  - Seierssjekken vil bli tregere jo større brett vi bruker
- **Den smartere algoritmen:**
  - Sjekker maksimalt 20 ruter
  - Seierssjekken går like fort uansett hvor stort brettet er
- **Store Θ-notasjon (Big-theta):**
  - Den naive algoritmen er $\Theta(n^2)$ - tidsbruk proporsjonalt med n^2
  - Den smartere algoritmen er Θ(1) – tidsbruk uavhengig av brettstørrelse
- **Tilsvarende for fallskjermeksemplet:**
  - Den beste algoritmen er Θ(1) – tidsbruk uavhengig av fallhøyde
  - Den naive algoritmen er Θ(n) der n er høyden i fot / 200
  - Spiller ikke så stor rolle, siden det er temmelig begrenset hvor stor n kan bli

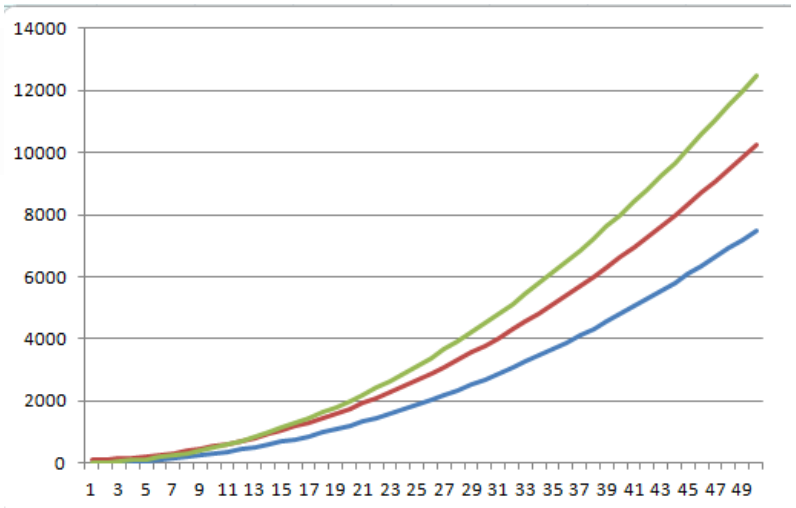
Hva betyr Θ ?

- Anta at ei algoritme sitt tidsforbruk er gitt som f(n)
  - Der 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

Illustrasjon

- Program P har tidsforbruk gitt ved $f(n) = 4n^2 + 3n + 100$
- Kan vi finne **g(n)** som kun ett ledd uten konstanter
- Og som er slik at $k_1 \cdot g(n) <= f(n) <= k_2 \cdot g(n)$?
  - NB: trenger kun være korrekt for **store n**
- JA! Funker f.eks. med
  - $g(n) = n^2$
  - $k_1 = 3$
  - $k_2 = 5$



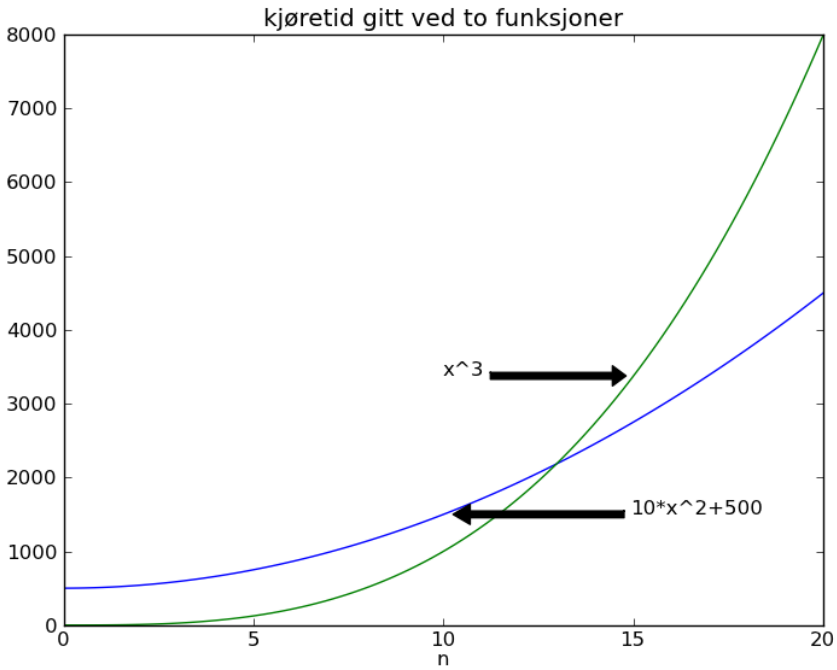


Trenger ikke stemme for små n

- Kan dermed konkludere med at P er **$\Theta(n^2)$**

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
  - Som figuren viser, en algoritme m. tidsbruk n^3 er raskere for små n, men for store n blir $10n^2+500$ fort bedre



Algoritmekompleksitet, div. eks.

- Θ (log n) Binærsøk i sorterte data, f.eks. vektor, (cell)array
- Θ (n) Sekvensielt søk i f.eks. vektor, (cell)array
- Θ (n log n) Sortering med mergesort, quicksort
  - » Nb: worst case hvis spesielt uheldig kan være n^2 for quicksort
- Θ (n^2) Sortering med insertion sort, bubblesort
- Θ (n^3) Naiv multiplikasjon av to n x n matriser
- Θ (2^n) Prøve alle mulige kombinasjoner av bits for et passord på binær form
- Θ (n!) ”Brute force” løsning av det såkalte ”travelling salesman” problemet

Passordknekking, mer eks.

- Kun små bokstaver (26 stk)
  - Lengde 4, alle mulige komb.: $26^4 = 456.976$
    - Vs. Angrep med ordliste (~500.000 ord): 500.000
  - Lengde 8, $26^8 = 208.827.064.576$
    - Angrep med ordliste: fortsatt 500.000
    - UNNGÅ BRUK AV VANLIGE ORD I PASSORD

- Hva lønner seg mest?
  - Øke lengden ytterligere?
  - Eller forlange bruk av store og små bokst., tall, spesialtegn?
  - 29^{16} = 25024647368034734878752 1
  - 100^8 = 10000000000000000
- Konklusjon: Lengde er viktigere enn rare tegn

Flere måter å spare tid på

- Implementere ”samme” algoritme på en bedre måte
  - Bl.a. flytte ut av løkker det som kan gjøres utenfor
  - For eksempel redusere tidsbruk fra 4 * n til n
  - Men har i begge tilfeller en algoritme som er Θ(n)

- Parallellisering, gjøre flere ting på en gang
  - Forutsetter maskin med flere prosessorer, de fleste har dette
  - F.eks. vektorisering og preallokering i Matlab
  - Reduserer tidsbruk, endrer vanligvis ikke kompleksitet

- Finne en lurere algoritme for å gjøre samme ting
  - F.eks. oppnå Θ(1) eller Θ(log n) i stedet for Θ(n)
  - Hvis n blir skikkelig stor, sparer dette mer enn de over

- Dele opp programmet lurt, skille fra hverandre…
  - Tidkrevende operasjoner som kan gjøres sjelden
  - Hyppige operasjoner som må gjøres raskt

Finne kompleksitet for egen kode?

- Rent sekvensielt program uten noen repeterte handlinger: Θ(1)
- Enkel løkke som går gjennom alle elementer i f.eks. vektor: Θ(n)
  - Der n er lengden til vektoren
    - NB: om løkka f.eks. bare ser på halvparten av elementene, også Θ(n)
    - Men om bare et konstant antall, f.eks. første 10 elementer: Θ(1)
- Dobbel løkke, ytre løkke går ~n ganger og indre ~n ganger: $\Theta(n^2)$

- NB: mye ser ut som enkle operasjoner, men skjuler løkker
  - F.eks. sum, max av liste, `max(L), sum(L)` - Θ(n)
    - Om ikke sum, max er forhåndslagret
  - Sortering `L.sort()` Θ(n log n)
  - Sjekk av tilstedeværelse av element m. for eksempel in
  - Jfr. f.eks. https://wiki.python.org/moin/TimeComplexity

- Hvis det er vanskelig å ”se” kompleksiteten direkte
  - prøvekjør programmet og mål tidsbruk
  - Hva skjer hvis datamengden (f.eks lengde av liste) dobles, tidobles etc.?

Iterasjon vs. Rekursjon
Søking og sortering

Kapittel 5.4-5.5

Iterasjon vs. Rekursjon

- Boka presenterer
  - Iterative algoritmer: Sekvensielt søk, sortering ved innsetting
  - Rekursiv algoritme: Binærsøk

- NB: **alle tre** *kan* programmeres iterativt – eller rekursivt
- Generelt:
  - Iterasjon – algoritmen benytter løkke(r) for stegvis løsning av problemet
  - Rekursjon – stegvis repetisjon oppnås ved at en funksjon kaller seg selv

Algoritme for sekvensielt søk (i sortert liste, s. 143 i Teoribok)

```
def Search(List, TargetValue)
    if (List is empty):
        Declare search a failure
    else:
        Select the first entry in List to be TestEntry
        while (TargetValue > TestEntry and more entries to be considered):
                Select the next entry in List as TestEntry
        if (TargetValue == TestEntry):
                Declare search a success
        else:
                Declare search a failure
```

Hvordan måtte vi endre algoritma hvis lista ikke var sortert?

Hva sparte vi ved at lista var sortert med denne algoritma?

Tidskompleksitet for sekvensielt søk?

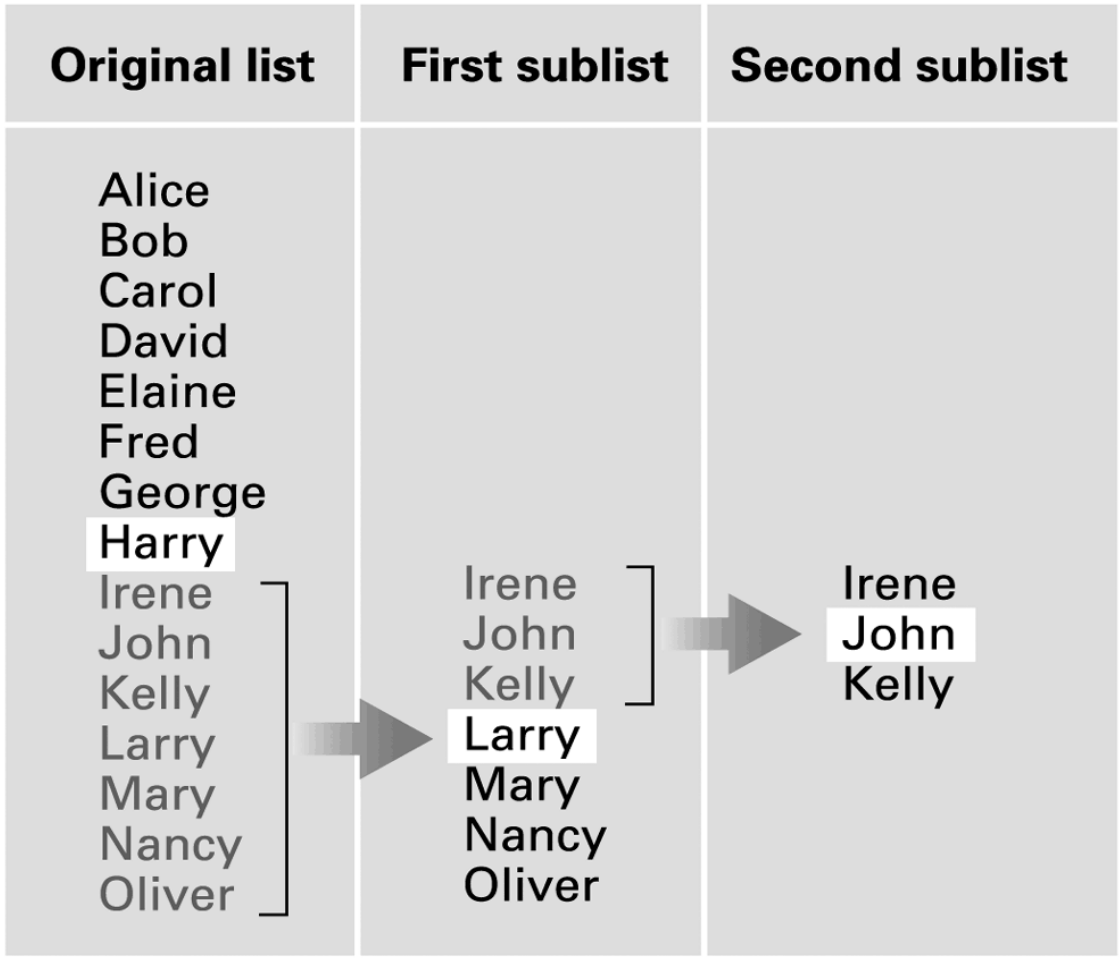
- Noen ganger er vi heldige, søkt element er tidlig i lista
- Andre ganger uheldige, sent i lista
  - Dvs. det vi leter etter kommer sent i alfabetet
- Gjennomsnittlig se på N/2 elementer, hvor N er lengde av lista

- Sekvensielt søk: $\Theta(n)$
  - Dvs. tidsforbruk proporsjonal med n
  - Hvis lista blir 5 ganger så lang, vil søkene også ta 5 ganger så lang tid

Binærsøk:
En smartere måte å søke på

- Forutsetter sortert liste
- Sekvensielt søk: se på første element, så på neste, så på neste, osv. – inntil funnet eller ikke
- Binærsøk:
  - Hvis lista er tom
    - FERDIG! (Elementet fins ikke i lista)
  - Ellers:
    - Se på midterste element
    - Hvis midterste element == søkt element:
      - **FERDIG! (Suksess: Element funnet)**
    - Ellers, hvis midterste element > søkt element:
      - Gjenta binærsøk i første halvdel av lista (foran elementet vi så på)
    - Ellers
      - Gjenta binærsøk i siste halvdel av lista (bak elementet vi så på)

Bruke strategien til å søke etter John i en liste



Binærsøk – rekursiv versjon

Adm.dir
Main

Let etter ’Foo’ i lista fra indeks 0 til 14

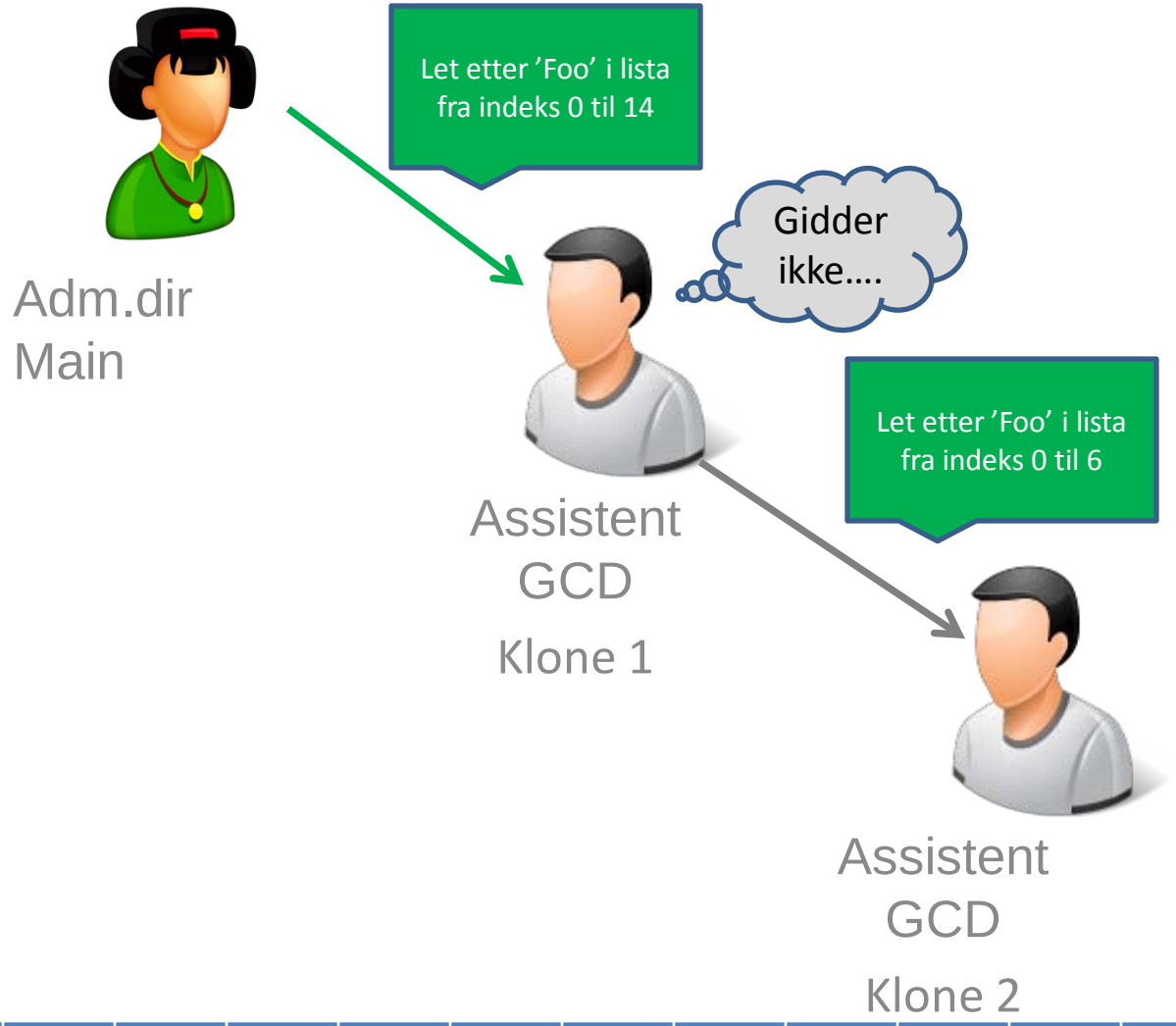
Assistent
BS

Klone 1

Er lista tom? (sluttindeks < startindeks). Nei! … Ser på det midterste elementet [7]. Nei, ikke Foo men Gil! Gidder ikke mer, lar assistenten min lete. Men siden lista er sortert og ’Foo’ er mindre enn ’Gil’, trenger han bare lete til venstre for midten… altså fra indeks 0 til 6

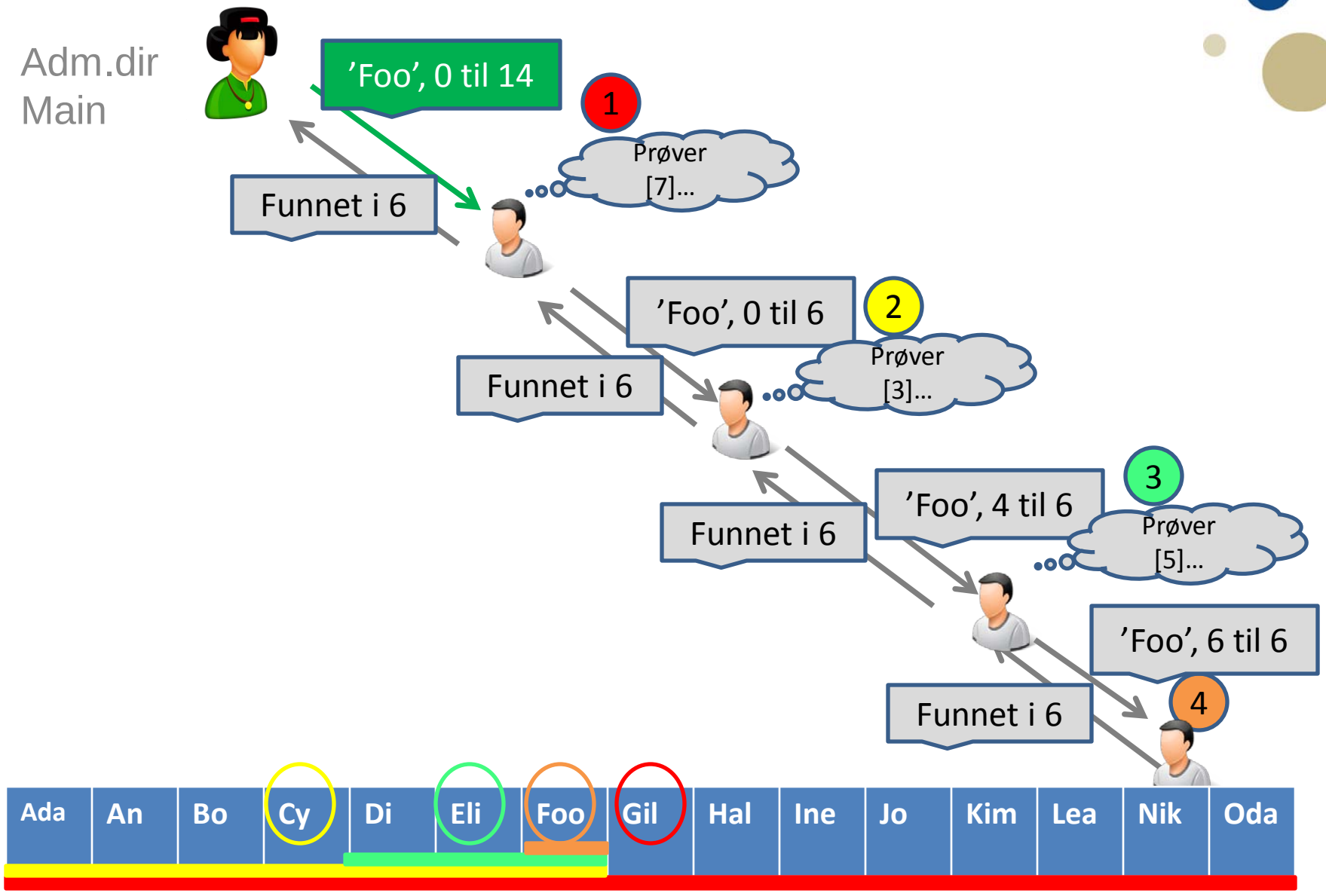
Ada	An	Bo	Cy	Di	Eli	Foo	Gil	Hal	Ine	Jo	Kim	Lea	Nik	Oda

Binærsøk – rekursiv versjon

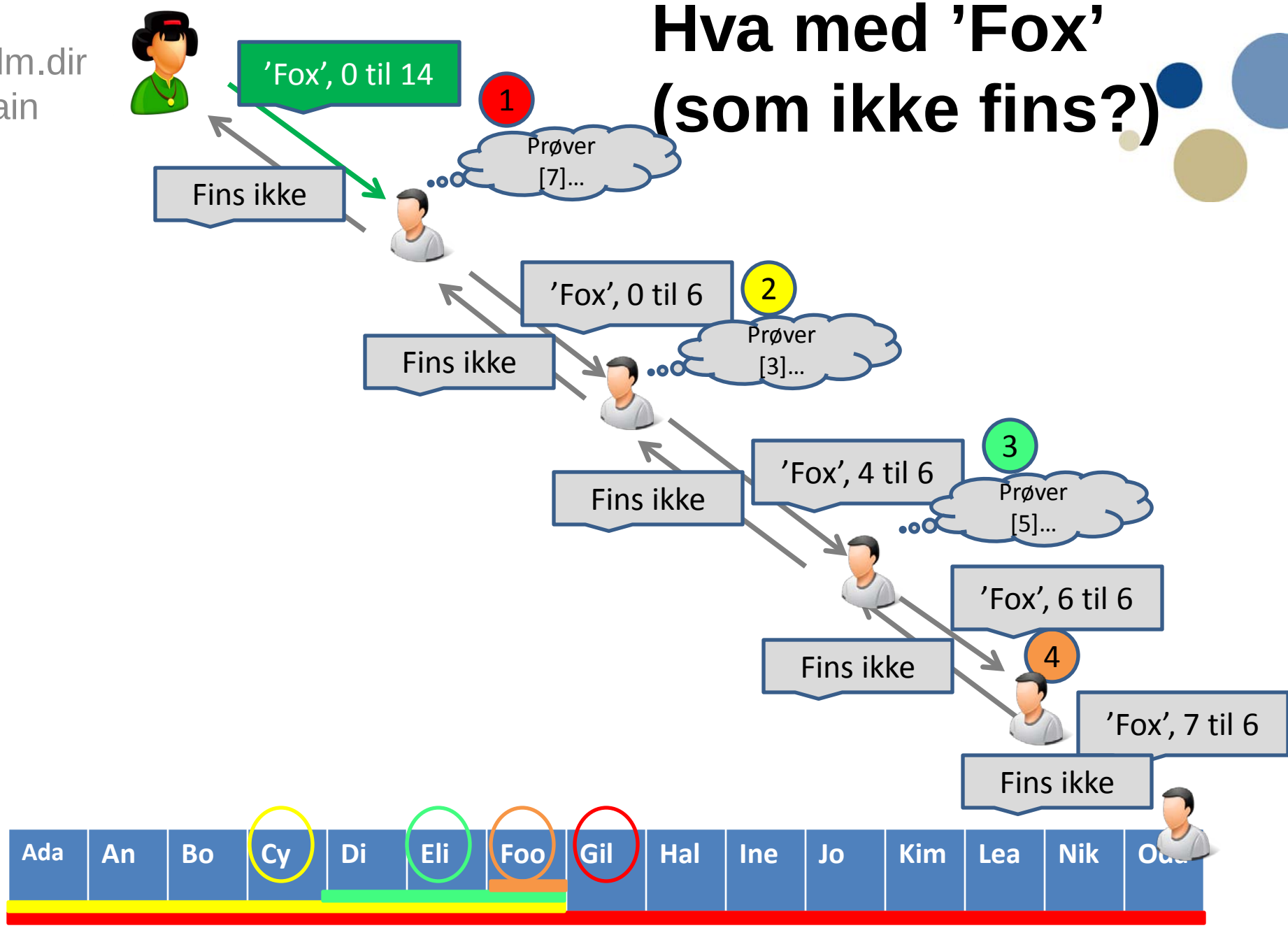


Ada	An	Bo	Cy	Di	Eli	Foo	Gil	Hal	Ine	Jo	Kim	Lea	Nik	Oda

Binærsøk – rekursiv versjon



Hva med 'Fox' (som ikke fins?)



Tidsforbruk for binærsøk?

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

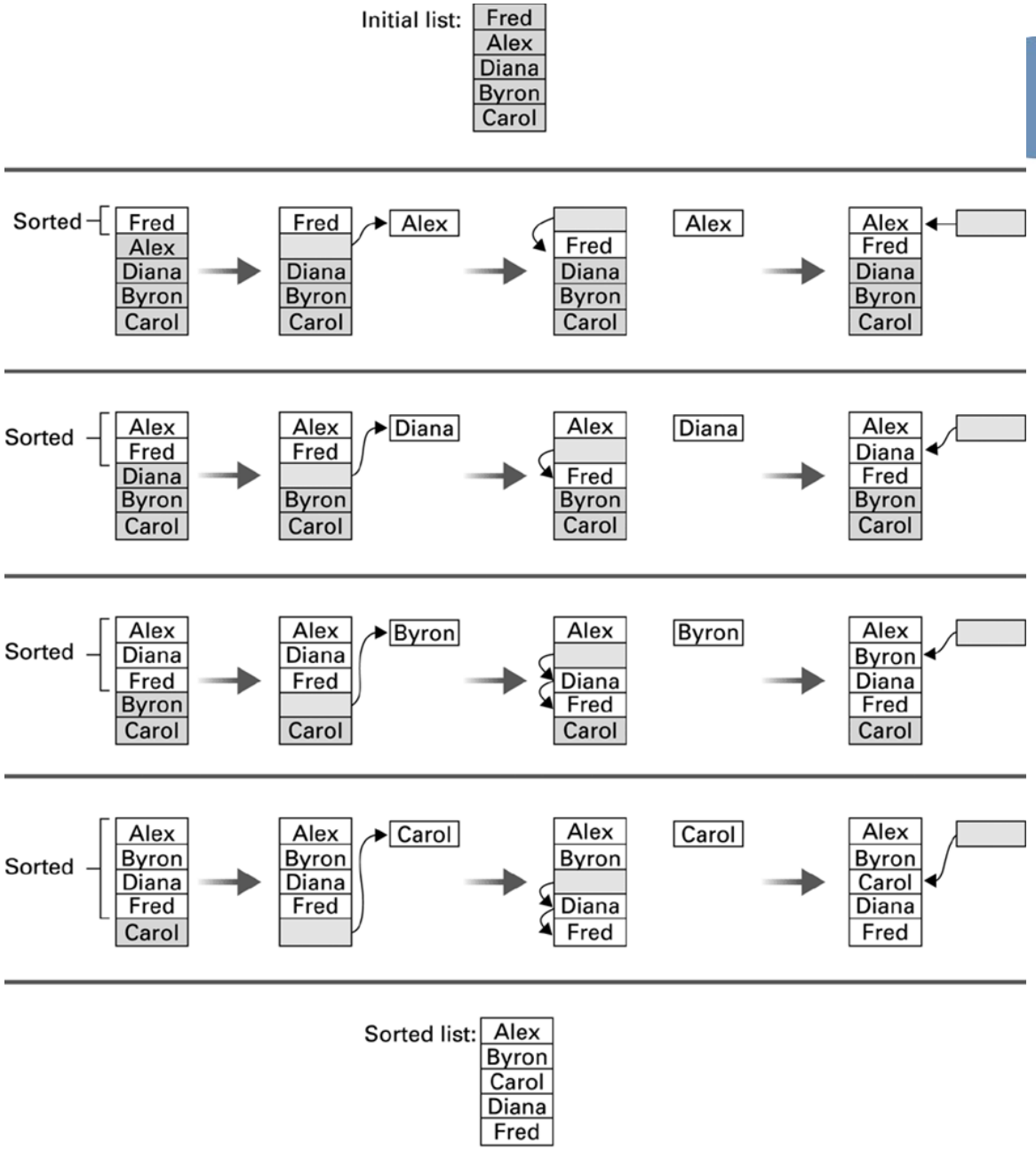
- Binærsøk: $\Theta(\log_2 n)$

- 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

Sortering: mange algoritmer

- **I pensum:**
  - Sortering ved innsetting (insertion sort)
- **Andre**:
  - Boblesortering (bubble sort)
  - Mergesort
  - Quicksort

Insertion sort Illustrasjon



Tidsforbruk for insertion sort?

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

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

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