next
UDDANNELSE KØBENHAVN

Studieområdeprojekt
Vestskoven Gymnasium, HTX 2021-2022

SOP - Opgaveformulering

Navn:	**Rasmus Kroghede Hansen**
Klasse:	**3v**

Opgaveformulering:

Giv en kort introduktion til forskellige bevismetoder i matematik.

Redegør i detaljer for induktionsbevismetoden herunder induktionsbasen, induktionsantagelsen og induktionstrinnet. Medtag gerne et konkret bevis-eksempel (du kan f.eks. eftervise Binets formel).

Introducér sorteringsproblemet og analysér og sammenlign de to sorteringsalgoritmer InsertionSort og MergeSort. Du kan f.eks. analysere og sammenligne deres køretider. Vis endvidere deres korrekthed ved at opstille passende løkkeinvarianter.

Diskutér kort lighederne mellem dit bevis ved matematisk induktion og metoden, du benyttede til at vise korrektheden af InsertionSort og MergeSort.

Studieretningsfag:	**Matematik A**
Vejleder:	**Hans Henrik Knudsen hhk@nextkbh.dk**

Andet fag:	**Programmering B**
Vejleder:	**Peter Sterner pets@nextkbh.dk**

Beviser og Sorteringsalgoritmer

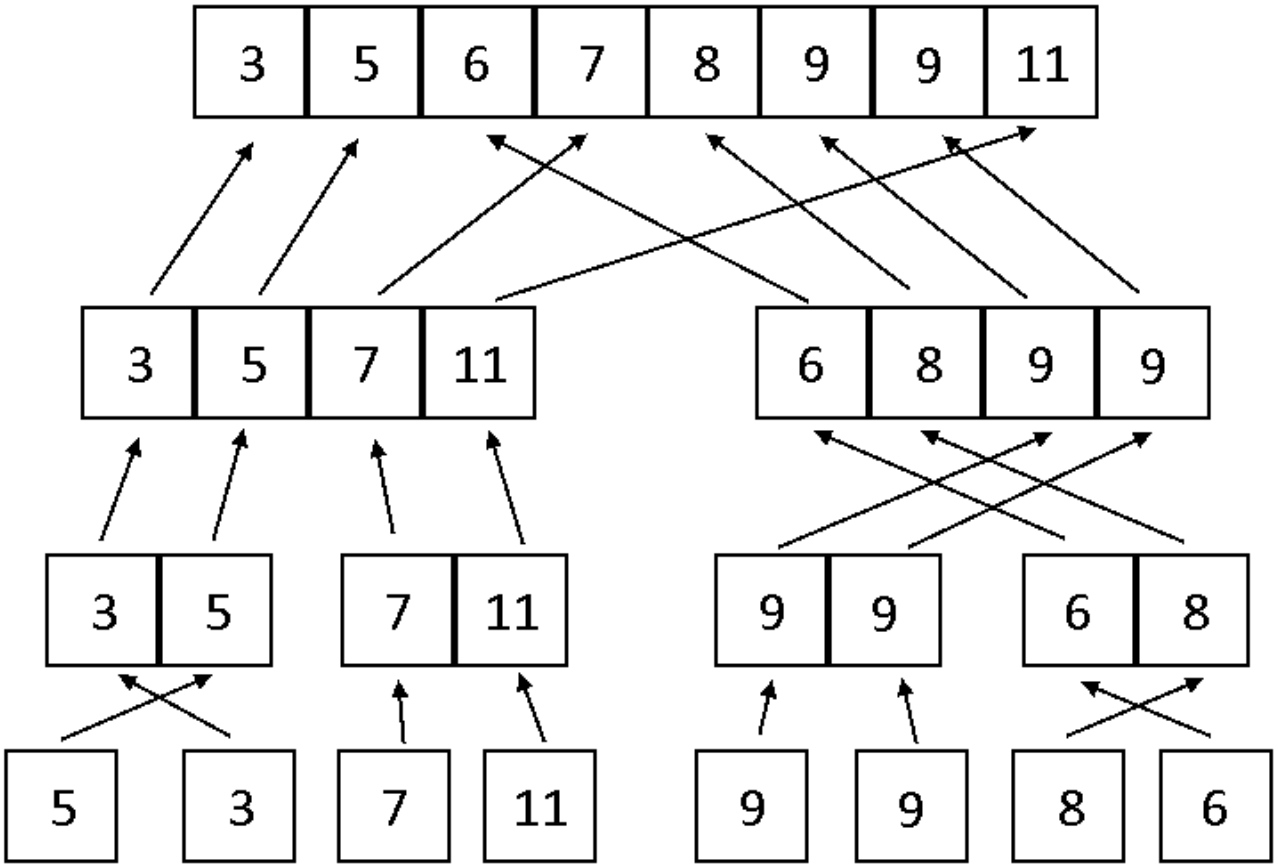
Matematik A og Programmering B

Af Rasmus Kroghede Hansen

3.v Vestskoven Gymnasium

2021

Vejledere: Hans Henrik Knudsen og Peter Sterner



Resume

Formålet med denne opgave var at undersøge forskellige bevismetoder og analysere de to sorteringsalgoritmer InsertionSort og MergeSort. I de indledende afsnit blev der redegjort for bevismetoderne direkte bevis, modstridsbevis og induktionsbevis. I det direkte bevis, beviste man sandheden af et udtryk gennem en række af trin, hvor ens udledelser byggede på allerede etablerede fakta. Denne metode kunne bruges til at vise at summen at to lige heltal giver et lige heltal. Modstridsbeviset var en måde at bevise at et udtryk var sandt ved at antage at det modsatte af udtrykket var sandt og vise at dette førte til en modstrid. Denne metode kunne bruges til at bevise at $\sqrt{2}$ er et irrationelt tal. I induktionsbeviset viste man at et udtryk var sandt for et bestemt positivt heltal (induktionsbasen), man antog herefter at udtrykket var sandt for et arbitrært positivt heltal k (induktionsantagelsen) og viste så at $k+1$ også blev sand (induktionstrinnet). Ud fra dette kunne man slutte at udtrykket var sandt for alle positive heltal. Denne metode blev brugt til at vise at udtrykket $1+2+3+\cdots+n=\frac{n\cdot(n+1)}{2}, n \in Z_+$ er sandt for alle positive heltal. Herefter viste en analyse af sorteringsalgoritmerne InsertionSort og MergeSort at MergeSort var hurtigst til at sortere tal, når mængden af tal var stor nok. Kun i tilfælde hvor tallene var sorterede i forvejen var InsertionSort hurtigere. Algoritmerne blev også testet i praksis ved at blive implementeret og kørt i JavaScript og resultaterne af dette kunne bekræfte analysen. Til sidst blev induktionsbevismetoden sammenlignet med løkkeinvariant-metoden, som der blev brugt til at vise korrektheden af sorteringsalgoritmerne. Det kunne konkluderes at induktionsbasen kunne sammenlignes med initialisering-trinnet og induktionstrinnet kunne sammenlignes med vedligeholdelses-trinnet. Endvidere kunne løkkeinvarianten sammenlignes med det udtryk man skulle bevise i induktionsbeviset. Forskellen på de to metoder var at induktionsbeviset fortsatte ud i det uendelige hvorimod løkkeinvariant-metoden stoppede når løkken havde kørt færdig.

Indhold

Indledning

I denne opgave vil emnerne bevisførelse og algoritmeanalyse blive undersøgt.

Der vil blive introduceret forskellige matematiske bevismetoder, herunder induktionsbevismetoden. Begreberne induktionsbasen, induktionsantagelsen og induktionstrinnet vil blive forklaret. Herefter vil der blive givet et konkret bevis-eksempel hvor induktionsbevismetoden benyttes.

Efter dette vil der blive givet en introduktion til sorteringsproblemet. Her vil de to sorteringsalgoritmer InsertionSort og MergeSort blive forklaret og korrektheden af disse vil blive vist ved at opstille passende løkkeinvarianter. Hernæst vil algoritmerne blive analyseret og sammenlignet ved at kigge på deres køretider.

Til slut vil der blive diskuteret lighederne mellem induktionsbevismetoden og metoden, der blev benyttet til at vise korrektheden af InsertionSort og MergeSort.

1 Bevismetoder i matematik

I matematikken bruger man beviser til at vise om udtryk er sande eller falske. Der er dog mange forskellige bevismetoder til at komme frem til sandhedsværdien af et udtryk. Dette afsnit vil introducere to af disse metoder: det direkte bevis og modstridsbeviset. I afsnit 2 vil der blive redegjort i detaljer for en anden bevismetode der kaldes induktionsbeviset.

1.1 Direkte bevis

Et direkte bevis er et bevis hvor man beviser sandhedsværdien af et udtryk ud fra allerede etableret viden, uden at gøre nogen yderlige antagelser (Wikipedia, 2021). Dette gøres ved at gå frem i en række af trin, hvor man i hvert trin udnytter en slutningsregel (Terstrup, 1974, p. 54).

Direkte bevis kan blandt andet bruges til at bevise, at summen af to vilkårlige lige heltal er lig med et lige heltal:

Givet: to lige heltal x og y

Da x og y er lige, kan de skrives som

$x = 2a$ og $y = 2b$, hvor a og b er heltal.

Summen bliver da $x + y = 2a + 2b = 2(a + b) = 2 \cdot c$, $hvor\ c = a + b$

Det kan ses at x+y har 2 som en faktor og er derfor et lige heltal. Derfor er summen af to vilkårlige lige heltal lig med et lige heltal (Wikipedia, 2021). Det kan ses at dette bevis blandt andet byggede på den etablerede viden om at et lige heltal kan skrives som $2 \cdot k$, hvor k er et heltal.

1.2 Modstridsbevis

Med modstridsbeviset (eller bevis ved modsigelse) bevises et udtryk sandt ved at antage at det modsatte udtryk er sandt og herefter vise at dette fører til en modstrid (Terstrup, 1974, p. 71).

Det kan f.eks. bevises at $\sqrt{2}$ er et irrationelt tal ved at antage at $\sqrt{2}$ er rationelt tal. Hvis $\sqrt{2}$ er rationelt kan det skrives som en uforkortelig brøk $\frac{a}{b}$ hvor a og b er heltal og ikke går op i hinanden, det vil sige mindst en af dem må være et ulige tal. Hvis $\frac{a}{b} = \sqrt{2}$ så er $a^2 = 2b^2$, derfor må a^2 være lige og siden kvadratet af et ulige tal er et ulige tal, så må a også være lige. Dette betyder at b må være det ulige tal i brøken. Men hvis a er lige, må a^2 kunne skrives på formen $4 \cdot c^2$ hvor c er et heltal. Hvis der reduceres, får man $2c^2 = b^2$.

b^2 er derfor lige og så må b også være lige. Det er nu vist at b er både lige og ulige. Dette er en modstrid og må betyde at antagelsen om at $\sqrt{2}$ er et rationelt tal er forkert. Derfor må $\sqrt{2}$ være et irrationelt tal.

2 Induktionsbevismetoden

Induktionsbevismetoden er en form for direkte bevis, der kan bruges til at bevise at et udtryk er sandt for alle $n \in Z_+$ eller for alle $n \in Z_+ \; og \; n > r, hvor \; r \; er \; et \; positivt \; heltal$ (Terstrup, 1974, p. 60).

Induktionsbevismetoden kan også anvendes hvis n tilhører en anden talmængde end Z_+ såfremt denne talmængde kan bringes i en enentydig overensstemmelse med de positive heltal (Terstrup, 1974, p. 60). Det vil sige at den for eksempel også kan anvendes på talmængderne N og N_0.

Induktionsbevismetoden består af 2 skridt: Induktionsbasen og induktionstrinnet.

I induktionsbasen eller basisskridtet, viser man at udtrykket, der skal bevises, er sandt for en bestemt værdi af n, for eksempel n=1 (Terstrup, 1974, p. 60).

I induktionstrinnet antager man først at udtrykket er sandt for en arbitrær værdi k. Denne antagelse kaldes induktionsantagelsen. Ud fra denne antagelse viser man at udtrykket også er sandt for $k + 1$ (Terstrup, 1974, p. 60).

Ud fra dette kan man slutte at udtrykket gælder for alle $n \in Z_+$, da det i induktionsbasen blev vist at udtrykket var sandt når $n = 1$. Hvis man lader $k = 1$, må udtrykket ifølge induktionstrinnet også være sandt for $k + 1 = 2$. Hvis det er sandt for 2, må det også være sandt for $2 + 1 = 3$ og sådan kan man fortsætte ud i det uendelige.

Man kan sammenligne induktionsbeviset med et uendeligt højt tårn af byggeklodser, som er stablet ovenpå hinanden. Hvis man i induktionsbasen viser, at den nederste byggeklods vælter og man i induktionstrinnet viser at hvis en byggeklods vælter, så vælter byggeklodsen oven på denne også, så kan man slutte at hele tårnet vil vælte.

Nogle gange kan man i induktionsbasen vise at udtrykket gælder for mere end én specifik værdi af n, for eksempel for både $n = 1$ og $n = 2$ og ligeledes i induktionsantagelsen, så antage at udtrykket gælder for mere end én arbitrær værdi. Dette skal for eksempel gøres hvis man vil bevise Binet's formel. Et bevis for Binet's formel kan findes i Bilag A.

2.1 Eksempel på induktionsbevis

Et eksempel på et udtryk som kan bevises med et induktionsbevis, er udtrykket:

$$1 + 2 + 3 + \cdots + n = \frac{n \cdot (n+1)}{2} \; , n \in Z_+ \tag{2.1}$$

I induktionsbasen skal det nu bevises at udtrykket er sandt for $n = 1$.

Når $n = 1$ bliver venstresiden blot 1.

Højresiden bliver $\frac{1 \cdot (1+1)}{2} = \frac{2}{2} = 1$

Da venstresiden og højresiden er lige med hinanden er udtrykket nu vist sand for $n = 1$

Som induktionsantagelse antages det at udtrykket er sandt for $n = k$. Udtrykket giver da følgende:

$$1 + 2 + 3 + \cdots + k = \frac{k \cdot (k+1)}{2} \tag{2.2}$$

Nu går induktionstrinnet ud på, på baggrund af ovenstående antagelse, at vise at udtrykket er sandt for $n = k + 1$.

Hvis udtrykket er sandt for $n = k + 1$ skal det give:

$$1 + 2 + 3 + \cdots + (k + 1) = \frac{(k + 1) \cdot \big((k + 1) + 1\big)}{2}$$

Dette kan også skrives som

$$1 + 2 + 3 + \cdots + k + (k + 1) = \frac{(k + 1) \cdot (k + 2)}{2} \qquad (2.3)$$

(*2.3*) skal nu vises fra (*2.2*). Hvis man lægger $k + 1$ til begge sider af (*2.2*) får man:

$$1 + 2 + 3 + \cdots + k + (k + 1) = \frac{k \cdot (k + 1)}{2} + (k + 1)$$

Hvis det nye $(k + 1)$ på højresiden sættes ind i brøken får man:

$$1 + 2 + 3 + \cdots + k + (k + 1) = \frac{k \cdot (k + 1) + 2 \cdot (k + 1)}{2}$$

Det ses at begge led i brøken ganges med $(k + 1)$, dette sættes udenfor en parentes:

$$1 + 2 + 3 + \cdots + k + (k + 1) = \frac{(k + 1) \cdot (k + 2)}{2}$$

Nu fremkommer (*2.3*). Det er derfor vist at udtrykket er sandt for $n = k + 1$, hvis det er sandt for $n = k$. Det vides fra induktionsbasen at udtrykket er sandt for $n = 1$, så hvis man lader $k = 1$, så må udtrykket også være sandt for $n = k + 1 = 2$. Hvis man nu lader $k = 2$, så bliver udtrykket sandt for $n = k + 1 = 3$ osv. Det er hermed bevist at udtrykket er sandt for alle $n \in Z_+$.

3 Sorteringsproblemet

Man møder sortering mange steder i hverdagen. Hvis man f.eks. shopper på nettet og gerne vil sorterer varerne efter pris. Hvis man arbejder med et datasæt, er det også nødvendigt at have det sorteret hvis man f.eks. gerne vil finde medianen. Der er mange forskellige situationer hvor man gerne vil have ændret rækkefølgen på et sæt tal, så de kommer i en opstigende rækkefølge, altså at de laveste tal står først og de højeste står sidst. Dette problem kaldes sorteringsproblemet og formelt er det defineret som følger:

Input: En rækkefølge bestående af n tal $< a_1, a_2, \dots, a_n >$.

Output: En permutation af denne rækkefølge $< a'_1, a'_2, \dots, a'_n >$, således at $a'_1 \leq a'_2 \leq \cdots \leq a'_n$ (Cormen, et al., 2009, p. 5)

Hvis inputtet for eksempel er $< 5,3,7,4,1 >$ så vil outputtet blive $< 1,3,4,5,7 >$.

For at løse sorteringsproblemet kan man anvende algoritmer. En algoritme er en præcist beskrevet procedure, som tager et input og producerer et output. En algoritme består altså af en række trin, der skal tages for at forvandle inputtet til outputtet (Cormen, et al., 2009, p. 5). En algoritme kan derfor opfattes som en bageopskrift, hvor inputtet er ingredienserne og outputtet er en kage. Bageopskriften fortæller hvad der skal gøres med ingredienserne for at lave kagen og ligeledes fortæller en algoritme hvad der skal gøres med inputtet for at lave outputtet.

En algoritme siges at være korrekt hvis den for ethvert input standser med det korrekte output. Man siger at en korrekt algoritme løser det givne problem (Cormen, et al., 2009, p. 6). Algoritmer der løser sorteringsproblemet kaldes for sorteringsalgoritmer. Dette afsnit vil kigge nærmere på sorteringsalgoritmerne InsertionSort og MergeSort, som løser sorteringsproblemet på vidt forskellige måder. Algoritmerne vil blive forklaret og deres korrekthed vil blive vist. Herefter vil deres køretid blive analyseret og sammenlignet.

3.1 InsertionSort

InsertionSort er en algoritme der løser sorteringsproblemet. InsertionSort fungerer på samme måde som mange mennesker sorterer en hånd når de spiller med kort. Man starter med en tom hånd og en bunke kort på bordet. Man tager herefter ét kort fra bunken ad gangen og sammenligner det med de kort man har på hånden, fra højre til venstre, indtil man finder et kort i hånden som er mindre end eller lig med det kort man prøver at indsætte. Man indsætter så kortet til højre for dette kort (Cormen, et al., 2009, p. 17). Når man har indsat det sidste kort fra bunken, står man tilbage med en sorteret hånd.

For at vise hvordan InsertionSort fungerer, kan den opskrives i pseudokode. Pseudokode er et uformelt programmeringssprog, hvor man beskriver en algoritmes trin så letforståeligt som muligt. Pseudokoden for InsertionSort ser således ud:

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      i = j - 1
4      while i > 0 and A[i] > key
5          A[i+1] = A[i]
6          i = i - 1
7      A[i+1] = key
```

Inden selveste InsertionSort pseudokoden bliver forklaret, vil der blive redegjort for de grundlæggende begreber i pseudokode.

Når der over linje 1 står INSERTION-SORT(A), betyder det man laver en såkaldt procedure kaldet INSERTION-SORT, som tager et input A. I dette tilfælde er A, den liste som skal sorteres og denne liste har en længde på n.

Når der skrives A[i] betyder det at man tilgår det i'te element i listen A. Når i bruges sådan, kaldes i for en indeks. Når man skriver A[$1..j$] indikerer man at man snakker om en delliste af A der indeholder elementerne $A[1], A[2], A[3]$ … $A[j]$

Man tildeler værdi til en variabel ved at bruge "=" tegnet. Hvis man f.eks. skriver b = 3, så tildeler man værdien 3 til b.

På linje 1 laves en **for**-løkke. Syntaksen for en for-løkke er "**for** a = b **to** c". Når en for-løkke laves, starter den med at sætte $a = b$. Herefter tjekker der om $a \leq c$. Er dette sandt kører den alt koden i løkkens krop én gang hvor $a = b$. Løkkens krop er alt under for-løkke linjen, som er indrykket i forhold til linjen. Når løkkens krop køres, kaldes det en iteration. Efter en iteration gøres a én større. Nu tjekkes igen om $a \leq c$, og i så fald køres endnu en iteration, hvor a denne gang er én større end forrige iteration. Sådan fortsættes det indtil $a > c$ og så går koden videre til den næste linje som ikke er rykket ind under for-løkken. Eller den stopper hvis der ikke er flere linjer. Selveste for-løkke linjen køres i alt $c - b + 2$ gange, da dennes linje køres alle gangene når $a \leq c$ og én gang når $a > c$ og løkkens krop køres $c - b + 1$ gange, da denne ikke køres når $a > c$.

På linje 4 laves en **while** løkke. Syntaksen for en while løkke er "**while** a" hvor a er et boolsk udtryk (SANDT/FALSK). while løkken starter med at tjekke om a er sandt, hvis a er sandt, udføres en iteration af løkkens krop. Herefter tjekkes igen om a er sandt, hvis a stadig er sandt, udføres endnu en iteration. Denne proces fortsætter til a er falsk, her stopper koden med at køre iterationer af while-løkken.

Nu når de basale elementer af pseudokode er på plads, vil pseudokoden for InsertionSort bliver forklaret. På linje 1 laves en **for**-løkke hvor $j = 2$ til start og den kører så længe $j \leq A.length$. A.length er længden af listen A. På linje 2 laves en variabel key som sættes til at være A[j]. Det er key, som skal sorteres i denne iteration af for-løkken. Herefter laves der på linje 3 en variabel $i = j - 1$. A[i] er derfor det tal, som er én plads til venstre for A[j] i listen.

På linje 4 laves en while løkke, der kører så længe $i > 0$ og $A[i] > key$. I løkkens krop, på linje 5, sættes $A[i + 1] = A[i]$. Med andre ord så rykkes A[i] én gang til højre, så det nu har pladsen A[$i + 1$]. Efter A[i] er rykket til højre gøres i én mindre på linje 6, næste gang vil der derfor blive sammenlignet med elementet til venstre for i.

Til sidst, når $i = 0$ eller $A[i] \leq key$ bliver $A[i+1] = key$ på linje 7. Hvis $i = 0$, blev der ikke fundet en værdi for i, hvor $\mathrm{A[i]} \leq \mathrm{key}$ og derfor bliver key indsat på A[1], den første plads i listen. Hvis **while**-løkken stoppede fordi $A[i] \leq key$ så indsættes key én plads til højre for A[i]. Dette giver mening da key, da må være $\geq A[i]$ og derfor skal key placeres til højre for A[i], for at være sorteret. Så er iterationen færdig og en ny kan begynde hvor j er én større end før.

3.1.1 Korrekthed af InsertionSort

Metoden til at bevise korrektheden af en algoritme, minder meget om induktionsbevismetoden. Lighederne mellem disse metoder vil blive diskuteret i afsnit 4. For at vise korrektheden af en algoritme opstiller man en såkaldt løkkeinvariant. En løkkeinvariant er et udtryk der kan bruges til at fortælle at en algoritme er korrekt. Der skal vises tre ting om løkkeinvarianten:

Initialisering: Løkkeinvarianten er sand før den første iteration af løkken.

Vedligeholdelse: Hvis løkkeinvarianten er sand før en iteration af løkken, forbliver den sand før den næste iteration af løkken. Det siges at den vedligeholdes.

Termination: Når løkken er kørt færdig, giver løkkeinvarianten en information der kan bruges til at vise at algoritmen er korrekt (Cormen, et al., 2009, p. 19).

Hvis man har vist initialiseringen og vedligholdelsen, kan man bruge samme proces som i induktionsbeviset til at slutte at invarianten må være sand gennem alle iterationer.

Hvis man vil vise korrektheden af InsertionSort, kan man opstille følgende løkkeinvariant: I starten af hver iteration af **for**-løkken på linje 1-7, består dellisten A[$1..j-1$] af de oprindelige elementer i A[$1..j-1$], men i en sorteret rækkefølge. (Cormen, et al., 2009, p. 18)

Det skal nu vises at denne løkkeinvariant er sand før den første iteration af for-løkken, at den vedligeholdes og at den ved løkkens afslutning kan bruges til at vise InsertionSorts korrekthed.

Initialisering: Før første iteration af for-løkken er $j = 2$ og det vil sige at dellisten A[$1..j-1$] bliver til A[$1..1$] og består derfor kun af ét element nemlig A[1], dette element er det oprindelige element A[1]. Siden der kun er ét element i dellisten, må dellisten være sorteret. Begge invariantens betingelser er derfor sande før den første iteration af løkken og derfor er løkkeinvarianten sand før den første iteration af løkken.

Vedligeholdelse: Hernæst skal det vises at hvis invarianten er sand før en iteration af løkken vil den også være sand inden den næste iteration af løkken. Her antages det altså at løkkeinvarianten er sand, det vil sige at A[$1..j-1$] består af de oprindelige elementer i A[$1..j-1$], men i en sorteret rækkefølge. Hvis man betragter **while**-løkkens krop så virker den på den måde at den på linje 5 rykker A[$j-1$], A[$j-2$] og så videre, én plads til højre indtil den korrekte plads for A[j] er fundet. A[j] sættes ind på denne plads på linje 7. Siden der kun rykkes på elementer i A[$1..j-1$] og de hver højest rykkes én plads til højre kan man nu sige at A[$1..j$] består af de oprindelige elementer i A[$1..j$].

Siden man har rykket alle elementerne som var større end A[j] én plads til højre og herefter har indsat A[j] mellem elementerne der er større end A[j] og elementer som var $\leq$ A[j], så må dellisten A[$1..j$] også være sorteret. Efter iterationen er kørt vil j blive gjort én større inden næste iteration, det der var A[$1..j$] bliver derfor nu A[$1..j-1$] og det er lige blevet vist at denne delliste indeholder de oprindelige elementer fra A[$1..j-1$], men i sorteret rækkefølge. Derfor er løkkeinvarianten sand inden næste iteration.

Termination: Betingelsen der gør at for-løkken afsluttes er at $j > A.length$. Siden j gøres én større efter hver iteration, må j være lig med $A.length + 1$, som var det samme som $n + 1$. Sættes dette ind på j's plads i løkkeinvarianten får man at dellisten A[1..n] består af de oprindelige elementer i A[1..n], men i en sorteret rækkefølge. Da hele listen havde n elementer må A[1..n] svare til hele listen og dermed er hele listen sorteret. Hermed er InsertionSorts korrekthed blevet bevist.

3.2 MergeSort

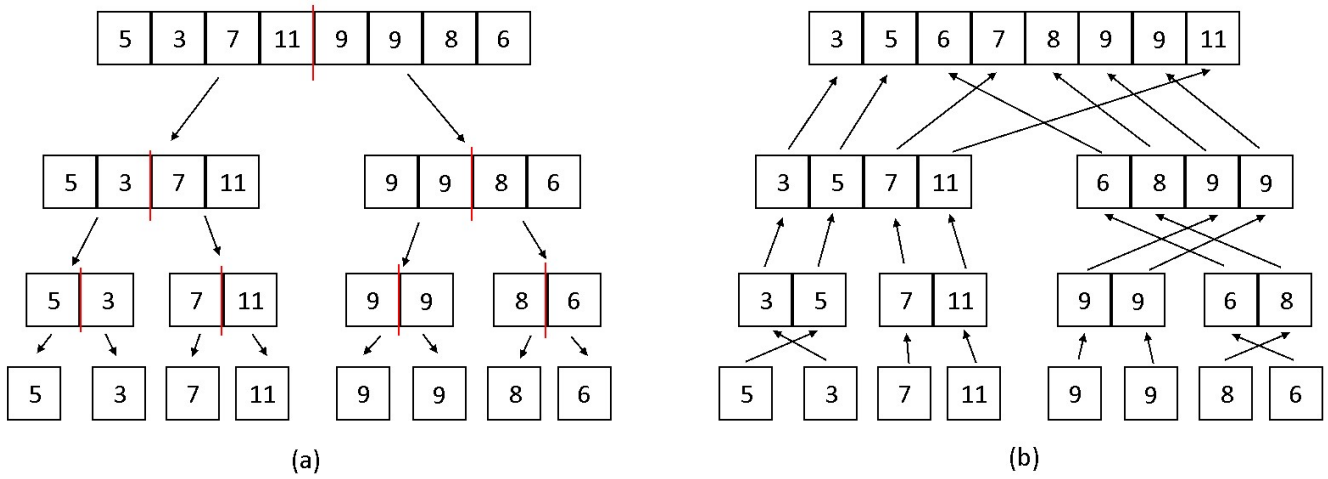
En anden algoritme, der kan bruges til at løse sorteringsproblemet, er MergeSort. MergeSort løser sorteringsproblemet rekursivt og gør brug af Del-og-hersk metoden. I del-og-hersk metoden gør man brug af tre simple trin, for hvert rekursionsniveau i algoritmen: Del, hersk og kombiner.

I del-trinnet deles problemet i flere underproblemer, som ligner det oprindelige problem. I MergeSort gøres dette ved at dele listen af tal midt over, så man har to halve lister, som man i stedet kan arbejde med.

I hersk-trinnet, besejrer man underproblemerne ved at løse dem rekursivt. Hvis størrelsen på underproblemerne er lille nok, løser man dem på en ligefrem måde. I MergeSort gøres dette ved rekursivt at halvere listerne indtil de ikke kan halveres længere, dvs. de har en længde på 1. Hvis listen har en længde på 1 er den per automatik sorteret.

I kombiner-trinnet, kombinerer man løsningerne på underproblemerne, sådan at man har løst det oprindelige problem. I MergeSort gøres dette ved at kombinere de to halve (og nu sorterede) lister, til én samlet sorteret liste (Cormen, et al., 2009, p. 30).

MergeSort bliver altså ved med at halvere listen på hvert rekursionsniveau indtil der til sidst er en masse lister med en længde på 1. Herefter kombineres disse lister så igen, sådan at man til sidst har en samlet sorteret liste. En illustration af denne proces kan ses på Figur 3.1 nedenfor.



*Figur 3.1 Illustration af MergeSort. **(a)** Viser hvordan MergeSort halverer input listen rekursivt, indtil man står tilbage med en masse lister, der kun indeholder ét tal. **(b)** Viser dellisterne blive kombineret til større og større sorterede lister, indtil man i toppen har en sorteret rækkefølge af den oprindelige liste.*

En vigtig del af MergeSort er kombineringen af underproblemerne. Man kan, forklare kombineringen i MergeSort med et kortspil. Man kan forestille sig at man har to sorterede bunker af kort ved siden af hinanden, som vender med forsiden opad, sådan at det mindste kort i hver bunke ligger øverst. Disse to bunker skal kombineres til én samlet sorteret bunke. Dette gøres ved at kigge på det øverste kort i de to bunker og se hvilket af dem er lavest, det kunne være kortet i venstre bunke. Dette kort tages nu fra

bunken og ligges med bagsiden nedad, i en ny bunke. Nu fremkommer et nyt kort i toppen af venstre bunke, dette sammenligner man nu med kortet i toppen af højre bunke. Igen tages det laveste kort og ligges i den tredje bunke. Processen fortsætter indtil en af bunkerne er tom. Nu kan resten af den ikke tomme bunke ligges oven i den nye bunke og så har man kombineret de to oprindelige bunker til én samlet sorteret bunke. Algoritmisk fungerer denne kombinering ved at lave en procedure der kaldes Merge(A, p, q, r), hvor A er en liste og p, q og r er indekser til denne liste således at $p \leq q < r$. Proceduren antager at A[$p..q$] og A[$q+1..r$] er sorterede. Den kombinerer disse to dellister til en sorterede delliste, der erstatter den nuværende delliste A[$p..r$] (Cormen, et al., 2009, p. 30). A[$p..q$] kan opfattes som den ene kortbunke og A[$q+1..r$] som den anden bunke og den sorterede delliste A[$p..r$], som bunken alle kortene blev puttet i. Pseudokoden for Merge proceduren er som følger:

```
MERGE(A,p,q,r)
 1   n1 = q - p + 1
 2   n2 = r - q
 3   lad L[1..n1+1] og R[1..n2+1] være nye lister
 4   for i = 1 to n1
 5       L[i] = A[p + i - 1]
 6   for j = 1 to n2
 7       R[i] = A[q+j]
 8   L[n1+1] = ∞
 9   R[n2+1] = ∞
10   i = 1
11   j = 1
12   for k = p to r
13       if L[i] <= R[j]
14           A[k] = L[i]
15           i = i + 1
16       else
17           A[k] = R[j]
18           j = j + 1
```

På linje 1 og 2 beregnes henholdsvis længden af dellisten A[$p..q$] og A[$q+1..r$] og disse gemmes som henholdsvis $n1$ og $n2$.

Herefter laves der på linje 3 to nye lister L og R, der henholdsvis gives længderne $n1+1$ og $n2+1$. Disse to lister skal holde en kopi af elementerne fra henholdsvis A[$p..q$] og A[$q+1..r$]. Det er L og R, der senere skal kombineres og danne en sorteret rækkefølge af tallene fra A[$p..r$]. L og R svarer derfor til de to kortbunker.

På linje 4 og 5 laves en løkke der kopierer elementerne fra A[$p..q$] til L og ligeledes laves der en løkke på linje 6 og 7 som kopierer elementerne fra A[$q+1..r$] til R.

Hernæst indsættes uendelig i slutningen af både L og R på linje 8 og 9.

Herefter sættes i og j til at være 1 på linje 10 og 11, disse to variabler skal holde styr på hvor langt man er nået i henholdsvis L og R, når de to lister bliver kombineret. i og j fortæller altså hvor langt man er nået i kortbunkerne.

På linje 12 laves en for-løkke hvor variablen $k = p$ til start og løkken kører så længe $k \leq r$, løkken vil derfor gennemløbe tallene fra p til og med r.

I hver iteration af for-løkken bliver der på linje 13 tjekket om $L[i] \leq R[j]$, altså om tallet man er nået til i L er mindre end eller lig med tallet man er nået til i R. Hvis dette er sandt, så sættes $A[k] = L[i]$ på linje 14 og i gøres én større på linje 15, ved næste sammenligning på linje 13, vil det derfor være det næste tal i L der bliver tjekket.

Hvis $R[j] < L[i]$ køres linje 17 og 18 i stedet. Her sættes $A[k] = R[j]$ og j gøres én større, ved næste sammenligning på linje 13 vil det derfor være det næste til i R der bliver tjekket. Hver iteration sættes det mindste kort af $L[i]$ og $R[j]$ altså ind på $A[k]$

Man indsætter uendelig i enden af de to lister for at undgå at tjekke om listerne eller "kortbunkerne" er tomme. Når man i stedet når til uendelig på linje 13, så ved man at det ikke kan være det mindste kort i toppen af de to bunker. Man kan derfor bare tage kort fra den anden bunke, medmindre toppen af begge bunker er uendelig. Men i så fald har man taget alle de andre kort fra bunkerne og så vil for-løkken ikke kører længere da den kun kører fra p til r altså $r - p + 1$ gange, men listerne indeholder $n1 + 1 + n2 + 1 = (q - p + 1) + 1 + (r - q) + 1 = r - p + 3$ tal, der er altså 2 tal tilbage og det er de to uendelig.

Når for-løkken fra linje 12 til 18 er kørt færdig er Merge-proceduren slut.

Merge proceduren kan nu bruges i en anden procedure kaldet Merge-Sort(A, p, r), som sorterer elementerne i dellisten A[$p..r$]. Pseudo-koden for denne procedure er:

```
MERGE-SORT(A,p,r)
 1   if p < r
 2       q = ⌊(p+r)/2⌋
 3       MERGE-SORT(A, p, q)
 4       MERGE-SORT(A, q+1, r)
 5       MERGE(A, p, q, r)
```

Hvis $p \geq r$ kan A[$p..r$] maksimalt have 1 element og er derfor allerede sorteret. Derfor køres linje 2 til 5 kun hvis $p < r$.

På linje 2 beregnes den midterste indeks i A[$p..r$]. Der tages den nedre heltalsgrænse af dette tal, da det ikke er muligt at have en brøk som indeks i en liste. Den midterste indeks gemmes i q. På linje 3 og 4 kaldes Merge-Sort rekursivt. Problemet deles altså i 2 og man får Merge-Sort til at sorterer det to dellister A[$p..q$] og A[$q + 1..r$]. Til sidst kombineres de to løsninger ved at kalde Merge(A, p, q, r) på linje 5. Merge proceduren kombinerer de to dellister A[$p..q$] og A[$q + 1..r$] til en sorteret liste og erstatter den nuværende A[$p..r$], så når linje 5 er kørt er A[$p..r$] nu sorteret.

For at sortere en hel liste $A =< A[1], A[2], \dots, A[n] >$ skal man da kalde Merge-Sort($A, 1, A.length$), hvor $A.length$ jo var længden på listen, altså n.

3.2.1 Korrekthed af MergeSort

For at vise korrektheden af MergeSort opstilles en løkkeinvariant for **for**-løkken på linjerne 12 til 18 i Merge-proceduren, da det er her at MergeSort ændrer på tallene i listen. Løkkeinvarianten er:

I starten af hver iteration af **for**-løkken på linje 12 til 18, består dellisten A[$p..k-1$] af de $k-p$ mindste elementer af L[$1..n1+1$] og R[$1..n2+1$] i en sorteret rækkefølge. Derudover er L[i] og R[j] de mindste tal i deres lister, som endnu ikke er blevet kopieret tilbage i A (Cormen, et al., 2009, p. 32).

Ligesom med InsertionSort, skal det vises at invarianten er sand, før den første iteration af løkken, at den bliver vedligeholdt i hver iteration af løkken og at den giver en brugbar egenskab, som kan vise korrektheden af MergeSort, når løkken terminerer.

Initialisering: Inden første iteration af løkken er $k = p$ og derfor er dellisten [$p..k-1$] tom. Den tomme delliste indeholder de $k-p = 0$ mindste elementer af L og R. Både i og j er 1, så L[i] og R[j] må være de mindste tal i deres lister, som ikke er kopieret tilbage til A. De er de mindste tal da L og R var kopier af henholdsvis A[$p..q$] og A[$q+1..r$] som blev antaget at være sorterede. De første elementer i listerne må da være de laveste. Det er altså nu vist at løkkeinvarianten er sand før den første iteration af løkken.

Vedligeholdelse: Hvis det antages at $L[i] \leq R[j]$, så må L[i] være det mindste element der ikke er kopieret tilbage i A. Ifølge løkkeinvarianten indeholder A[$p..k-1$] de $k-p$ mindste elementer i L og R, så når L[i] på linje 14 sættes ind på A[k], må A[$p..k$] indeholde de $k-p+1$ mindste tal i L og R. Inden næste iteration vil k blive gjort én større, A[$p..k$] bliver da til A[$p..k-1$], som nu indeholder de $k-p$ mindste tal fra L og R.

Efter L[i] bliver sat ind i A[k], må L[$i+1$] være det mindste tal fra L som stadig ikke er kopieret tilbage til A (da L jo er en sorteret liste), derfor gøres i én større på linje 15, sådan at L[i] nu er det mindste tal fra L som stadig ikke er kopieret tilbage til A. Nu er løkkeinvarianten vedligeholdt såfremt L[i] var $\leq$ R[j].

Processen er det samme hvis R[j] er mindre end L[i]. Der køres linje 17 og 18 i stedet og R[j] bliver sat ind på A[k], da R[j] må være det mindste tal, som ikke er kopieret ind i A. På linje 18, gøres j én større så den nye R[j] er det mindste tal i R, som ikke er kopieret tilbage i A. Løkkeinvarianten er altså nu fuldt vedligeholdt.

Termination: For-løkken kører til og med r, så ved termination må $k = r+1$. Ifølge løkkeinvarianten får man da at A[$p..k-1$], som er det samme som A[$p..r$], indeholder de $k-p = r-p+1$ mindste elementer af L[$1..n1+1$] og R[$1..n2+1$] i en sorteret rækkefølge (Cormen, et al., 2009, p. 33). De to lister L og R, indeholdte i alt $r-p+3$ tal. Siden de $r-p+1$ mindste af dem blev kopieret tilbage i A i en sorteret rækkefølge, betyder det at de 2 største tal ikke blev kopieret ind i A. De 2 største tal var selvfølgelig de 2 uendeligt store tal. Alle de ikke uendeligt store tal er derfor blevet kopieret tilbage i A i en sorteret rækkefølge og derfor må den nye A[$p..r$] være en sorteret rækkefølge af den oprindelige A[$p..r$].

Merge-proceduren blev kaldt fra Merge-Sort-proceduren. Første gang Merge-proceduren bliver kaldt vil det blive for at kombinere to lister med længder på 1 (som per automatik er sorteret), til en sorteret liste med længden 2. Denne liste med længden 2, vil så blive kombineret med en anden liste til at lave en sorteret liste med længden 4 og så videre. Siden man starter fra bunden af og det først er listerne med længder på 1 som kombineres så sørger man for at antagelsen i Merge om at A[$p..q$] og A[$q+1..r$] er sorterede holder. Når Merge-Sort til sidst kalder Merge($A, 1, q, A.length$) kombineres de to halve lister med størrelser på n/2 til den samlede sorterede liste og sorteringsproblemet er da løst.

3.3 Analyse af køretider

Når man skal analysere algoritmer, kigger man oftest på deres køretider. Man kan benytte sig af forskellige modeller til at analysere køretid. I denne opgave vil der blive brugt RAM-modellen (Random Access Machine). RAM-modellen har en masse grundlæggende instruktioner, såsom addition, division, lagring og hentning af data, osv. Det vigtigste ved RAM-modellen er at hver af disse instruktioner tager en konstant tid at udføre (Cormen, et al., 2009, p. 23).

Tiden det tager at sortere med både InsertionSort og MergeSort afhænger af det input man giver dem. De er for eksempel begge meget langsommere om at sortere 1000 tal end de er om at sortere 10 tal. I InsertionSorts tilfælde kan det også tage forskellige tider at sortere et input på samme størrelse, afhængigt af hvor sorteret inputtet er i forvejen. Generelt kan man dog sige at tiden det tager for en algoritme at fuldende sit arbejde, vokser med størrelsen af det input man giver den. Derfor er det normalt at beskrive køretiden for en algoritme, som en funktion af inputtets størrelse (Cormen, et al., 2009, p. 24). For sorteringsalgoritmer svarer inputtets størrelse til det antal tal der skal sorteres. Man skal altså beskrive køretiden som en funktion af antallet af tal som skal sorteres. Man betegner normalt inputstørrelsen med bogstavet n.

Køretiden for en algoritme er det antal af primitive operationer eller trin, som algoritmen udfører. For at holde sig indenfor RAM-modellens rammer, antages det at hver linje i algoritmens kode tager et konstant antal trin at udføre, forskellige linjer kan godt tage forskellige konstante antal trin, men linje i vil altid tage c_i trin for at udføre (Cormen, et al., 2009, p. 25).

3.3.1 Analyse af InsertionSort

For at analysere InsertionSort, skal der som nævnt findes en funktion af n, der beskriver antallet af trin som InsertionSort skal udføre ved en inputstørrelse på n. For at bestemme dette skal man udregne hvor mange gange hver linje i pseudo-koden for InsertionSort bliver kørt.

Linje 1, starten af for-løkken, må køre n gange da $n-2+2=n$ (se definitionen af for-løkke i afsnit 3.1). Det må betyde at linje 2,3 og 7 køres $n-1$ gange. While-løkken køres et forskelligt antal gange afhængigt af hvad j er og hvorvidt inputtet er sorteret i forvejen. Hvis man lader t_j beskrive antallet af gange som linje 5 køres for en given værdi af j, må antallet af gange linje 4 køres i alt være summen af t_j for alle de mulige værdier af j, altså $\sum_{j=2}^{n} t_j$, while-løkkens krop må køre én gang mindre end dette for hver iteration af for-løkken. Linje 5 og 6, må derfor være samme sum, men hvor der trækkes 1 fra t_j, altså $\sum_{j=2}^{n}(t_j-1)$. Hvis man noterer prisen (antallet af trin for en primitiv operation) for linjen i som c_i kan man lave følgende tabel med priser for hver linje, samt hvor mange gange hver linje køres:

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      i = j - 1
4      while i > 0 and A[i] > key
5          A[i+1] = A[i]
6          i = i - 1
7      A[i+1] = key
```

Pris	Antal
c_1	n
c_2	$n-1$
c_3	$n-1$
c_4	$\sum_{j=2}^{n} t_j$
c_5	$\sum_{j=2}^{n}(t_j-1)$
c_6	$\sum_{j=2}^{n}(t_j-1)$
c_7	$n-1$

Det samlede antal primitive operationer en linje udføres må kunne findes ved at gange linjens pris, med antallet af gange linjen bliver kørt. For eksempel bliver det for linje 1 $c_1 \cdot n$. Det samlede antal operationer kan så findes ved at lægge antallet af operationerne for hver linje sammen. Hvis dette gøres for InsertionSort, får man følgende funktion $T(n)$ for køretiden:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Som det kan ses, så kan T(n) til en bestemt værdi af n, stadig give forskellige resultater alt afhængigt af værdien af t_j. I det bedste tilfælde vil listen i forvejen være sorteret og så vil t_j være 1 for alle de mulige værdier af j, da tallet til venstre for j altid vil være $\leq A[j]$. I så fald bliver linje 5 og 6 aldrig kørt og linje 4 køres kun 1 gang per iteration altså lige så mange gange som linje 2,3 og 7. I det bedste tilfælde bliver køretiden derfor:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \end{aligned}$$

I det bedste tilfælde fremkommer der altså en lineær funktion af formen $an + b$, hvor a og b er konstanter som afhænger af priserne på de forskellige linjer (Cormen, et al., 2009, p. 26).

I værste tilfælde vil listen være omvendt sorteret, altså vil de højeste tal være først. I så fald skal A[j] sammenlignes med alle elementerne til venstre for j. I dette tilfælde bliver $t_j = j$ for alle de mulige værdier af j, altså $j = 2,3, \dots, n$ (Cormen, et al., 2009, p. 27). Nu bliver summerne der skal udregnes altså

$2 + 3 + 4 + \cdots + n$ og $1 + 2 + 3 + \cdots + (n-1)$

Disse summer kan udregnes med følgende formler som kan udledes fra ligning (*2.1*):

$$\sum_{j=2}^{n} j = \frac{n \cdot (n+1)}{2} - 1$$

og

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

Hvis disse indsættes i T(n) og man reducerer udtrykket, får man følgende funktion for det værste tilfælde af køretiden i InsertionSort:

$$T(n) = \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right) n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right) n - (c_2 + c_3 + c_4 + c_7)$$

Det kan ses at der fremkommer et andengradspolynomium på formen $an^2 + bn + c$ hvor a, b, c er konstanter som er afhængige af priserne på de forskellige linjer.

I et gennemsnitligt tilfælde hvor tallene i inputtet er tilfældigt blandet, vil halvdelen af A[$1..j-1$] i gennemsnit være mindre end A[j] og den anden halvdel vil være større, så når dellisten A[$1..j-1$], skal tjekkes for at finde en plads til A[j] vil der i gennemsnit skulle tjekkes halvdelen af A[$1..j-1$], inden man finder en plads. I det gennemsnitlige tilfælde bliver t_j, derfor $\frac{j}{2}$, dette vil også genere et andengradspolynomium (Cormen, et al., 2009, p. 28).

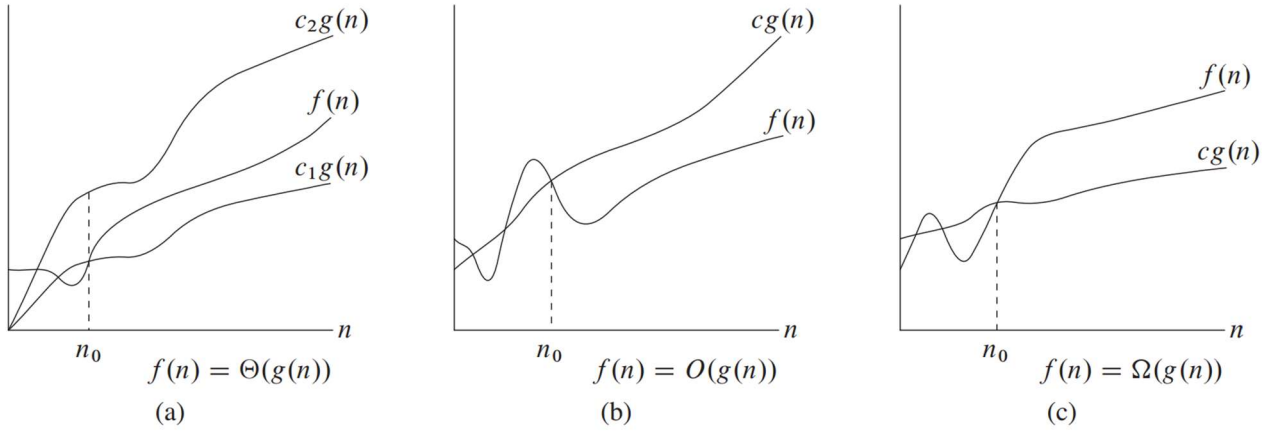
Ovenover blev udtrykkene for køretiderne simplificeret så man i stedet for priserne for hver linje, brugte konstanterne a, b, c. Køretiden kan dog simplificeres endnu mere, idet man kun er interesseret i at kigge på vækstordenen af funktionen. Vækstorden er en måde at beskrive hvordan en funktion vokser på når størrelsen på dens input bliver meget stort. Til at beskrive vækstordenen af en funktion kan man bruge det der kaldes asymptotisk notation. En særligt brugbar asymptotisk notation er Θ (store theta). For en given funktion g(n) er $\Theta(g(n))$ givet ved følgende sæt af funktioner:

$$\Theta\big(g(n)\big) = \{f(n): der\ eksisterer\ positive\ konstanter\ c_1, c_2\ og\ n_0, således\ at \\ 0 \leq\ c_1 g(n) \leq f(n) \leq c_2 g(n)\ for\ alle\ n \geq n_0\ \} \quad (3.1)$$

(Cormen, et al., 2009, p. 44)

Hvis en funktion $f(n)$ tilhører $\Theta(g(n))$ noteres dette som $f(n) \in \Theta(g(n))$ eller blot som $f(n) = \Theta(g(n))$

Hvis en funktion $f(n)$ = $\Theta(g(n))$ så kan det siges om $f(n)$ at den for store nok værdier af n, vil vokse lige så hurtigt som $g(n)$. Foreksempel hvis $f(n) = \Theta(n^2)$, så vil $f(n)$ for store nok værdier af n, vokse lige så hurtigt som et andengradspolynomium. Det kan ses på Figur 3.2 **(a)** at f(n) bliver bundet mellem to konstante faktorer af g(n).

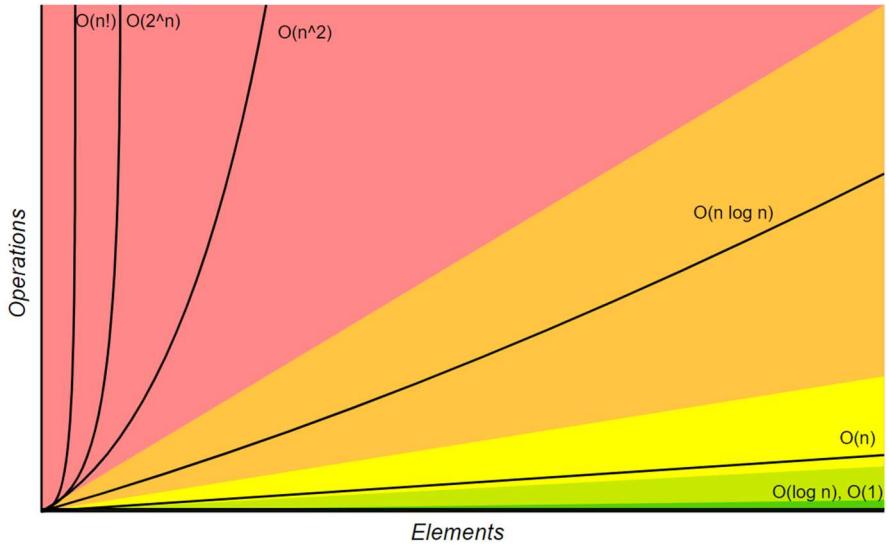


Figur 3.2 (Cormen, et al., 2009, p. 45) Illustrationer af Θ , O og Ω. ***(a)*** *Viser Θ-notation her ligger f(n) altid mellem eller på $c_1 g(n)$ og $c_2 g(n)$ når $n \geq\ n_0$.* ***(b)*** *Viser O-notation. Her ligger f(n) altid under eller på cg(n), når $n \geq\ n_0$.* ***(c)*** *Viser Ω-notation. Her ligger f(n) altid over eller på cg(n), når $n \geq\ n_0$*

Når man bruger Θ notation, kan man se bort fra konstante faktorer, da disse blot vil forskyde $c_1\ og\ c_2$ med en konstant faktor. Derfor er $3n^2$ stadig $\Theta(n^2)$.

Hvis en funktion består af flere led, behøver man kun at kigge på det led der vokser hurtigst. Når n er stor nok vil selv den mindste brøk af dette led nemlig dominere de andre led og det er derfor det eneste af leddene, som vil have en reel indflydelse på væksten (Cormen, et al., 2009, p. 46). Følgende række viser hvordan nogle typiske funktioner vokser i forhold til hinanden. Dem der står til venstre, vokser hurtigst (se også Figur 3.3 for illustration):

$n!\ , 2^n, n^2, n \cdot \log(n)\ , n, \log(n)\ , 1$ (Rowell, et al., u.d.)



Figur 3.3 (Rowell, et al., u.d.) Illustration af forskellige funktioners vækst. Henad x-aksen ses antallet af elementer og y-aksen viser antallet af operationer. De sorte grafer viser hvordan antallet af operationer udvikler sig for forskellige funktioner, når antallet af elementer bliver større.

Hvis man har en funktion $n \cdot \log(n) + n$, behøver man da kun at kigge på $n \cdot \log(n)$ og væksten bliver da $\Theta(n \cdot \log(n))$

Derfor kan det også siges, at for et hvilket som helst polynomium af grad d givet ved $p(n) = \sum_{i=0}^{d} a_i n^i$, hvor a_i er konstanter og $a_d > 0$ gælder det at $p(n) = \Theta(n^d)$ (Cormen, et al., 2009, p. 46). Med andre ord behøver man i et polynomium kun at kigge på leddet med den højeste grad. Enhver konstant er i virkeligheden et 0-gradspolynomium, så en konstant funktion kan noteres som $\Theta(n^0)$ eller blot $\Theta(1)$ (Cormen, et al., 2009, p. 46).

Θ notationen kan benyttes på køretiderne for InsertionSort. I InsertionSorts værste tilfælde og gennemsnitlige tilfælde blev køretiden et andengradspolynomium på formen $an^2 + bn + c$. Ifølge ovenstående regler kan man se bort fra konstanterne og leddene af lavere grad. Hvis man giver InsertionSort en omvendt sorteret liste eller blandet liste af tal bliver køretiden altså $\Theta(n^2)$. For store nok værdier af n vokser køretiden derfor med samme hastighed som et andengradspolynomium.

På samme måde kan køretiden for InsertionSort i bedste tilfælde også findes. Her kunne køre tiden skrives på formen $an + b$ og dette bliver da til $\Theta(n)$.

I de tre køretider beskrevet ovenover kigger man på enkelte tilfælde for InsertionSort, nemlig hvis inputtet allerede er sorteret eller hvis inputtet er omvendt sorteret eller hvis det er blandet. Generelt vil man dog gerne have udtryk for den generelle køretid, altså hvis man ikke kender til den måde tallene er placeret på når man starter sorteringen. Her kan man benytte to andre asymptotiske notationer O-notation (store O) og Ω-notation (store omega). For en given funktion g(n) er $\mathrm{O}(g(n))$ givet ved følgende sæt af funktioner:

$$O(g(n)) = \{f(n): der\ eksisterer\ positive\ konstanter\ c\ og\ n_0, således\ at\ 0 \leq f(n) \leq cg(n)\ for\ alle\ n \geq n_0\} \quad (3.2)$$

(Cormen, et al., 2009, p. 47)

Hvis en funktion $f(n) = O(g(n))$ kan det siges at for store nok værdier af n, vokser $f(n)$ enten lige så hurtigt som $g(n)$ eller langsommere end $g(n)$. En illustration af dette kan ses på Figur 3.2 **(b)**. O-notation

beskriver en øvre grænse for væksten af $f(n)$. $f(n)$ kan ikke vokse hurtigere end $cg(n)$ for store nok værdier af n.

Ω-notation kan opfattes som det omvendte af O-notation. For en given funktion g(n) er $\Omega(g(n))$ givet ved følgende sæt af funktioner:

$$\Omega\big(g(n)\big) = \{f(n): der\ eksisterer\ positive\ konstanter\ c\ og\ n_0, således\ at\ 0 \leq cg(n) \leq f(n) for\ alle\ n \geq n_0\} \quad (3.3)$$

(Cormen, et al., 2009, p. 48)

Hvis en funktion $f(n) = \Omega(g(n))$ kan det siges at for store nok værdier af n, vokser $f(n)$ enten lige så hurtigt som $g(n)$ eller hurtigere end $g(n)$. En illustration af dette kan ses på Figur 3.2 **(c)**. Ω-notation beskriver en nedre grænse for væksten af $f(n)$. $f(n)$ kan ikke vokse langsommere end $cg(n)$ for store nok værdier af n.

Ud fra definitionerne på de tre asymptotiske notationer kan man udlede følgende teorem:

$$For\ to\ vilkårlige\ funktioner\ f(n)\ og\ g(n), har\ vi\ at\ f(n) = \Theta\big(g(n)\big), hvis\ og\ kun\ hvis\ f(n) = O\big(g(n)\big)\ og\ f(n) = \Omega(g(n)) \quad (3.4)$$

(Cormen, et al., 2009, p. 48)

Køretiden for bedste tilfælde med InsertionSort var $\Theta(n)$. Som følge af (*3.4*) er køretiden i dette tilfælde altså også $O(n)$ og endnu mere vigtigt, så er køretiden også $\Omega(n)$. Ligeledes var køretiden ved værste tilfælde (en omvendt sorteret liste) $\Theta(n^2)$, så i dette tilfælde er køretiden også $\Omega(n^2)$ og $O(n^2)$. Da O og Ω kunne giver en henholdsvis øvre og nedre grænse på væksten af køretiden, kan disse dog også bruges til at sige noget om køretiden for et hvilket som helst input. Hvis køretiden i det værste tilfælde er $O(n^2)$, så må køretiden for et hvilket som helst tilfælde ligeledes også være $O(n^2)$, da køretiden ikke kan vokse hurtigere end den vokser i det værste tilfælde. Hvis køretiden for det bedste tilfælde var $\Omega(n)$, så må køretiden for et hvilket som helst tilfælde også være $\Omega(n)$, da køretiden ikke kan vokse langsommere end det bedste tilfælde. O-notation kan derfor bruges til at beskrive den værste mulige vækst for køretiden, til et hvilket som helst input og Ω-notation kan bruges til at beskrive den bedste mulige vækst for køretiden, til et hvilket som helst input.

3.3.2 Analyse af MergeSort

For at finde køretiden på MergeSort, skal man bruge en anden tilgang, da MergeSort er rekursiv. Der skal dog stadig findes en funktion T(n), som beskriver køretiden.

Hvis problemets størrelse er lille nok, i MergeSorts tilfælde hvis $n = 1$, så må løsningen tage konstant tid altså $\Theta(1)$. Når $n \geq 1$ må køretiden for hvert rekursionsniveau være summen af tiden det tager at dele problemet i to underproblemer, tiden det tager at løse de to underproblemer og tiden det tager at kombinerer de to løsninger til underproblemerne. Siden MergeSort deler problemet i to lige store underproblemer af størrelsen $n/2$, må tiden det tager at løse hvert underproblem være lig med $T(\frac{n}{2})$. Siden der er to underproblemer, må den samlede tid for at løse dem begge være $2 \cdot T(\frac{n}{2})$. Hvis man betegner tiden det tager at dele listen i to som en funktion D(n) og tiden det tager at kombinerer underproblemer som en funktion C(n), kan følgende rekursive ligning opstilles for køretiden af MergeSort:

$$T(n) = \begin{cases} \Theta(1) & hvis\ n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + D(n) + C(n) & hvis\ n > 1 \end{cases}$$

Her bruges $\Theta(1)$ i en ligning. Når dette er tilfældet, betyder det blot at man snakker om en eller anden funktion som er en del af sættet $\Theta(1)$, dvs en konstant funktion (Cormen, et al., 2009, p. 49).

Opdelingen af problemet sker på linje 2 i MergeSort, hvor midten af listen beregnes. Denne linje køres kun én gang for hvert rekursionsniveau og den tager derfor konstant tid. Derfor er $D(n) = \Theta(1)$. Kombineringen sker i Merge proceduren. Linje 1 til 3 og 8 til 11 i Merge kører alle sammen én gang det vil sige konstant tid. For løkken på linje 4 kører $n1$ gange og løkken på linje 6 kører $n2$ gange. For-løkken på linje 12 til 18 kører i alt n gange. Hver af løkkernes kroppe kører i konstant tid $\Theta(1)$ for hver iteration. Man får altså tre lineære funktioner $n1 \cdot \Theta(1), n2 \cdot \Theta(1)\ og\ n \cdot \Theta(1)$. Lægger man de tre lineære funktioner sammen er resultatet også en lineær funktion. Køretiden for kombineringen af underproblemer ved hvert rekursionsniveau tager altså lineær tid. Derfor $C(n) = \Theta(n)$

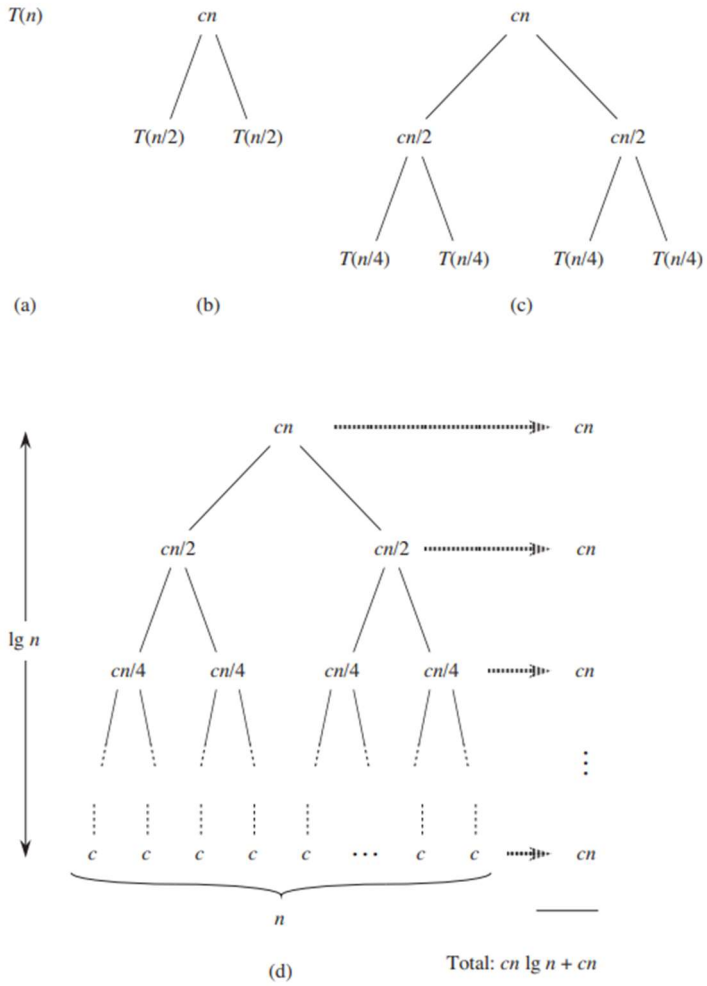
Da man kun behøver kigge på det hurtigst voksende led, skal man i $D(n) + C(n) = \Theta(1) + \Theta(n)$, blot ignorere det konstante led og den rekursive ligning bliver derfor nu:

$$T(n) = \begin{cases} \Theta(1) & hvis\ n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n) & hvis\ n > 1 \end{cases}$$

Hvis man lader konstanten c være den tid det tager at løse problemer på størrelsen 1 og den tid det tager per element at opdele og kombinere problemerne kan ligningen omskrives som følger:

$$T(n) = \begin{cases} \mathrm{c} & hvis\ n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + \mathrm{cn} & hvis\ n > 1 \end{cases}$$

Hvis man antager at n er en potens af 2 kan man opstille et rekursionstræ, som det der kan ses på Figur 3.4 nedenfor. Som det kan ses på figuren, tilføjes der cn til køretiden, for hvert rekursionsniveau. Den samlede køretid kan derfor findes ved at gange cn med antallet af rekursionsniveauer. Bemærk at der på andet rekursionsniveau er 2 underproblemer og dette antal fordobles (ganges med 2) for hvert efterfølgende rekursionsniveau indtil man til sidst har n underproblemer. Hvordan finder man ud af hvor mange gange 2 skal opløftes for at få n? Man bruger totals-logaritmen. $\log_2 n$ er netop løsningen til ligningen $2^x = n$. Antallet af rekursionsniveauer bliver da $\log_2 n + 1$, da det øverste rekursionsniveau, hvor der kun er et problem, også skal medregnes. Ganges dette med cn får man $cn \log_2 n + cn$. Man kan se bort fra konstanterne og det højre led, da $n\ \log_2 n$ vokser hurtigere end cn. Derfor bliver køretiden for MergeSort $\Theta(n \log_2 n)$. Man kan ændre grundtallet på enhver logaritme ved at gange med en konstant, så køretiden kan også skrives som $\Theta(n \log n)$ (Cormen, et al., 2009, p. 37).



Figur 3.4 (Cormen, et al., 2009, p. 38) Rekursionstræ

Køretiden for MergeSort vil være $\Theta(n \log n)$ i alle tilfælde, da der i modsætning til InsertionSort ikke er nogen værdi tilsvarende t_j, som afhang af hvordan inputtet var sorteret i forvejen. MergeSorts køretid er derfor også $O(n \log n)$ i alle tilfælde og $\Omega(n \log n)$ i alle tilfælde.

3.3.3 Sammenligning af InsertionSort og MergeSort

Resultatet af køretidsanalyserne for InsertionSort og MergeSort kan ses i Tabel 3.1 nedenfor

Algoritme	Bedste tilfælde	Gennemsnitligt	Værste tilfælde
InsertionSort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
MergeSort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$

Tabel 3.1 Køretiderne for InsertionSort og MergeSort

Som det blev nævnt tidligere, så vokser n^2 hurtigere end $n \log n$, som vokser hurtigere end n. Køretiden for InsertionSort vokser derfor i både det gennemsnitlige og det værste tilfælde hurtigere end MergeSort, når størrelsen på inputtet bliver stort nok. Kun når inputtet er sorteret i forvejen vokser MergeSort hurtigere for store nok værdier af n. Så hvis man gerne vil have sorteret tal, giver det bedst mening at anvende MergeSort ved større input, da køretiden for MergeSort vokser langsomst og tallene vil derfor blive sorteret hurtigere.

3.4 Test af køretider

For at undersøge køretiderne i praksis, implementeres de to algoritmer i JavaScript (se Bilag B for koden). Køretiden for de to algoritmer blev testet ved forskellige inputstørrelser. Der blev lavet 10 tests for hver inputstørrelse og den tid der fremgår af tabellerne nedenfor, er den gennemsnitlige testtid over de 10 tider. På Tabel 3.2 og Tabel 3.3 nedenfor kan ses køretiderne ved blandede og omvendt sorterede datasæt.

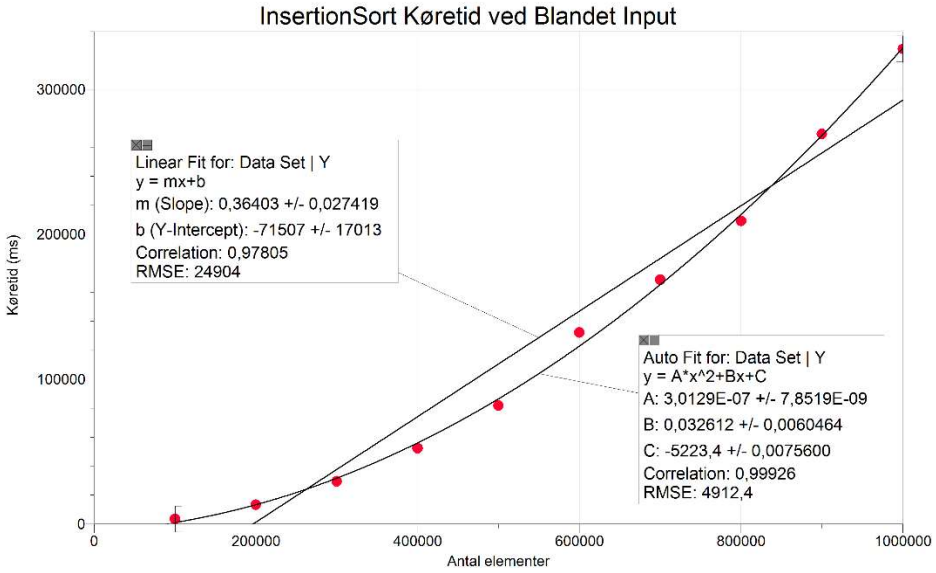
Gennemsnitskøretider ved blandede datasæt		
Antal elementer	Insertion Sort køretid (ms)	Merge Sort køretid (ms)
100000	3493	38
200000	13322	75
300000	29454	131
400000	52337	175
500000	81706	225
600000	132190	276
700000	168613	311
800000	209093	366
900000	269124	415
1000000	327756	459

Tabel 3.2 Køretider for InsertionSort og MergeSort ved blandede datasæt

Gennemsnitskøretider ved omvendt sorterede datasæt		
Antal elementer	Insertion Sort	Merge Sort
100000	5217	26
200000	21279	60
300000	49207	89
400000	86912	117
500000	136457	152
600000	195701	181
700000	267691	202
800000	349525	241
900000	447065	280
1000000	547521	314

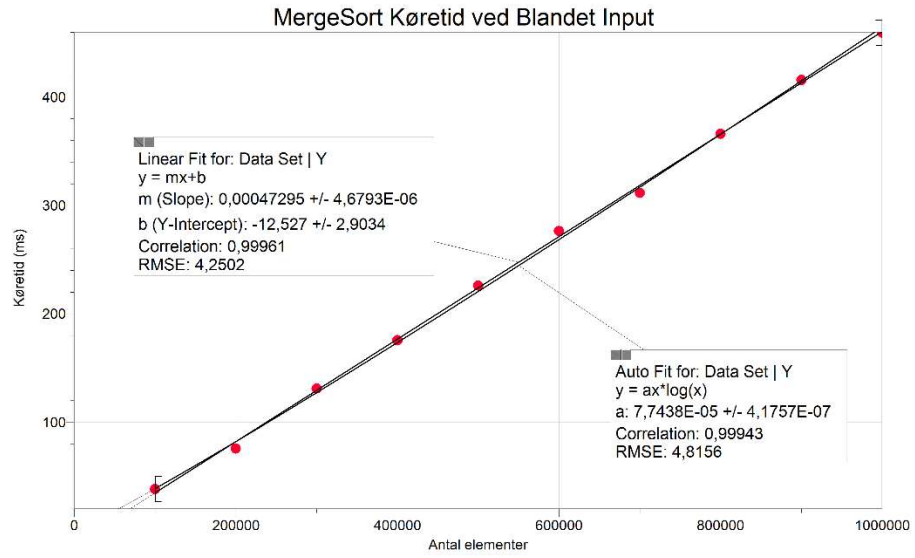
Tabel 3.3 Køretider for InsertionSort og MergeSort ved Omvendt sorterede datasæt.

Som det fremgår af Tabel 3.2 og Tabel 3.3, er InsertionSort meget langsommere end MergeSort, når tallene er blandede eller omvendt sorterede. InsertionSorts køretid bliver næsten 100 gange større ved 1.000.000 elementer i forhold til 100.000, hvorimod MergeSort kun bliver lidt over 10 gange større. Dette stemmer godt overens med de vækstordener der blev fundet i analysen. For at undersøge hvor godt køretiderne i tabellerne passer til de teoretiske køretider, anvendes regressionsanalyse med værktøjet Logger Pro. Resultatet af regressionsanalysen kan ses på Figur 3.5 og Figur 3.6 nedenfor.



Figur 3.5 Regressionsanalyse af InsertionSort ved blandet input. Ved omvendt sorteret input var det en meget lignende regression, så denne vil blive udeladt i opgaven. Den kan dog findes i Bilag C

Som det fremgår af Figur 3.5, har den kvadratiske regression (højre) en bedre korrelation (også kendt som R^2) end den lineære model (venstre). Dataene passer derfor bedre på den kvadratiske model. Dette stemmer overens med at InsertionSort havde en gennemsnitskøretid på $\Theta(n^2)$ og en værste køretid på $O(n^2)$



Figur 3.6 Regressionsanalyse af MergeSort ved blandet input. Ved omvendt sorteret input var det en meget lignende regression, så denne vil blive udeladt i opgaven. Den kan dog findes i Bilag C

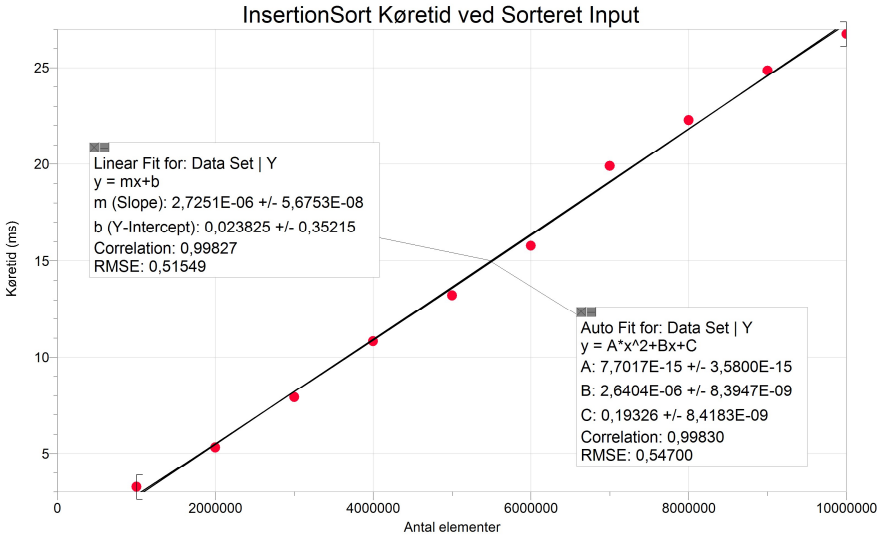
På Figur 3.6 passer den lineære model dog bedre på MergeSort end $n \log n$-modellen. Dette skyldes sandsynligvis at logaritmer vokser meget langsomt, så på så kort en rækkevidde i antallet af elementer har $\log n$ faktoren ikke en stor indflydelse i regressionsmodellen og det er derfor kun n som virkelig påvirker køretiden.

I Tabel 3.4 nedenfor kan køretiderne for allerede sorterede sæt ses

Gemmesnitskøretider ved allerede sorterede sæt		
Antal elementer	Insertion Sort køretid (ms)	Merge Sort køretid (ms)
1000000	3	355
2000000	5	748
3000000	8	1160
4000000	11	1553
5000000	13	1986
6000000	16	2435
7000000	19	2940
8000000	22	3274
9000000	24,84823	3690,77616
10000000	26,7513	4086,50411

Tabel 3.4 Køretider for InsertionSort og MergeSort ved allerede sorterede sæt. Læg mærke til at alle element antallene er 10 gange større her end i Tabel 3.2 og Tabel 3.3, da tiderne ellers blev for lave for InsertionSort til at være præcise.

Som det fremgår af Tabel 3.4 er InsertionSort meget hurtigere end MergeSort ved allerede sorterede sæt. Det kan ses at InsertionSort vokser meget mere lineært end før. Dette passer med at InsertionSort havde en køretid på $\Omega(n)$. Denne sammenhæng tjekkes nu med regressionsanalyse, se Figur 3.1.



Figur 3.7 Regressionsanalyse af InsertionSort ved allerede sorteret input.

På Figur 3.7, kan det ses at korrelationen på den lineære model er i meget mere tæt løb med den kvadratiske, i forhold til hvad den var ved blandet input. Hvis man kigger på a-koefficienten for den kvadratiske vækst, så er den meget lav, så i dette interval er den kvadratiske model i bund og grund også en lineær model, derfor kan man sige at køretiden $\Omega(n)$ passer godt til InsertionSort ved allerede sorteret input.

Generelt kan det på baggrund af dataene konkluderes at MergeSort er hurtigere til at sortere tal end InsertionSort når mængden af tal er stor nok, medmindre de er sorterede i forvejen. MergeSort er derfor den mest praktiske af de to algoritmer at bruge på store mængder tal, hvis man gerne vil have dem sorteret hurtigt.

4 Lighederne mellem induktionsbeviset og løkkeinvariant-metoden

Som det fremgik i afsnit 3, så mindede metoden man brugte til at vise korrektheden af InsertionSort og MergeSort meget om induktionsbevismetoden, som blev introduceret i afsnit 2. Lighederne mellem disse metoder vil blive diskuteret yderligere i dette afsnit.

For at vise korrektheden af sorteringsalgoritmerne blev der opstillet løkkeinvarianter, som var udtryk, man skulle vise var sande for hver iteration af en løkke i algoritmen. Løkkeinvarianten kan sammenlignes med det udtryk eller den formel man prøver at bevise i et induktionsbevis, det kunne for eksempel være udtrykket $1 + 2 + 3 + \cdots + n = \frac{n \cdot (n+1)}{2}$, da dette udtryk ligeledes skulle vises at være korrekt for alle positive heltal.

I initialisering-trinnet skulle man vise at invarianten var sand før den første iteration af løkken. Dette kan sammenlignes med induktionsbasen, hvor man skulle vise at udtrykket var sandt for en specifik værdi af n. Den specifikke værdi af n, kan ses som den første iteration af løkken.

I vedligeholdelses-trinnet antog man at invarianten var sand før en iteration og viste da, at den var sand inden næste iteration. Dette kan sammenlignes med induktionstrinnet, hvor man antog at udtrykket var sandt for en arbitrær værdi k (induktionsantagelsen) og man da viste at udtrykket så også var sandt for $k + 1$.

De to metoder adskiller sig dog på ét punkt. I induktionsbeviset fortsatte man ud i det uendelige og viste at udtrykket var sandt for alle positive heltal. Ved løkkeinvarianten stoppede man derimod når løkken stoppede og det var kun her løkkeinvarianten kunne bruges til at vise algoritmens korrekthed. Den store forskel på de to metoder er derfor at man i induktionsbeviset forsætter i det uendelige, hvorimod man med løkkeinvarianten stopper når løkken er kørt færdig.

Konklusion

I de indledende afsnit blev der redegjort for bevismetoderne direkte bevis, modstridsbevis og induktionsbevis. Den mest relevante af disse var induktionsbeviset. I induktionsbeviset viste man at et udtryk var sandt for et bestemt positivt heltal (induktionsbasen), man antog herefter at udtrykket var sandt for et arbitrært positivt heltal k (induktionsantagelsen) og viste så at $k + 1$ også blev sand (induktionstrinnet). Ud fra dette kunne man slutte at udtrykket var sandt for alle positive heltal.

Analysen af sorteringsalgoritmerne InsertionSort og MergeSort viste at MergeSort generelt var hurtigst til at sortere tal, når mængden af tal var stor nok. Algoritmerne blev også testet i praksis og resultaterne af dette kunne bekræfte analysen.

Til sidst blev induktionsbevismetoden sammenlignet med løkkeinvariant-metoden, som der blev brugt til at vise korrektheden af sorteringsalgoritmerne. Det blev konkluderet at løkkeinvarianten kunne sammenlignes med det udtryk man beviste i induktionsbeviset. Induktionsbasen kunne sammenlignes med initialisering-trinnet og induktionstrinnet kunne sammenlignes med vedligeholdelses-trinnet. Forskellen på de to metoder var at induktionsbeviset fortsatte ud i det uendelige hvorimod løkkeinvariant-metoden stoppede når løkken havde kørt færdig.

Bibliografi

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C., 2009. *Introduction to Algorithms.* 3. red. Cambridge, Massachusetts: Massachusetts Institute of Technology.

Rowell, E. et al., u.d. *Big-O Algorithm Complexity Cheat Sheet (Know Thy Complexities!) @ericdrowell.* [Online]
Available at: https://www.bigocheatsheet.com/
[Senest hentet eller vist den 28 11 2021].

Terstrup, J., 1974. *Logik 2 - Bevis.* 1. red. København: Open University / Gyldendal.

Wikipedia, 2021. *Direct Proof - Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/Direct_proof
[Senest hentet eller vist den 01 12 2021].

Wikipedia, 2021. *Golden ratio - Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/Golden_ratio
[Senest hentet eller vist den 27 11 2021].

Bilag

Bilag A - Induktionsbevis for Binet's formel

Et eksempel på et udtryk som kan bevises ved brug af matematisk induktion, er Binet's formel:

$$x_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right), n \in N_0 \quad (0.1)$$

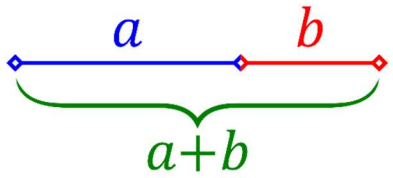
Binet's formel er en formel der kan bruges til at beregne det n'te tal i Fibonacci-talfølgen som er en talfølge der er givet ud fra følgende rekursive ligning:

$$x_n = x_{n-1} + x_{n-2}, n = 2,3,4 \ldots \quad (0.2)$$

Startbetingelserne til talfølgen er:

$$x_0 = 0 \; og \; x_1 = 1$$

Før Binet's formel bliver bevist er det relevant at kigge på værdierne $\frac{1+\sqrt{5}}{2}$ og $\frac{1-\sqrt{5}}{2}$ som indgår i Binet's formel. $\frac{1+\sqrt{5}}{2}$ kaldes det guddommelige forhold og kan også betegnes med det græske bogstav ϕ (phi). $\frac{1-\sqrt{5}}{2}$ betegnes som ϕ' (phi-mærke). Disse to værdier kan udledes fra definitionen af det gyldne snit. Det gyldne snit er en måde at opdele et linjestykke i to mindre linjestykker, således at forholdet mellem det største linjestykke og det mindste linjestykke er lig med forholdet mellem hele linjestykket og det største linjestykke. En illustration af det gyldne snit kan ses på Figur 0.1 nedenfor.



Figur 0.1 (Wikipedia, 2021) Illustration af linjestykkerne i det gyldne snit

Hvis man kalder det største linjestykke i det gyldne snit for a og det mindste linjestykke for b, kan forholdet mellem a og b udtrykkes med følgende ligning:

$$\frac{a}{b} = \frac{a+b}{a}$$

Ud fra ovenstående fås følgende:

$$\frac{a}{b} = 1 + \frac{b}{a}$$

Hvis man lader $x = \frac{a}{b}$ så bliver $x^{-1} = \frac{b}{a}$, indsættes disse fås følgende:

$$x = 1 + x^{-1}$$

Nu ganges der igennem med x og man får følgende udtryk, som skal bruges senere i induktionstrinnet:

$$x^2 = x + 1 \quad (0.3)$$

Hvis alting rykkes hen på venstresiden, får man følgende andengradsligning:

$$x^2 - x - 1 = 0$$

Herfra bestemmes diskriminanten:

$$d = (-1)^2 - 4 \cdot 1 \cdot (-1) = 5$$

Løsningerne til andengradsligningen er givet ved:

$$x = \frac{-b \pm \sqrt{d}}{2 \cdot a}$$

Løsningerne bliver altså:

$$\frac{1+\sqrt{5}}{2} \; og \; \frac{1-\sqrt{5}}{2}$$

Indutkionsbasen vil nu blive at vise at Binet's formel gælder for n=0 og n=1. For at vise at formlen gælder for begyndelsesbetingelsen $x_0 = 0$ indsættes 0 på n's plads i formlen:

$$x_0 = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^0 - \left(\frac{1-\sqrt{5}}{2}\right)^0\right) = \frac{1}{\sqrt{5}}(1-1) = 0$$

Formlen gælder altså når n er 0. For at vise at formlen gælder for begyndelsesbetingelsen $x_1 = 1$ indsættes 1 på n's plads i formlen.

$$x_1 = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^1 - \left(\frac{1-\sqrt{5}}{2}\right)^1\right) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}-1+\sqrt{5}}{2}\right) = \frac{1}{\sqrt{5}}\left(\frac{2\sqrt{5}}{2}\right) = \frac{2\sqrt{5}}{2\sqrt{5}} = 1$$

Formlen gælder derfor også når n er 1. Nu er induktionsbasen færdiggjort.

Som induktionsantagelse antages det nu at Binet's formel gælder for tallene k-1 og k-2. Induktionstrinnet vil da gå ud på at vise at Binet's formel gælder for tallet k, når den gælder for tallene k-1 og k-2.

Ifølge Binet's formel vil x_k være lig med:

$$x_k = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^k - \left(\frac{1-\sqrt{5}}{2}\right)^k\right) \tag{0.4}$$

Det skal derfor vises at man kan komme frem til ovenstående resultat når Binet's formel gælder for k-1 og k-2.

Hvis Binet's formel gælder for tallet k-1 vil x_{k-1} være lig med:

$$x_{k-1} = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{k-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{k-1}\right)$$

På samme måde vil x_{k-2} være lig med:

$$x_{k-2} = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{k-2} - \left(\frac{1-\sqrt{5}}{2}\right)^{k-2}\right)$$

Ifølge (*0.2*) er x_k givet ved:

$$x_k = x_{k-1} + x_{k-2}$$

Nu kan ovenstående værdier for x_{k-1} og x_{k-2} indsættes i formlen

$$x_k = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{k-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{k-1}\right) + \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{k-2} - \left(\frac{1-\sqrt{5}}{2}\right)^{k-2}\right) \quad (0.5)$$

Nu substitueres $\phi = \frac{1+\sqrt{5}}{2}$ og $\phi' = \frac{1-\sqrt{5}}{2}$ ind i (*0.5*):

$$x_k = \frac{1}{\sqrt{5}}\left(\phi^{k-1} - \phi'^{k-1}\right) + \frac{1}{\sqrt{5}}\left(\phi^{k-2} - \phi'^{k-2}\right)$$

Herfra kan $\frac{1}{\sqrt{5}}$ sættes udenfor en parentes

$$x_k = \frac{1}{\sqrt{5}}\left(\phi^{k-1} - \phi'^{k-1} + \phi^{k-2} - \phi'^{k-2}\right)$$

Nu kan der rykkes lidt rundt på leddene:

$$x_k = \frac{1}{\sqrt{5}}\left(\phi^{k-1} + \phi^{k-2} - \left(\phi'^{k-1} + \phi'^{k-2}\right)\right)$$

Herefter sættes ϕ^{k-2} og ϕ'^{k-2} udenfor parenteser.

$$x_k = \frac{1}{\sqrt{5}}\left(\phi^{k-2}\cdot(\phi+1) - \left(\phi'^{k-2}\cdot(\phi'+1)\right)\right) \quad (0.6)$$

Som det blev vist ovenover var både ϕ og ϕ' løsninger til ligningen (*0.3*) $x^2 = x + 1$. Derfor kan udtrykkene $\phi + 1$ og $\phi' + 1$ i (*0.6*) erstattes med henholdsvis ϕ^2 og ϕ'^2:

$$x_k = \frac{1}{\sqrt{5}}\left(\phi^{k-2}\cdot\phi^{\,2} - \left(\phi'^{k-2}\cdot\phi'^2\right)\right)$$

Til slut anvendes potensreglen $a^b \cdot a^c = a^{b+c}$:

$$x_k = \frac{1}{\sqrt{5}}\left(\phi^{k-2+2} - \phi'^{k-2+2}\right) = \frac{1}{\sqrt{5}}\left(\phi^k - \phi'^k\right)$$

Nu genindsættes $\frac{1+\sqrt{5}}{2}$ på ϕ's plads og $\frac{1-\sqrt{5}}{2}$ på $\phi's$ plads:

$$x_k = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{k} - \left(\frac{1-\sqrt{5}}{2}\right)^{k}\right)$$

Som det kan ses, fremkommer nu (0.4), som var det resultat x_k skulle være lig med ifølge Binet's formel. Det er dermed vist at hvis Binet's formel gælder for tallene k-1 og k-2, så gælder Binet's formel også for tallet k.

På baggrund af det der er blevet vist i induktionsbasen og induktionstrinnet kan det nu udledes at formlen gælder for alle værdier af $n, n \in N_0$. Hvis k-2 sættes til at være 0 og k-1 sættes til at være 1, så bliver k lig med 2. Ifølge det der blev vist i induktionstrinnet må det gælde at hvis Binet's formel gælder for tallene n=0 og n=1, så vil Binet's formel også gælde for n=2. I induktionsbasen blev det vist at Binet's formel lige netop gælder for tallene n=0 og n=1, derfor må Binet's formel også gælde for tallet n=2. Nu kan k-2 sættes til at være 1 og k-1 til at være 2, k bliver da 3. Da det nu er vist at Binet's formel gælder for både 1 og 2, så må den jævnfør induktionstrinnet også gælde for 3. Denne argumentation kan fortsættes ud i det uendelige og derfor er det nu bevist at Binet's formel gælder for alle tal $n, n \in N_0$

Bilag B - JavaScript kode

Alt koden til projektet (inklusiv kode til at visualisere de to algoritmer) kan tilgås på https://github.com/Hjijijing/SOPWorkspace

Nedenstående er en forklaring af de vigtigste kodeelementer, der blev brugt til at komme frem til tiderne i Afsnit 3.4. Nogle af kodeeksemplerne er en smule simplificerede i forhold til den rigtige kode, der blev brugt, for nemmere at forklare deres betydning.

Hvis man gerne vil benytte algoritmerne på sin computer, er det ikke nok at have pseudokoden. Man bliver nødt til at implementere algoritmerne i et reelt programmeringssprog. Der findes mange programmeringssprog, men i denne opgave vil der blive brugt programmeringssproget JavaScript. InsertionSort kan implementeres i JavaScript på følgende måde:

```
1   function insertionSort(A) {
2     for (let j = 1; j < A.length; j++) {
3       let key = A[j];
4       let i = j - 1;
5       while (i >= 0 && A[i] > key) {
6         A[i + 1] = A[i];
7         i -= 1;
8       }
9       A[i + 1] = key;
10    }
11  }
```

Som det kan ses, minder koden meget om den pseudokode, der før blev vist for InsertionSort. Der er dog nogle små forskelle. Først og fremmest bruger man ikke indrykning til at specificere kroppe i JavaScript, man bruger i stedet tuborg-klammer. Så "{" markerer starten af en krop og "}" markerer slutningen af en krop. Det kan herfra udledes at for-løkken i den ovenstående kode f.eks. har en krop fra linje 2 til 10.

Første gang man introducerer en variabel i JavaScript putter man også ordet **let** foran variabel navnet, som der for eksempel gøres på linje 3 og 4.

Syntaksen for en for-løkke er også lidt anderledes, som det kan ses på linje 2 laver man en parentes efter **for**. I parentesen laver man da først en variabel med en start værdi, her er det j som starter med at være 1. Herefter specificere man en betingelse, her $j < A.length$. Løkken vil tjekke denne betingelse før hver iteration og når den ikke længere er opfyldt stopper for-løkken med at køre. Til sidst beskriver man en ændring af variablen, her $j++$. Denne ændring bliver gjort på variablen efter hver iteration. $j++$ svarer til at gøre j én større. Det vil sige at j gøres én større efter hver iteration.

I while-løkken på linje 5 sætter man også betingelserne til løkken ind i en parentes. I stedet for at skrive **and** skriver man && (to "og"-symboler), det er den måde man skriver **and** i de fleste programmeringssprog.

I pseudokoden var det første element i listerne $A[1]$, i JavaScript er det derimod $A[0]$, derfor skal alle indekser gøres mindre i JavaScript koden. Derfor starter j med at være 1 i stedet for 2 og den kører indtil $j < A.length$ i stedet for $j \leq A.length$. Ligeledes kører while-løkken så længe $i \geq 0$ i stedet for $i > 0$.

Når der på linje 7 står $i{-}{=}\ 1$ betyder det at i gøres én mindre. Hvis man vil gøre en variabel én større kan man ligeledes skrive $i{+}{=}\ 1$.

På linje 1 laves der en funktion (kan opfattes som det samme som en procedure) kaldet insertionSort og den tager inputtet A, som er en liste.

Resten af koden er ligesom pseudokoden for InsertionSort.

Her er koden for mergeSort

```
1  function mergeSort(A, p, r) {
2    if (p < r) {
3      let q = Math.floor((p + r) / 2);
4      mergeSort(A, p, q);
5      mergeSort(A, q + 1, r);
6      merge(A, p, q, r);
7    }
8  }
```

Som det kan ses er det stort set det samme som pseudokoden for MergeSort. For at finde q bruges dog funktionen Math.floor(). Denne er en indbygget funktion i JavaScript, som blot beregner den nedre heltalsgrænse af hvad end der står indenfor dens parenteser. Resten af koden er det samme. Når man kalder mergeSort skal man dog skrive $mergeSort(A, 0, A.length - 1)$ i stedet for at bruge inputtet $(A, 1, A.length)$. Dette er, som nævnt før, fordi lister i JavaScript starter ved 0 og de slutter derfor også 1 plads før deres længde.

Nedenfor ses koden til Merge proceduren, denne er praktisk talt identisk med pseudokoden for merge.

```
1  function merge(A, p, q, r) {
2    let n1 = q - p + 1;
3    let n2 = r - q;
4    let L = [];
5    let R = [];
6    for (let i = 1; i <= n1; i++) {
7      L[i] = A[p + i - 1];
8    }
9    for (let j = 1; j <= n2; j++) {
10     R[j] = A[q + j];
11   }
12   L[n1 + 1] = Infinity;
13   R[n2 + 1] = Infinity;
14
15   let i = 1;
16   let j = 1;
17   for (let k = p; k <= r; k++) {
18     if (L[i] <= R[j]) {
19       A[k] = L[i];
20       i += 1;
21     } else {
22       A[k] = R[j];
23       j += 1;
24     }
25   }
26 }
```

På linje 4 og 5 i ovenstående kode laves listerne L og R. For at lave en liste i JavaScript skriver man blot en firkantet parentes []

For at tage tid på sorteringerne bruges en indbygget funktion i JavaScript kaldet performance.now(). Denne giver en præcis måling i millisekunder for hvor lang tid der er gået siden programmet startede. Så hvis man bruger denne før og efter man har kaldt en af sorteringsalgoritmerne, kan man finde forskellen mellem efter og før og så har man tiden i millisekunder, som sorteringsalgoritmen har brugt. Det kan se således ud:

```
1 let tidFør = performance.now();
2 insertionSort(A);
3 let tidEfter = performance.now();
4 print(tidEfter - tidFør);
```

Her gemmes tiden før og efter insertionSort bliver kaldt i variablerne tidFør og tidEfter. Til sidst udskrives forskellen mellem de to på skærmen, ved brug af den indbyggede funktion print().

For at lave et sorteret datasæt bruges følgende funktion:

```
1 function lavSorteretDataSæt(n) {
2   let resultat = [];
3   for (let i = 0; i < n; i++) {
4     resultat[i] = i;
5   }
6   return resultat;
7 }
```

På linje 2 laves en liste kaldet resultat. Herefter laves en for-løkke på linje 3, som itererer igennem tallene $i = 0, 1, 2, \dots, n-1$. I hver iteration sættes det i'te element i listen til at være lig med i. Resultaten bliver derfor en liste med tallene $0,1,2, \dots, n-1$.

På linje 6 returnes resultat. Dette betyder at man sender resultat listen tilbage til hvorend funktionen lavSorteretDataSæt. Så hvis man skriver følgende kode

```
1 let A = lavSorteretDataSæt(100);
```

Så vil en sorteret liste med tallene 0 til 99 bliver sendt tilbage fra lavSorteretDataSæt og gemt i variablen A.

Koden for at lave et omvendt sorteret datasæt er nærmest den samme men i stedet for at sætte resultat[i] til at være i sætter man det blot til at være $n-1-i$. Dette vil give en liste med tallene $n-1, n-2, \dots, 1, 0$

Til slut skal der laves en funktion til at lave et blande tallene. Til det bruges følgende kode:

```
1 function blandTal(A) {
2   for (let i = A.length - 1; i > 0; i--) {
3     let j = Math.floor(Math.random() * (i + 1));
4     [A[i], A[j]] = [A[j], A[i]];
5   }
6 }
```

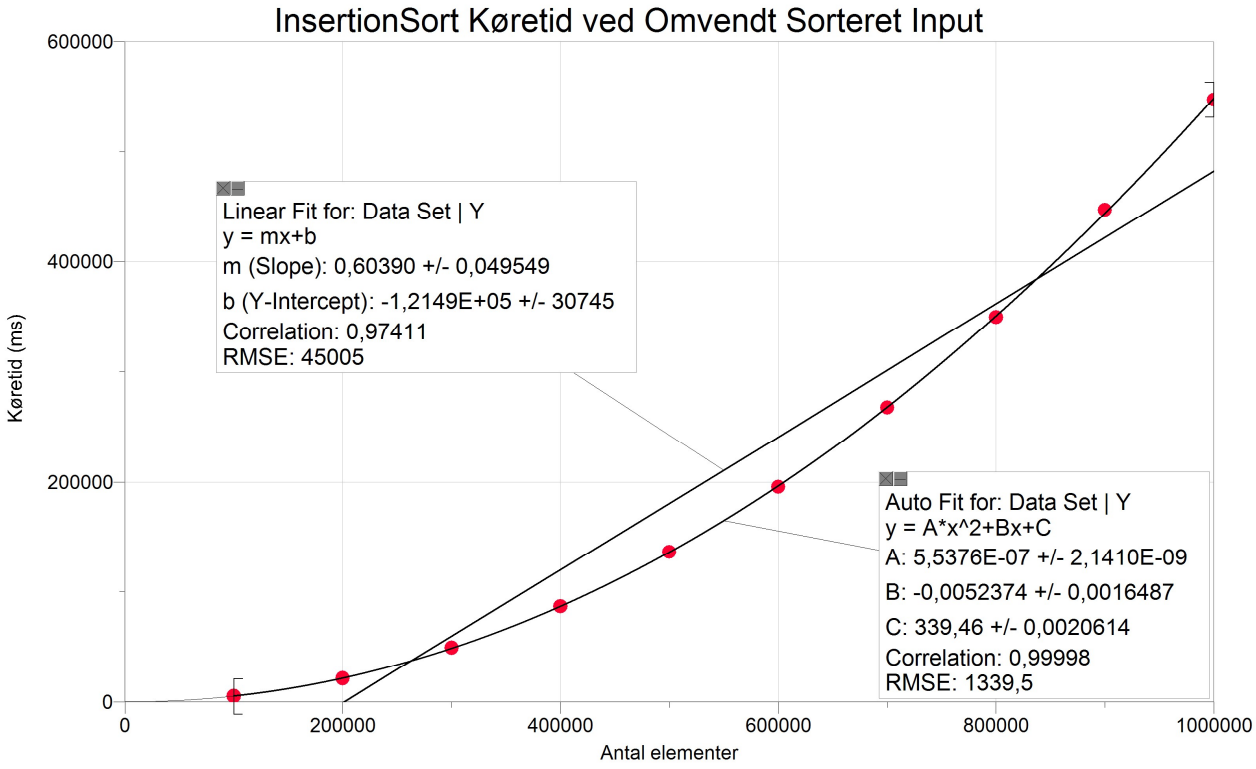
Denne funktion tager en liste A, som input og blander den. På linje 2 laves en for-løkken hvor i starter med at være $A.length - 1$ og så gøres den mindre efter hver iteration ($i--$), indtil $i > 0$ ikke længere er sand. Den kører altså igennem alle indekser i A (udover 0) men i en omvendt rækkefølge. I løkken laver man en værdi j, som sættes til at være den nedre heltalsgrænse af Math.random()*i(+1). Math.random() laver et tilfældigt decimaltal mellem 0 og 1 (1 er ikke inkluderet). Dette ganges så med i+1, det vil sige at man får et tal mellem 0 og i+1 (ikke inkluderet). Så når det rundes ned med den nedre heltalsgrænse, bliver j et tilfældigt tal mellem 0 og i (i inkluderet). Linje 4 ser lige farlig ud, men koden betyder blot at man bytter pladserne for A[i] og A[j] i listen med hinanden, sådan at A[i] kommer på A[j]'s plads og omvendt. Siden j er tilfældig i hver iteration, vil A være blandet efter funktionen er kørt. Så hvis man gerne vil lave et blandet datasæt kan man først lave et datasæt med en af de to datasæt funktioner ovenfor og så herefter putte datasættet ind i blandTal. F.eks:

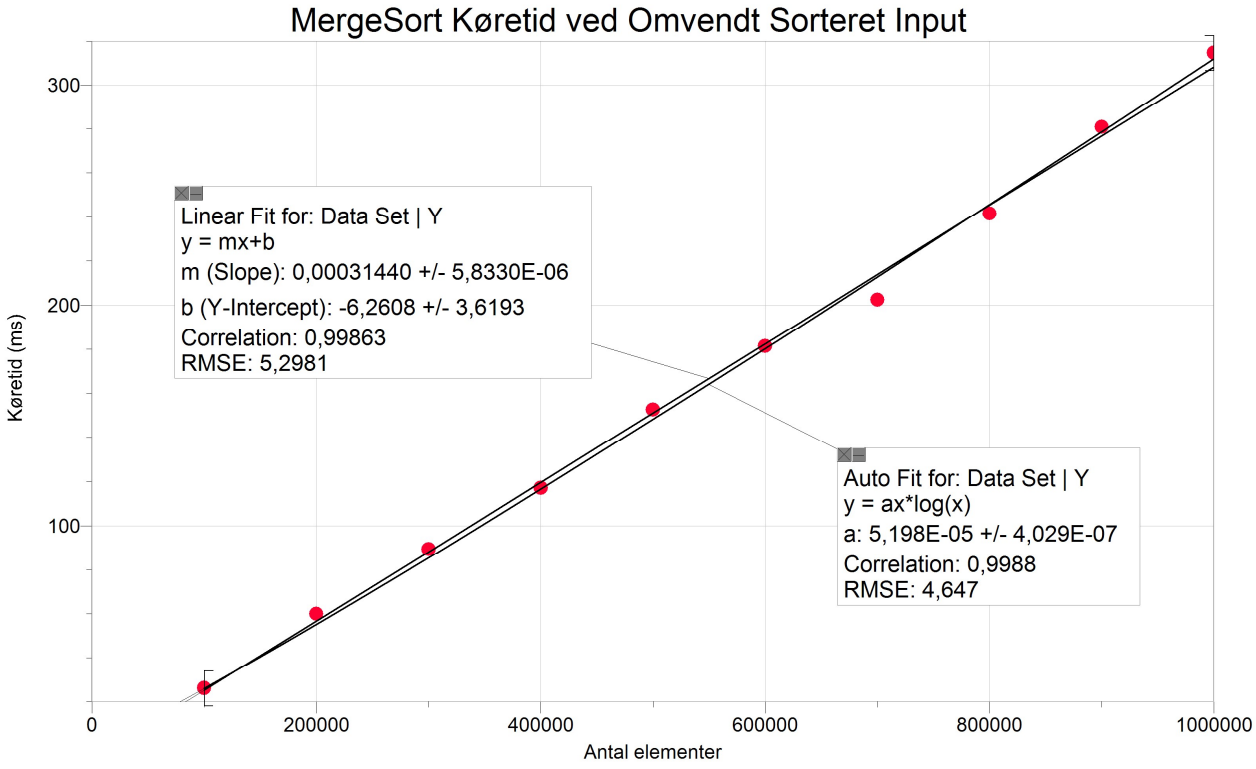
```
1 let A = lavSorteretDataSæt(100);
2 blandTal(A);
```

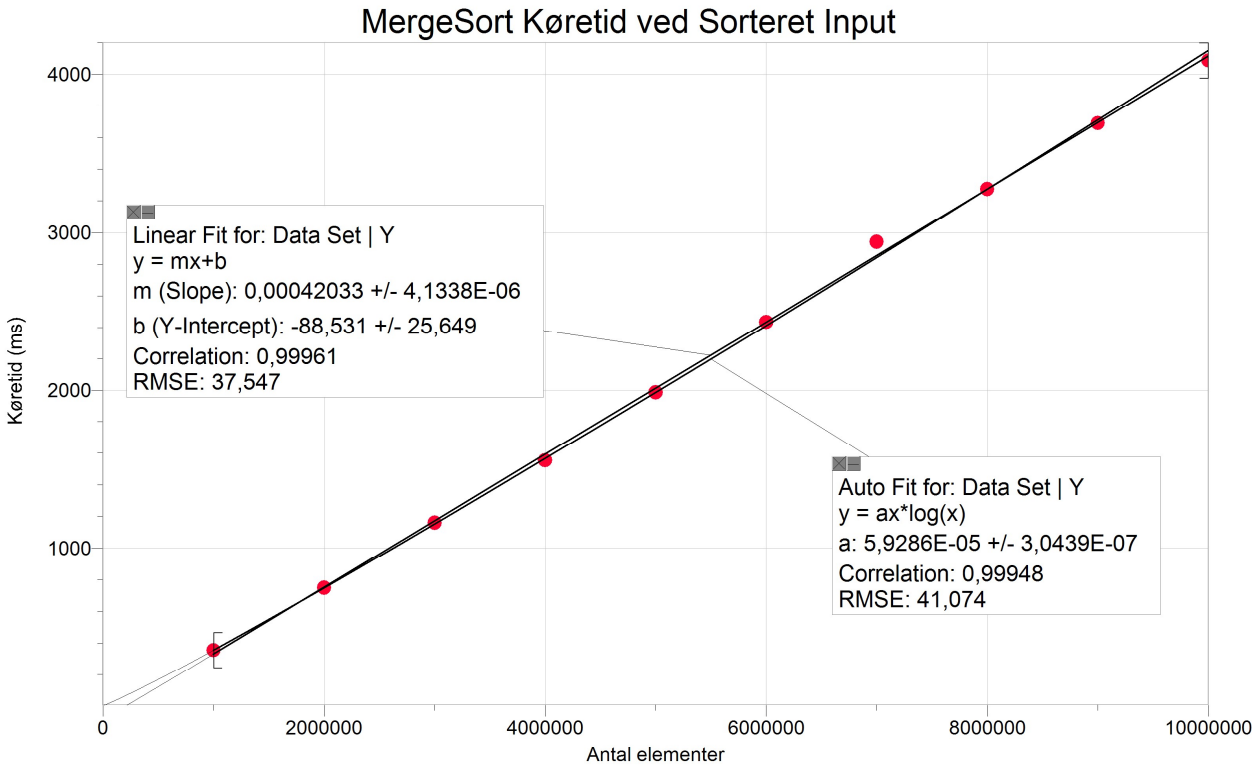
Efter denne kode er kørt vil A være en blandet rækkefølge af tallene $0,1,2, \dots, 99$

Det var alt koden der var nødvendig at forstå for at se hvordan man kunne finde ud af tiderne det tager de to algoritmer at sortere data. De ovenstående funktioner til at lave og ændre på datasæt kan bruges til at lave et enten blandet, sorteret eller omvendt sorteret datasæt, som man kan give til mergeSort eller insertionSort og tage tid på hvor hurtigt datasættet bliver sorteret.

Bilag C - Andre regressioner






MergeSort Køretid ved Sorteret Input
Køretid (ms)
4000
3000
2000
1000
0
2000000
4000000
6000000
8000000
10000000
Antal elementer
Linear Fit for: Data Set | Y
y = mx+b
m (Slope): 0,00042033 +/- 4,1338E-06
b (Y-Intercept): -88,531 +/- 25,649
Correlation: 0,99961
RMSE: 37,547
Auto Fit for: Data Set | Y
y = ax*log(x)
a: 5,9286E-05 +/- 3,0439E-07
Correlation: 0,99948
RMSE: 41,074