

# Min forside

# Resume

Indhold

[Min forside 2](#_Toc89278164)

[Resume 3](#_Toc89278165)

[Indledning 5](#_Toc89278166)

[1 Giv en kort introduktion til forskellige bevismetoder i matematik 6](#_Toc89278167)

[1.1 Direkte bevis 6](#_Toc89278168)

[1.2 Modstridsbevis 6](#_Toc89278169)

[1.3 Geometrisk bevis 6](#_Toc89278170)

[2 Redegør i detaljer for induktionsbevismetoden herunder induktionsbasen, induktionsantagelsen og induktionstrinnet. Medtag gerne et konkret bevis eksempel (du kan f.eks. eftervise Binets formel) 7](#_Toc89278171)

[2.1 Bevis for Binet’s formel 7](#_Toc89278172)

[3 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 8](#_Toc89278173)

[3.1 Introduktion til sorteringsproblemet 8](#_Toc89278174)

[3.2 InsertionSort 8](#_Toc89278175)

[3.2.1 Korrekthed af InsertionSort 10](#_Toc89278176)

[3.3 MergeSort 11](#_Toc89278177)

[3.3.1 Korrekthed af MergeSort 14](#_Toc89278178)

[3.4 Analyse af køretider 15](#_Toc89278179)

[3.4.1 Analyse af InsertionSort 16](#_Toc89278180)

[3.4.2 Analyse af MergeSort 20](#_Toc89278181)

[3.4.3 Sammenligning af InsertionSort og MergeSort 21](#_Toc89278182)

[3.5 Implementering i JavaScript 22](#_Toc89278183)

[4 Diskutér kort lighederne mellem dit bevis ved matematik induktion og metoden, du benyttede til at vise korrektheden af InsertionSort og MergeSort. 24](#_Toc89278184)

[Konklusion 24](#_Toc89278185)

[Bibliografi 25](#_Toc89278186)

[Bilag 26](#_Toc89278187)

[4.1 Induktionsbevis for Binet’s formel 26](#_Toc89278188)

# Indledning

Til at starte med vil der i afsnit 1 blive givet en introduktion til forskellige bevismetoder i matematik.

Bagefter vil der i afsnit 2 blive redegjort for induktionsbevismetoden. Begreberne induktionsbasen, induktionsantagelsen og induktionstrinnet vil blive forklaret. Herefter vil det blive vist hvordan induktionsbevismetoden kan bruges til at bevise Binet’s formel.

Efter dette vil der i afsnit 3 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 algoritmernes køretider blive analyseret. Til sidst i afsnittet vil algoritmerne blive implementeret i programmeringssproget JavaScript og deres køretider vil blive testet, for at sammenligne de to algoritmer.

Til slut vil der i afsnit 4 blive diskuteret lighederne mellem induktionsbevismetoden og metoden, der blev benyttet til at vise korrektheden af InsertionSort og MergeSort.

# Giv en kort introduktion til forskellige bevismetoder i matematik

## Direkte bevis

Et direkte bevis er et bevis hvor man beviser sandhedsværdien af et udtryk ud fra allerede etableret viden, såsom aksiomer eller teoremer, 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). Hvis man for eksempel er givet A og B, så viser man at C gælder

Direkte bevis kan bruges til at bevise at summen af to vilkårlige lige heltal er lig med et lige heltal.

Givet to lige heltal x og y, kan disse, da de er lige, skrives som

og , hvor a og b er heltal.

Summen bliver da

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).

## Modstridsbevis

Modstridsbeviset (eller bevis ved modsigelse) er en måde at bevis sandhedsværdien af et udtryk på ved at antage at det modsatte er sandt og herefter vise at dette fører til en modstrid (Terstrup, 1974, p. 71).

Det kan f.eks. bevises at er et irrationelt tal ved at antage at er rationelt tal. Hvis er rationelt kan det skrives som en uforkortelig brøk 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 så er , det vil sige at er 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 er lige, må kunne skrives på formen hvor c er et heltal. Hvis der reduceres, får man .

er derfor lige og så må b også være lige. Det er nu vist at b er både lig og ulige. Dette er en modstrid og må betyde at antagelsen om at er rationel er forkert. Derfor må være irrationel.

## Geometrisk bevis

Peter vil komme med formel der ikke gælder uendeligt

Strukturel induktion

Fuldstændig induktion. Stærk induktion

Det jeg laver i Binets formle er svag induktion

Direkte bevis, modstridsbevis sqrt2 er irrationel, eksistensbevis, geometriske beviser (anal. plan), entydighedsbevis

Sprogligt overbeviser mange. Godt mat bevis overbeviser alle

# Induktionsbevismetoden

Induktionsbevismetoden er en metode der kan bruges til at bevise at et udtryk er sandt for alle eller for alle (Terstrup, 1974, p. 60).

Induktionsbevismetoden kan også anvendes hvis n tilhører en anden talmængde end 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 og .

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.

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.

Ud fra dette kan man slutte at udtrykket gælder for alle , da det i induktionsbasen blev vist at udtrykket var sandt når n=1. Hvis man vælger 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.

## Bevis for Binet’s formel ÆNDRE TITEL HER

INDSÆT ET BEVIS HER

# Sorteringsproblemet

## Introduktion til 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 altså mange forskellige situationer hvor man gerne vil have ændret rækkefølgen på en mængde af tal, således at de er i en opstigende rækkefølge, altså at det laveste tal står først, næstlaveste står næst og så videre. Dette problem kaldes sorteringsproblemet og formelt er det defineret som følger:

**Input:** En rækkefølge bestående af n tal

**Output:** En permutation af denne rækkefølge , således at (Cormen, et al., 2009, p. 5)

Hvis inputtet for eksempel er så vil outputtet blive . Man kalder en inputsekvens som denne for en instans af sorteringsproblemet. En instans af et problem består generelt af det input der skal bruges til at løse problemet (Cormen, et al., 2009, p. 5).

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 altså 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 korrekt output. Man kan sige at en korrekt algoritme løser det givne problem (Cormen, et al., 2009, p. 6). Algoritmer der løses sorteringsproblemet kaldes for sorteringsalgoritmer. I dette afsnit vil der blive kigget 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 til sidst vil de blive implementeret i JavaScript og testet.

## InsertionSort

InsertionSort er en algoritme der kan anvendes til at løse 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 allerede 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 så tilbage med en sorteret hånd.

For at vise hvordan InsertionSort fungerer, kan man skrive den op i pseudokode. Pseudokode kan opfattes som en form for programmeringssprog, hvor man beskriver tingene så der er så letforståelige som muligt. Pseudokoden for InsertionSort ser således ud:



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 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, kan man også kalde 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 min\_variabel = 3, så tildeler man værdien 3 til min\_variabel.

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 til at være b. Herefter tjekker der om . Hvis dette er sandt kører den alt koden i løkkens krop én gang hvor a = b. Løkkens krop er alt under **for**-løkken, som er indrykket i forhold til for-løkken. Når løkkens krop køres, kaldes det en iteration. Herefter gøres a én større. Nu tjekkes det igen om , og i så fald udføres endnu en iteration, hvor a denne gang er én større end forrige iteration. Sådan fortsættes det indtil 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 og én gang når og løkkens krop køres c-b+1 gange, da denne ikke køres når .

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, hvorefter koden går videre til den næste linje under **while** løkken som ikke er med i løkkens krop.

Nu når de basale elementer i pseudokode er på plads, vil pseudokoden for InsertionSort bliver forklaret. På linje 1 laves en **for**-løkke hvor j starter med at være 2 og vil køre så længe , Hvor 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, som er 1 mindre end j. 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 og . I løkkens krop, på linje 5, sættes det tal i listen som er til højre for det i’te tal, altså tal nummer i+1 til at være lig med 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 er 0 eller bliver A[i+1] sat til key på linje 7. Hvis , blev der ikke fundet en værdi for i, hvor og derfor bliver key indsat på A[1], den første plads i listen. Hvis **while**-løkken stoppede fordi så indsættes key én plads til højre for A[i]. Dette giver mening da key, da må være og derfor skal key i en sorteret rækkefølge være placeret til højre for A[i]. Så er iterationen færdig og en ny kan begynde hvor j er én større end før.

### 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 i 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 at InsertionSort er korrekt.

**Initialisering:** Først vises det at løkkeinvarianten gælder før den første iteration af for-løkken. Før første iteration 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 så 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], da A[1..j] må bestå af elementerne i A[1..j-1] som enten er rykket eller ej og key som var A[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 mindre end eller lig med 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 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:** Til sidst skal det i termination-trinnet vises at løkkeinvarianten kan bruges til at bevise algoritmens korrekthed når for-løkken afsluttes. Betingelsen der gjorde at for-løkken afsluttes er at  
. 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.

## MergeSort

En anden algoritme, der kan bruges til at løse sorteringsproblemet, er MergeSort. InsertionSort løste sorteringsproblemet ved at sortere ét tal ad gangen, ind i den allerede sorterede del af listen. MergeSort løser derimod problemet rekursivt og gør brug af Del-og-hersk metoden (Divide-and-conquer på engelsk). 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 problemet. 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 nem og åbenlyse 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).

For at opsummere så bliver MergeSort 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.

Et billede, der indeholder tekst, tastatur

Automatisk genereret beskrivelse

Figur . 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 vigtigt del af MergeSort er kombinationen af de halverede lister, til en samlet sorteret liste. 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 som stadig var i toppen af højre bunke. Igen tages det laveste af disse 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 . 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). Hvis man skal sammenligne dette med kortene, så kan A[p..q] opfattes som den ene bunke og A[q+1..r] som den anden bunke og den sorterede delliste A[p..r], som den resulterende bunke som man puttede alle kortene heni. Pseudokoden for Merge proceduren er som følger:



På linje 1 beregnes længden af dellisten A[p..q] og gemmes som n1. Ligeledes beregnes der på linje 2, længden af dellisten A[q+1..r]. 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 elementer fra henholdsvis A[p..q] og A[q+1..r]. Det er L og R, der senere skal kombineres sammen til at 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 hver liste 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. Hvis man kigger tilbage til analogien om kortbunkerne vil i og j altså fortælle hvor langt man er nået i hver kortbunke.

På linje 12 laves der en for-løkke hvor variablen k starter med at være p og løkken kører så længe, det vil altså sige at løkken vil gennemløbe tallene fra p til og med r.

For hver iteration af for-løkken bliver der på linje 13 tjekket om L[i] er mindre end eller lig med R[j], altså om det tal man er nået til i L er mindre end eller lig med det tal man er nået til i R. Hvis dette er sandt, så sættes A[k] til at være L[i] på linje 14 og i gøres én større på linje 15, det vil sige at næste gang der bliver sammenlignet på linje 13, vil det være det næste tal i L der bliver tjekket.

Hvis R[j] derimod er mindre end L[i] bliver linje 17 og 18 kørt i stedet. Her bliver A[k] i stedet sat til at være R[j] og j bliver gjort én større, det vil sige at til næste sammenligning på linje 13 vil det være det næste til i R der bliver tjekket.

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  
 tal, der er altså 2 tal tilbage og det er de to uendelig.

Rent intuitivt kan man måske nu se at man vil opnå en sorteret kombinering af L og R i A[p..r] når k gennemløber tallene fra p til og med r og det ved hver iteration er det mindste af de to tal man er nået til i L og R, der bliver sat ind på A[k].

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 ser ud som følger:



Hvis kan A[p..r] maksimalt have 1 element og er derfor allerede sorteret. Derfor køres linje 2 til 5 kun hvis p er mindre end 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, man kan f.eks. ikke sige A . Denne midterste indeks gemmes i variablen q. På linje 3 og 4 kaldes Merge-Sort rekursivt. Problemet deles altså i 2 og man får MergeSort 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 kombinerede jo de to dellister A[p..q] og A[q+1..r] til en sorteret liste og erstattede den nuværende A[p..r], så når linje 5 er kørt er A[p..r] nu sorteret.

For at sortere en hel liste skal man da kalde Merge-Sort(A,1,A.length), hvor A.length jo var længden på listen, altså n.

### Korrekthed af MergeSort

For at vise korrektheden af MergeSort opstilles der 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 der kan bruges til at vise korrektheden af MergeSort 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 nu vises at denne løkkeinvariant er sand, før den første iteration af løkken, at løkkeinvarianten bliver vedligeholdt i hver iteration af løkken og at løkkeinvarianten giver en brugbar egenskab, som kan vise korrektheden af MergeSort, når løkken terminerer.

**Initialisering:** Inden første iteration af løkken er k lig med p og derfor er delliste [p..k-1] tom. Den tomme delliste indeholder de k-p = 0 mindste elementer af L og R. Siden både i og j er 1, måde L[i] og R[j] være de mindste tal i deres lister, som ikke er kopieret tilbage til A. Det vides at 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] er mindre end eller lig med R[j], så må L[i] være det mindste element der endnu 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, sådan at A[p..k] nu bliver 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 gør man på linje 15 i én større, 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 mindre end eller lig med R[j]. Processen er det samme hvis R[j] er mindre end L[i], der er det linje 17 og 18 der køres 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 så én større så den nye R[j] er det mindste tal i R, som ikke kopieret tilbage i A. Løkkeinvarianten er altså nu fuldt vedligeholdt.

**Termination:**  For-løkken kører til og med r, så ved termination må k være 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, har i alt indeholdt  
 tal. Siden de r-p+1 mindste af dem blev kopieret tilbage i A i en sorteret rækkefølge, må det betyde at de 2 største tal ikke blev kopieret ind i A. De 2 største tal var selvfølgelig de 2 uendeligt store tal. Det vil sige at alle de ikke uendeligt store tal er 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 selvfølgeligt kaldt fra MergeSort-proceduceren, som rekursivt kaldte sig selv indtil input listen var delt i lister med længder på 1. Første gang Merge-proceduren bliver kaldt vil det da blive for at kombinere to lister med længder på 1 (som jo 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 bliver kombineret så sørger man for at antagelsen 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.

## Analyse af køretider

Når man skal analysere algoritmer, kigger man på hvor mange ressourcer de kommer til at bruge. Man kan f.eks. analysere hvor meget hukommelse eller båndbredde en algoritme bruger. Den mest normale ressourcer at analysere er dog køretiden. 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 1 million 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). Definitionen af inputtets størrelse kan variere alt afhængigt at hvilket problem ens algoritme løser, men når det er sorteringsalgoritmer man analyserer, bruger man normalt antallet af tal, der skal sorteres, som inputtets størrelse. Man skal altså beskrive køretiden som en funktion af antallet af tal som skal sorteres. Man betegner normalt antallet af tal 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 trin for at udføre.

### Analyse af InsertionSort

Nu når det grundlæggende er på plads er det tid til at analysere køretiden af InsertionSort. For at analysere InsertionSort, skal der som nævnt findes en funktion af n, der beskriver antallet af trin som InsertionSort skal udfører ved en inputstørrelse på n. For at bestemme dette skal man regne ud hvor mange gange hver linje i pseudo-koden for InsertionSort bliver kørt.

Linje 1, starten af for-løkken, må køre n gange. Én gang for hver j=2, j=3….. j=n+1. Generelt gælder det at linjen hvor man definerer en løkke kører én gang mere end linjerne i løkkens krop, da linjen hvor man definerer løkken tjekker løkkens betingelse (i dette tilfælde ) før hver iteration, det vil sige den køres for alle gange hvor denne betingelse er sand, men også den ene gang hvor betingelsen så bliver falsk, hvorimod løkkens krop kun bliver kørt for alle de gange hvor betingelsen var sand. Linje 1 bliver altså kørt n gange og 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 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 for alle de mulige værdier af j, altså , som det blev nævnt før så kører løkkernes krop én gang mindre, så linje 5 og 6, må derfor være samme sum, men hvor der trækkes 1 fra , altså . Hvis man som det blev beskrevet før, noterer prisen (antallet af trin for en primitiv operation) for linje i som kan man lave følgende tabel med priser for hver linje, samt hvor mange gange hver linje køres:



Siden prisen betegner antallet af primitive operationer det tager at køre hver linje og antallet beskriver antallet af gange hver linje køres, kan man finde det samlede antal primitive operationer en linje udfører, ved at gange linjens pris med antallet af gange linjen køres. For eksempel bliver det for linje 1 . 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 for køretiden:

Som det kan ses, så kan T(n) til den en bestemt værdi af n, stadig give forskellige resultater alt afhængigt af værdien af , altså hvor mange gange while-løkken køres for hver værdi af j. I det bedste tilfælde vil listen i forvejen være sorteret og så vil være 1 for alle de mulige værdier af j, da tallet til venstre for j altid vil være mindre end eller lig med A[j]. I så fald bliver linje 5 og 6 aldrig kørt og linje 4 køres kun 1 gang per iteration altså ligeså mange gange som linje 2,3 og 7. I det bedste tilfælde bliver køretiden altså:

I det bedste tilfælde fremkommer der altså en lineær funktion af formen , hvor a og b er konstanter som afhænger af priserne på de forskellige linjer.

I det 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 for alle de mulige værdier af j, altså j = 2,3,…,n (Cormen, et al., 2009, p. 27). Nu bliver summerne der skal udregnes altså

og

Disse summer kan heldigvis udregnes med følgende formler:

og

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

Nu kan det ses at der fremkommer et andengradspolynomium på formen 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 , derfor , dette vil også genere et andengradspolynomium (Cormen, et al., 2009, p. 28).

I bedste tilfælde vokser køretiden af InsertionSort altså lineært og i det værste og gennemsnitlige tilfælde vokser den, ligesom et andengradspolynomium.

Ovenover blev udtrykkene for køretiderne simplificeret så man i stedet for priserne for hver linje, brugte konstanterne a, b og c. Normalvis kan køretiden 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 asymptotiske notationer. En særligt brugbar asymptotisk notation er (store theta). For en given funktion g(n) er givet ved følgende sæt af funktioner:

|  |  |
| --- | --- |
|  | (3.1) |

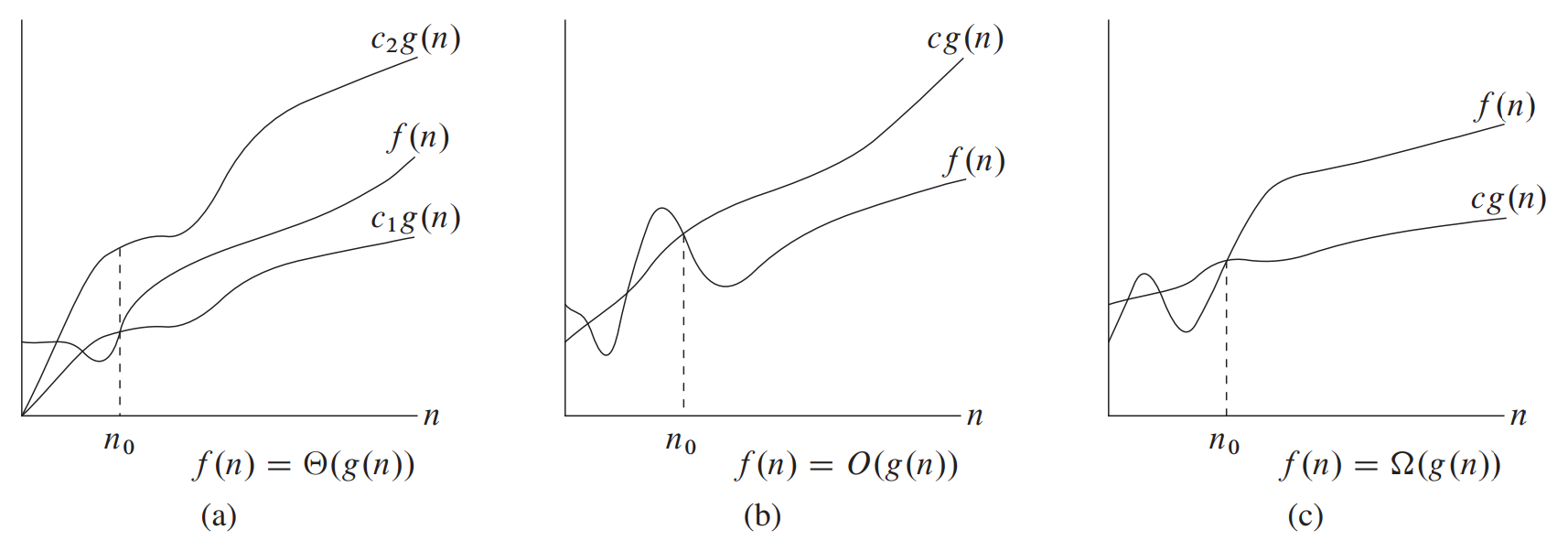
(Cormen, et al., 2009, p. 44)

Det vil sige at hvis der for en funktion eksister positive konstanter og således at

, så tilhører sættet .

Dette kan noteres som eller blot som

Hvis en funktion = så kan det siges om at den for store nok værdi af n, vil vokse ligesom . Foreksempel hvis , så vil for store nok værdier af n, vokse ligesom et andengradspolynomium. Det kan ses på Figur 3.2 **(a)** at f(n) bliver bundet mellem to konstante faktorer af g(n). g(n) giver altså en øvre og nedre grænse for væksten af f(n).



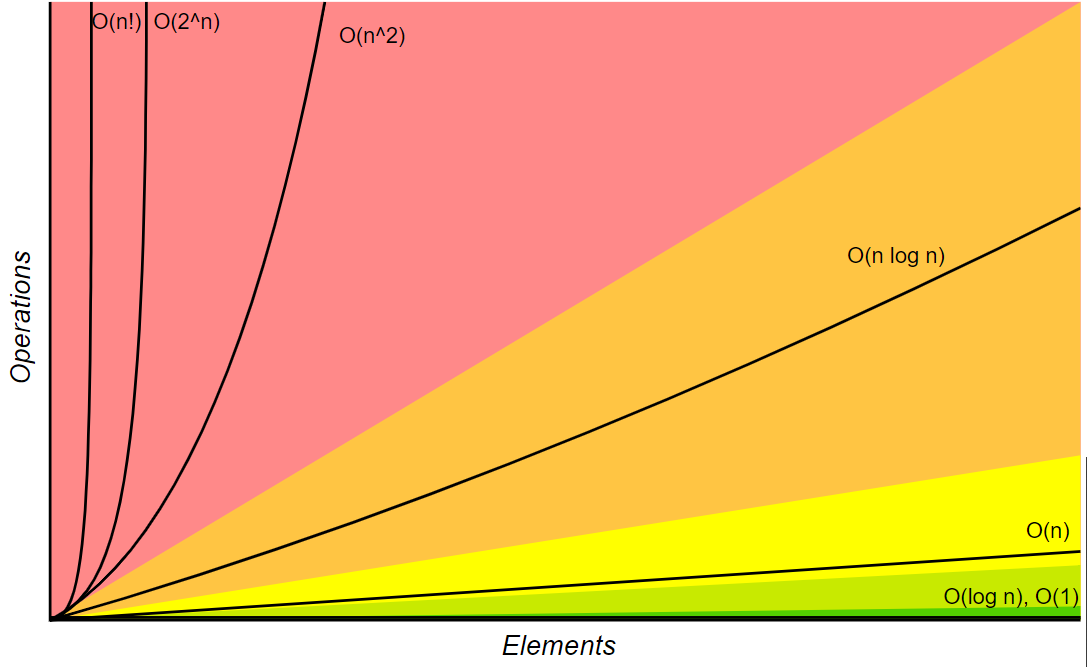
Figur . (Cormen, et al., 2009, p. 45) Illustrationer af . **(a)** Viser -notation her ligger f(n) altid mellem eller på og når . **(b)** Viser -notation. Her ligger f(n) altid under eller på cg(n), når . **(c)** Viser -notation. Her ligger f(n) altid over eller på cg(n), når

For at vise at en funktion er , skal man finde finde 3 konstanter , så uligheden for er opfyldt.

Når man bruger notation, kan man se bort fra konstante faktorer, da disse blot vil forskyde med en konstant faktor. Derfor er stadig .

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 dominerer 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, hvor dem der står til venstre, vokser hurtigst (en illustration af disse kan også ses på Figur 3.3):

(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 tidskompleksiteter, når antallet af elementer bliver større.

Hvis man har en funktion , behøver man da kun at kigge på og væksten bliver da

Derfor kan det også siges, at for et hvilket som helst polynomium af grad d givet ved , hvor er konstanter og gælder det at (Cormen, et al., 2009, p. 46). Med andre ord behøver man i et polynomium kun at kigge på leddet med en grad på d og man kan altså se bort fra alle leddene af lavere grad. Enhver konstant er i virkeligheden et 0-gradspolynomium. Så en konstant funktion kan noteres som eller blot (Cormen, et al., 2009, p. 46).

Nu kan notationen benyttes på køretiderne for InsertionSort. I InsertionSorts værste tilfælde kunne køretiden skrives som et andengradspolynomium på formen . Ifølge ovenstående regler kan man se bort fra konstanterne og leddene af lavere grad. Hvis man giver InsertionSort en omvendt sorteret liste af tal bliver køretiden altså .

I tilfældet af at inputtet var tilfældigt blandet blev køretiden også et andengradspolynomium, så i det gennemsnitlige tilfælde vil køretiden også blive

På samme måde kan køretiden for InsertionSort i bedste tilfælde også findes. Her kunne køre tiden skrives på formen og dette bliver da til .

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 nu benytte to andre asymptotiske notationer -notation (store O) og -notation (store omega). For en given funktion g(n) er givet ved følgende sæt af funktioner:

|  |  |  |
| --- | --- | --- |
|  |  | (3.2) |

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

Hvis en funktion kan det siges at for store nok værdier af n, vokser enten lige så hurtigt som eller langsommere end . En illustration af dette kan ses på Figur 3.2 **(b)**. -notation beskriver altså en øvre grænse for væksten af . kan ikke vokse hurtigere end for store nok værdier af n.

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

|  |  |  |
| --- | --- | --- |
|  |  | (3.3) |

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

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

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

|  |  |  |
| --- | --- | --- |
|  |  | (3.4) |

Hvis man sortererede en allerede sorteret liste (bedste tilfælde) med InsertionSort var køretiden . Som følge af (*3*.*4*) er køretiden i dette tilfælde altså også og endnu mere vigtigt, så er køretiden også . Ligeledes var køretiden ved værste tilfælde (en omvendt sorteret liste) , så i dette tilfælde er køretiden også og da og 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 , så må køretiden for et hvilket som helst tilfælde ligeledes også være , da køretiden jo ikke kan vokse hurtigere end den vokser i det værste tilfælde. Hvis køretiden for det bedste tilfælde var , så må køretiden for et hvilket som helst tilfælde også være , da køretiden ikke kan vokse langsommere end det bedste tilfælde. -notation kan altså bruges til at beskrive den værste mulige vækst for køretiden, til et hvilket som helst input og -notation kan bruges til at beskrive den bedste mulige vækst for køretiden, til et hvilket som helst input.

### Analyse af MergeSort

For at finde køretiden på MergeSort, skal man bruge en lidt 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 . Dette kan ses på linje 1 i MergeSort, som kun vil blive kørt 1 gang i alt hvis , altså hvis n er 1. Når n er større end 1 må køretiden for hvert rekursionsniveau være lig med 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 , må tiden det tager at løse hvert underproblem være lig med og siden der er to underproblemer, må den samlede tid for at løse dem begge være . 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:

Her bliver -notationen brugt i en ligning. Når dette er tilfældet betyder det blot at man snakker om en eller anden funktion som er en del af sættet , dvs en konstant funktion. Hvis der i en ligning står , så snakker man ligeledes om en funktion som er en del af sættet

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 . Kombineringen sker i Merge proceduren. Her er ved hvert rekursionsniveau. Linje 1 til 3 og 8 til 11 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 krop kører i konstant tid for hver iteration. Man får altså tre lineære funktioner . 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

Da man kun behøver kigge på det hurtigste voksende led, skal man i , blot ignorere det konstante led og den rekursive ligning bliver derfor nu:

Som det fremgår af ligningen, så tilføjer hvert rekursionsniveau til køretiden. Hvis man vil finde køretiden, skal man derfor finde antallet af rekursionsniveauer. Hvert rekursionsniveau deler problemet op i to mindre problemer, så det vil sige at antallet af problemer ganges med 2 for hvert niveau. På det sidste niveau er der n problemer, det er nemlig her listerne er blevet delt så meget at det kun er i de enkelte tal der står tilbage. Man bliver altså ved med at gange antallet af underproblemer med 2 indtil man får n. Hvilken funktion kan bestemme hvor mange gange man skal gange 2 med sig selv for at få n? Det kan totals-logaritmen! er netop løsningen til ligningen . Antallet af rekursionsniveauer vokser altså ligesom totals-logaritmen og er derfor eller, siden man kan ændre grundtallet på enhver logaritme ved at gange med en konstant, så vokser antallet af rekursionsniveauer også som . Da der blev tilføjet til køretiden ved hvert rekursionsniveau og der var rekursionsniveauer bliver køretiden for MergeSort altså .

Ligemeget hvordan inputtet er sorteret inden MergeSort bliver kaldt, vil køretiden stadig være , da der i modsætning til InsertionSort ikke er nogen værdi tilsvarende , som afhang af hvordan inputtet var sorteret. MergeSorts køretid er derfor også i alle tilfælde og i alle tilfælde.

### Sammenligning af InsertionSort og MergeSort

Resultatet af køretidsanalyserne for InsertionSort og MergeSort blev som følger:

|  |  |  |  |
| --- | --- | --- | --- |
| Algoritme | Bedste tilfælde | Gennemsnitligt | Værste tilfælde |
| InsertionSort |  |  |  |
| MergeSort |  |  |  |

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

## Implementering i JavaScript

Hvis man gerne vil benytte algoritmerne på sin computer, er det dog ikke nok at have pseudokoden. Man bliver nødt til at implementere dem 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:



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 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.





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

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 såkaldte 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 Binet’s formel.

For løkkeinvarianten skulle man i initialisering-trinnet vise at løkkeinvarianten var sand før den første iteration af løkken. I dette trin viser man altså at algoritmen er sand for én specifik given tilstand, nemlig den tilstand der er inden første iteration af løkken. Dette kan sammenlignes med induktionsbasen i induktionsbeviset, hvor man også skulle vise at udtrykket var sandt for en eller flere specifikke værdier (tilstande) af n.

# Konklusion

# 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

## Induktionsbevis for Binet’s formel

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

|  |  |  |
| --- | --- | --- |
|  |  | (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:

|  |  |  |
| --- | --- | --- |
|  |  | (0.2) |

Startbetingelserne til talfølgen er:

Før Binet’s formel bliver bevist er det relevant at kigge på værdierne og som indgår i Binet’s formel. kaldes det guddommelige forhold og kan også betegnes med det græske bogstav (phi). betegnes som (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.

Et billede, der indeholder tekst, ur

Automatisk genereret beskrivelse

Figur . (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:

Ud fra ovenstående fås følgende:

Hvis man lader så bliver , indsættes disse fås følgende:

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

|  |  |  |
| --- | --- | --- |
|  |  | (0.3) |

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

Herfra bestemmes diskriminanten:

Løsningerne til andengradsligningen er givet ved:

Løsningerne bliver altså:

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 indsættes 0 på n's plads i formlen:

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

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 være lig med:

|  |  |  |
| --- | --- | --- |
|  |  | (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 være lig med:

På samme måde vil være lig med:

Ifølge (*0*.*2*) er givet ved:

Nu kan ovenstående værdier for og indsættes i formlen

|  |  |  |
| --- | --- | --- |
|  |  | (0.5) |

Nu substitueres og ind i (*0*.*5*):

Herfra kan sættes udenfor en parentes

Nu kan der rykkes lidt rundt på leddene:

Herefter sættes og udenfor parenteser.

|  |  |  |
| --- | --- | --- |
|  |  | (0.6) |

Som det blev vist ovenover var både og løsninger til ligningen (*0*.*3*) . Derfor kan udtrykkene og i (*0*.*6*) erstattes med henholdsvis og :

Til slut anvendes potensreglen :

Nu genindsættes på ’s plads og på plads:

Som det kan ses, fremkommer nu (*0*.*4*), som var det resultat 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 . 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