Miniprojekt 1:

Store-O notation og kompleksitet af algoritmer

I dette miniprojekt interesserer vi os alene for tidskompleksitet. Kompleksitet af hukommelse behandles ikke. Vi benytter matematikprogrammet Maple.

På siden `http://first.math.aau.dk/dan/software/maple/` findes information om installation og brug af Maple.

Screencast 2 (og 3) viser hvordan man anvender Maple i nogle eksempler.

Løkker og kompleksitet

Betragt følgende legetøjs-algoritme, som du kan kopiere ind i Maple:

```
forfour:=proc(n::integer)
local i,j,k,l;
local a;
a:=0;
for i from 1 to n do
 for j from 1 to n do
  for k from 1 to n do
   for l from 1 to n do
    a:=a+1;
   od;
  od;
 od;
od;
return a;
end proc;
```

Man kan nu benytte kommandoen forfour med et vilkårligt input af formen et ikke-negativt heltal. Eksempelvis giver

```
forfour(0)
```

svaret 0, og

```
forfour(7)
```

giver svaret 2401.

Opgave 1:
Betragt algoritmen forfour.

- *Vis, at worst-case-kompleksiteten er $\mathcal{O}(n^4)$.*
- *Vis, at gennemsnits-kompleksiteten er $\mathcal{O}(n^4)$.*

Hvis man vil vide, hvor lang tid Maple bruger på at regne en given ting ud, da benyttes kommandoen "time()".

Opgave 2:

- *Indtast i Maple time(forfour(20)). Maple returner den forbrugte tid. Klik på kommandolinien 10 gange og beregn gennemssnitstidsforbruget.*
- *Gentag med time(forfour(40))*
- *Gentag til sidst med time(forfour(80))*
- *Eftervis, at køretiden bliver cirka 16 gange så stor hver gang du fordobler problemstørrelsen (Altså lader input vokse som: $20 \to 40 \to 80$).*
- *Forklar, hvordan det hænger sammen med kompleksitetsestimaterne.*

Vi modificerer nu algoritmen forfour sådan at dele af algoritmen ikke altid udføres.

```
forfourrand:=proc(n::integer)
local i,j,k,l,dice;
local a;
global b,c;
a:=0;
dice:=rand(1..10);
b:=dice();
c:=dice();
for i from 1 to n do
 if not b=2 then
  for j from 1 to n do
   for k from 1 to n do
    if not c=2 then
     for l from 1 to n do
      a:=a+1;
     od;
     else a:=7;
    fi;
   od;
  od;
 fi;
od;
return a;
end proc;
```

Programlinien

```
dice:=rand(1..10);
```

definerer en tilfældighedsgenerator, som returnerer værdier mellem 1 og 10. Kaldene:

```
b:=dice();
c:=dice();
```

tildeler variablerne b og c tilfældige værdier mellem 1 og 10. Vi ser, at algoritmen stopper meget hurtigere end forfour-algoritmen, hvis b er tildelt værdien 2. Den stopper også hurtigere end forfour-algoritmen hvis b er tildelt en værdi forskellig fra 2; men c er tildelt værdien 2.

Opgave 3:

- *Find worst-case-kompleksiteten af "forfourrand".*
- *Find gennemsnits-kompleksiteten af "forfourrand".*
- *Test din viden om kompleksiteterne ved hjælp af kommandoen "time()".*

Til sidst modificerer vi "forfour" på en anden vis:

```
forfourvild:=proc(n::integer)
local i,j,k,l;
local a;
a:=1;
for i from 1 to n do
 for j from 1 to n do
  for k from 1 to n do
   for l from 1 to n do
    a:=2*a;
   od;
  od;
 od;
od;
return a;
end proc;
```

Opgave 4:
I denne opgave testes algoritmen "forfourvild". Kompleksitetsberegningerne viser sig ikke at give så meget mening, indtil vi opdager hvorfor.

- *Indse, at kompleksitetsberegningerne fra opgave 1 stadig holder.*
- *Udfør tests som i opgave 2, men med meget mindre problemstørrelser. Går Maple i kredsløb om månen, så stop processen ved at klikke på "stop"-knappen.*
- *Indse, at kompleksitetsberegninerne ikke siger noget om tidsforbruget fordi de tal der ganges med 2 bliver større og større undervejs i algoritmen. Alt andet lige tager det længere tid at gange 2 med 2000009045 end med 56.*
- *Man kan forfine kompleksitetsmodellen, så den tager hensyn til ovenstående problem. En måde er at tælle binære operationer i stedet for operation i $\mathbb{Z}$. Dette behandles dog ikke i dette miniprojekt.*

Beregning af determinant

I den resterende del af miniprojektet arbejdes der med determinanter af $n \times n$ matricer

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}.$$

Husk fra lineær-algebra-kurset, at du har lært to metoder til beregning af determinanten. Disse metoder omtaltes i det følgende som Metode 1 og Metode 2.

Metode 1: Lad A være en $n \times n$ matrix.

- Hvis $n = 1$, så gælder $A = [a_{11}]$ og vi sætter $\det A = a_{11}$.
- Hvis $n \geq 2$, så definerer vi A_{ij} til at være den matrix, der fremkommer ved at fjerne række i og søjle j fra A. (Dette er en $(n-1) \times (n-1)$ matrix). Vi har:

  $$\det A = (-1)^{1+1} a_{11} \det A_{11} + (-1)^{1+2} a_{12} \det A_{12} + \cdots + (-1)^{1+n} a_{1n} \det A_{1n}. \tag{1}$$

  Dette svarer som bekendt til at udvikle efter første række.

Opgave 5:
Brug Metode 1 til at finde $\det\left(\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{bmatrix}\right)$ *og til at finde* $\det\left(\begin{bmatrix} 0 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{bmatrix}\right)$

Anvendes Metode 1 på en 2×2 matrix A (altså $n = 2$), da fås

$$\det A = a_{11} a_{22} - a_{12} a_{21}. \tag{2}$$

Anvendes Metode 1 på en 3×3 matrix A (altså $n = 3$), da fås

$$\det A = a_{11}a_{22}a_{33} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31}. \tag{3}$$

Opgave 6.
Tjek efter, at ovenstående udregninger (formlerne (??) og (??) er en sum af $n!$ led, der hver især er $+/-$ et produkt af n elementer.

Sætning
For generel $n \times n$ matrix A gælder der, at Metode 1 svarer til en systematisk sum af $n!$ led, der hver især er $+/-$ et produkt af n elementer.

Opgave 7:

- *Vis ved hjælp af ovenstående sætning, at Metode 1 har en worst case kompleksitet på $\mathcal{O}\big((n!)n\big)$.*
- *Vis, at dette betyder, at worst case kompleksiteten er $\mathcal{O}\big((n+1)!\big)$.*
- *Vis, at worst case kompleksiteten af Metode 1 er $\Theta\big((n!)n\big)$.*

Metode 2:
Bring A på trappeform (ikke nødvendigvis på *reduceret* trappeform) ved hjælp af Gauss elimination, hvor alene følgende to operationer anvendes:

R1: "$\mathbf{r}_i \leftrightarrow \mathbf{r}_j$" (for $i \neq j$). Altså rækkeombytning.

R2: "$\mathbf{r}_i + c\mathbf{r}_j \rightarrow \mathbf{r}_i$" (for $i \neq j$). Altså læg konstant gange en række til en anden række.

Lad s være antallet af foretagne operationer af typen R1. lad B være trappeformen. Der gælder da:

$$\det A = (-1)^s b_{11}b_{22}\cdots b_{nn}. \tag{4}$$

Opgave 8:
I denne opgave estimeres kompleksiteten af Metode 2.

- *Vis, at worst-case-kompleksiteten af Gauss-eliminationen i Metode 2 er $\mathcal{O}(n^3)$.*
- *Vi kan antage, at Gauss-eliminationen højst benytter n række-ombytninger (operationer af typen R1). Hvorfor?*
- *Når Gauss-eliminationen er fuldført udregnes højresiden af (??). Vis, at dene udregning højst kræver $\mathcal{O}(n)$ operationer.*
- *Vis, at worst-case-kompleksiteten af Metode 2 er $\mathcal{O}(n^3)$.*

Vi sammenligner nu Metode 1 og Metode 2. Vi ser bort fra de ukendte konstanter, der ligger gemt i udtrykkene $\mathcal{O}\big((n!)n\big)$ og $\mathcal{O}(n^3)$. Eller mere præcist, lad os for enkelthedens skyld sige, at Metode 1 kræver $(n!)n$ operationer og at Metode 2 kræver n^3 operationer.

Opgave 9:
Hvis en Computer udfører $1000000000 = 10^9$ *operationer i sekundet. Hvor lang tid tager Metode 1 og Metode 2 for en* $n \times n$ *matrix med:*

- $n = 20$*?*
- $n = 21$*?*
- $n = 22$*?*
- *Hvorfor giver det mening, at ignorere konstanter i udtrykkene* $\mathcal{O}\big((n!)n\big)$ *og* $\mathcal{O}(n^3)$*?*

Bemærkning:
I ovenstående analyse af kompleksiteten af Metode 1 og Metode 2 har vi alene regnet antallet af multiplikationer og additioner i $\mathbb{R}$. Vi har set bort fra, at tallene undervejs i udregningerne kan eksplodere, når en lang række af tal ganges sammen. Skal man tage hensyn til dette i sin analyse af Metode 1 versus Metode 2, så bliver det mere indviklet. Man kan vise, at Metode 2 stadig har fornuftig kompleksitet, mens Metode 1 selvfølgelig blot får endnu værre kompleksitet.