Programmering og Modellering (PoM)
Ugeseddel 9 — Uge 45 — Deadline Ingen aflevering

Kim Steenstrup Pedersen, Katrine Hommelhoff Jensen, Knud Henriksen,
Mossa Merhi og Hans Jacob T. Stephensen

30. oktober 2014

1 Plan for ugen

Denne uge tager vi fat på et nyt emne: Algoritmer og beregningskompleksitet. De fleste kender til problematikken at et program, som kører fint på een computer, opfører sig yderst problematisk på en anden. Det kunne være et videofremvisningsprogram, der på den ene computer gør som det forventes - afspiller videoen - men på den anden hakker og viser billederne i en grov opløsning. Når programmer kører uoptimalt er det naturligt at undersøge ressourcerne - er det en kraftig nok CPU? Måske mangler der hukommelse? Det kunne også være man skulle anvende et andet styresystem? Faktisk kunne jeg godt bruge en ny computer....o.s.v. Lad os nu sige, at to forskellige programmer med den samme funktion klarer sig vidt forskelligt på den *samme* computer. Det kunne på den ene side have noget at gøre med, hvordan de to programmer bruger ressourcerne på computeren, men det kunne på den anden siden også have noget at gøre med hvor *effektivt* de to programmer udfører deres opgave. Dette svarer til at stille spørgsmålene:

- Hvor mange gange bliver CPU'en bedt om at udføre en instruktion?
- Hvor meget lagerplads bruger programmet, og hvor ofte tilgås det?

Disse spøgsmål er helt centrale i algoritmisk beregningskompleksitet, der handler om at sammenligne algoritmer på *ide-niveau* - at fastsætte kompleksiteten af en algoritmerne *uafhængigt* af programmeringssprog, hardware og CPU instruktionssæt. I kompleksitets- eller køretidsanalyse udledes antallet af *beregningsskridt* i en algoritme, hvilket kan være f.eks. tildeling af en værdi til en variabel, opslag i en liste eller en aritmetisk operation. I kompleksitetsanalyse ser man på antallet af beregningsskridt som funktion af størrelsen af inddata og tillader derved at estimere en algoritmes faktiske tidsforbrug, når tiden for de enkelte instruktioner er kendt: Hvis køretiden for en algoritme er 1 sekund når størrelsen af inddata er 1000, hvad er den så når størrelsen af inddata fordobles? Vil den være det samme, det dobbelte eller f.eks. fire gange længere? Det kunne også lyde: Hvis en algoritme til en webapplikation virker fint med 1000 brugere, hvor godt virker den så for 2000? Dertil kan man give et bud på, hvordan køretiden for en algoritme kan nedsættes hvis metoden modificeres således at antallet af instruktioner reduceres. Den overordnede kompleksitet af en algoritme, som udledes af det mest dominerende beregningsled, kan på en simpel måde forbindes med den faktiske køretid og er et meget vigtigt værktøj i udvikling og programmering.

Til torsdag:

Læs: Guttag kap. 9 (Algorithmic complexity) og 10 (Simple algorithms).
Til forelæsningen gennemgås:

- Køretidsanalyse
- Asymptotisk notation
- Klasser af kompleksiteter
- Eksempler på algoritmer og deres beregningskompleksitet

Bemærk: Der er ingen obligatorisk afleveringsopgave i denne uge og ingen forelæsning og øvelser tirsdag. Torsdag foregår alle øvelsestimerne hos Mossa.

1.1 Torsdagsøvelser

Besvarelser af disse opgaver skal ikke afleveres, men opgaverne forventes løst inden tirsdag i efterfølgende uge.

9to1 De følgende funktioner indikerer køretiden, eller antallet af instruktioner, for problemstørrelse n, udført af en algoritme. Bestem det dominerende led for hver algoritme og klassificer den asymptotiske kompleksitet med store-O notation:

- $f(n) = 7n + 45$
- $f(n) = 324$
- $f(n) = n^2 + 3n + 45$
- $f(n) = n^3 + 2500n + 100$
- $f(n) = n + \sqrt{n}$
- $f(n) = 2\log(n) + \log(\log(n))$

9to2 Den følgende funktion `hasDuplicates` tager en liste `A` som inddata og returnerer `True` hvis listen indeholder mindst eet duplikat, ellers `False`:

```
def hasDuplicates(A):
    n = len(A)
    for i in range(n):
        for j in range(n):
            if i != j and A[i] == A[j]:
                return True
    return False
```

Lad n være længden af `A`. Foretag en kompleksitetsanalyse af `hasDuplicates`:

1. Udled køretiden som funktion af n. Forklar de enkelte led.
2. Bestem den asymptotiske kompleksitet $O(?)$.

9to3 Algoritme A har køretiden n^2 og algoritme B har køretiden $\frac{1}{2}n^2 + \frac{1}{2}n$, for problemstørrelse n. For hvilke problemstørrelser har A en bedre køretid end B, og for hvilke har B en bedre køretid end A?

9to4 Betragt følgende linier kode:

```
a=5
b=6
c=10
for i in range(n):
   for j in range(n):
      x = i * i
      y = j * j
      z = i * j
for k in range(n):
   w = a*k + 45
   v = b*b
d = 33
```

Uden at kende noget videre til kodens funktion eller n, foretag da en kompleksitetsanalyse af koden:

1. Udled køretiden som funktion af n. Forklar de enkelte led.
2. Bestem den asymptotiske kompleksitet $O(?)$.

9to5 Den følgende funktion `isAnagram` tager to strenge `str1` og `str2` og returnerer `True` hvis den ene streng er et anagram af den anden, ellers `False`:

```
def isAnagram(str1, str2):
    str2list = list(str2)
    pos1 = 0
    res = True
    while pos1 < len(str1) and res:
        pos2 = 0
        found = False
        while pos2 < len(str2list) and not found:
            if str1[pos1] == str2list[pos2]:
                found = True
            else:
                pos2 = pos2 + 1
        if found:
            str2list[pos2] = None
        else:
            res = False
        pos1 = pos1 + 1
    return res
```

Lad begge strenge have længden n. Foretag en kompleksitetsanalyse af `isAnagram`:

1. Udled køretiden som funktion af n. Forklar de enkelte led.
2. Bestem den asymptotiske kompleksitet $O(?)$.

Hvad er best-case og worst-case beregningskompleksiteten? Kan man fastsætte en average-case beregningskompleksitet?

9to6 Den følgende funktion `integerToString` konverterer et positivt heltal `intNum` af længden n til en streng.

```
def integerToString(intNum):
     digits = '0123456789'
     if intNum == 0:
          return '0'
     res = ''
     while intNum > 0:
          res = digits[intNum%10] + res
          intNum = intNum/10
     return res
```

Foretag en kompleksitetsanalyse af `integerToString`:

1. Udled køretiden som funktion af n. Forklar de enkelte led.
2. Bestem den asymptotiske kompleksitet $O(?)$.

9to7 Vi vil nu studere en algoritme `maximumSum` der finder den største sum af enhver mulig delsekvens af en liste af heltal. Dette problem er trivielt, hvis hele listen udelukkende indeholder positive heltal - nemlig listen selv. Hvis listen derimod indeholder en blanding af positive og negative tal, er løsningen ikke helt så oplagt. Som et eksempel vil delsekvensen med den største sum af talsekvensen $[2, 1, -4, 10, 15, -2, 22, -8, 5]$ være [10,15,-2,22], hvis sum er 45. `maximumSum` kan skrives som:

```
def maximumSum(intList):
    n = len(intList)
    maxSum = 0
    for i in range(n):
        for j in range(i, n):
            curSum = 0
            for k in range(i, j+1):
                curSum = curSum + intList[k]
            if curSum > maxSum:
                maxSum = curSum
    return maxSum
```

Denne implementation har den asymptotiske beregningskompleksitet $O(n^3)$. En af problemerne med implementationen er, at beregningerne af mange delsummer gentages unødigt. For eksempel beregner vi summen af index 3 til 5

```
intList[3] + intList[4] + intList[5]
```

og senere summen af index 3 til 6

```
intList[3] + intList[4] + intList[5] + intList[6]
```

hvorved summen af index 3 til 5 beregnes to gange. Med denne observation, skriv en ny version af `maximumSum` der forbedrer beregningskompleksiteten til at være $O(n^2)$.

9to8 Udfordring: Skriv en version af `maximumSum` ovenfor, der forbedrer beregningskompleksiteten til at være $O(n)$.