Introduksjon

IN2010 – Algoritmer og Datastrukturer

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Høsten 2022

VELKOMMEN

Litt om algoritmer

- Hvis du gir en *presis* beskrivelse for hvordan man skal løse et *problem*, så har du beskrevet det vi kaller en algoritme
- Algoritmer er en serie med små og entydige steg
- En algoritme må
  - terminere etter et endelig antall steg
  - hvert steg må være helt presist definert
  - ta null eller flere input
  - produsere et output som står i forhold til input
- En algoritme *bør* være effektiv

Litt om datastrukturer

- En samling av verdier som følger en fast struktur utgjør en datastruktur
- Algoritmer er avhengig av gode datastrukturer for å bli enkle og raske
- Datastrukturer er avhengig av gode algoritmer for å opprettholde strukturen
- Et godt valg av datastruktur bør gjøre data vi trenger ofte lett tilgjengelig

Perspektiv

- Vi vil fokusere på å forstå, analysere, anvende og lage algoritmer
- Vi vil bygge forståelse ved å studere, og helst implementere, mange eksempler
- Vi vil analysere dem i form av å forutsi hvor effektive de er
- Dere vil anvende algoritmer når dere løser oppgaver
  - da må dere finne en algoritme som løser problemet
- Ofte vil dere ikke finne algoritmen dere leter etter, og dere må lage den selv
  - da gjelder det å studere *lignende* problemer
  - eller løse et enklere problem først

Noen råd

- Vær en god medstudent
- Dra på gruppetimer
- Skriv *litt* kode hver dag
- Finn på egne oppgaver
- Gjør en av disse om til din nye hobby
  - `https://open.kattis.com/`
  - `https://leetcode.com/`
  - `https://projecteuler.net/`
  - `https://cses.fi/problemset/`

Undervisningstilbud

- Forelesninger fredag kl. 10:15
  - Forelesningene vil foregå fysisk
  - Opptak vil publiseres etter forelesning
  - Videoene fra høsten 2020 kan brukes som ekstra læringsressurser
- Det er ukentlige gruppetimer
  - Vi har 9 enestående gruppelærer dere kan lære masse av!
- Det blir fem innleveringsoppgaver
  - Den første og siste vil ikke være obligatoriske
  - En oversikt publiseres på semestersiden

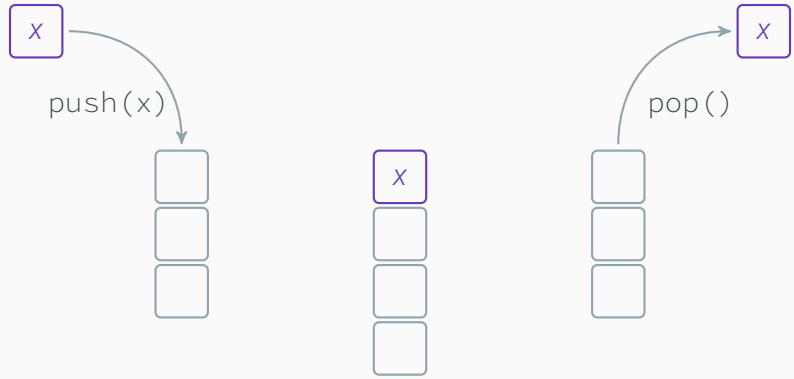
Abstrakte datatyper

Abstrakte datatyper

- En abstrakt datatype (ADT) sier om *oppførsel*, men ingenting om implementasjon
- En abstrakt datatype kan ha flere ulike konkrete implementasjoner
- En abstrakt datatype realiseres som en konkret datastruktur
- Grensesnitt (`interface`) er Java sin måte å snakke om abstrakte datatyper
- En stor del av IN2010 handler om å finne *effektive* implementasjoner for noen sentrale abstrakte datatyper

Stack

- Legg til elementer på toppen av stacken: `stack.push(x)`
- Plukk ut elementet som sist ble lagt på stacken: `stack.pop()` $\mapsto x$



- Kan for eksempel implementeres med lenkede lister eller dynamisk arrayer

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om *x* forekommer i mengden *A*: $x \in A$
  - Sett inn i mengden: `Insert(A, x)`
  - Fjern et element fra mengden: `Remove(A, x)`
  - Operasjonene union, snitt og differanse:

$A \cup B$



`Union(A,B)`

$A \cap B$



`Intersection(A,B)`

$A \setminus B$



`Difference(A,B)`

$B \setminus A$

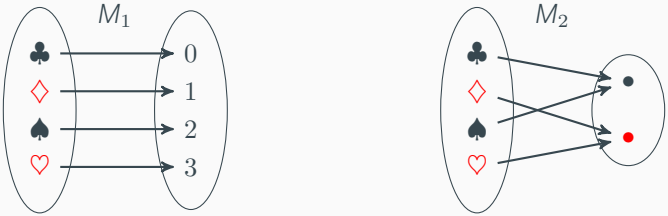


`Difference(B,A)`

- Implementeres oftest som ordnede trær (`TreeSet`) eller ved hashing (`HashSet`)
- Hverken rekkefølge eller antall forekomster noen rolle i mengder
- `OrderedSet`
  - Vi forventer å kunne gjøre akkurat de samme operasjonene som på et `Set`
  - I tillegg kommer elementene sortert ved iterering

Map

- Et `Map`, eller *ordbok* (`dict`), assosierer en nøkkel med nøyaktig én verdi



- Den abstrakte datatypen for maps krever at vi kan
  - sette inn: M_2.`put`(□, ●)
  - slå opp: M_2.`get`(□) ↦ ●
  - slette: M_2.`remove`(□)
- Implementeres oftest som ordnede trær (`TreeMap`) eller ved hashing (`HashMap`)
- `OrderedMap`
  - Vi forventer å kunne gjøre akkurat de samme operasjonene som på et `Map`
  - I tillegg kommer nøklene sortert ved iterering

Litt om pseudokode

Litt om pseudokode

- Vi bruker pseudokode for å formidle algoritmer
- Vi kan se for oss at pseudokode ligger langs en skala:



- Slideren er posisjonert *veldig omtrentlig* der vi ønsker (ta det med en klype salt)

Notasjon

- Notasjonen vi bruker inneholder:
  - Aritmetiske uttrykk — $a + b - \frac{c}{2}$
  - Sammenligninger — $a \leq b$
  - Tilordninger — $i \leftarrow 0$
  - While-løkker — `while` test `do` body
  - For-løkker — `for` $i \leftarrow 0$ `to` $n - 1$ `do` body
- Pseudokoden vi skriver skal være lett å oversette til et programmeringsspråk

Binærsøk

Søk – spesifikasjon

Algorithm: Søk (spesifikasjon)

Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner `true` ellers `false`

- Dette er en spesifikasjon for en algoritme som avgjør om et element er tilstede i et array eller ikke
- Vi har to input, et array A og et element x
- Output er `true` eller `false`, avhengig av om x er med i A eller ikke
- En algoritme som *oppfyller spesifikasjonen* må
  - terminere på et endelig antall steg
  - gi riktig svar uansett hva A og x er

Rett-frem søk – implementasjon

Algorithm: Rett frem søk

Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner `true` ellers `false`

```
1 Procedure Search(A, x)
2     for i ← 0 to |A| − 1 do
3         if A[i] = x then
4             return true
5     return false
```

- Denne algoritmen oppfyller spesifikasjonen fra forrige slide
- I *verste tilfelle* må vi løpe gjennom hele arrayet
  - Det vil si vi må gjøre $|A|$ sammenligninger
  - Her angir $|A|$ størrelsen på A
  - Vi bruker 0-indekserte arrayer
  - (Merk at boken bruker 1-indekserte arrayer)

Binærsøk – spesifikasjon

Algorithm: Binærsøk (spesifikasjon)

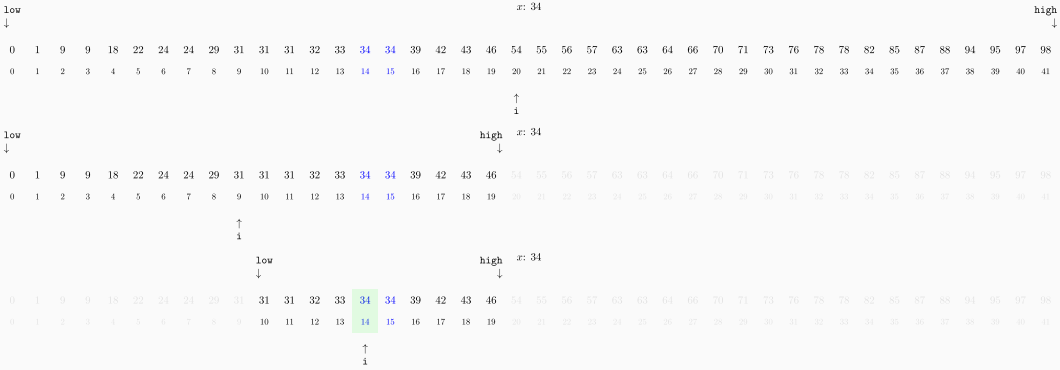
`Input:` Et ordnet array `A` og et element x
`Output:` Hvis x er i arrayet `A`, returner `true` ellers `false`

- Merk ordet *ordnet*
- Det vil si at hvis $0 \leq i \leq j < |\texttt{A}|$, så $\texttt{A[i]} \leq \texttt{A[j]}$
- Et rett-frem søk (som på forrige slide) vil fungere fint!
- Ved å anta at arrayet er ordnet, kan vi finne på noe mye lurere

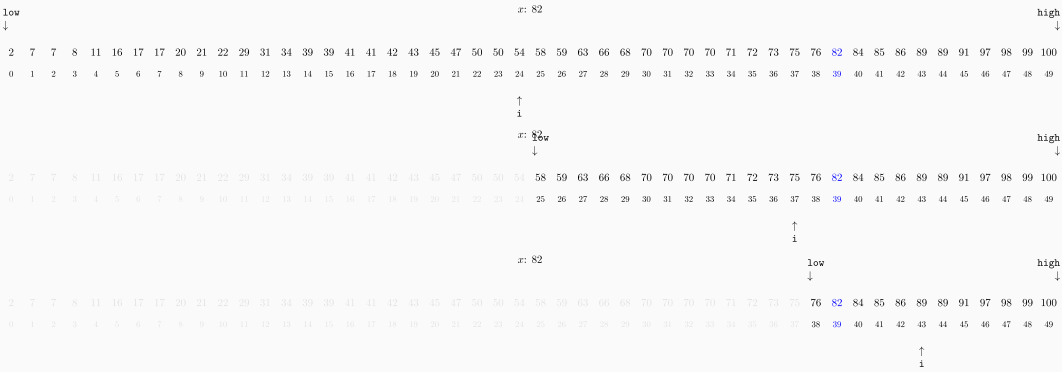
Binærsøk – idé

- Vi bruker samme idé som du helt naturlig ville brukt, dersom du skal slå opp et navn i en ordliste
- Utfordringen er å formulere dette som en presis algoritme
  - Altså oversette fremgangsmåten din til entydige steg

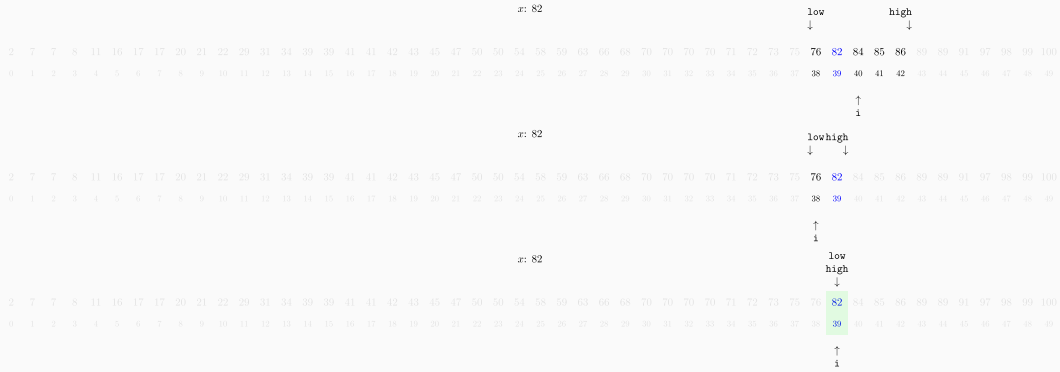
Binærsøk – eksempel 1



Binærsøk – eksempel 2



Binærsøk – eksempel 2



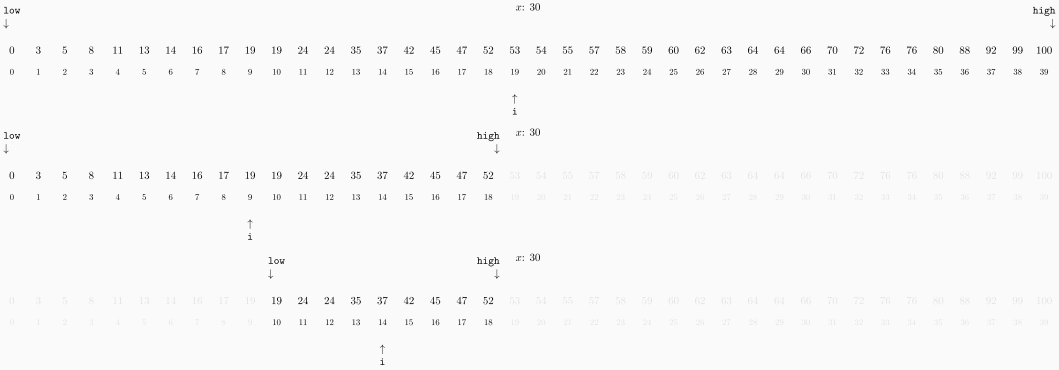
Binærsøk – implementasjon

Algorithm: Binærsøk

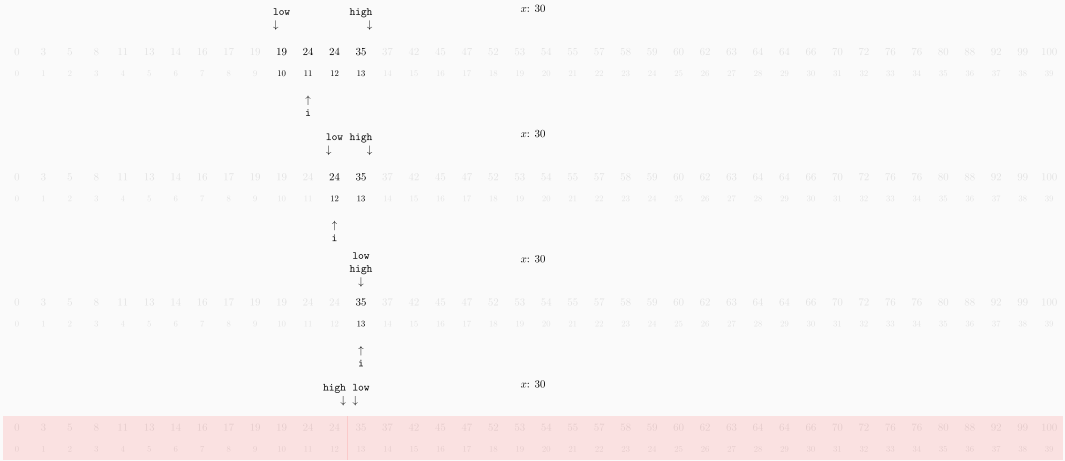
Input: Et ordnet array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
1  Procedure BinarySearch(A, x)
2      low ← 0
3      high ← |A| − 1
4      while low ≤ high do
5          i ← ⌊(low+high)/2⌋
6          if A[i] = x then
7              return true
8          else if A[i] < x then
9              low ← i + 1
10         else if A[i] > x then
11             high ← i − 1
12     return false
```

Binærsøk – eksempel 3



Binærsøk – eksempel 3



Introduksjon til kjøretidskompleksitet

Noe mer presist enn raskt

- Det er vanskelig å si om et program er *raskt* eller ikke
- Hvordan skille mellom hastigheten til programmet og til maskinen det kjører på?
  - Det *samme programmet* kjører mye tregere på en gammel datamaskin enn en ny
- I stedet for å snakke om *hastighet* snakker vi om *effektivitet*

Effektivitet og ytelse

- Vi definerer effektivitet som å *ikke gjøre unødvendig arbeid*
  - En effektiv algoritme er en algoritme som ikke gjør mange unødvendige steg
- Vi definerer ytelse som *å utføre arbeidet raskt*
  - En algoritme kan ikke ha god ytelse i seg selv
  - Den kan gi opphav til god ytelse under ulike omstendigheter
- I dette kurset bryr vi oss primært om effektivitet, og lite om ytelse
  - Det er fordi en effektiv løsning er effektiv for alltid
  - For å snakke om ytelse trenger vi vite alt om maskinvaren, kompilatoren, temperaturen i rommet, og så videre…

Kompleksitet

- Kompleksiteten til en algoritme defineres av hvor mye ressurser den bruker
- De viktigste ressursene er
  - Tid
  - Minne
- Kompleksitet defineres *relativt til størrelsen på input*
- Tid måles i antall steg algoritmen bruker i forhold til hvor stort input er
  - Vi kaller dette *kjøretidskompleksiteten* til algoritmen
- Minne måles i hvor mange elementer som lagres i samlinger
- I dette kurset fokuserer vi på kjøretidskompleksitet

Størrelsen av input

- Et input kan være stort eller lite
  - Vi kan jobbe med en kort eller lang binærstreng
  - En liste med få eller mange noder
  - Et array med få eller mange elementer
  - En mengde med få eller mange elementer
  - Et tall med lav eller høy verdi
  - ⋮
- Størrelsen på input angis konvensjonelt ved en variabel n
  - Merk at det ikke er noe spesielt med n

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$
  - Sammenligninger $a < b$
  - Aksessering på index i arrayer `A[i]`
  - Returnering `return` *a*
- De koster alle 1 i tidsbruk
- En `while`-løkke arver kostnaden fra testen og kroppen `while test do` body
  - Der kostnaden ganges med antall iterasjoner
- Tilsvarende for `for`-løkker
- Kostnaden arves også fra prosedyre- og metodekall

En enkel omskrivning

- Den samme koden som tidligere, der `for` byttes med `while`

```
1 Procedure Search(A, x)
2 |  for i ← 0 to |A| − 1 do
3 |  |  if A[i] = x then
4 |  |  |  return true
5 |  return false
```

⇝

```
1 Procedure Search(A, x)
2 |  i ← 0
3 |  while i < |A| do
4 |  |  if A[i] = x then
5 |  |  |  return true
6 |  |  i ← i + 1
7 |  return false
```

Telle steg (eksempel)

- La $n = |A|$ og anta at x *ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av n
- Variabelen i tilordnes en verdi; det utgjør 1 primitivt steg
- Testen til `while`-løkken kjøres $n + 1$ ganger
  - Hver gang gjøres det én sammenligning, altså 1 primitivt steg
- `while`-løkken vil ha n iterasjoner:
  - I `if`-testen sammenlignes `A[i]` med x
    - En aksessering og en sammenligning utgjør 2 primitive steg
  - I hver iterasjon økes i med én
    - En tilordning og en addisjon utgjør 2 primitive steg
- Til slutt returnerer vi, som utgjør 1 primitivt steg
- Samlet antall steg blir $5 \cdot n + 3$, fordi
  $1 + (n + 1) + n \cdot (2 + 2) + 1 = 3 + n + n \cdot 4 = 5 \cdot n + 3$

```
1 Procedure Search(A, x)
2     i ← 0
3     while i < |A| do
4         if A[i] = x then
5             return true
6         i ← i + 1
7     return false
```

Verste tilfelle

- Som regel ønsker vi en *verste tilfelle* analyse
  - Det vil si at vi finner det høyeste antall steg algoritmen *kan* bruke
- En analyse av gjennomsnittet er mye vanskeligere å gjennomføre
  - Vi ville trengt sannsynlighetsfordelingen for alle mulig input
- Med verste tilfelle kan vi finne trygget i at det ikke kan bli verre enn verst!
- Det viktigste spørsmålet er: Hvordan utvikler kjøretiden seg når input blir stort?

Store $\mathcal{O}$

- Vi bruker et verktøy, kalt store $\mathcal{O}$-notasjon, for å uttrykke kjøretidskompleksitet
- Intuisjonen for store $\mathcal{O}$ er å se for seg at input blir *veldig* stort
- Da tillater vi oss å gjøre alle konstanter om til 1:

$$\mathcal{O}(5 \cdot n + 3) = \mathcal{O}(n + 1)$$

- Og å stryke alt bortsett fra det *største leddet*:

$$\mathcal{O}(n + 1) = \mathcal{O}(n)$$

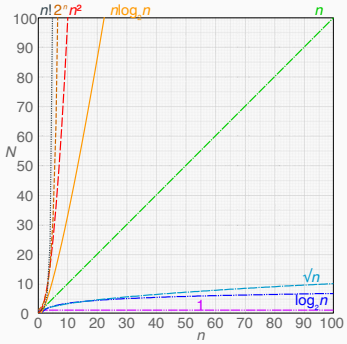
- Det lar oss regne mye grovere
- Og bevare hovedtrenden i hvordan kjøretiden vokser når input blir stor

Vanlige uttrykk for kjøretidskompleksitet

- De fleste algoritmene vi ser på i dette kurset faller inn under en av disse kategoriene

Notasjon	Uttrykk
$\mathcal{O}(1)$	Konstant tid
$\mathcal{O}(\log(n))$	Logaritmisk tid
$\mathcal{O}(n)$	Lineær tid
$\mathcal{O}(n \cdot \log(n))$	Lineæritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	
$\mathcal{O}(n^k)$	polynomiell tid
$\mathcal{O}(2^n)$	eksponensiell tid

Vanlige uttrykk for kjøretidskompleksitte (graf)



Bildet er hentet fra Wikipedia

Kodeeksempler

```
1 Procedure Constant(n)
2 |  return n · 3
```

```
1 Procedure Log(n)
2 |  i ← n
3 |  while i > 0 do
4 |  |  Constant(i)
5 |  |  i ← i/2
```

```
1 Procedure Linear(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Constant(i)
```

```
1 Procedure Linearithmic(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Log(n)
```

```
1 Procedure Quadratic(n)
2 |  for i ← 0 to n − 1 do
3 |  |  for j ← 0 to n − 1 do
4 |  |  |  Constant(i)
```

```
1 Procedure Exponential(n)
2 |  if n = 0 then
3 |  |  return
4 |  a ← Exponential(n − 1)
5 |  b ← Exponential(n − 1)
6 |  return a + b
```