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.

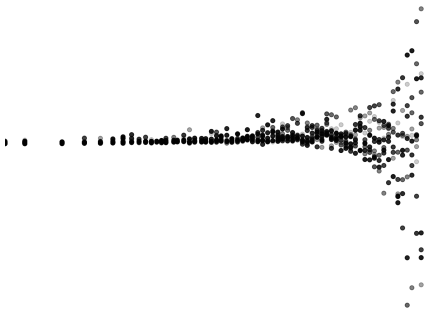
Beskriv sorteringsproblemer generelt, og forklar algoritmerne for insertion- og merge-sort. Vis at insertion-sort har en værste-tilfælde-udførelse på $O(n^2)$.

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

ns. $\approx 15{,}88$

Resume

Denne opgave handler om sorteringsalgoritmer, analysen af dem ved hjælp af store-O og test af deres reelle køretider. Store-O kan bruges til at analysere sorteringsalgoritmers tidskompleksitet. Denne opgave tager udgangspunkt i sorteringsalgoritmerne insertionsort og mergesort. Det viste sig at insertionsort havde tidskompleksiteten $O(n^2)$, og at mergesorts altid var $\Theta(n \cdot \log n)$. I testen af algoritmernes reelle køretider viste det sig, at insertionsort gennemsnitligt var hurtigere end mergesort, hvis $n < 37$. Denne information kunne bruges til at konstruere en hybridalgoritme, der i gennemsnit var hurtigere end mergesort. Dog kan vi ikke være sikre på, at denne hybridalgoritme altid er bedre end mergesort, da vi ikke har fortaget store-O-analysen af algoritmen.

Indhold

Figurer

Indledning

Alt sorteres. Navne, bøger, lande, slik, biler, ALT. Hver gang vi søger på Google, sorteres søgeresultaterne efter relevans, og når vi scroller på Tiktok eller Facebook, er opslagene sorteret på baggrund af tidspunktet, de er postet og den personlige data, der står i deres databaser.

Sortering er en måde at ordne data, så de er nemme at finde igen. Servere verden rundt bruger store dele af deres ydeevne på netop at sortere de data, som de lagrer. Optimering af disse algoritmer er derfor essentiel for det moderne samfunds gigantiske dataafhængighed, da de drastisk kan nedsætte databehandlingens udførelsestid.

Denne opgave omhandler sorteringsalgoritmer og analysen af dem ved hjælp af store-O og kompleksitetsklasser. Først defineres store-O, og herefter tages der udgangspunkt i sorteringsalgoritmerne insertionsort og mergesort, hvis tidskompleksitet diskuteres og bevises. Blandt andet vises det, at insertionsort har en værste-tilfælde-vækstrate på $O(n^2)$. Binære træer bruges i afsnittet efter, til at bevise at den nedre grænse for værste-tilfælde-vækstraten er $O(n \cdot \log n)$ for alle sorteringsalgoritmer. Det vil også blive forklaret hvorfor, insertionsort med en bedste-tilfælde-vækstrate på $O(n)$ ikke overskrider denne grænse. Til sidst implementeres algoritmerne i Python og deres reelle køretider undersøges. Algoritmerne samles til en hybrid-algoritme, der sammenlignes med mergesort.

1 Algoritmers Udførelsestid

1.1 Tid som Funktion af Inputtets Størrelse

Dette afsnit bygger primært på side 24 og 25 i bogen "*Algoritmer og Datastrukturer*" [2]. I følgende afsnit redegøres for definitionen af store-O.

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 kan nu beskrives som funktion af instansen $T(I)$. Med det menes der, at en algoritmes udførselsestid er afhængig af det input algoritmen får. Vi kan bruge dette til at beskrive tre egenskaber af algoritmens udførselsestid; algoritmens maksimale udførselsestid, algoritmens minimale udførselsestid og algoritmens gennemsnitlige udførselsestid. Den maksimale udførselsestid er udførselsestiden i værste tilfælde ($T(I_{\text{værst}})$), den minimale udførselsestid kan modsat beskrives som køretiden på det bedste tilfælde ($T(I_{\text{bedst}})$), 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:

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

Alle størrelserne beskriver et aspekt af algoritmens udførselsestid, dog er den mest pålidelige størrelse udførselsestiden i værste tilfælde. Det er samme koncept, som når postvæsnet siger at det vil tage tre dage for din pakke at nå frem, selvom det godt kan ske hurtigere. Det værste tilfælde *garanterer*, 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. Tildels kan det forklares af Allan Turings "halting problem"[5], hvor han beviste, at man aldrig kan vide om et program stopper eller ender i en uendelig løkke. Problemet i vores kontekst, hvor vi forsøger at bestemme en køretid for algoritmen, er, at hvis man kendte udførselsestiden for en algoritme, så ville man vide, at algoritmen stoppede før den havde kørt. Det var netop dette, Allan Turing viste, at man ikke kan. En anden forklaring er, at alle

computere vil køre den samme algoritme med 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 cpu'ens temperatur ændrer sig. Der således er mange faktorer, som bestemmer, hvor lang tid det tager en algoritme at køre færdig, hvilket gør det meget svært at bruge 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 forholde sig til udførselsestidens vækstrate, og algoritmens opførsel når n bliver meget stor. Denne form for analyse hedder store-O-analyse.

1.2 Store-O-Analyse

I store-O-analyse forholder vi os til, hvordan udførselsestidens stiger, når n bliver meget stor. I denne form for analyse inddeler vi grupper efter deres vækstrate. En funktions vækstrate beskriver, hvor hurtigt funktionen stiger i forhold til andre funktioner ved meget høje n-værdier. Hvis en funktion $f(n)$ har en højere vækstrate end funktionen $g(n)$, betyder det, at funktionen $f(n)$ altid vil være større end $g(n)$ ved store nok n-værdier. Det har ingen betydning 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 en funktion $f(n)$ har en højere vækstrate end $g(n)$, hvis $f(n) \geq c \cdot g(n)$ (hvor c er en konstant, og n er tilstrækkeligt stor). Hvis funktionerne har *samme* vækstrate gælder det at $c \leq \frac{f(n)}{g(n)} \leq d$ (hvor d er en konstant, og det samme gælder for n og c som før). I praksis betyder det, at funktioner som n^2 og $n^2 + 3n - 10$ har den samme vækstrate og, at deres vækstrate er højere end en funktion som $n \cdot log(n)$.

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

Figur 1.1: Definition af $O(f(n))$, $\Omega(f(n))$ og $\Theta(f(n))$ [2, s. 26].

Vi kan nu bruge disse regler til at definere store-O-notation (se figur 1.1). 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 1.2). 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 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. Man kan også sige, at algoritmens *tidskompleksitet* er $O(n^2)$ [11]. Da det kun er vækstraten, vi interesserer os for, betyder det, at det kun er leddet med den højeste vækstrate, der betyder noget, fordi de små led vil være ubetydelige ved store n-værdier. En funktion som $n! + n^3 + \log n$ vil blive reduceret til tidskompleksiteten $O(n!)$, da $n!$ er leddet med den højeste væk-

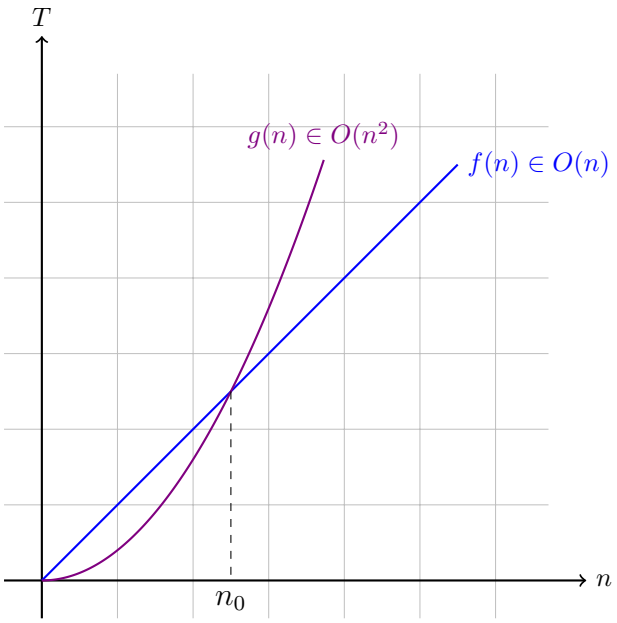
strate. Selv en funktion som $100 \cdot n!$, vil også have tidskompleksiteten $O(n!)$, da konstanten foran det størst voksende led ikke ændrer funktionens tidskompleksitet.

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

Der er en enkel 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 $\Theta(f(n))$. 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 i lige lang teoretisk udførselsestid.

1.3 Kompleksitetsklasser

Med denne definition af store-O kan vi klassificere algoritmer i forhold til deres værste-tilfælde-vækstrate. Store-O inddeler algoritmer i kompleksitetsklasser efter deres tidskompleksitet [4]. Algoritmer som er en del af klassen $O(n^2)$ har i værste tilfælde samme vækstrate som n^2 . Det kunne også være, at algoritmen var $O(n)$, altså ville den have en lineær vækstrate, og hvis en algoritme har konstant vækstrate, er den del af kompleksitetsklassen $O(1)$. Det er vigtigt at huske, at disse kompleksitetsklasser ikke kun betegner vækstraten i værste tilfælde, men at de også indeholder alle bedre vækstrater. Det vil altså sige, at alle kompleksitetsklasser med en bestemt værste-vækstrate også indeholder alle kompleksitetsklasser med lavere værste-vækstrate (se figur 1.3).



Figur 1.2: Her ses skæringspunktet n_0, hvorefter $g(n)$ altid er højere end $f(n)$. Altså vækstraten af $g(n)$ højere end vækstraten af $f(n)$.

$$O(1) \subset O(n) \subset O(n \cdot \log n) \subset O(n^2) \subset O(n!)$$

Figur 1.3: Eksemplel på hvordan kompleksitetsklasserne indeholder hinanden

2 | Sorteringsalgoritmer

2.1 Hvad er sortering?

Sortering er helt lavpraktisk at sætte en mængde data i rækkefølge på baggrund af disse datas attributter [10]. Det kunne f.eks. være alfabetisk eller efter farve eller størrelse. Når man som menneske sorterer en hånd med kort er det ikke altid, at vi tænker over, hvordan vi gør. Det er dog ikke altid klart, hvilken sorterings-strategi, der vil være hurtigst. Disse sorterings-strategier kan også kaldes sorterings-algoritmer, og det er ikke kun mennesker, der kan benytte sorterings-algoritmer til f.eks. at sortere kort. Computere kan også, og de gør det for det meste også langt hurtigere. Resten af denne opgave omhandler to sorterings-algoritmer: insertionsort og mergesort, og deres forskellige måder 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 2.1). Derudover vil algoritmernes opgave altid være at sortere liste i stigende rækkefølge.

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

Figur 2.1: Eksempel på sortering af en liste

2.2 Insertionsort

Insertionsort er en af de mere simple sorteringsalgoritmer. Pseudokoden til algoritmen kan ses i figur 2.2 på side 7. Koden er baseret på bogen *Algoritmer og datastrukturer* [2] s. 104.

Det er ikke en tilfældighed, at denne algoritme hedder insertionsort. Den fungerer nemlig ved at gennemgå hvert element i listen og placere det, hvor det passer ind i de elementer, der allerede er sorterede. Dette er nok den måde, man ville sortere sin hånd i Uno eller 500.

2.2.1 Insertionsort Procedure

Vi kan allerede i linje 2 se, at algoritmen gennemgår alle elementerne i listen fra en ende af, da algoritmen her begynder med en for-løkke, 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 2.3).

```
1        funktion insertionsort(liste) {
2           for i = 1 til i = n {           # her er n længden af listen
3           e = liste[i]                    # dette er elementet i listen
4
5              if e < liste[0] {            # hvis elementet er større end det første element i
                   listen
6                 for j = j til j = 0{
7                    liste[j] = liste[j-1]  # ryk elementerne på pladserne 0 til j et tak frem
8                 }
9              liste[0] = e                 # sætter dette element forrest i listen
10          }
11          else{
12             j = i                      # j er en nu tæller der starter på i
13             while liste[j-1] > e {     # kør mens liste[j-1] er størren end elementet
14                liste[j] = liste[j-1]   # ryk liste[j] et tak til højre
15                j -= 1
16             }
17             liste[j] = element         # indsæt elementet hvor det passer ind
18          }
19       }
20       return(liste)                    # returnerer den sorterede liste
21    }
```

Figur 2.2: Pseudokode til insertionsort [2, s. 104].

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 2.3: Procedure for at indsætte element først i listen. Elementet er markeret med rødt, og tal der allerede er sorterede er markeret med blåt. Linjer refererer til koden i figur 2.2.

Det er vigtigt at pointere, at computere ikke kan "rykke" tal i en liste, men kun overskrive dem. Det er derfor at det ser ud som om at, computeren har dubleret 5-tallet i figur 2.3. Det har den nemlig; den har taget de tre første elementer i listen, og skrevet dem over på andet, tredje og fjerde element. Computeren kan nu overskrive det første tal i listen, da det er en dublet.

Vi er nu nået til linje 11. Hvis ikke elementet er mindre end det første element i listen, sker følgende (se figur 2.4): Algoritmen ser på tallet før elementet og spørger: "er dette tal større end elementet". Hvis det er, rykkes det til højre. Herefter stiller algoritmen samme spørgsmål til tallet *to* pladser før elementet. Hvis dette tal er mindre and elementet, placerer algoritmen elementet efter dette tal i listen, 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 listen af den første del af proceduren (linje 5-10).

2.2.2 Egenskaber af Insertionsort

Insertionsort er en god sorteringsalgoritme, idet proceduren er forholdsvis nem at forstå, og fordi listen til venstre for det element algoritmen er nået til (e), 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. Denne egenskab har mergesort ikke.

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 2.4: Procedure for at indsætte et element hvor det passer i den sorterede liste. Elementet er markeret med rødt, og tal der allerede er sorterede er markeret med blåt. Linjer refererer til figur 2.2.

Se afsnit 2.4 for store-O analysen af insertionsort.

2.3 Mergesort

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

2.3.1 Mergesort Procedure

Pseudokoden til denne algoritme kan findes i figur 2.5 på side 9. Koden er basseret på kilde [2, s. 106].

Mergesortalgoritmen består af to funktioner. Lad os begynde med funktionen $merge$, da det er den mest simple. Koden til funktionen kan findes på figur 2.5 linje 11-29. Funktionen tager to lister (a og b) som argumenter, og fletter dem sammen til en tredje liste (c) (se figur 2.6). Denne c liste er dog ikke nødvendigvis 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 dog at den kan flette to sorterede lister sammen til én samlet sorteret 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.

Linjenumre i det næste afsnit refererer til pseudokoden til algoritmen på figur 2.5 på side 9.

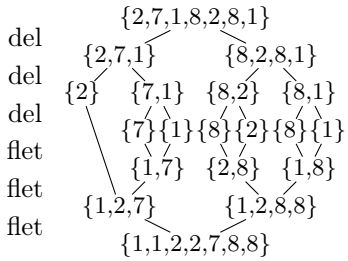
Mergesorts kompleksitet kommer dog først rigtigt til udtryk, når vi tager hele algoritmen i betragtning. Mergesortfunktionen (l. 1) tager en enkel liste som argument og checker først, om listen kun indeholder ét element (l. 2). Hvis den gør, så returnerer den listen uændret (l. 3). Dette giver intuitivt 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

```
1   funktion mergesort(liste) {
2      if liste.length == 1{    # hvis listen har en længde på 1: returner listen uændret
3         return(liste)
4      }
5      else {
6         # Her deler algoritmen først listen i to dele. Derefter kalder den sig selv på hver
              del. Dette gør den til listerne kun er indeholder et element. Derefter samles
              dellisterne til en samlet og sorteret liste af mergefunktionen
7         return(merge(mergesort(liste[0 ... n/2]), mergesort(liste[n/2 + 1 ... n])))
8      }
9   }
10
11  funktion merge(a,b) {
12     c = []                   # Listen a og b flettes ind i
13
14     while(true) {            # Lykken kører til der returneres en værdi
15        if a.length == 0 {    # Hvis der ikke er mere i a
16           return(c + b)      # Returnerer c sammenføjet med b
17        }
18        elif b.length == 0 {  # Ellers hvis der ikke er mere i b
19           return(c + a)      # Returnerer c sammenføjet med a
20        }
21        elif a[0] <= b[0] {   # Ellers hvis det første element i a er mindre end det i b
22           tilføj a[0] til c
23           fjern a[0] fra a
24        }
25        else {                # Ellers
26           tilføj b[0] til c
27           fjern b[0] fra b
28        }
29     }
30  }
```

Figur 2.5: Pseudokode til Mergesort [2, s. 106].

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 2.6: 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 2.5.



Figur 2.7: Hvordan mergesort først opdeler, og derefter samler listen med $merge$. [2, s. 106]

algoritmen til at fungere (l. 5-8): den deler listen 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 $mergesort$ 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 den samler er sorterede (se figur 2.7 på side 10). 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 altid to sorterede lister som input, og spytter derfor altid en samlet sorteret liste ud som output.

2.3.2 Tidskompleksiteten af Mergesort

Mergesort har tidskompleksiteten $O(n \cdot \log n)$ [6]. Algoritmens værste-tilfælde-vækstrate er det samme som vækstraten af $n \cdot \log n$.

$$T_{\text{mergesort}} \in O(n \cdot \log n)$$

2.4 Store-O-Analyse af Insertionsort

I dette afsnit gøres der brug af tidligere definerede begreber og koncepter fra afsnit 1.

2.4.1 Insertionsort i Værste Tilfælde

Når man analyserer en algoritme teoretisk handler det ikke om den faktiske udførelsestid, men nærmere hvor mange operationer computeren skal køre for at fuldføre algoritmen. På den måde kan man abstrahere helt væk fra fysisk tid og regne med teoretiske operationer, hvis reelle udførelsestid ikke har nogen indflydelse. Det er fordi udførelsestiden er proportional med antallet af operationer, der kører på computerens cpu. Den eneste faktor for antallet af operationer, som en sorteringsalgoritme kører, er kardinaliteten af inputtet algoritmen udføres på. Dette gør det muligt at beskrive antallet af operationer (der er propportionalt med udførelsestiden) udelukkende som funktion af n (inputtets kardinalitet). [2, s. 42]

Insertionsort består af to løkker: en indre og en ydre. Den ydre løkke kører fra $i = 1$ til (og uden) $i = n$. Altså kører koden inde i løkken $n - 1$ gange. Den indre løkke kører i værste tilfælde fra $j = i$ til $j = 1$ for hver i værdi i den ydre løkke. Det betyder, at operationerne, som den indre løkke udfører, stiger sammen med i (se figur 2.8). Vi ender med at kunne beskrive det totale antal gange som koden i den indre løkke

kører, som summen af alle i-værdierne, som den ydre løkke gennemgår. Koden i den indre lykke køres $i-1$ gange for hver i.

$$\sum_{i=2}^{n} i-1$$

Den indre løkkes kode køres i gange, da løkken gennemgår fra $j=i$ til og med $j=1$, altså kører løkken i gange. Vi kan nu omskrive summen således:

$$\sum_{i=1}^{n-1} i$$

For at simplificere udtrykket endnu mere kan vi bruge reglen at en sum som $1+\cdots+n-1$ kan omskrives på denne måde.

$$\sum_{i=1}^{n-1} i \quad = \quad \frac{n(n-1)}{2} \quad = \quad \frac{1}{2}\cdot n^2 - \frac{1}{2}\cdot n$$

Dette er altså et udtryk for, hvor mange gange koden i den indre løkke kører som funktion af kardinaliteten af algoritmens input. Vi kunne nu begynde at gange denne funktion med hvor mange operationer, der sker inde i den indre løkke. Og lægge operationerne der sker n gange til, fordi de er med i den ydre løkke. Til sidst kunne vi lægge de operationer, der kun sker én gang til, og resultatet ville se nogenlunde sådan ud:

$$a\cdot\frac{1}{2}\cdot n^2 - a\cdot\frac{1}{2}\cdot n + b\cdot n + c$$

Her er a antallet af operationer i den indre løkke, b er antallet af operationer kun i den ydre løkke, og c er operationerne, som er uden for begge løkker. Det smarte ved store-O-notation er at vi faktisk ikke behøver at kende de reelle værdier for a, b og c, da de alle er konstanter, og derfor ikke har nogen indflydelse på algoritmens vækstrate. Selv ikke udførelsestiden for hver at disse operationer har nogen indflydelse på vækstraten. Vi kan derfor sige, at algoritmens udførelsestid er $O(n^2)$.

$$T_{\text{insertionsort}}(n) \in O(n^2)$$

Med det menes, at udførelsestiden i værste tilfælde har tidskompleksiteten $\Theta(n^2)$.

i	j	kørsler i j løkken
2	$2\ldots1$	1
3	$3\ldots1$	2
4	$4\ldots1$	3
5	$5\ldots1$	4
$\vdots$	$\vdots$	$\vdots$
n	$n\ldots1$	$(n-1)$

$\Longrightarrow$

Total antal kørsler i j-løkken:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Figur 2.8: Insertionsort: Antal kørsler af koden i den indre j-løkke, hvis inputlisten har kardinaliteten n

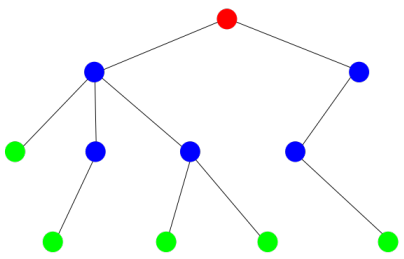
3 Binære Træer

3.1 Hvad er et Træ

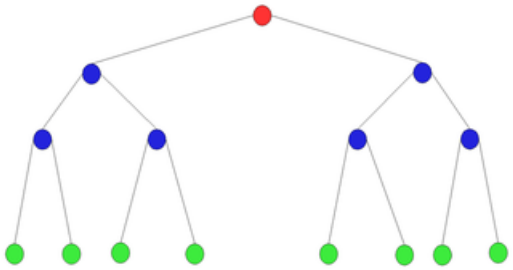
Træer, som vi kender dem, har en stamme, hvorfra grene springer ud. Fra grenene springer mindre grene, og på den måde ender træet med at have mange små grene, som stammer fra samme stamme. Informatikkens træer er ikke langt fra denne forståelse. I informatik er træer en datastruktur. Det er et netværk af knudepunkter forbundet af kanter. Træet starter med roden, der typisk er tegnet øverst (se figur 3.1). En rod er, hvor træet begynder, og har ingen forældre. Fra roden forgrener træet sig til flere børn, der også kan have flere børn. Træstrukturen er sådan, at man altid kan vælge et knudepunkt og lave et subtræ med knudepunktet som rod. Knudepunkterne, hvor træet ender, kalder man blade. Informatikkens træer har også en dybde. Hvis man starter ved roden, er dybden 0. Hver gang man går et trin "ned", i træstrukturen, stiger dybden med 1. Det betyder, at den maksimale dybde svarer til træets højde. [8]

3.2 Et Binært Træ

Man tegner normalt binære træer som på figur 3.2 bestående af knudepunkter og forbindelser mellem dem. Det specielle ved *binære* træer er, at hvert knudepunkt højst må forgrene sig to gange (altså binært). Denne lille specialisering er brugbar i mange sammenhænge som f.eks. som datastruktur til søgealgoritmer [1], men det viser sig også at være en snedig vej til at beskrive sorteringsalgoritmers opførsel.



Figur 3.1: Eksempel på træ.



Figur 3.2: Eksempel på binært træ. Træets blade er markeret med grønt, og roden med rødt. [7]

3.3 Blade i et Fuldt Binært Træ

Beviset er baseret på kilden [3].

Et fuldt binært træ er et binært træ, hvor alle bladene har samme dybde. Træet på figur 3.2 er et fuldt binært træ.

Intuitivt giver det mening, at hvert knudepunkt har to børn, og at antallet af blade derfor er givet ved 2^d, hvor d er træets dybde. Dette udtryk kan bevises således:

Vi antager at et træ med en maksimal dybde d, har 2^d blade.

Hvis vores dybde er $d+1$, skal antallet af blade være 2^{d+1}.

I et fuldt binært træ fordobles antallet af blade, hver gang dybden stiger med 1. Det er fordi alle knudepunkter i et fuldt binært tre altid forgrener sig til to børn. Altså må antallet af blade ved dybden $d+1$ være det dobbelte af antallet af blade ved dybden d.

$$2 \cdot (2^d)$$

Vi kan nu omskrive udtrykket således:

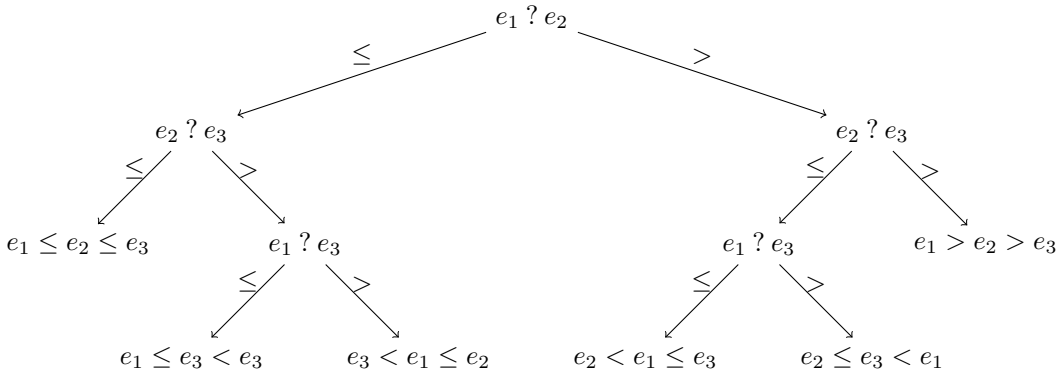
$$2 \cdot (2^d) \quad = \quad 2^1 \cdot 2^d \quad = \quad 2^{d+1}$$

Altså er antagelsen, at antallet af blade i et fuldt binært træ er 2^d, bekræftet. ■

Det er vigtigt at pointere, at alle binære træer med dybden d maksimalt har lige så mange blade som det fulde træ med samme dybde. Altså er 2^d også det maksimale antal blade for er generelt binært træ, med dybden d.

3.4 Binære Træer og Sorteringsalgoritmer

Ved første blik kunne man tilgives for ikke at se, hvordan binære træer har relevans for sorteringsalgoritmer, men det kræver bare, at man giver knudepunkterne og grenene meningsfulde betydninger. I bund og grund



Figur 3.3: Binært Træ. [2, s. 109]

er det som en sorteringsalgoritme gør, at fortage en masse sammenligninger af elementerne i dens input. Algoritmen bestemmer, hvilke operationer den skal køre udelukkede på baggrund af disse sammenligninger. Hvis vi tænker hvert knudepunkt som en sammenligning af to elementer fra algoritmens input (se figur 3.3), er det jo givet, at det ene element enten vil være større end det andet eller ikke større. Denne sammenligning guider algoritmens operationer og derved den næste sammenligning. Dette gentages, ind til algoritmen har gjort nok sammenligninger til at kunne sortere dens input. Vi kan derfor tænke alle træets blade som en måde for algoritmen at sortere et bestemt input: Hvert blad har kun én specifik vej, og vejen til bladet er udelukkede givet af algoritmens input. Altså er hvert af træets blade en kollektion af de sammenligninger, som algoritmen gjorde for at sortere inputtet. [2, s. 109]

3.4.1 Det Mindste antal Sammenligninger

I dette store teoretiske træ med alle dets blade kunne man jo så spørge: Hvor mange sammenligninger skal der så højest til for at sortere et input? Eller omformuleret: Hvor højt er træet? Denne oversættelse holder stik, da vi skal lave lige så mange sammenligninger for at komme ned til nederste blad, som træet er højt.

3.5 Højden af et Binært Træ

Dette bevis er basseret på kilden [2, s. 109].

Bevis: I et sammenligningstræ for en sorteringsalgoritme med inputlængden n, hvor π og σ er permutationer af listen $\{1 \dots n\}$, er bladene, der svarer til π og σ (l_π og l_σ), forskellige.

Dette er et modstridsbevis, hvilket betyder, at vi starter med at antage det modsatte af det, vi prøver at bevise: Vi antager altså, at π og σ leder til samme blad i sammenligningstræet; l_π og l_σ er ens.

Under denne antagelse kan to forskellige lister, $\{e_1 \dots e_n\}$ og $\{e'_1 \dots e'_n\}$, gennemgå de *præcis* samme operationer og derved sortere listen. Dette er en modstrid, da man aldrig vil kunne benytte præcis de samme operationer til at sortere to forskellige permutationer af $\{1 \dots n\}$. ■

Alle permutationer af listen $\{1 \dots n\}$ har altså et tilsvarende blad. Derfor må sammenligningstræet have mindst $n!$ blade, da der skal være et for hver permutation.

Det maksimale antal blade på et binært træ er 2^d, hvor d er træets dybde (som vi beviste i afsnit 3.3). Vi kan nu opstille denne ligning:

$$2^d \geq n! \quad \Leftrightarrow \quad d \geq \log n!$$

For at bygge videre kan vi nu bruge regnereglen $n! \geq \left(\frac{n}{e}\right)^n$

$$d \geq \log n! \geq \log\left(\left(\frac{n}{e}\right)^n\right)$$

Ved hjælp af logaritmeregneregler kan vi omskrive udtrykket længst til højre således:

$$\log\left(\left(\frac{n}{e}\right)^n\right) = n \cdot \log\left(\frac{n}{e}\right) = n \cdot \log n - n \cdot \log e$$

Dette er altså et udtryk for højden af et sammenligningstræ:

$$d \geq n \cdot \log n - n \cdot \log e$$

Da d er træets dybde, er det også det maksimale antal sammenligninger en given sorteringalgoritme skal gøre for at sortere en liste. Udtrykket siger, at den nedre grænse for d er $n \cdot \log n - n \cdot \log e$. Det er altså ikke muligt at sortere en liste med færre sammenligninger i værste tilfælde. Da antallet af sammenligninger er proportionalt med algoritmens udførelsestid, må det være sandt, at alle sorteringsalgoritmer har en nedre værste-tilfælde-vækstrategrænse på $O(n \cdot \log n)$.

3.6 Store-O er Værste Tilfælde

Det er en vigtigt detalje, at denne nedre grænse er for vækstraten i *værste* tilfælde. Insertionsort kan i bedste tilfælde sortere en liste med $O(n)$ [6], men det er værste tilfælde, som vækstrate-grænsen gælder for. Det er muligt at sortere en liste hurtigere end $O(n \cdot \log n)$; det kræver bare, at det ikke er det værste tilfælde, og at algoritmen (som insertionsort) kan sortere listen hurtigere i bedste tilfælde. Man burde hellere tænke grænsen således: Man kan aldrig lave en sorteringalgoritme, der virker ved hjælp af sammenligninger, som vi kan være sikre på, er hurtigere end $O(n \cdot \log n)$.

3.7 Mergesort og Den Nedre Grænse

Vi ved nu, at alle sorteringsalgoritmers værste-tilfælde-vækstrate, er afgrænset af $O(n \cdot \log n)$. Vi ved også, at mergesort har tidskompleksiteten $O(n \cdot \log n)$. Mergesort har altså den teoretisk bedste tidskompleksitet for en sammenlignings-sortering-algoritme. Ikke nok med det, betyder det også, at mergesort både i bedste og værste tilfælde er $O(n \cdot \log n)$. Mergesort er altså $\Theta(n \cdot \log n)$.

$$T_{\text{mergesort}}(n) \ \in \ \Theta(n \cdot \log n)$$

4 | Implementering og Test

I dette afsnit testes algoritmerne fra afsnit 2. Algoritmerne implementeres i python.

4.1 Python

Sproget Python har et højt abstraktionsniveau og kan på mange punkter ligne vores pseudokode fra tidligere [2, s. 68]. Det betyder, at vi på mange punkter overgiver lidt kontrol til compileren for at gøre koden mere læsbar og tilgængelig. Det betyder også, at vi aldrig helt kan vide, hvilke instruktioner computeren udfører. Derudover er Python et relativt langsomt sprog, da koden kompileres samtidigt med, at programmet kører [12]. Dette gør, at sproget ikke egner sig særligt godt til at skrive optimerede algoritmer, der skal køre på meget store datasæt. Heldigvis kan vi se bort fra dette, da vi med store-O-analyse er ligeglade med den reelle udførelsestid og i stedet er interesseret i vækstrater. Vækstraten for en algoritme er nemlig ens, ligegyldigt hvor langsomt hvert skridt er. På nogle punkter er det måske endda en fordel, at algoritmerne kører langsommere, da det vil gøre forskellen i udførelsestiderne større og lettere at forholde sig til.

4.2 Implementering

Implementering i Python er relativt simpelt. Python bruger dynamiske variabler [12], derfor er det ikke nødvendigt at fortælle Python, om en variabel f.eks. er et heltal eller en liste.

4.3 Test af Sorteringsalgoritmer

For at teste sorteringsalgoritmerne fra afsnit 2 og se hvordan deres reelle udførelsestid som funktion af n relaterer til deres teoretiske vækshastighed og store-O-analyse, skal algoritmerne køres på en kontroleret måde, hvor vi systematisk kan tage tid på algoritmens udførelsestid ved forskellige input.

På figur 4.1 på side 17 ses koden til test af algoritmerne. Variablen $functions$ på linje 1 indeholder en liste med de algoritmer, der skal testes. I vores tilfælde indeholder den [$insertionsort$,$mergesort$]. Inde i denne løkke tester vi funktionerne. Det, at vi behandler algoritmerne ens, sikrer, at de bliver testet på samme måde. Det næste vi gør, er at definere lister til opbevaring af kardinaliteten af den liste, som algoritmen sorterer (l. 4), og til den reelle tid, det tager for algoritmen at sortere listen (l. 6). I linje 9 sætter vi frøet for de pseudotilfældige tal [9], som vi senere skal bruge til at generere listerne, som sorteringsalgoritmerne skal sortere. Det er vigtigt, at vi sætter frøet til den samme værdi, før vi tester hver algoritme, da vi på den måde sikrer os, at det er de samme pseudotilfældige lister, som algorimerne sorterer. Vi kan derfor med god vilje

```
1   for function in functions:
2
3       # i denne liste gemmes antallet af elementer at den liste som algoritmen sorterer for hvert
            datapunkt.
4       ns = []
5       # i denne liste gennes den tid det tager at sorterer listen med n elementer
6       times = []
7
8       # Bruger det samme seed til test at hver algoritme. på den måde er det de samme
            pseudo-tilfældige liste som algoritmerne sorterer
9       random.seed(seed)
10
11      # Vi laver testen et antal (trials) gange pr. n-værdi
12      for trial in range(0,trials):
13
14          # En lykke der køre et abitrært antal gange (jo højere en i-værdi jo højere maks antal
                elementer i listen)
15          for i in range(0,80):
16
17              # Jeg bruger en potensfunktion til at fa flere datapunker tættere på y-aksen og færre
                    lange oberationer (pga. lange liste)
18              n = round(pow(1.1,i))
19
20              # lidt feedback
21              print(f"function=\"{function.__name__}\": Trial: [{trial+1}/{trials}] {i=},{n=}")
22
23              # gennererer en tilfældig liste med længden n
24              l = createRandomList(n)
25
26              # gem størrelsen af listen der skal sorteres
27              ns.append(n)
28              # gem den tid det tager at sortere listen
29              times.append(test(function,l))
```

Figur 4.1: Kode til test af algoritmerne. Variablen $functions$ er en liste med de to algoritmer der testes ([insertionsort,mergesort]).

sammenligne algoritmernes køretider, da vi er sikre på at deres input, var det samme under testen. Efter denne opsætning begynder vi to løkker (l. 12 og 15). Den første sørger for, at vi gentager den samme test flere gange. I vores tilfælde er variablen $trials$ sat til 15, hvilket resulterer i, at vi tester sorteringen af hver n lange liste 15 gange. Den indre løkke tæller fra $i = 0$ til $i = 80$. Vi bruger herefter i-værdien til at generere vores n-værdier i linje 18. Til det bruges formlen $n = \lfloor 1.1^i \rfloor$. Valget af test med exponentielt stigende n er gjort af to grunde: For det første kan det tage lang tid for en algoritme at sortere lister med store n, så det gør hele processen meget hurtigere, hvis man ikke tester så mange sorteringer med store n. Den anden og nok bedre grund er, at vi er meget interesserede i at se, hvordan algoritmerne måles med hinanden, når n *ikke* er stort. Det ville jo være interessant, hvis vi kunne identificere et n_0 som i afsnit 1. Til dette skal vi bruge punkter, der ved store n ikke behøver at ligge tæt.

Nu er det hele sat op, og vi kan teste algoritmen på en n lang liste med tilfældige tal. n-værdien, og den tid sorteringen tog, gemmes i listerne og eksporteres til en csv fil til brug i databehandlingen (se koden i bilag 1 og 2).

4.4 De Reelle Udførelsestider

Algoritmernes udførelsestider er plottet i figur 4.2 på side 19.

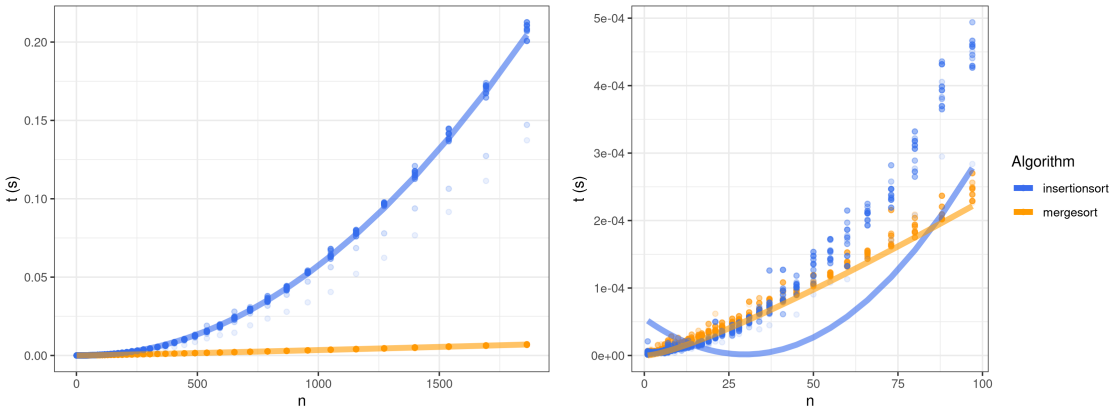
Som det tydeligt fremgår på graferne, er mergesort langt hurtigere end insertionsort ved store n. Vi kan også se, at algoritmerne nogenlunde følger deres regressioner, men hvorfor egentlig det? Det er kun i værste tilfælde, at $T_{insertionsort}(n) \in \Theta(n^2)$, ikke? Jo, men det lader til, at insertionsorts gennemsnitlige vækstrate er tættere på $\Theta(n^2)$ end på $\Theta(n)$. Det er dog alligevel tydeligt, at andengradsregression ikke passer helt på insertionsorts gennemsnitlige køretid, hvis vi zoomer ind på plottet til højre på figur 4.2. Faktisk passer regressionen slet ikke på punkterne på dette stykke af grafen, hvilket tyder på, at algoritmens udførelsestid ikke bare kan beskrives med et andengradspolynomium. Hvad angår mergesort, følger den regressionen rigtig godt. Det tyder på, at antagelsen, at mergesorts vækstrate altid er $\Theta(n \log n)$ fra afsnit 3.7, må være sand.

En anden finurlighed kan findes i residualplottet (figur 4.3). Residualplottet for begge grafer er rigtig flot, med en jævn punktfordeling på begge sider af regressionen, men til allersidst er der et par af insertionsorts punkter, der er langt under regressionen. Det kan være af flere grunde, blandt andet at computerens styresystem lige i dette tilfælde, tildelte flere ressourcer til programmet. En anden grund kunne være, at algoritmen netop i disse tilfælde var heldig at få en delvist sorteret liste, der gjorde, at listen kunne sorteres langt hurtigere end $O(n^2)$. Dog tyder det tydelige nedadgående mønster i outlier-residualerne på, at afvigelsen ikke er en tilfældighed.

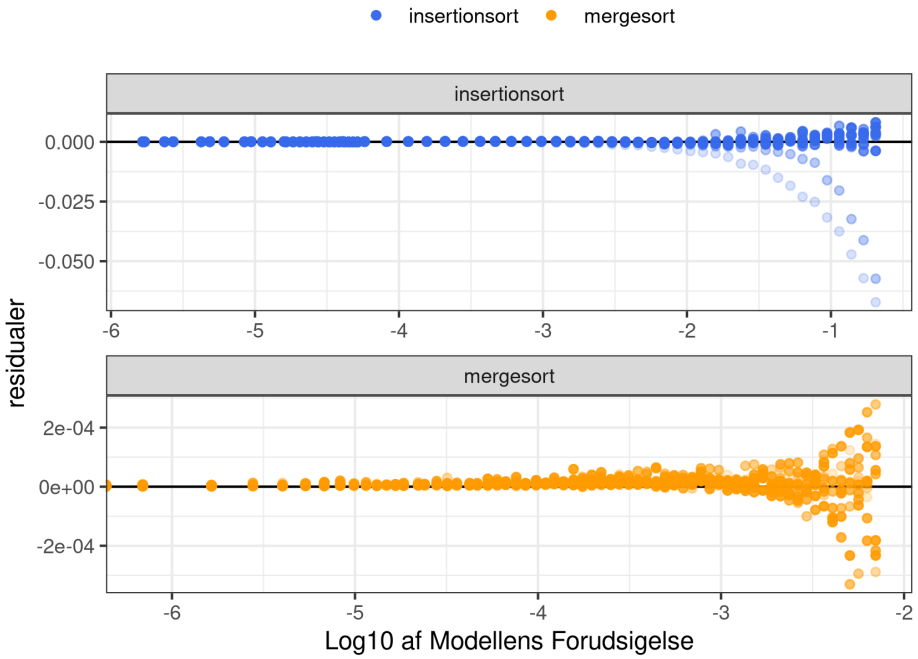
Når man ser på residualplottene, er et andet mønster også tydeligt: variansen stiger. Udførelsestiderne har større og større usikkerhed, jo større n er. Dette giver intuitivt mening, hvis man sætter det i kontekst. Hvis 10 personer løber et 50m løb, vil alle løbere afslutte inden for et forholdsvist lille tidinterval. Hvis vi herefter bad dem om at løbe et maraton, ville deres tider ikke bare variere med sekunder og minutter, men med timer. Variansen i deres tider øges altså af størrelsen på opgaven. Det samme er sandt for vores sorteringsalgoritmer, hvilket tydeliggøres af residualplottene.

4.4.1 Hvornår er Insertionsort Hurtigst?

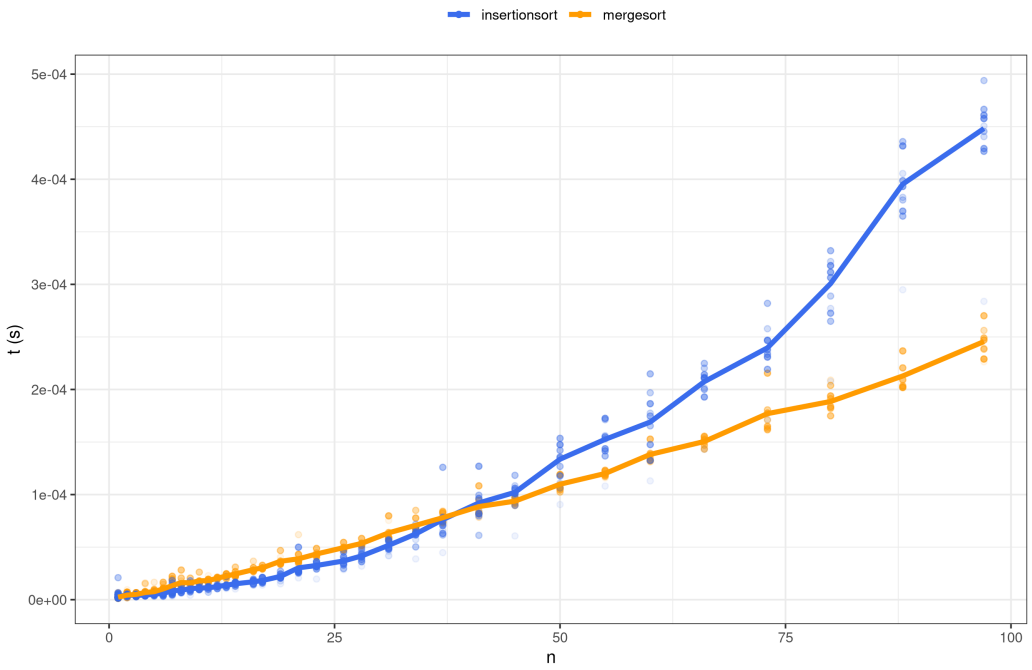
Det er tydeligt, at mergesort er langt bedre til at sortere store lister, men ved små n kan vi også se, at insertionsort ikke er langt bagefter. Faktisk er den *gennemsnitlige* sortering hurtigere eller nogenlunde ens før $n = 37$ (se figur 4.4). Det bekræfter teorien fra afsnit 1, idet store-O ikke siger noget om algoritmens opførsel ved små n, men kun ved meget store. Her er det tydeligt, at en sådan form for analyse ikke er tilstrækkelig for at bedømme, hvilken algoritme der er bedst til hvilke problemer. Store-O-analyse er også blind overfor den gennemsnitlige udførelsestid. Ved små n er den gennemsnitlige køretid brugbar, da den maksimale køretid aldrig bliver astronomisk stor, og vi kan derfor med god vilje bruge gennemsnitlig udførelsestid i disse overvejelser. Det ville gennemsnitligt være hurtigst, at bruge insertionsort ved små n og mergesort ved store. Punktet der adskiller, hvilken algoritme der er hurtigst, er ca. $n = 37$. Dette punkt kan siges at svare til n_0 fra afsnit 1, idet mergesort gennemsnitligt altid er hurtigere efter punktet. Alligevel er det ikke helt sandt, da n_0 fra afsnit 1 forholder sig til værste-tilfælde-udførelsestiden, men vi kan i denne reelle kontekst bruge n_0 som punktet, hvorefter mergesort *gennemsnitligt* er hurtigst.



Figur 4.2: Sammenligning af insertionsort og mergesort. Punkterne er de målte køretider og kurverne er regressioner. Regressionerne er henholdsvis en andengradsregression for insertionsort, og en $a \cdot n \cdot \log n$-regression for mergesort. Til venste ses figuren for alle n, og til højre ses samme plot zoomet ind til $n < 100$



Figur 4.3: Residualplot for modellerne i figur 4.2.



Figur 4.4: Dette er samme data som i figur 4.2, men zoomet ind til indervallet $n \leq 100$. Her er regressionerne erstattet af en linje, der går mellem gennemsnittet af t for hver n. Det interessante ved grafen er, at man tydeligt kan de, hvordan insertionsort gennemsnitligt er hurtigst indtil $n \approx 37$, hvorefter mergesort er hurtigst.

4.5 Optimering af Mergesort

At insertionsort faktisk normalt er hurtigere end mergesort, hvis $n < n_0$, betyder, at den hurtigste måde at sortere en liste afhænger af listens længde: Er listen på under n_0 elementer? Så brug insertionsort. Er listen over n_0 elementer? Så er mergesort gennemsnitligt hurtigere. Dette kunne lede os til at lave en ny sorteringsalgoritme således:

```
1     def sort(liste) {
2        if liste.length <= 37 {
3           return(insertionsort(liste))
4        }
5        return(mergesort(liste))
6     }
```

Figur 4.5: Algoritme der bruger insertionsort hvis inputlisten er kortere end 37, og ellers mergesort.

Her bruger vi insertionsort, hvis listens længde er under eller lig 37, og mergesort hvis ikke. Det er en forbedring af algoritmen, men kun hvis $n < 37$. Dog er der en endnu mere snedig måde at inkorporere denne ide. Hvis vi tænker tilbage på mergesorts procedure fra afsnit 2, ved vi, at algoritmen deler en usorteret liste op, indtil der kun er lister med enkle elementer tilbage. Funktionen $merge$ sætter herefter listerne sammen igen og sørger for, at den samlede liste altid vil være sorteret, da den altid får sorterede lister som input. Vi ved dog nu, at denne process ikke er effektiv ved n-værdier under 37. Det er derfor en smart ide at bruge insertionsort til at sortere listerne, når mergesort har opdelt listen i tilstrækkeligt små bidder. Koden for denne hybridalgoritme kan ses på figur 4.6. I stedet for at opdele listen helt til der kun er et enkelt element i hver delliste, stopper denne algoritme opdelingen, så snart insertionsort kan sortere

```
1  from insertionsort import *
2
3  n0 = 30
4
5  def hybrid(l):
6     if len(l) <= n0:
7        return(insertionsort(l))
8     else:
9        return(merge(hybrid(l[:len(l)//2]),hybrid(l[len(l)//2:])))
10
11 def merge(a,b):
12    c = []
13    while True:
14       if (len(a) == 0):
15          return(c + b)
16       elif (len(b) == 0):
17          return(c + a)
18       elif (a[0] <= b[0]):
19          c.append(a[0])
20          a.pop(0)
21       else:
22          c.append(b[0])
23          b.pop(0)
```

Figur 4.6: Mergesort hvor lister mindre end 30 sorteres effektivt af insertionsort.

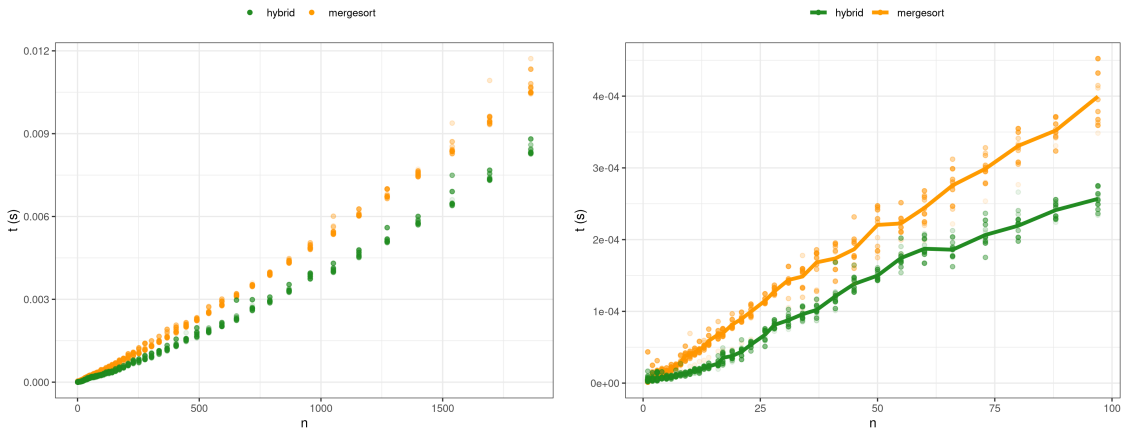
listen hurtigere. Det er vigtigt at pointere, at algoritmernes udførelsestid er nogenlunde ens ved $n \approx 37$, og at det derfor ikke ville være hurtigere at splitte listen op med mergesort, til alle dellisterne havde en længde på 37, og derefter sortere med insertionsort. For at sikre at insertionsort kun sorterer lister, hvor den er hurtigst, splitter hybridalgoritmen inputtet til dellisterne er højest 30 elementer lange (se linje 3 og 6 på figur 4.6).

4.5.1 Sammenligning af Optimerede Algoritmer

Udførelsestiderne er plottet på figur 4.7 på side 22.

Når man ser på grafen er det klart, at hybridalgoritmen gennemsnitligt er hurtigst, men ikke i samme grad som mergesort var insertionsort overlegen ved høje n. Det ligner på grafen, at hybridalgoritmen ligesom mergesort gennemsnitligt kører med $\Theta(n \cdot \log n)$. Vi kan dog ikke være sikker på hybridalgoritmens tidskompleksitet, da vi ikke har fortaget den teoretiske analyse af algoritmen. Hvad der er sikkert, er at hybridalgoritmen er hurtigere end mergesort på computeren, som algoritmen er testet på. Det ser dog ud til, at algoritmerne i hvert faldt gennemsnitligt har nogenlunde samme vækstrate. En anden måde vi kan argumentere for, at hybridalgoritmens tidskompleksitet i hvert fald ikke er mindre end $O(n \log n)$, er (som vi beviste i afsnit 3.5) at enhver sorteringsalgoritme der fungerer ved brug af sammenligninger, ikke kan have en mindre tidskompleksitet end $O(n \cdot \log n)$. Hybridalgoritmen har enten den samme eller værre tidskompleksitet end mergesort. Det, der skaber forskellen, må være en et mindre led i hybridalgoritmens $T(n)$ eller en konstant, der ganges på det dominerende led.

Nu ser det ud som om, at hybridalgoritmen altid er det bedste valg ved store n, men det kan vi faktisk ikke være sikre på. Problemet er, at det eneste vi ved om hybridalgoritmens tidskompleksitet, er at algoritmen i



Figur 4.7: Reelle udførelsestider for mergesort og hybridalgoritme. Plottet til højre er samme plot som til venste bare zoomet ind, og med en linje der gør mellem gennemsnitet af t for hver n

værste fald er $\Omega(n \cdot \log n)$ ligesom alle andre sammenlignings-sorterings-algoritmer. Det betyder, at vi ikke ved, om hybridalgoritmen ved et exceptionelt dårligt udfald har en astronomisk lang udførelsestid. Der er altså ingen garanti for algoritmens udførelsestid. Mergesort på den anden side ved vi har tidskompleksitet $\Theta(n \cdot \log n)$, hvilket garanterer udførelsestidens vækstrate. Selvfølgelig ville dette problem forsvinde, hvis det viste sig, at hybridalgoritmen også var $\Theta(n \cdot \log n)$, men da vi ikke ved det med sikkerhed, kan vi ikke være sikre på, at hybridalgoritmen altid er bedre ved store n-værdier. Hvis det derimod viste sig, at hybridalgoritmen havde en tidskompleksitet på $O(n^2)$ ligesom insertionsort (som den jo gør brug af), ville det betyde, at algoritmen netop ville have en ekstrem lang udførelsestid i værste tilfælde. Så ville den pæne graf på figur 4.7 være ret så ligegyldig, da et input med høj kardinalitet i værste tilfælde ville tage utrolig lang tid. Den lille mængde tid hybridalgoritmen i gennemsnit sparer, ville da ikke være risikoen for meget lange udførelsestider værd.

5 Konklusion

Sorteringsalgoritmers reelle køretider er svære at forudsige, selvom deres opgave lader til at være simpel. De sorterer data på baggrund af dataens attributter. For at klassificere algoritmernes opførsel inddeler vi dem i kompleksitsklasser ved hjælp af store-O-analyse og værste-tilfælde-vækstrate. Binære træer kan bruges til at finde en generel nedre grænse for en sorteringsalgoritmes værste-tilfælde-vækstrate. Denne grænse er en værste-tilfælde-grænse, derfor har insertionsorts bedste-tilfælde-tidskompleksitet på $\Theta(n)$ ingen betydning. Sorteringsalgoritmerne insertionsort og mergesort sorterer begge lister, men er gode til forskellige opgaver: mergesort er god til store input, hvilket forklares af dens tidskompleksitet på $\Theta(n \cdot \log n)$. Det viste sig i testen, at insertionsort i gennemsnit var hurtigere, på computeren algoritmerne blev testet på, hvis $n < 37$. At insertionsort var langt værre til sortering af store lister, var ikke en overraskelse, da vi i afsnit 2.4 analyserede algoritmen og kom frem til, at algoritmen havde tidskompleksiteten $O(n^2)$. Det er en meget højere værste-tilfælde-vækstrate end mergesorts. På baggrund af denne analyse kunne de to algoritmer sammensættes til en hybridalgoritme med det formål at optimere algoritmens udførelsestid. Hybridalgoritmen så ud til at være en optimeret version af mergesort, men vi ved ikke tilstrækkeligt om algoritmens tidskompleksitet til at kunne bestemme, om den altid vil være et bedre valg end mergesort.

Litteraturliste

1. Binary Search Trees: BST Explained with Examples. https://www.freecodecamp.org/news/binary-search-trees-bst-explained-with-examples (nov. 2019).
2. Dietzfelbinger, M. & Mehlhorn, K. *Algoritmer og datastrukturer* https://github.com/thorehusfeldt/algoritmer-og-datastrukturer/blob/master/ad-book.pdf ().
3. Farolino, D. Binary Tree Inductive Proofs. https://blog.domfarolino.com/Binary-Tree-Inductive-Proofs/#perfect-binary-tree-proof-3 (jul. 2016).
4. Huang, S. *What is Big O Notation Explained: Space and Time Complexity* jan. 2020. https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c.
5. Jago, M. *Turing & The Halting Problem - Computerphile* (red. Riley, S.) aug. 2014. https://www.youtube.com/watch?v=macM_MtS_w4.
6. Know Thy Complexities! https://www.bigocheatsheet.com.
7. Leclaire, N. *Build a Linked List For Each Layer in a Binary Tree* mar. 2014. https://nathanleclaire.com/blog/2014/03/01/build-a-linked-list-for-each-layer-in-a-binary-tree.
8. Nayyar, R. S. Tree Data Structure. *Scaler Topics.* https://www.scaler.com/topics/data-structures/tree-data-structure (dec. 2021).
9. random — Generate pseudo-random numbers. https://docs.python.org/3/library/random.html.
10. *Sorting Algorithm* https://www.programiz.com/dsa/sorting-algorithm.
11. Thompson, J. Big O Notation Examples – Time Complexity and Algorithm Efficiency Explained. https://www.freecodecamp.org/news/big-o-notation-examples-time-complexity-explained (mar. 2021).
12. What is Python? Executive Summary. https://www.python.org/doc/essays/blurb.

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]):
```

```
15          c.append(a[0])
16          a.pop(0)
17       else:
18          c.append(b[0])
19          b.pop(0)
```

Kode til test af algoritmerne

```
1  import random
2  import time
3  import pandas as pd
4  import os
5
6
7  # Denne funktion timer køretiden af en funktion med input l og returnerer funktionen køretid i
       milisekunder
8  def test(fun,l):
9     start_time = time.perf_counter()
10
11    fun(l)
12
13    return(time.perf_counter() - start_time)
14
15 # Denne funktion returnerer en liste af tilfældige tal mellem 0 og 1000, med n elementer
16 def createRandomList(n):
17    return([random.randint(0,1000) for i in range(n)])
18
19 # Laver en mappe i filsystemet hvis der ikke allerede er en med stien
20 def makeIfNeeded(dir_path):
21    if(os.path.isdir(dir_path) == False):
22       print(f"made dir: {dir_path}")
23       os.mkdir(dir_path)
24    return(dir_path)
25
26 # Finder det næste versionsnummer for til navngivning af fil på baggrund af indholdet i en folder
27 def newVersionNumber(dir_path,extention):
28    file_names = os.listdir(dir_path)
29    version = 0
30
31    thisfilename = f"{version}{extention}"
32
33    while(thisfilename in file_names):
34       version += 1
35       thisfilename = f"{version}{extention}"
36
```

```
37      return(thisfilename)
38
39  # Dette er funktionen der tester en liste med funktioner og gemmer deres køretider
40  def fullTest(functions):
41
42
43      data_dir = "../data/"
44      version_number = newVersionNumber(data_dir,"")
45
46      # hvor mange datapunkter pr. n-værdi
47      trials = 10
48
49      # Bruger tidspunkt som frø til pseudotilfældige tal.
50      seed = time.time()
51      print(f"Seed: {seed}")
52
53      for function in functions:
54
55          # i denne liste gemmes antallet af elementer at den liste som algoritmen sorterer for hvert
                datapunkt.
56          ns = []
57          # i denne liste gennes den tid det tager at sorterer listen med n elementer
58          times = []
59
60          # Bruger det samme seed til test at hver algoritme. på den måde er det de samme
                pseudo-tilfældige liste som algoritmerne sorterer
61          random.seed(seed)
62
63          # Vi laver testen et antal (trials) gange pr. n-værdi
64          for trial in range(0,trials):
65
66              # En lykke der køre et abitrært antal gange (jo højere en i-værdi jo højere maks antal
                    elementer i listen)
67              for i in range(0,80):
68
69              # Jeg bruger en potensfunktion til at fa flere datapunker tættere på y-aksen og færre
                    lange oberationer (pga. lange liste)
70                  n = round(pow(1.1,i))
71
72                  # lidt feedback
73                  print(f"function=\"{function.__name__}\": Trial: [{trial+1}/{trials}] {i=},{n=}")
74
75                  # gennererer en tilfældig liste med længden n
76                  l = createRandomList(n)
77
78                  # gem størrelsen af listen der skal sorteres
79                  ns.append(n)
80                  # gem den tid det tager at sortere listen
```

```
81            times.append(test(function,l))
82
83
84            # gemmer data
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)
```

Bilag 2 - Databehandling og Plots

Kode til databehandling og generering af plots

```
library(ggplot2)
library(plyr)
#library(pubr)
#library(tikzDevice)

#working dir
setwd("/home/Balder/Documents/Skole/Gym/SRP/data/9")

#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)

# punktmængder for hver algoritme
m_merge = subset(M,M$algorithm=="mergesort")
m_insertion = subset(M,M$algorithm=="insertionsort")
```

```
34
35 # laver modeller
36 model_merge = nls(t~a*n*log2(n), data=m_merge, start=list(a=0.000001))
37 model_insertion = nls(t~a*n^2 + b*n + c, data=m_insertion, start=list(a=1,b=1,c=1))
38
39
40 # Sætter ny path til hvor outputtet skal være
41 setwd("/home/Balder/Documents/Skole/Gym/SRP/img")
42
43 # gemmer r2-værdierne i to filer
44 writeLines(toString(round(with(m_merge,cor(t,n)),digits=3)),"r2-merge.txt")
45 writeLines(toString(round(with(m_insertion,cor(t,n)),digits=3)),"r2-insertion.txt")
46 print("r2 saved to files")
47
48
49 # laver modelerede v?rdier for hver n
50 m_merge$model = predict(model_merge)
51 m_insertion$model = predict(model_insertion)
52
53 m_merge$residual = resid(model_merge)
54 m_insertion$residual = resid(model_insertion)
55
56 # kombinerer de to
57 M = rbind(m_merge,m_insertion)
58
59 summary(M)
60
61
62 farve = c("#3a6ced","#ff9b00")
63
64
65 ggplot(M, aes(x=n, y=t, colour=Algorithm)) +
66    scale_color_manual(values = farve)+
67    labs(y="t (s)")+
68    geom_point(size=1.5,alpha=0.1,shape=19) +
69    geom_line(aes(x=n, y=model,color=Algorithm), size=2, alpha=0.6) +
70    guides(colour = guide_legend(override.aes = list(alpha = 1))) + # lav legend alpha 1
71    theme_bw()+
72       theme(legend.position="none")
73
74    ggsave("toAlgoritmer.png",width=8,height=6,scale=0.7)
75
76    # laver zoomed
77 ggplot(subset(M,n <= 100), aes(x=n, y=t, colour=Algorithm)) +
78    labs(y="t (s)")+
79    scale_color_manual(values = farve)+
```

```
80    geom_point(size=1.5,alpha=0.1,shape=19) +
81    geom_line(aes(x=n, y=model,color=Algorithm), size=2, alpha=0.6) +
82    theme(legend.position = c(.9, .9)) + # virker ikke!!
83    guides(colour = guide_legend(override.aes = list(alpha = 1))) + # lav legend alpha 1
84       theme_bw()
85
86 ggsave("toAlgoritmerZoomed.png",width=8,height=6,scale=0.7)
87
88
89
90
91
92
93 ggplot(M, aes(x=log10(model), y=residual, colour=Algorithm)) +
94    geom_hline(yintercept = 0)+
95    scale_color_manual(values = farve)+
96    geom_point(size=1.5,alpha=0.2,shape=19) +
97    labs(x="Log10 af Modellens Forudsigelse", y="residualer") +
98    facet_wrap(~algorithm,scales="free",ncol=1) +
99    guides(colour = guide_legend(override.aes = list(alpha = 1))) + # lav legend alpha 1
100   theme_bw() +
101      theme(legend.position="top",legend.title = element_blank())
102
103
104   ggsave("toAlgoritmerResidual.png",width=8,height=6,scale=0.7)
105
106
107
108
109
110 # Laver ges data.frame
111 f = function(x) {
112    data.frame(
113         t = mean(x$t)
114    )
115 }
116
117 gns = ddply(subset(M,n <= 100), .(Algorithm, n), f)
118
119 ggplot(subset(M,n <= 100), aes(x=n, y=t, colour=Algorithm)) +
120    labs(y="t (s)")+
121    scale_color_manual(values = farve)+
122    geom_point(size=1.5,alpha=0.08,shape=19) +
123    geom_line (size=1.5, data=gns) +
124    guides(colour = guide_legend(override.aes = list(alpha = 1))) + # lav legend alpha 1
125    theme_bw() +
```

```
126       theme(legend.position="top",legend.title = element_blank())
127
128     ggsave("toAlgoritmerZoomedGns.png",height=6, width=9)
```