Oppsummering

IN2010 – Algoritmer og Datastrukturer

Lars Tveito

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

Høsten 2024

Oppsummering

Hovedtemaene vi har sett på

- Binærsøk
- Trær
- Binære søketrær
- Selvbalanserte søketrær
- Binære heaps
- Sortering
- Grafer

Temaer fremover

- Litt mer grafer
- Hashing
- Litt om beregnbarhet og kompleksitet

Felles for alle temaer

- Vi bryr oss om kjøretidskompleksitet
  - Dette er effektivitet «i det store bildet»
- Verktøyet vi bruker for å snakke om kjøretidskompleksitet er $\mathcal{O}$*-notasjon*
- Vi bryr oss *ikke* om spesifikke programmeringsspråk
  - Vi snakker heller om generelle teknikker for å løse problemer
  - En annen måte å si det samme er at vi heller snakker om *algoritmer*
  - Disse teknikkene kan overføres til et hvilket som helst programmeringsspråk
- Verktøyet vi bruker for å beskrive algoritmer er *pseudokode*

Litt om O-notasjon

O-notasjon er relativt

- Det er vanskelig å snakke om den konkrete kjøretiden til et program
  - Hvor rask er datamaskinen?
  - Hvor effektiv maskinkode genererer kompilatoren?
  - Hva annet kjøres på datamaskinen i øyeblikket programmet kjører?
  - Hvilket tiår er vi i?
- Den konkrete kjøretiden er et *eksperiment*
  - Man kan trekke verdifull informasjon fra et eksperiment
  - Men det er skummelt å trekke slutninger på bakgrunn av et eksperiment
- O-notasjon lar oss si noe mer generelt
  - Kjøretidskompleksiteten til en algoritme var den samme i 1960 som i 2021
- O-notasjon gir oss dette gennom å si noe *relativt*, og ikke absolutt
  - Konkret kjøretid er absolutt
  - O-notasjon *relaterer* størrelsen på input og antall steg

O-notasjon handler om vekst

- En algoritme tar alltid utgangspunkt i null eller flere input
  - og produserer et output som er relatert til input
- Spørsmålet vi er interessert i er
  - hva er et øvre estimat for hvor lang tid algoritmen bruker
  - der vi kan skalere tiden med en konstant
- Konstanten kan fange opp mange ulike momenter
  - for eksempel hastigheten på maskinen og antall instruksjoner

Eksempel: 50-tallet vs. 2020

- Anta at vi kjører på en UNIVAC I (1951)
  - Den kan kjøre ca. 2000 instruksjoner per sekund
  - Den utfører et binærsøk på et sortert array med n elementer
- Anta at vi kjører på en Raspberry Pi
  - Den kan kjøre ca. 4000000000 (altså $4 \cdot 10^9$) instruksjoner per sekund
  - Det vil si den er to millioner ganger raskere enn UNIVAC I!
  - Den utfører et rett-frem søk på et sortert array med n elementer
- Vi antar at binærsøket kan gjøres med $50 \cdot \log_2(n)$ instruksjoner
- Vi antar at rett-frem søket kan gjøres med $20 \cdot n$ instruksjoner
- Hvor stor må n være før du heller vil beregne på en maskin fra 50-tallet?
  - Ca. 150000000, altså 150 millioner (se graf)

Hva er poenget med O-notasjon?

- Eksempelet fra forrige slide illustrerer hvordan O-notasjon sier noe grunnleggende
  - Kjøretidskompleksiteten er relevant, selv i møte med 70 år med teknologisk utvikling
- I eksemplet har vi store konstantfaktorer, der vi ser at

$$\frac{50}{2000} \cdot \log_2(n) < \frac{20}{4 \cdot 10^9} \cdot n$$

  bare vi lar n bli stor nok (ca. 150 millioner)
- Det sier noe om hvorfor $\mathcal{O}(\log(n))$ er *vesentlig* forskjellig fra $\mathcal{O}(n)$
- Vi kunne fortalt samme historie dersom
  - UNIVAC I kjørte et program i $\mathcal{O}(n)$
  - og Raspberry Pi kjørte et program i $\mathcal{O}(n^2)$
  - Da ville UNIVAC I tatt igjen Raspberry Pi når n ble større enn 5 millioner

Hvorfor O-notasjon er enkelt

- O-notasjon lar oss *ignorere* konstantfaktorer
- O-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

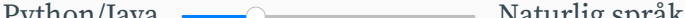
$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n\log(n))$	logaritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	
$\mathcal{O}(n^k)$	polynomiell tid
$\mathcal{O}(2^n)$	eksponensiell tid

- Og gjøre dette for hvert input
  - (i tilfellet hvor vi har en algoritme som er avhengig av mer enn ett input)

Litt om pseudokode

Pseudokode er formidling

- Pseudokode er *ikke et læringsmål i seg selv* for IN2010
- Det er et verktøy for å kunne formidle egen forståelse
- Pseudokoden er god dersom en (kvalifisert) leser forstår den
- 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)
- Gode eksempler finnes i slidene fra forelesning, boka og wikipedia

Pseudokode – Dos & don'ts

Gjør dette

- Bruk innrykk
- Bruk kontrollstrukturer
  - `if`, `while`, etc.
- Bruk fornuftige variabelnavn
  - evt. forklar navnene i teksten rundt
- Vær konsekvent
- Vær presis
- Vær konsis

Ikke gjør dette

- Ikke importer biblioteker
  - bruk *antagelser* i stedet
- Ikke vær *vag*
- Ikke vær språkspesifikk
  - Slicing i Python
  - `System.out.println`

Pseudokode – Bittelitt om navngivning

- Vi foretrekker ofte (men ikke alltid) konsise variabelnavn
  - I konteksten av IN2010 er det fordi vi ofte jobber med veldig generelle ting
- Her er to heuristikker for variabelnavn:
  1. Lengden på navnet bør være proporsjonalt med størrelsen av skopet
  2. Lengden på navnet bør være proporsjonalt med antall antagelser om variabelen
- Variabelnavnet x er veldig fint dersom prosedyren fungerer uansett hva x er
- Variabelnavnet s er veldig fint dersom prosedyren fungerer på alle strenger
- Variabelnavnene i, j, k er gode navn på tellere
- Verdien av lange variabelnavn dukker opp når
  - det er viktig informasjon å formidle om verdien som er bundet til variabelen
  - dersom variabelen brukes «langt unna» der den ble definert