Studieretningsprojekt 2022

Elev id: 3x 06
Elev: Balder Westergaard Holst

Fag:	Vejleder:
Matematik A	Jens Christian Larsen Tlf.: 26279734 Email: jl@soroeakademi.dk
Informatik C	Kristian Kjeldgaard Hoppe Tlf.: 28825466 Email: kh@soroeakademi.dk

Område:
Store-O notation og sorteringsalgoritmer

Opgaveformulering:
Gør rede for kompleksitetsklasserne givet ved Store-O notation, idet du giver eksempler på repræsentanter af forskellige klasser.

Beskriv sorteringsproblemer generelt, og forklar algoritmerne for insertion- og merge-sort. Vis at insertion-sort har en værste-tilfælde-udførelse på $O(n^2)$.

Implementer algoritmerne insertion- og merge-sort i Python og undersøg den faktiske udførelsestid af de to algoritmer på lister af tilfældige tal af forskellig størrelse. Vurder algoritmernes udførelsestid.

Indfør begrebet træ, og bevis relevante sætninger om træer, især højden af et træ. Bevis, at en vilkårlig sorteringsalgoritme baseret på sammenligning, vil være nedadtil begrænset af i dens værste-tilfælde-udførelsestid med $n \cdot \lg(n)$ (Stirlings formel må antages).

Insertion-sort har en bedste-tilfælde-udførelsestid på $O(n)$: Hvorfor er det ikke et modbevis til sætningen om at den nedre grænse for sorteringsalgoritmer er $n \cdot \lg(n)$?

Omfang 15-20 ns.

Sorteringsalgoritmer og O-notation

Matematik A og Informatik C

Vejledere: Jens Christian Larsen og Kristian Kjeldgaard Hoppe

Balder Westergaard Holst

28. marts 2022

Indhold

Figurer

1 | Indledning

Dette er min indledning pt.

2 Resume

Dette er hvad jeg har skrevet og fundet ud af.
Kompleksitetsklasser

3 Algoritmers Udførelsestid

3.1 Tid som Funktion af Argumentets Kardinalitet

Dette afsnit bygger primært på side 24 og 25 i bogen "*Algoritmer og Datastrukturer*" [2].

Når vi analyserer algoritmer, er det primære formål at skabe udsagn, der kort og præcist beskriver algoritmers opførsel. I vores tilfælde er vi interesserede i algoritmens udførselsestid. Vi lader $\mathcal{I}_n$ være mængden af mulige argumenter til algoritmen, n være kardinaliteten (størrelsen) af $\mathcal{I}_n$ og I være en instans af $\mathcal{I}$ (altså $I \in \mathcal{I}_n$). Algoritmens udførselsestid T når kan nu beskrives som funktion af instansen $T(I)$. Vi kan bruge dette til at beskrive tre egenskaber af algoritmens udførselsestid: algoritmens maksimale udførselsestid, algoritmens minimale udførselsestid og algoritmens gennemsnitlige køretid. Den maksimale udførselsestid er udførselsestid ved det værste tilfælde ($T(I_{vrst})$), den minimale udførselsestid kan modsat beskrives som køretiden på det bedste tilfælde ($T(I_{bedst})$), og den gennemsnitlige udførselsestid kan findes ved at tage gennemsnittet af alle udførselsestiderne ($\frac{1}{|\mathcal{I}|}\sum_{I\in\mathcal{I}_n} T(I)$). Med disse udtryk kan vi nu opskrive algoritmens mulige udførselsestider som funktion af kardinaliteten n af $\mathcal{I}_n$ således:

$$T(n) = \begin{cases} T(I_{vrst}) : I_{vrst} \in \mathcal{I}_n & \text{værste tilfælde} \\ T(I_{bedst}) : I_{bedst} \in \mathcal{I}_n & \text{bedste tilfælde} \\ \frac{1}{|\mathcal{I}_n|}\sum_{I\in\mathcal{I}_n} T(I) & \text{i gennemsnit} \end{cases}$$

Alle disse størrelser beskriver et aspekt af algoritmens udførselsestid, dog er den mest interessante udførselsestiden i værste tilfælde. Det er samme koncept som når postvæsnet siger at det vil tage tre bankdage for din pakke at nå frem, selvom det godt kan ske hurtigere. Det værste tilfælde *garanterer* at hvornår algoritmen er færdig. Dette værste tilfælde kan sammen med det bedste tilfælde bestemme variansen på udførselsestiden. Hvis en algoritme viser stor varians i dens udførselsestid kan gennemsnitsudførselsestiden sige noget om hvordan man kan forvente at algoritmen opfører sig.

Denne form for algoritmeanalyse er dog ikke helt problemfri: for det første kan vi aldrig definere nogen fast forskrift for $T(n)$, der ville gøre det muligt at beregne den præcise udførselsestid. Dette kan tildels forklares af Allan Turings "halt problem"[kilde] hvor han viste at man aldrig kan vide om en algoritme stopper eller ender i en uendelig lykke. Problemet med at kunne bestemme en køretid for algoritmen er at man så ville vide at algoritmen stoppede før den havde kørt. En anden forflaring er at alle computere vil køre den samme algoritme i forskellig hastighed pga. varians i computerens komponenter. Derudover vil selv den samme computer aldrig udføre en handling i præcis samme tidsinterval, da andre processer på computeren kan bruge dele af computerens ydeevne, eller fordi at cpu'ens temperatur ændre sig. Der er mange faktorer der bestemme hvor lang tid det tager en algoritme at fuldføre, hvilket gør det meget svært at denne form for analyse praktisk.

Uanset hvilken forklaring man bruger, er det svært at sige meget om den faktiske køretid for en algoritme før man har kørt algoritmen. Derfor er det nemmere, og mere brugbart at forfolde sig til udførselsestidens vækstrate og algoritmens opførsel når n bliver meget stort. Denne form for analyse hedder store-O-analyse.

$$O(f(n)) = g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)$$
$$\Omega(f(n)) = g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \geq c \cdot f(n)$$
$$\Theta(f(n)) = O(f(n)) \cup \Omega(f(n))$$

Figur 3.1: Definition af $O(f(n))$, $\Omega(f(n))$ og $\Theta(f(n))$ [2, s. 26].

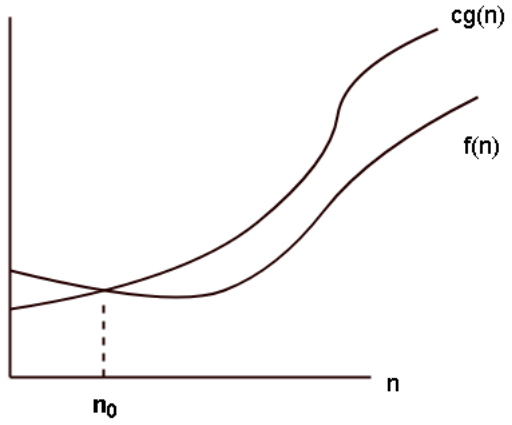
3.2 Store-O-Analyse

I store-O-analyse forholder vi os til hvordan udførselsestidens stiger, når n bliver meget stort. I denne form for analyse indeler vi grupper efter deres vækstrate. En funktions vækstrate beskriver hvor hurtigt en funktion stiger i forhold til andre funktioner ved meget høje n-værdier. Hvis en funktion $f(n)$ har en højere vækstrate end funktionen $g(x)$, betyder det at funktionen $f(x)$ altid vil være større end $g(x)$ ved store nok n. Det har ingenbetydning om $f(n)$ er mindre end $g(n)$ ved små n, da $f(n)$ altid vil overstige $g(n)$ pga. dens højere vækstrate. Matematisk kan man sige at an funktion $f(n))$ har en højere vækstrate end $g(n)$, hvis n $f(n) \geq c \cdot g(n)$ (hvor c er en konstant, og n er tilstrækkeligt stort). Hvis funktionerne har *samme* vækstrate gælder det at $c \leq \frac{f(n)}{g(n)} \leq d$ [skal jeg have den med?] (hvor d er en konstant, og det samme gælder for n og c som før). I praksis betyder dette at funktioner som n^2 og $n^2 + 3n - 10$ har den samme vækstrate og, at den er højere end en funktion som $n \cdot log(n)$. Vi kan nu bruge disse regler til at definere store-O-notattion (se figur **??**). Lad os begynde med $O(f(n))$. En funktion er del af mængden $O(f(n))$, hvis den kan sættes ind som $g(n)$. Med den matematiske definition menes der: $g(n)$ skal være sådan at (:) der findes en værdi c, som er større end 0, sådan at (:) der findes et positivt helt tal n_0, sådan at der for alle n større end n_0, gælder at $g(n)$ er mindre eller lig $c \cdot f(n)$, dvs. $g(n)$ har mindre eller samme vækstrate som $f(n))$. I praksis betyder dette, at der altid vil være en positiv n_0, hvorefter $f(n)$ altid vil være højere end eller lig $g(n)$ (se figur **??**). Man kan også tænke mængden $O(f(n))$ som mængden af funktioner der *ikke vokser hurtigere end* $f(n)$.

Når man siger at en algoritme er $O(n^2)$, menes der altså at vækstraten for $T(n)$ er en del af mængden $O(n^2)$, og at $T(n)$ i værste tilfælde har samme vækstrate som n^2.

$\Omega(f(n))$ indeholder modsat $O(f(n))$ alle funktioner der vokser mindst lige så hurtigt som $f(n)$.

Der er dog en vækstrate som både $O(f(n))$ og $\Omega(f(n))$ indeholder: nemlig vækstraten af $f(n)$.For at beskrive *præcis* denne mængde funktioner bruger vi $\Theta(n)$. Hvis en algoritme er $\Theta(n^3)$ betyder det altså at $T(n)$ *altid* vil have samme vækstrate som n^3. Det ville også betyde at alle input af samme kardinalitet vil resultere lige lang udførselsestid.



Figur 3.2: Her ses skæringspunktet n_0, hvorefter $cg(n)$ altid er højere end $f(n)$. Altså vækstraten af $cg(n)$ højere end vækstraten af $f(n)$ [1].

4 Sorteringsalgoritmer

4.1 Hvad er sortering?

Sortering er helt lavpraktisk at sætte en mængde data en rækkefølge på baggrund af dataens attributter. Det kunne f.eks. være alfabetisk eller efter farve eller størrelse. Det er dog ikke entydigt hvilken sorterings-strategi der ville være hurtigst. Disse sorterings-strategier kan også kaldes sorterings-algoritmer, det er ikke kun mennesker der kan benytte sorterings-algoritmer til f.eks. at sortere kort, men computere kan også, og de gør det for det meste også hurtigere. De næste to afsnit omhandler to forskellige sorterings-algoritmer, og deres måde at sortere en liste med tal.

Herfra vil jeg kun forholde mig til sortering af lister med tal, og sortere dem på baggrund af deres størrelse (se figur **??**)

$$[4,2,5,3,1] \longrightarrow [1,2,3,4,5]$$

Figur 4.1: Eksempel på sortering af en liste

4.2 Insertionsort

Insertionsort er en af de mere simple sorteringsalgoritmer. Pseudokoden til algoritmen kan ses i figur ??
Det er ikke en tilfældighed at denne algoritme hedder insertionsort. Den fungerer nemlig ved at gennemgå gennemgå hvert element i listen, og placere det hvor det passer ind i de elementer der allerede er sorterede. Dette er nok til dels den måde man f.eks. ville sortere sin hånd i Uno eller 500.

4.2.1 Insertionsort Procedure

At algoritmen gennemgår alle elementerne i listen kronologisk kan vi se allerede i linje 2, da algoritmen her begynder med en for-lykke, der tæller for hvert element i listen, dog starter den ved 2. element i listen. For at gøre koden mere læsbar sættes elementet som algoritmen er nået til ind i variablen e i linje 3. Det næste algoritmen gør, er at checke om elementet har en mindre værdi end det første element i listen, hvis dette er sandt rykkes alt før elementet et tak til højre, og elementet sættes ind først i listen (se figur ??).

Forklaring	Eksempel	Linje
Liste hvor `e < liste[0]`	[5,7,9,4,6,5,2]	5
Rykker til højre	[5,5,7,9,6,5,2]	6-8
Sætter elementet ind først	[4,5,7,9,6,5,2]	9

Figur 4.2: Procedure for at indsætte element først i listen. Elementet er markeret med rød, og tal der på allerede er sorterede er markeret med blå. Linjer refererer til koden i figur **??**.

Vi er nu nået til linje 11. Hvis ikke elementet er mindre end det første element i listen sker følgende (se figur ??): Algoritmen ser på tallet før elementet og spørger: "er dette tal større end elementet". Hvis det er, rykkes det hen på elementets plads. Herefter stiller algoritmen samme spørgsmål, til tallet to pladser før elementet. Hvis dette tal er mindre and elementet placerer algoritmen elementet efter dette tal, hvis ikke stiller den samme spørgsmål til tallet 3 pladser før elementet og så videre. Det er vigtigt at pointere, at der altid vil være et tal der er mindre end elementet, da elementet ellers ville være blevet placeret først i liste af den første del af algoritmenen (linje 5-10).

Forklaring	Eksempel	Linje
Liste hvor `e >= liste[0]`	[4,5,7,9,6,5,2]	11
Da $9 \geq 6$ rykkes tallet til højre	[4,5,7,9,9,5,2]	13-14
Da $7 \geq 6$ rykkes tallet til højre	[4,5,7,7,9,5,2]	13-14
Da $5 \ngeq 6$ sættes elementet ind efter 5	[4,5,6,7,9,5,2]	13 og 17

Figur 4.3: Procedure for at indsætte et element hvor det passer i den sorterede liste. Elementet er markeret med rød, og tal der på allerede er sorterede er markeret med blå. Linjer refererer til figur **??**.

4.2.2 Egenskaber af Insertionsort

Insertionsort er en god sorteringsalgoritme, idet at proceduren er forholdsvis nem at forstå, men også at listen til venstre for det element algoritmen er nået til, altid vil være sorteret. Hvis man stopper en insertionsort-algoritme halvvejs gennem dens køretid, vil man altid have en liste hvor den første halvdel er sorteret og den anden halvdel usorteret. Dette er ikke en egenskal i den næste algoritme.

(se afsnit ?? for store-O analysen)

```
funktion insertionsort(liste) {
   for i = 1 til i = n {           # her er n længden af listen
   e = liste[i]                    # dette er elementet i listen

      if e < liste[0] {            # hvis elementet er større end det første element i listen
         for j = j til j = 0{
            liste[j] = liste[j-1]  # ryk elementerne på pladserne 0 til j et tak frem
         }
      liste[0] = e                 # sætter dette element forrest i listen
      }
      else{
         j = i                     # j er en nu tæller der starter på i
         while liste[j-1] > e {    # kør mens liste[j-1] er størren end elementet
            liste[j] = liste[j-1]  # ryk liste[j] et tak til højre
            j -= 1
         }
         liste[j] = element        # indsæt elementet hvor det passer ind
      }
   }
   return(liste)                   # returnerer den sorterede liste
}
```

Figur 4.4: Pseudokode til insertionsort [2, s. 104].

4.3 Mergesort

Hvor insertionsort er en lykke-baseret algoritme, er mergesort rekursiv idet at den ikke benytter en lykke til at gentage instruktioner, men kalder sig selv i stedet. Dette leder til en fraktallignende sorteringsmetode, der deler problemet i mindre og mindre bidder.

4.3.1 Mergesort Procedure

Pseudokode til denne algoritme kan findes i figur **??**.

Mergesortalgoritmen består af to funktioner. Lad os begynde med funktionen $merge$, da det er den mest simple. Funktionen tager to lister (a og b) som argumenter, og fletter dem sammen til en tredje liste (c) (se figur **??**). Denne c liste dog ikke altid sorteret. Eksempelvis kunne dette være en returneret liste:

$$merge([1,2,4,8],[9,5,6,4,7]) \longrightarrow [1,2,4,8,9,5,6,4,7]$$

Her er resultatet en delvist sorteret liste bestående af flere sorterede dele. Altså kan vi ikke nøjes med $merge$ for at sortere en liste. En vigtig egenskab af $merge$ er at, den kan flette to sorterede lister sammen, til én samlet liste. Det kunne for eksempel være i dette tilfælde:

$$merge([1,3,4,8],[2,3,6,7,7]) \longrightarrow [1,2,3,3,4,6,7,7,8]$$

Her er begge argumenter (a og b) sorterede lister, der flettes sammen til én samlet sorteret liste.

Mergesorts kompleksitet kommer dog først rigtigt til udtryk, når vi tager hele algoritmen i betragtning. Mergesortfunktionen tager en enkel liste som argument, og checker først om listen kun indeholet ét element. Hvis den gør, så returnerer den bare listen uændret. Dette giver intuitivt mening, da en liste med et enkelt element altid vil være sorteret. Hvis listen er *mere* end 1 lang, gør den det, som får hele algoritmen til at fungere: den deler listen på i to, og kalder sig selv på hver halvdel. Herefter sættes de to dele sammen igen af $merge$-funktionen. Det er dog ikke helt så enkelt som det lyder, idet at $mergesort$ rekursivt kalder sig selv. Det leder til at algoritmen splitter listen op igen og igen, indtil der kun er et element i de mange lister. Herefter samler $merge$ alle de små lister til en stor liste, hvor $merge$ løbende sørger for at de lister

Forklaring	a	b	c	Linje
Da $a[0] \leq b[0]$ føjes $a[0]$ til c	[3,6,6]	[4,5,7,9]	[ ]	18-21
Da $a[0] \not\leq b[0]$ føjes $b[0]$ til c	[6,6]	[4,5,7,9]	[3]	22-25
Da $a[0] \not\leq b[0]$ føjes $b[0]$ til c	[6,6]	[5,7,9]	[3,4]	22-25
Da $a[0] \leq b[0]$ føjes $a[0]$ til c	[6,6]	[7,9]	[3,4,5]	18-21
Da $a[0] \leq b[0]$ føjes $a[0]$ til c	[6]	[7,9]	[3,4,5,6]	18-21
Da $a.length = 0$ føjes b til c	[ ]	[7,9]	[3,4,5,6,6]	12-14
Returnerer c	[ ]	[ ]	[4,5,3,6,1,7,9]	13

Figur 4.5: Eksempel med delfunktionen *merge* i mergesort. Her kalder vi merge([3,6,6], [4,5,7,9]). I hvert trin sammenlignes de røde tal. Blå tal er en del af den færdige c liste, og lilla tal er de tal der sidst blev føjet til c. Læg mærke til at den endelige c liste er sorteret fordi listerne a og b på forhånd var sorterede. Linjer refererer til figur **??**.

Funktion	Lister
mergesort	[3,6,4,7,2,7,5,2]
mergesort	[3,6,4,7] [2,7,5,2]
mergesort	[3,6] [4,7] [2,7] [5,2]
merge	[3] [6] [4] [7] [2] [7] [5] [2]
merge	[3,6] [4,7] [2,7] [2,5]
merge	[3,4,6,7] [2,2,5,7]
Færdig!	[2,2,3,4,5,6,7,7]

Figur 4.6: Hvordan *mergesort* først splitter listen op, og derefter samler den med *merge* (se figur **??**). Farver indikerer de par af lister der sættes ind i *merge*-funktionen.

den samler er sorterede. De lister der er samlet af *merge*, vil altid være sorterede, da udgangspunktet er lister med et enkelt element. Disse enkelt-element-lister samles af *merge* til lister med to elementer der er sorterede. Og disse sorterede lister sættes sammen med andre sorterede lister. *merge* får altså i altid to sorterede lister som input, og spytter derfor altid en samlet sorteret liste ud som output.
tilfælde hvor med en liste med ulige længde?

```
1         funktion mergesort(liste) {
2            if liste.length == 1{
3               return(liste)
4            }
5            else {
6               return(merge(mergesort(liste[0 ... n/2]), mergesort(liste[n/2 + 1 ... n])))
7            }
8         }
9
10        funktion merge(a,b) {
11           c = []
12
13           while(true) {
14              if a.length == 0 {
15                 return(c + b)
16              }
17              elif b.length == 0 {
18                 return(c + a)
19              }
20              elif a[0] <= b[0] {
21                 tilføj a[0] til c
22                 fjern a[0] fra a
23              }
24              else {
25                 tilføj b[0] til c
26                 fjern b[0] fra b
27              }
28           }
29        }
```

Figur 4.7: Pseudokode til Mergesort [2, s. 106].

4.4 Analyse af Insertionsort

5 | Binære Træer

6 | Store-O og Bedste Tiilfælde

7 | Implementering og test af Algoritmerne

8 | Konklusion

Her er mine konkluderende sætninger

Litteraturliste

1. Admin. Algorithms Order Of Growth. https://notesformsc.org/algorithms-order-of-growth.
2. Dietzfelbinger, M. & Mehlhorn, K. *Algoritmer og datastrukturer* https://github.com/thorehusfeldt/algoritmer-og-datastrukturer/blob/master/ad-book.pdf ().

Bilag 1 - Algoritmer og Datageneration

Insertionsort Algoritmen

```
def insertionsort(l):
   for i in range(1,len(l)):
      element = l[i]

      if element < l[0]:
         for j in range(i,0,-1):
            l[j] = l[j-1]
         l[0] = element
      else:
         j = i
         while(l[j-1]>element):
            l[j] = l[j-1]
            j -= 1
         l[j] = element
   return(l)
```

Mergesort Algoritmen

```
def mergesort(l):
   if len(l) <= 1:
      return(l)
   else:
      return(merge(mergesort(l[:len(l)//2]),mergesort(l[len(l)//2:])))

def merge(a,b):
   c = []
   while True:
      if (len(a) == 0):
         return(c + b)
      elif (len(b) == 0):
         return(c + a)
      elif (a[0] <= b[0]):
         c.append(a[0])
         a.pop(0)
      else:
         c.append(b[0])
         b.pop(0)
```

Kode til test af algoritmerne

```python
import random
import time
import pandas as pd
import os

import sys
sys.path.insert(1, './algoritmer')
from mergesort import *
from insertionsort import *

# Denne funktion timer køretiden af en funktion med input l og returnerer funktionen køretid i milisekunder
def test(fun,l):
   start_time = time.perf_counter()

   fun(l)

   return(time.perf_counter() - start_time)

# Denne funktion returnerer en liste af tilfældige tal mellem 0 og 1000, med n elementer
def createRandomList(n):
   return([random.randint(0,1000) for i in range(n)])

# Laver en mappe i filsystemet hvis der ikke allerede er en med stien
def makeIfNeeded(dir_path):
   if(os.path.isdir(dir_path) == False):
      print(f"made dir: {dir_path}")
      os.mkdir(dir_path)
   return(dir_path)

# Finder det næste versionsnummer for til navngivning af fil på baggrund af indholdet i en folder
def newVersionNumber(dir_path,extention):
   file_names = os.listdir(dir_path)
   version = 0

   thisfilename = f"{version}{extention}"

   while(thisfilename in file_names):
      version += 1
      thisfilename = f"{version}{extention}"

   return(thisfilename)

# Dette er funktionen der tester en liste med funktioner og gemmer deres køretider
def fullTest(functions):

   # hvor mange datapunkter pr. n-værdi
   trials = 10

   data_dir = "../data/"
   version_number = newVersionNumber(data_dir,"")

   seed = time.time()
   print(f"Seed: {seed}")

```

```
55    for function in functions:
56
57        # i denne liste gemmes antallet af elementer at den liste som algoritmen sorterer for hvert
              datapunkt.
58        ns = []
59        # i denne liste gennes den tid det tager at sorterer listen med n elementer
60        times = []
61
62        # Bruger det samme seed til test at hver algoritme. på den made er det de samme
              pseudo-tilfældige liste som algoritmerne sorterer
63        random.seed(seed)
64
65        # Vi laver testen et antal (trials) gange pr. n-værdi
66        for trial in range(0,trials):
67
68            # En lykke der køre et abitrært antal gange (jo højere en i-værdi jo højere maks antal
                  elementer i listen)
69            for i in range(0,80):
70
71                  # Jeg bruger en potensfunktion til at fa flere datapunker tættere på y-aksen og
                        færre lange oberationer (pga. lange liste)
72                n = round(pow(1.1,i))
73
74                print(f"function=\"{function.__name__}\": Trial: [{trial+1}/{trials}] {i=},{n=}") #
                      lidt feedback
75
76                # gennererer en tilfældig liste
77                l = createRandomList(n)
78
79                # gem størrelsen af listen der skal sorteres
80                ns.append(n)
81                # gen den tid det tager at sortere listen
82                times.append(test(function,l))
83
84
85            data = {
86                "n": ns,
87                "t": times
88                }
89
90            version_dir = makeIfNeeded(data_dir + version_number + "/")
91            algorithm_dir = makeIfNeeded(version_dir + function.__name__ + "/")
92            full_path = algorithm_dir + newVersionNumber(algorithm_dir,".csv")
93
94            print(f"\ndata saved to \"{full_path}\"\n")
95
96            pd.DataFrame(data).to_csv(full_path,index = False)
97
98
99    if __name__ == "__main__":
100       functions = [mergesort,insertionsort]
101
102       fullTest(functions)
```

Bilag 2 - Databehandling og Plots

Kode til databehandling og generering af plots

```
library(ggplot2)
#library(tikzDevice)

#working dir
setwd("/home/Balder/Documents/Skole/Gym/SRP/data/5")


#import data
dataset <- read.csv("mergesort/0.csv",header=TRUE,sep=",")

#dir = "1"
algorithm_dirs = list.files()

M = NULL
for (j in 1:length(algorithm_dirs)){
   algorithm_dir = algorithm_dirs[j]

   files = list.files(algorithm_dir)

   for (i in 1:length(files)){
      file_path = paste(algorithm_dir,files[i],sep="/")
      print(file_path)
      m = read.csv(file_path,header=TRUE,sep=",")
      m$Algorithm = algorithm_dirs[j]
      M = rbind(M, m)
   }

}

M$algorithm = factor(M$Algorithm)
summary(M)

# punktm?ngder for hver algoritme
m_merge = subset(M,M$algorithm=="mergesort")
m_insertion = subset(M,M$algorithm=="insertionsort")

# laver modeller
model_merge = nls(t~a*n*log2(n), data=m_merge, start=list(a=0.000001))
model_insertion = nls(t~a*n^2 + b*n + c, data=m_insertion, start=list(a=1,b=1,c=1))


# Sætter ny path til hvor outputtet skal være
setwd("/home/Balder/Documents/Skole/Gym/SRP/img")
```

```
44
45 # gemmer r2-værdierne i to filer
46 writeLines(toString(round(with(m_merge,cor(t,n)),digits=3)),"r2-merge.txt")
47 writeLines(toString(round(with(m_insertion,cor(t,n)),digits=3)),"r2-insertion.txt")
48 print("r2 saved to files")
49
50
51 # laver modelerede v?rdier for hver n
52 m_merge$model = predict(model_merge)
53 m_insertion$model = predict(model_insertion)
54
55 m_merge$residual = resid(model_merge)
56 m_insertion$residual = resid(model_insertion)
57
58 # kombinerer de to
59 M = rbind(m_merge,m_insertion)
60
61 summary(M)
62
63
64 ggplot(M, aes(x=n, y=t, colour=Algorithm)) +
65    geom_point(size=1.5,alpha=0.1,shape=19) +
66    geom_line(aes(x=n, y=model,color=Algorithm), size=2, alpha=0.6) +
67    labs(title="To Sorteringsalgoritmer") +
68    theme(legend.position = c(.9, .9)) + # virker ikke!!
69    guides(colour = guide_legend(override.aes = list(alpha = 1))) + # lav legend alpha 1
70    theme_bw()
71
72    ggsave("toAlgoritmer.png")
73
74    ggplot(M, aes(x=log10(model), y=residual, colour=Algorithm)) +
75       geom_point(size=1.5,alpha=0.1,shape=19) +
76       labs(title="Residualer") +
77       facet_wrap(~algorithm,scales="free",ncol=1) +
78       theme_bw() +
79       theme(legend.position="none")
80    ggsave("toAlgoritmerResidual.png")
81
82    C = data.frame(
83             Algorithm = unique(M$Algorithm),
84             R2 = c(with(m_merge, cor(t,n)), with(m_insertion, cor(t,n)))
85    )
86
87    write.table(C, "r2.txt", quote=FALSE,sep="\t", row.names=FALSE)
```