

# Min forside

# Resume

Indhold

[Min forside 2](#_Toc89090492)

[Resume 3](#_Toc89090493)

[Indledning 5](#_Toc89090494)

[1 Giv en kort introduktion til forskellige bevismetoder i matematik 6](#_Toc89090495)

[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](#_Toc89090496)

[2.1 Bevis for Binet’s formel 7](#_Toc89090497)

[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 12](#_Toc89090498)

[3.1 Introduktion til sorteringsproblemet 12](#_Toc89090499)

[3.2 InsertionSort 12](#_Toc89090500)

[3.2.1 Korrekthed af InsertionSort 13](#_Toc89090501)

[3.3 MergeSort 14](#_Toc89090502)

[3.4 Køretider 17](#_Toc89090503)

[3.5 Implementering og analyse 18](#_Toc89090504)

[3.6 Sammenligning 20](#_Toc89090505)

[3.7 Korrekthed 20](#_Toc89090506)

[4 Diskutér kort lighederne mellem dit bevis ved matematik induktion og metoden, du benyttede til at vise korrektheden af InsertionSort og MergeSort. 21](#_Toc89090507)

[Konklusion 21](#_Toc89090508)

[Bibliografi 22](#_Toc89090509)

[Bilag 23](#_Toc89090510)

# 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

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

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

Induktionsbevismetoden er en metode der kan bruges til at bevise at et udtryk er gyldigt for alle eller for alle .

Det vil sige at induktionsbevismetoden kan bruges til at bevise udtryk af følgende form:

Eller udtryk som:

Hvor r er et positivt heltal.

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 eller flere bestemte værdier af n. Det kunne for eksempel være at vise at udtrykket er sandt når n er 1.

I induktionstrinnet antager man først at udtrykket er sandt for

Herefter viser man i induktionstrinnet at hvis man antager at udtrykket er gyldigt for tallet n, så er udtrykket også gyldigt for tallet n+1.

## Bevis for Binet’s formel

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

|  |  |  |
| --- | --- | --- |
|  |  | (2.1) |

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:

|  |  |  |
| --- | --- | --- |
|  |  | (2.2) |

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 2.1 nedenfor.

Et billede, der indeholder tekst, ur

Automatisk genereret beskrivelse

Figur 2.1 (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:

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:

|  |  |  |
| --- | --- | --- |
|  |  | (2.3) |

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:

|  |  |  |
| --- | --- | --- |
|  |  | (2.4) |

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 (*2*.*2*) er givet ved:

Nu kan ovenstående værdier for og indsættes i formlen

|  |  |  |
| --- | --- | --- |
|  |  | (2.5) |

Nu substitueres og ind i (*2*.*5*):

Herfra kan sættes udenfor en parentes

Nu kan der rykkes lidt rundt på leddene:

Herefter sættes og udenfor parenteser.

|  |  |  |
| --- | --- | --- |
|  |  | (2.6) |

Som det blev vist ovenover var både og løsninger til ligningen (*2*.*3*) . Derfor kan udtrykkene og i (*2*.*6*) erstattes med henholdsvis og :

Til slut anvendes potensreglen :

Nu genindsættes på ’s plads og på plads:

Som det kan ses, fremkommer nu (*2*.*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

# 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

## 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 ligesom 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 dog beskriver tingene på en mere letlæselige måde end de normalt ville forekomme i de populære programmeringssprog, som bruges i industrien. Pseudokoden for InsertionSort ser således ud:



INDSÆT DETTE ET ELLER ANDET STED SOM IKKE ER HER

På linje laves en for-løkke hvor j starter med at være 2 og vil kører til og med A.length, altså længden (n) af listen. På linje 2 laves en variabel key som sættes til at være A[j], 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. Det vil sige at A[i] er det tal, der er én plads til venstre for A[j] i listen.

På linje 4 laves en while løkke der kører så længe i er større end 0 og A[i] er større end key. 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 i listen så det nu har pladsen A[i+1]. Siden i er j-1 til at starte med, vil den første iteration af while løkken rykke A[i] over på A[j]’s plads altså over på key’s plads. Efter A[i] er rykket til højre gøres i én mindre på linje 6, det vil sige at der kigges én gang mere til venstre.

Til sidst, når i er 0 eller A[i] ikke er større end key bliver A[i+1] sat til key på linje 7. Så hvis i er lig med 0, betyder det at der ikke blev fundet en A[i] som var mindre end eller lig med key og derfor bliver key indsat på A[1], altså den første og mest venstre plads i listen. Hvis while-løkken stoppede fordi A[i] var mindre end eller lig med key så indsættes key én plads til højre for A[i], hvilket giver god mening da key enten må værre større eller lig med A[i] og derfor skal key i en sorteret rækkefølge være placeret til højre for A[i]

HERFRA

For at forstå hvordan algoritmen fungerer, er det vigtigt at have en grundlæggende viden om hvordan pseudokode opskrives. Derfor vil nogle af de grundlæggende begreber indenfor pseudokode nu blive introduceret.

Når der skrives A[i] betyder det at man tilgår det i’te element i listen A. 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 variabler ved at bruge er lig med tegnet (=), så hvis man for eksempel skriver min\_variabel = 3, så tildeler man værdien 3 til variablen min\_variabel. Ligeledes tildeler man for eksempel værdien A[j] til variablen key på linje 2.

På linje 1 laves der en **for** løkke, her sættes variablen j til at være 2 til at starte med. Hele løkkens krop, det vil sige alt under linje 1, som er indrykket (dvs. linje 2-8), vil nu blive udført én gang hvor j = 2. Herefter vil j blive gjort én større, dvs. j = 3. Nu køres hele koden igennem hvor j = 3. Herefter vil j blive gjort én større igen og løkken vil blive kørt igen. Denne proces fortsætter indtil j er lig med A.length dette bliver udtrykt når der skrives ”**to** A.length”. Det vil sige løkkens krop køres én gang for værdierne j = 2,3,4…. A.length. Når koden har kørt for j = A.length, vil j bliver gjort én større dvs. j = A.length + 1 og for-løkken vil ikke længere blive kørt, i pseudokode vil j dog beholde sin værdi. Når en løkke udfører sin krop, kaldes det er iteration.

En anden form for løkke, som der gøres brug af, er en **while** løkke. Denne bruges på linje 5. En **while** vil køre sin krop så længe udtrykket der står efter **while** er sandt, med andre ord vil løkken køre indtil udtrykket er falskt. Det vil sige at **while** løkken på linje 5 vil køre så længe i er større end 0 og A[i] er større end key. Rent praktisk fungerer **while** løkken ved at den først tjekker om udtrykket er sandt og hvis dette er tilfældet, så udføres kroppen én gang. Herefter tjekkes det igen om udtrykket er sandt og i så fald udføres kroppen igen og dette fortsætter indtil udtrykket bliver falskt. Det vil sige at hvis i på et tidspunkt i løkkens krop bliver sat til 0, så vil løkken stoppe med at køre når udtrykket tjekkes igen.

### 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 hvorfor en algoritme er korrekt. Der skal vises tre ting om denne løkkeinvariant:

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

**Vedligeholdelse:** Hvis løkkeinvarianten er sand før en given iteration af løkken, så forbliver den sand før den næste iteration af løkken.

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

Når man har vist at løkkeinvarianten er sand før den første iteration af løkken og at den er sand før den næste iteration såfremt at den var sand før en given iteration af løkken, kan man slutte at løkkeinvariant må være sand før enhver iteration af løkken.

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)

Nu skal det altså vises at denne løkkeinvariant er sand før den første iteration af for-løkken, at hvis løkkeinvarianten er sand før en iteration af løkken, forbliver den også sand inden den næste iteration af løkken og at løkkeinvarianten 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 A[1..j-1] bliver til A[1..1] og består derfor kun af ét element nemlig A[1], dette element er selvfølgelig også det oprindelige element A[1]. Siden der kun er ét element i dellisten, må dellisten være sorteret. Det er nu vist at alle betingelserne for løkkeinvarianten er 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 løkkeinvarianten 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 for-løkkens kroppe så virker den på den måde at den på linje 5 og 6 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 er rykket på elementer i A[1..j-1] og de hver højest er rykket én plads til højre kan man nu sige at A[1..j] består af de oprindelige elementer i A[1..j], netop fordi A[1..j] består af elementerne i A[1..j-1] som enten er rykket eller ikke er rykket og så key eller A[j]. Siden man har rykket alle elementerne som var større end A[j] én plads til højre og herefter har indsat A[j] i mellem elementerne der er større end A[j] og mindre end eller lig med A[j], så må dellisten A[1..j] også være sorteret. Når iterationen er kørt vil j bliver gjort én større inden den næste iteration, det vil sige at det der var A[1..j] er nu A[1..j-1] og siden det lige er blevet vist at denne delliste indeholder de oprindelige elementer fra A[1..j-1], men i sorteret rækkefølge, så betyder det at løkkeinvarianten er 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 blev afsluttet er at j blev større end A.length. Siden j blev gjort én større efter hver iteration må det betyde at j er lig med A.length + 1, som var det samme som n + 1. Hvis dette sættes 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 sorteringsproblemet på en rekursiv måde, det vil sige at den kalder sig selv med mindre sorteringsproblemer. MergeSort 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 deler man problemet i flere underproblemer, som ligner det oprindelige problemet. I MergeSort gøres dette ved at dele listen af tal midt over, sådan at 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 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 vigtigt del af MergeSort er kombination af de halverede lister, til samlet sorteret liste. Man kan, ligesom med InsertionSort, sammenligne kombineringen i MergeSort med et kortspil. Hvis man forestiller sig at man har to sorterede bunker af kort ved siden af hinanden, som vender man 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 for eksempel være kortet i venstre bunke. Dette kort tager man nu fra bunken og ligger det med bagsiden nedad, i en ny bunke. Nu er der kommet et nyt kort i toppen af venstre bunke, dette sammenligner man nu med kortet som stadig var i toppen af højre bunke og man tager igen det laveste af disse kort og ligger i den tredje bunke. Denne proces fortsætter man indtil en af bunkerne er tom, så 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 så 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 man opfatte A[p..q], som den ene bunke og A[q+1..r], som den anden bunke og den sorterede 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 altså 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. Man indsætter uendelig i enden af de to lister for at undgå at tjekke om listerne om ”kortbunkerne” er tomme. Når man i stedet når til uendelig, så ved man at det ikke kan være det mindste kort i toppen af de to bunker og 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.

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 indtil k er større end r, 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 altså 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 (som følge af, at der står else på linje 16). 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.

Rent intuitivt kan man måske nu se at 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] at man så opnår en sorteret kombinering af L og R i A[p..r]

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 beregne den midterste indeks i A[p..r], ved at lægge p og r sammen og dividere med 2. 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 sorterede 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 ændre 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 når løkken terminerer, som kan vise korrektheden af MergeSort.

**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 listen L var en kopi af A[p..q], som det blev antaget var sorteret når Merge proceduren blev kaldt, og derfor må det første element i listen L, da også være det laveste. Det samme gælder for listen R, som var en kopi af listen A[q+1..r], som også var en sorteret liste. 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], så må A[p..k] indeholde de k-p + 1 mindste tal i L og R. Inden næste iteration vil k da 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], indeholde 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 n1+1+n2+1 = (q-p+1)+1+(r-q)+1=r-p+3 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. Først 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 alting er sorteret når man så til sidst kombinerer de to halve lister med størrelser på n/2 til den samlede sorterede liste.

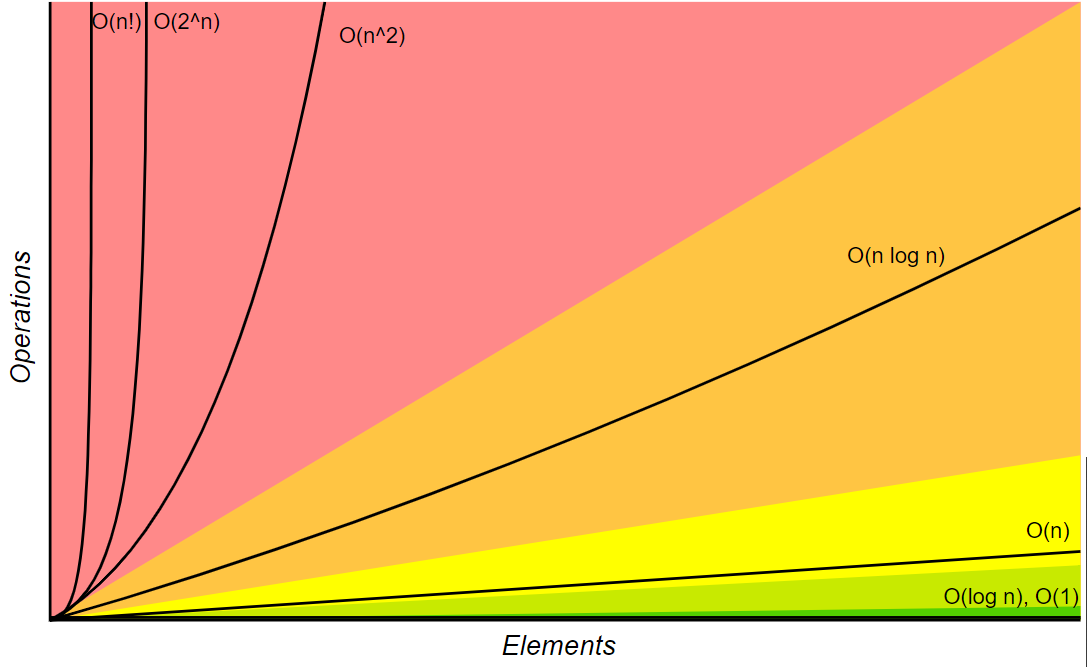
## Analyse af køretider

Når man skal sammenligne to algoritmer, kan det være relevante at kigge på den tid det tager dem at udføre deres opgaver, det man kalder køretid. Det giver dog ikke så meget mening at kigge på den specifikke tid det tager algoritmerne at sortere et specifikt antal tal, da denne tid kan variere meget alt afhængigt af hvor hurtig maskinen der kører algoritmen er. Det kan derfor være mere relevant at kigge på hvordan køretiden udvikler sig når man øger antallet af tal, som skal sorteres.

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

|  |  |  |
| --- | --- | --- |
|  |  | (3.2) |
|  |  | (3.3) |

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



Figur 3.2 (Rowell, et al., u.d.) Illustration af forskellige tidskompleksiteter. 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.

Man kan drage paralleller mellem den asymptotiske notation for funktioner f og g og den algebraiske sammenligning af to reelle tal a og b.

er ligesom

er ligesom

er ligesom a = b

er ligesom a < b

er ligesom

Denne analogi holder dog ikke i alle tilfælde (Cormen, et al., 2009, p. 52).

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

# Bilag