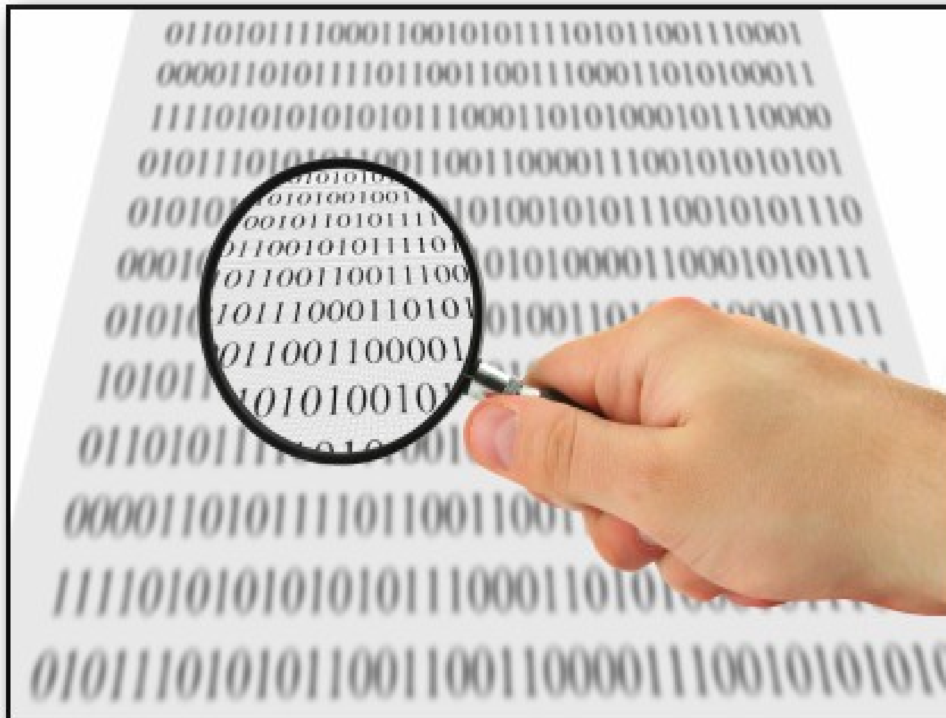
Søking



Søkeproblemet

- Gitt en datastruktur med *n* elementer:

  Finnes et bestemt element (eller en bestemt verdi) *x* lagret i datastrukturen eller ikke?

- Effektiviteten til søkealgoritmer avhenger av:
  - Om datastrukturen er sortert eller usortert
  - Om datastrukturen er bygget opp slik at den kan utnyttes til å gjøre raske søk
  - Hvor "smart" selve algoritmen er
  - Noen ganger også av dataenes verdier

Datastrukturer for rask søking

- I alg.dat. kurset skal vi senere se på:
  - Binære søketrær
  - Heap og prioritetskøer
  - Multivei søketrær og B-trær
  - Hashtabeller
- I denne delen om søking holder vi oss til:
  - Usorterte arrayer
  - Sorterte arrayer

Hvor raskt klarer vi å søke?

- Må alltid gjøre minst ett “oppslag” i data-strukturen bare for å se om *x* er lagret på en bestemt “posisjon”
- Hvis vi ikke har noe ekstra informasjon om dataene som kan brukes til å effektivisere søket: $O(n)$
- Hvis dataene er *sorterte*: $O(\log n)$
- Hvis dataene er sorterte og verdiene er *jevnt fordelt*: $O(\log(\log n))$
- Hvis vi *vet mer* om dataene og lagrer dem i en “smart” datastruktur *kan* søking være helt ned i $O(1)$

Søking i arrayer – forenkling

- Læreboka bruker *generiske* metoder i Java som kan søke i arrayer som inneholder “alt”, så lenge dataene er `Comparable`
- For å fokusere på algoritmene og effektiviteten, og ikke på Java, forenkler vi søkeproblemet til:
  - Arrayer som bare innholder heltall
  - Boolske metoder som søker etter et tall *x* i en array, og som returnerer:
    - `true` hvis *x* finnes i arrayen
    - `false` ellers

Sekvensielt / lineært søk

- Hvis en datastruktur ikke er sortert, og vi ikke har noe ekstra informasjon om dataene som kan utnyttes, er sekvensielt søk det beste vi klarer å få til
- Starter med første element og søker etter *x*, et og et element om gangen, inntil vi enten finner *x* eller har gått gjennom alle *n* elementer: *O(n)*
- Kan implementeres for alle datastrukturer som tilbyr en standard *iterator* for å gå gjennom alle elementer
- Java-kode: `searchAlgorithms.java`

Søking i sorterte arrayer

- Sekvensielt søk er “dumt”:
  - Vi “spør” hvert element om det er lik *x*
  - Jfr. gjetteleken “20 spørsmål”
- Hvis vi *vet* noe mer om dataene, kan dette utnyttes til å lage *mye* raskere algoritmer
- Noen søkealgoritmer for *sorterte* arrayer:
  - Binærsøk
  - Ternært søk (oppgave)
  - Interpolasjonssøk

Binærsøk i (stigende) sortert array

- Sammenligner verdien *x* vi søker etter med elementet *midt i* arrayen
- Hvis *x* er *lik* midtre element er søket ferdig
- Hvis *x* er *mindre enn* midtre element søker vi videre på samme måte i *nedre* halvdel av array
- Hvis *x* er *større enn* midtre element søker vi videre på samme måte i *øvre* halvdel av array
- Binærsøk bruker "smarte spørsmål": Vi "kvitter oss med" halvparten av de mulige gjenværende elementene i hvert steg

Binærsøk: Eksempel, $x = 22$, $n = 17$

2 6 10 14 22 27 28 29 35 40 50 63 77 82 88 93 99

2 6 10 14 22 27 28 29 35 40 50 63 77 82 88 93 99

2 6 10 14 22 27 28 29 35 40 50 63 77 82 88 93 99

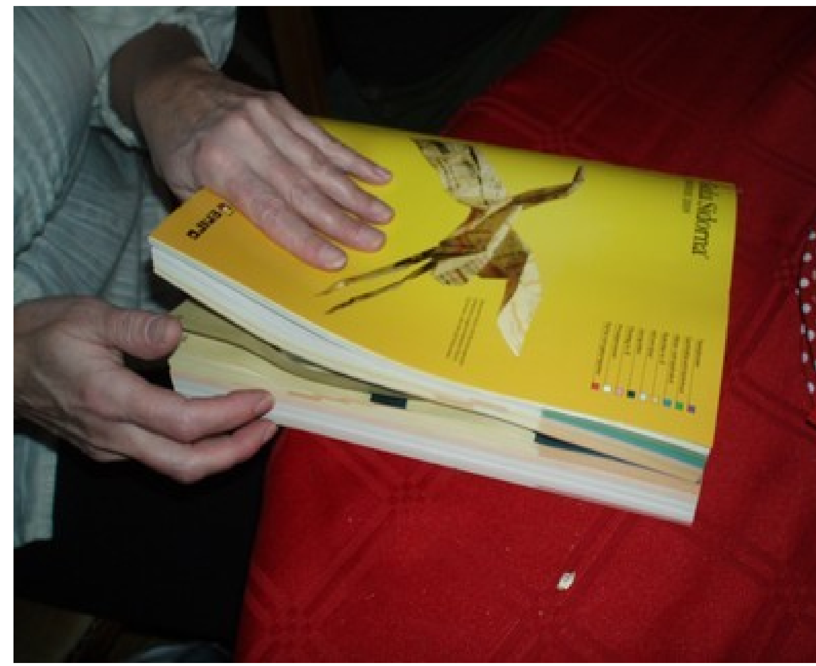
2 6 10 14 22 27 28 29 35 40 50 63 77 82 88 93 99

Binærsøk: Effektivitet og implementasjon

- Effektivitet for array av lengde *n*:
  - Worst case: Elementet vi søker finnes ikke i arrayen
  - Antall steg er lik antall ganger *n* er delelig med 2
  - Konklusjon: Binærsøk er $O(\log n)$ – “supereffektivt”
- Implementasjon:
  - Iterativt: Med en løkke der vi hele tiden oppdaterer nedre, øvre og midtre indeks for den delen av arrayen der søkt element kan befinne seg
  - Rekursivt: Enklere kode, mindre effektivt (oppgave)
- Java-kode: `searchAlgorithms.java`

Litt smartere: Interpolasjonssøk

- Eksempel: Leter etter “B”, i en telefonkatalog fra A-Å på papir
- Binærsøk: Åpner katalogen på midten, deretter på 1/4 av sidene etc.
- “Menneskelig søk”: Åpner katalogen omtrent der vi *tror* “B” befinner seg



- Interpolasjonssøk: Etterligner “menneskelig søk”, prøver å *beregne* hvor søkt element antagelig ligger lagret

Interpolasjonssøk: Eksempel

- *A* sortert array med heltall, $n = 50$, $x = 180$
- $A[0] = 101$, $A[49] = 200$
- Hvis tallene i *A* er noenlunde jevnt fordelt, kan vi anta at verdien 180 ligger på ca. 80% av lengden på arrayen:

  (180 – 101 + 1) / (200 -101 + 1) = 0.8

- I stedet for å dele *A* i to like store halvdeler, deler vi i den indeksen der vi *antar* at 180 ligger:

  (50 · 0.8) - 1 = 40 - 1 = 39

- Fortsetter å dele på samme måte inntil ferdig
- Java-kode: `searchAlgorithms.java`

Binærsøk vs. interpolasjonssøk

- Binærsøk:
  - Deler alltid i to like store deler, alltid $O(\log n)$
- Interpolasjonssøk:
  - Deler også i to deler, men på det stedet der vi *forventer* at søkt element ligger, hvis verdiene er noenlunde jevnt fordelt
  - Kan bevises: $O(\log(\log n))$ hvis jevn fordeling av verdier
- Sammenligning, $n = 1\,000\,000$
  - Binærsøk: $O(\log n) \approx 20$
  - Interpolasjonssøk: $O(\log(\log n)) \approx 4$
- Testprogram: `searchTest.java`

Arrayer som datastruktur: Effektivitet

- Usortert array:
  - Innsetting: $O(1)$
  - Fjerning: $O(n)$
  - Søking: $O(n)$
  - OK bare for *små* datamengder
- Sortert array
  - Innsetting: $O(n)$
  - Fjerning: $O(n)$
  - Søking: $O(\log n)$
  - OK for *statiske* datasett med lite fjerning/innsetting

- For generelle problemer med mye data må vi lagre dataene i strukturer som er raskere enn en array
- Trenger datastrukturer der operasjonene søking, fjerning og innsetting alle er $O(\log n)$ eller bedre