SORTERINGSALGORITMER

JÄMFÖRELSE OCH UNDERSÖKNING AV ALGORITMER FÖR SORTERING I JAVA

Av: Johan Ekdahl

Klass: EVXJUH21

2023-03-03

INNEHÅLLSFÖRTECKNING

SAMMANFATTNING

Sorteringsalgoritmer är en viktig del av systemutveckling och används för att sortera data i en viss ordning. I denna rapport undersöks fem vanliga sorteringsalgoritmer - Bubblesort, Selectionsort, Insertionsort, Mergesort och Quicksort - genom att implementera och testa deras prestanda på olika datamängder. Resultaten visar att vissa algoritmer är snabbare och mer effektiva än andra beroende på storleken på datamängden och användningsområde. Sorteringsalgoritmer används i många olika applikationer, inklusive sökmotorer, databaser, bildbehandling och många andra datadrivna plattformar.

Därför är det viktigt att utvecklare har en god förståelse för de olika sorteringsalgoritmerna och deras prestanda, för att kunna välja rätt algoritm för rätt uppgift och för att kunna skapa effektiva och optimerade program. Som exempel fann rapporten att Insertionsort var speciellt lämpad för listor som är nästan helt sorterade. Mergesort och Quicksort var speciellt lämpade för sortering av större listor. Bubblesort och Quicksort är bra på mindre datamängder samt är lämpliga startpunkter för en novis att lära sig sorteringsalgoritmer.

FÖRORD

Jag är glad och tacksam över att presentera denna rapport som en del av mitt examensarbete. Jag vill uttrycka min uppskattning till EC-utbildning, min klass Javautvecklare-21, och mina lärare därifrån för deras stöd och uppmuntran under hela min studietid. Jag vill också tacka instruktör Andrei Neagoie och hans webbkurs "Master the Coding Interview: Data Structures + Algorithms" för att ge mig den grundläggande kunskapen och färdigheterna för att kunna utföra detta arbete. Jag vill tacka alla som har hjälpt mig under min utbildning, mitt LIA-företag Sigma och mina handledare där, och avslutningsvis mina vänner och familj.

INLEDNING

När en systemutvecklare söker jobb eller blir kontaktad av en rekryterare och kommer vidare i en anställningsprocess så uppenbarar det sig snabbt att företag och rekryterare lägger stor vikt vid att personen kan datastrukturer och algoritmer. Detta leder ofta för eller senare att ett kodprov ges ut där utvecklaren ska visa förståelse för dessa begrepp och kan praktiskt implementera dessa.

Utvecklaren presenterar sina lösningar och rekryteraren vill gärna se olika lösningar med olika komplexitet för samma problem, samt en motivering för vilka fördelar det finns för den mer avancerade lösningen. Normalt sett får utvecklaren alltid möjlighet att göra denna motivering under testets gång eller efter testet har skickats in i. Rekryteraren vill se att personen kan lösa problemet på bästa möjliga sätt men också problemlösning som en iterativ process där utvecklaren kan göra förbättringar på sin lösning.

BAKGRUND

Skälet till att detta är grundförfarandet i anställningsprocesser är att datastrukturer och algoritmer utgör kärnan i programmering, för att bli en framgångsrik och produktiv utvecklare är det nödvändigt att ha en djup förståelse för dessa koncept. I dagens snabbt digitaliserande värld med enorma mängder data som processas så kan en lite snabbare, eller lite mindre minneskrävande algoritm spara stora mängder resurser i operationskostnader för datacenter.

Användare av digitala tjänster förväntar sig också en tjänst med minimal responstid idag. Skulle tjänster som exempelvis Instagram eller Twitter ha laddningstider som webbsidor hade på 90-talet skulle användarupplevelsen varit oacceptabel. Mjukvaruföretag vill därför försäkra sig om att deras utvecklare har kapaciteten att resonera kring olika implementationer och producera kod av hög kvalitét som är både skalbar och kostnadseffektiv.

En viktig del av en utvecklares förmåga att lösa problem och bygga effektiv skalbar kod är att ha en god förståelse om olika sorteringsalgoritmer. Sorteringsalgoritmer är en stor del av programmering eftersom de används för att ordna och organisera data på ett snabbt och effektivt sätt. Genom att olika sorteringsalgoritmer används i olika scenarion kan tjänster och applikationer hantera stora datamängder på bästa möjliga tid.

Det är viktigt att poängtera att alla utvecklare nödvändigtvis inte behöver ha kunskapen att skriva sorteringsalgoritmer med bästa möjliga prestanda. Det är framför allt på företag med tjänster för miljoner användare såsom Twitter och Instagram, där optimeringar är av absolut nödvändighet. Även om alla utvecklare inte behöver ha denna kunskap är det dock viktigt för alla utvecklare att ha en grundläggande förståelse om datastrukturer, algoritmer och sorteringsalgoritmer, oavsett vilken nivå man är på som utvecklare.

SYFTE OCH FRÅGESTÄLLNING

I rapporten kommer jag att undersöka och jämföra olika sorteringsalgoritmer och deras prestanda i olika testscenarion. Rapporten kommer vara ifrån perspektivet av en utvecklare som har små tidigare erfarenheter

av att implementera egna sorteringsalgoritmer och hoppas kunna ge en bättre förståelse för hur dessa algoritmer fungerar och vilka faktorer som påverkar deras prestanda. För att uppnå detta syfte har jag följande frågeställningar

- Vilka sorteringsalgoritmer finns det och hur fungerar dem
- Vilka faktorer påverkar prestandan hos dessa sorteringsalgoritmer och hur kan de optimeras för bättre prestanda
- Vilken sorteringsalgoritm är mest effektiv för olika datamängder

AVGRÄNSNINGAR

Jämförelse av sorteringsalgoritmer kan göras på många olika sätt och på olika typer av data. Jag kommer i rapporten mäta den relativa hastigheten det tar för en algoritm att sortera en array av integers. Det vill säga hur lång tid det tar att exekvera en lista av siffror för varje algoritm. De olika listorna är Small Array, Medium Array, Large Array och Almost Sorted Array.

- Small Array – Tio slumpartade siffror från 1 till 10
- Medium Array – Tusen slumpartade siffror från 1 till 1000
- Large Array – Hundra tusen slumpartade siffror från 1 till 100 000
- Almost Sorted Array – Hundra siffror som är nästan sorterade från 1 till 100

Den sista listan av siffror som är nästan sorterad innebär i praktiken att bara en siffra har bytt plats ett fåtal steg. Detta förekommer ofta i verkliga scenarion där ett nytt element har lagts till i en datastruktur och där önskan är att hålla denna datastruktur sorterad för snabbare hastighet vid sökningar.

METOD

För att få en djupare förståelse för sorteringsalgoritmerna tänker jag praktiskt implementera dem i ett programmeringsspråk. Min utbildning som har grund i Javaspråket får mig att välja Java. Jag har ingen anledning att misstänka att det inte skulle kunna gå att implementera sorteringsalgoritmerna i vilket språk som helst. Men det kan finnas skillnader, åtminstone i sättet som de implementeras och möjligen i hastighet och minnesanvändning.

Algoritmerna ska sedan köras med listorna som valts ut i avgränsningen och tiden det tar att utföra 10 körningar kommer att föras in i ett Excelblad. Tidsmätningen görs i koden då jag inte ser någon anledning att använda ett tredjepartsverktyg, uppfattningen är att det görs med hög precision och är fullständigt adekvat när det bara är tiden som ska uppskattas. Med tiderna och med hjälp av verktygen som erbjuds av Excel så kommer jag kunna sammanställa den insamlade datan på ett lättöverskådligt sätt och kunna framställa den visuellt med stapeldiagram i Excel.

För att göra testarbetet vetenskapligt och konsistent så kommer alla tester köras på samma sätt och mätas på samma vis med samma dator. De 10 testerna kommer köras programmatiskt efter varandra för varje algoritm och algoritmerna kommer köras med samma förutsättningar och då menas att inga andra fönster av program kommer köras ex Chrome, Word osv. Detta är för att säkerställa att annan belastning på datorn ska ha så minimal påverkan på resultatet som möjligt.

Förutom att genomföra testningen av algoritmerna så kommer också en betydelsefull del av arbetet bestå av att samla in kunskap om hur algoritmerna implementeras. Jag har begränsad men inte obefintlig förkunskap

om sorteringsalgoritmer så för att testerna ska gå väl så behöver förkunskaperna kompletteras med studier. Eventuella förbättringar som kan göras på mina algoritmer kommer också studeras och samlas in på samma sätt och följa med in i rapporten.

HUVUDDEL

ARBETETS BEGYNNELSE

Första steget i arbetet kommer vara att ta reda på vilka sorteringsalgoritmer som finns och som är betydelsefulla inom programmering. Jag har tidigare påbörjat en webbkurs ”Master the Coding Interview: Data Structures + Algorithms” av instruktören Andrei Neagoie på e-learning plattformen Udemy. Det var från denna webbkurs jag fick inspiration till arbetet och det är från denna kurs jag kommer inhämta kunskaper ifrån och använda som källa.

I webbkursens sektion 13 ”Algorithms: Sorting” behandlas sorteringsalgoritmer med denna arbetsprocess: Instruktören, Andrei Neagoie, beskriver först de abstrakta egenskaperna hos en algoritm, för att sedan låta eleven (jag) genomföra en övning och försöka implementera algoritmen utifrån hans beskrivning. I nästkommande avsnitt så går instruktören steg-för-steg igenom lösningen i Javascript.

De tre första algoritmerna som behandlas kallas ”Elementära sorteringsalgoritmer” och är vanligt förekommande algoritmer som nybörjare får lära sig. De har detta anseende för att de implementeras på ett sätt som känns intuitivt, så som en nybörjare först hade angripit ett sorteringsproblem utan förkunskaper.

DE ELEMENTÄRA SORTERINGSALGORITMERNA

I kommande tabeller så kommer begreppen Time Complexity och Space Complexity förekomma och är två mått på tidseffektivitet och minneseffektivitet. Det är mått för att jämföra algoritmers skalbarhet med varandra och skrivs som Big O (N), uttalas Big O av N, och betyder då att tidskomplexiteten skalar upp linjärt med värdet N som är termen för antalet element som sorteras. Likaså skulle tidskomplexiteten för en algoritm med Big O(N^2) skalas upp kvadratiskt med ett högre värde på N. Minneskomplexiteten skrivs på samma form och en algoritm med minneskomplexitet Big O(1) betyder att minnesåtgången är konstant, det vill säga listan som sorteras gör det utan att allokera nytt minne i datorn.

Bubblesort	
Bubblesort är en enkel sorteringsalgoritm som lämpar sig för små listor. Algoritmen går igenom listan flera gånger och jämför två intilliggande element i taget. Om de är i fel ordning byter algoritmen plats på dem. Genom att upprepa detta steg kommer de större elementen att flyta uppåt i listan som "bubblor". Till slut kommer elementen hamna på rätt plats i listan. Även om Bubblesort är enkel att förstå och implementera, är den inte särskilt effektiv för stora listor.	
Time Complexity	**Space Complexity**
Worst Case: O(N^2)	Worst Case: O(1)

Selectionsort	
Selectionsort sorterar genom att hitta det minsta elementet i listan och flytta det till den första positionen. Därefter upprepas processen för att hitta det näst minsta elementet och placera det på den andra positionen, och så fortsätter man tills hela listan är sorterad. Algoritmen fungerar genom att iterera över listan och jämföra varje element med det minsta elementet som hittills hittats. Om det aktuella elementet är mindre än det minsta elementet, byter algoritmen plats på dem. Selectionsort är en enkel sorteringsalgoritm att implementera, men är ineffektiv för stora listor.	
Time Complexity	**Space Complexity**
Worst Case: O(N^2)	Worst Case: O(1)

Insertionsort	
Insertionsort sorterar genom att betrakta varje element i listan och placera det i rätt position i den sorterade delen av listan. Algoritmen bygger successivt upp den sorterade delen av listan genom att jämföra det aktuella elementet med det föregående elementet i den sorterade delen. Om det aktuella elementet är mindre än det föregående, byter algoritmen plats på dem tills elementet hamnar på rätt position. Insertionsort är enkel att implementera och effektiv för små listor och för listor som är nästan färdigsorterade.	
Time Complexity	**Space Complexity**
Worst Case: O(N^2) Best: O(N)	Worst Case: O(1)

Bland de tre elementära sorteringsalgoritmerna så är det enkelheten av implementeringen som är gemensamt. Insertionsort utmärker sig dock på det viset att den har ett attraktivt Best Case scenario för nästan sorterade listor då den bara behöver iterera en gång och skalar därmed linjärt O(N). Det gör Insertionsort kommersiellt gångbar i vissa scenarion, även i jämförelse med mer avancerade algoritmer.

AVANCERADE SORTERINGSALGORITMER: MERGE SORT OCH QUICK SORT

I webbkursens senare avsnitt introduceras två algoritmer som är två av de mest använda sorteringsalgoritmerna för att sortera stora datamängder, med en stor fördel jämfört med de elementära algoritmerna. Båda använder sig av metoden ”Divide and conquer” vilket praktiskt betyder att listan delas på

mitten för att senare sortera varje del-lista. Detta gör att listor som sorteras med mergesort eller quicksort gör det med tidskomplexiteten Big O(N log N) vilket gör dem mycket effektiva på att sortera stora datamängder.

Mergesort	
Mergesort är en effektiv sorteringsalgoritm som fungerar genom att dela upp listan i mindre delar och sedan sortera varje del för sig. Därefter kombineras de sorterade delarna för att bygga upp den slutgiltiga, sorterade listan. Algoritmen fungerar genom att dela upp listan i hälften och upprepa processen på varje halva tills man når små, enskilda delar av listan. Sedan kombineras delarna genom att jämföra och placera elementen i rätt ordning. Mergesort är en stabil och effektiv sorteringsalgoritm som fungerar väl på stora listor, men kräver mer minne än andra sorteringsalgoritmer.	
Time Complexity	**Space Complexity**
Worst Case: O(N Log N)	Worst Case: O(N)

Quicksort	
Quick sort fungerar genom att välja ett pivot-element i listan och sedan dela upp listan i två delar: en del som innehåller element som är mindre än pivot-elementet och en del som innehåller element som är större än pivot-elementet. Därefter sorteras de två delarna var för sig genom att upprepa processen med pivot-elementet tills varje del består av enskilda element. Sedan kombineras de sorterade delarna för att bygga upp den slutgiltiga, sorterade listan. Quicksort är en effektiv sorteringsalgoritm som fungerar väl på stora listor, men kan vara instabil i vissa implementationer.	
Time Complexity	**Space Complexity**
Avg Case: O(N Log N) Worst: O(N^2)	Worst Case: O(Log N)

Från tabellerna för Mergesort och Quicksort så används begreppen ”Stabil” och ”Instabil” för algoritmerna. Enkelt förklarat betyder en stabil algoritm att om två element i en lista har samma värde, som i detta exempel: [4,2,2,1], så behåller elementen samma relativa position i minnet: [1,2,2,4] (tvåorna har inte bytt plats). Det har betydelse om det är element med andra attribut i listan som sorteras. Quick sort klassas därmed som en instabil algoritm, för den kan inte garantera att element med samma värde håller kvar sin position. Det finns också ett gränsfall i Quick sort när ett pivot-element är valt som är antingen det högsta eller lägsta i listan som orsakar att algoritmen har ett worst case scenario (N^2). Det finns tekniker för att undvika detta så Quick sort anses ändå ha en tidskomplexitet på O(N Log N), även om man i strikt mening alltid bör utgå från worst case scenariot när man jämför algoritmer.

Från kursen nämns ytligt att motorn i Webbläsaren Chrome använder sig utav en kombination av Insertionsort och Quicksort. Medan webbläsaren Firefox använder sig utav en hybrid av Insertionsort och Mergesort, kallad ”TimSort”. Från och med version 70 av Chrome används dock en stabil sorteringsalgoritm även i Chrome.

De 5 algoritmerna som tidigare beskrivits ska ingå i testprocessen; Quicksort, Mergesort, Insertionsort, Selectionsort och Bubblesort. De elementära algoritmerna implementerar jag självständigt i högre grad medan Mergesort och Quicksort använder jag mig mer utav kursmaterialet från webbkursen.

Implementationerna jag gjort kan hittas på github.com/j-development/

Jag har säkerställt att mina implementationer kompilerar och fungerar i ett snabbtest, och att algoritmerna är implementerade så de lever upp till deras vedertagna funktionalitet. Men det finns givetvis förbättringar som kan göra implementationerna bättre. Jag kommer lista upp de vanligaste optimeringarna här.
Bubblesort, med ”Early termination” kan man avsluta sorteringen i förtid om listan är helt sorterad vid första iterationen. Detta görs genom att använda en flagga som håller ett värde som indikerar att listan inte ändrats när man kommit till slutet av listan.
Selectionsort, genom att använda en binär sökalgoritm för att hitta det minsta elementet kan man minska antalet jämförelser som behövs för att hitta det minsta elementet.
Insertionsort, för att hitta det minsta elementen kan man använda en binär sökalgoritm för att hitta rätt position att sätta in det nya elementet. Detta kan minska antalet jämförelser som krävs för att hitta rätt position.
Mergesort, En vanlig förbättring av Mergesort är att använda en annan sorteringsalgoritm för att sortera små delar av listan. När listan delas upp i mindre delar, kan en annan algoritm användas för att sortera delarna om de är tillräckligt små. Detta kan minska antalet jämförelser och kopieringar som krävs för att sortera listan.
Quicksort, För att hantera problemet där ett pivot-element väljs ut som råkar vara det högsta eller lägsta i listan kan man implementera en strategi där pivot-elementet väljs ut som är medianen av de tre första elementen. Detta minskar risken för att Quicksort hamnar i det värsta scenariot med tidskomplexiteten O(N^2).

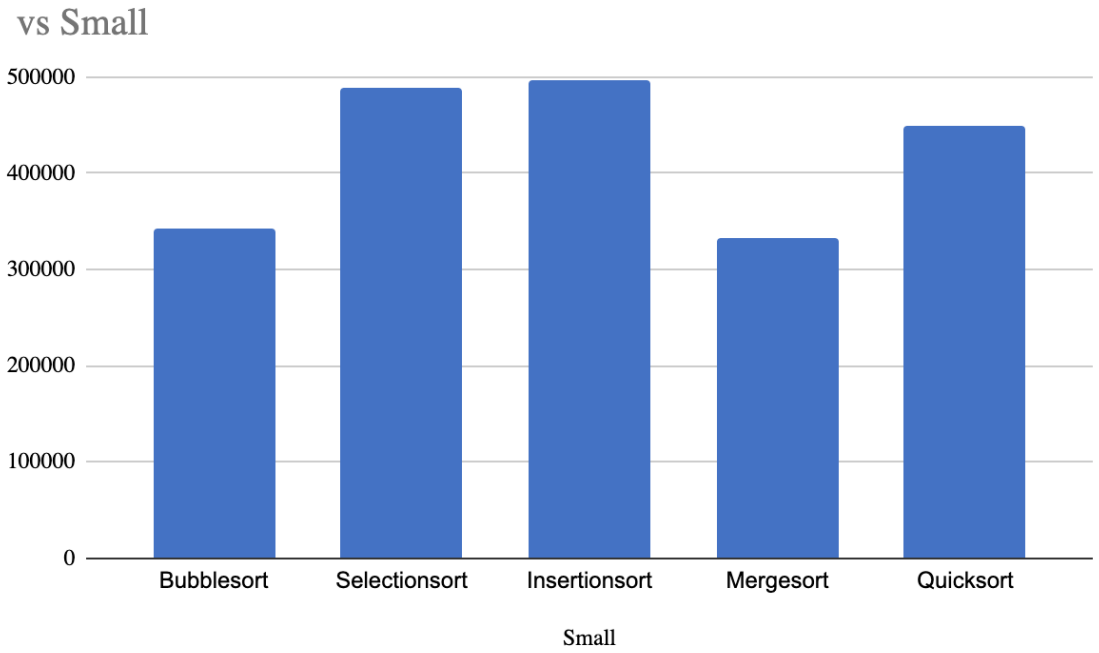
ANALYS AV RESULTAT

Testerna har utförts och datan samt resultatet av testerna har lagts in i ett Excelblad som kan hittas här: Github/Sorting Algorithms.xlsx

Det som uppenbarade sig redan vid första testet var att första iterationen av 10 kunde gå 100 gånger långsammare än de efterföljande 9. Det de kan bero på är att JVM:en (Java virtual machine) måste kompilera och ladda in nödvändiga klassfiler och vid efterföljande iterationer är dessa filer redan cachade(Cache är en snabbtillgänglig form av datalagring) i minnet och kan således utföra algoritmen mycket snabbare. Jag försökte korrigera för detta genom att bygga JAR-filer av algoritmerna och köra dem direkt i förhoppningen att det skulle ta bort overhead på första iterationen. Det gjorde ingen skillnad.

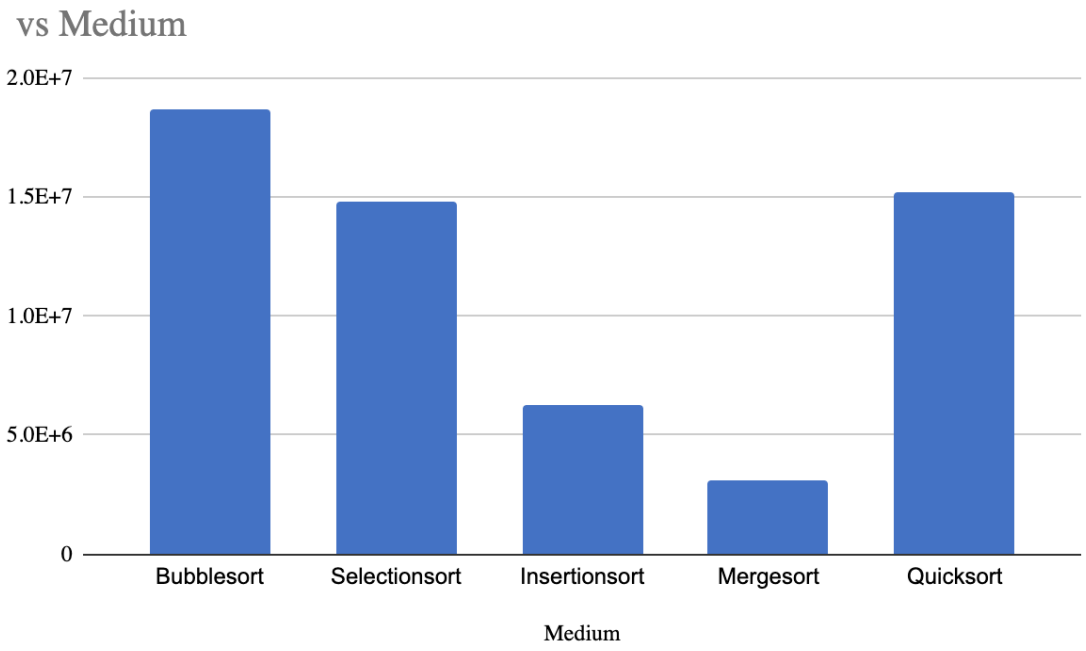
Jag övervägde alternativ så som att göra om testet till att testa 1 iteration åt gången men landade i att jag inte bör ändra testscenariot utefter upptäckter under testfasen.

- Small Array – Tio slumpartade siffror från 1 till 10



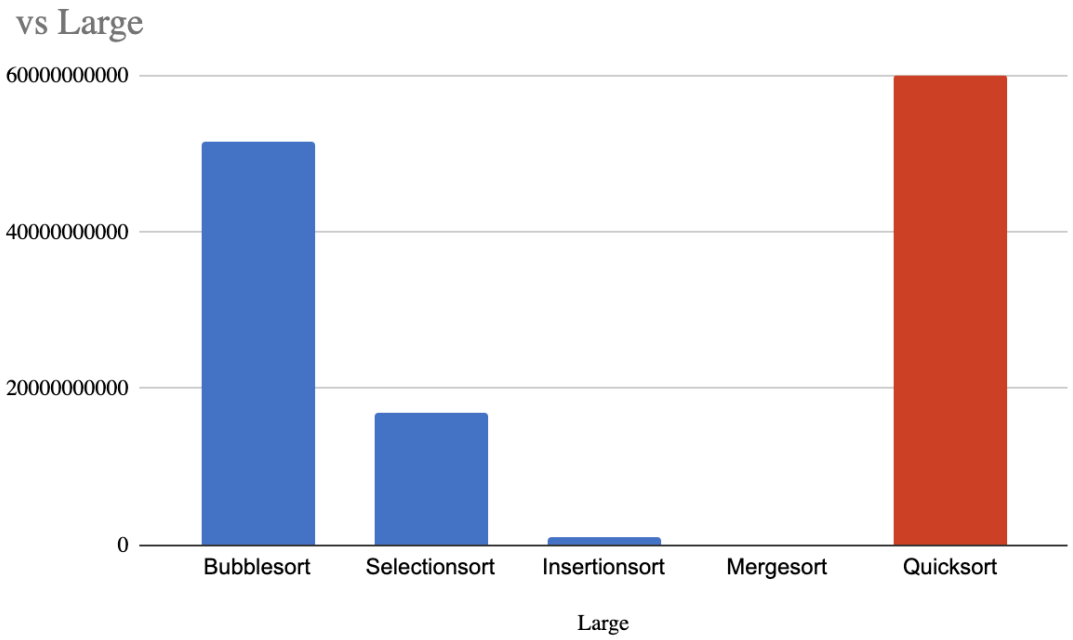
I det första testscenariot med en lista på 10 siffror så ser man ingen betydelsefull skillnad mellan algoritmerna. Bubblesort som är en simpel algoritm står sig väl mot de andra elementära algoritmerna och är även lite snabbare än den mer sofistikerade Quicksort. Den adderade overheaden på första iterationen gällde också samtliga algoritmer och det verkar därmed inte vara någon algoritm som dragit fördel av att vara mer optimerad än någon annan.

- Medium Array – Tusen slumpartade siffror från 1 till 1000



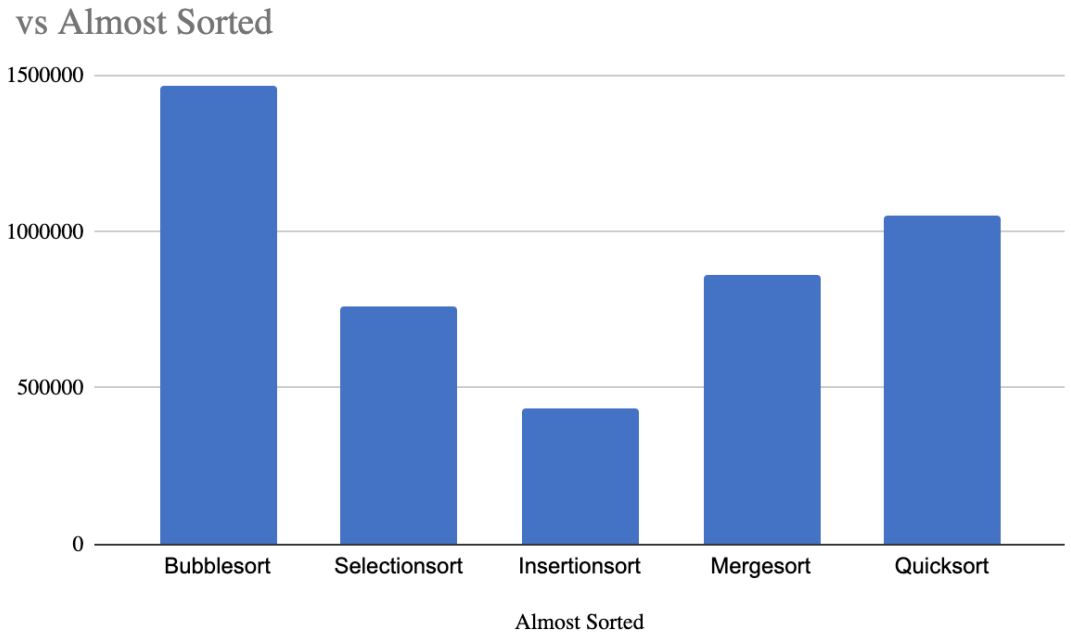
På den mellanstora listan med 1000 siffror så tappar Bubblesort markant från tidigare och placerar sig sist av alla och sist av de elementära algoritmerna. Vilket inte är så chockerande för vad som betraktas som den enklaste sorteringsalgoritmen. Den sofistikerade algoritmen Mergesort tar plats nummer ett, det som sticker ut är Quicksort som är besläktad med Mergesort genom att de båda använder Divide and Conquer-strategin. Quicksort placerar sig här näst sist. Detta verkar bero på testscenariot där de efterföljande iterationerna tynger ner Quicksort.

- Large Array – Hundra tusen slumpartade siffror från 1 till 100 000



I den stora listan med 100 000 siffor så är fokus igen på Quicksort som inte ens kan slutföra de 10 iterationerna av testet. Efter undersökning så verkar Quicksort ha problem med att sortera samma lista flera gånger, Quicksort har inte samma problem när listan slumpas om för *varje* iteration. Jag ställs återigen inför dilemmat om testscenariot ska ändras men landar i att det är viktigt att hålla fast vid de testscenarion som jag lagt fram från start. Mergesort är en solklar etta och visar på behovet av en mer sofistikerad algoritm vid större listor. För de elementära algoritmerna så är det Insertionsort som sticker ut då den presterar bra med tanke på att listan som testas är mycket stor.

- Almost Sorted Array – Hundra siffror som är nästan sorterade från 1 till 100



Den sista listan som testas är Almost Sorted-listan och här blir resultatet ganska väntat, Insertionsort bekräftar sitt anseende som den snabbaste för listor som är nästan helt sorterade. De avancerade algoritmerna Mergesort och Quicksort står sig inte mot Insertionsorts linjära tidskomplexitet O(N).

AVSLUTANDE DISKUSSION

Jag har nu inhämtat kunskap om de vanligast förekommande sorteringsalgoritmerna och hur de fungerar, framför allt ifrån webbkursen ”Master the Coding Interview: Data Structures + Algorithms” av instruktören Andrei Neagoie. Ifrån denna källa har jag sen implementerat 5 olika sorteringsalgoritmer i Java som jag sedan har genomfört en serie av tester på. Vi delar in sorteringsalgoritmerna i två kategorier:

De elementära sorteringsalgoritmerna: Bubblesort, Selectionsort, Insertionsort
De sofistikerade/avancerade sorteringsalgoritmerna: Mergesort, Quicksort

Innan resultatet av testerna framkommit så var det en förväntan av resultatet baserat på de material jag stött på över nätet och ifrån webbkursen. Jag har i resultatanalysen förmedlat dessa uppfattningar och framför allt vad som ligger i linje med förväntningarna och vad som sticker ut.

För samtliga algoritmer så blev det första testet i serien mycket långsammare än de efterföljande 9. Detta beror på att JVM (Java Virtual Machine) läser in och cachar klassfiler och som gör att overheaden är högre på första iterationen, men lägre på efterföljande iterationer. Jag beslutade att låta teststrategin ligga kvar då det upplevdes som ovetenskapligt och ohederligt att ändra strategin allt eftersom. Dessutom, om alla algoritmer har samma förutsättningar så kan ändå värdefull kunskap komma ifrån testresultatet.

Resultatet i första testserien Small Array så sticker ingen algoritm ut, det är en liten lista på 10 siffor där skalbarhet inte är en faktor. De avancerade algoritmerna presterade jämförbart med de elementära, vilket gör det svårt att utse en tydlig vinnare.

För Medium Array som är listan med 1000 siffror så förlorar Bubblesort mer markant mot vinnaren Mergesort. Vilket var väntat då det är en enkel algoritm med tidskomplexitet på O(N^2) mot Mergesorts O(N Log N). Det anmärkningsvärda var att Quicksort som har samma tidskomplexitet som Mergesort, får ett mycket lägre medelvärde. Det kan finnas flera förklaringar till detta och det mest troliga är att Quicksort inte cachar resultatet när den får samma lista att sortera i nästa iteration

En annan förklaring är att pivot-elementet inte väljs ut på ett sätt som minskar antalet stack frames i ett acceptabelt intervall. Denna förklaring lämpar sig bättre på varför Quicksort förlorar även i Large Array scenariot.

I det testet så sorteras 100 000 siffror och där kunde Quicksort inte ens fullfölja de 10 iterationerna av testet utan att krascha av en ”Stack Overflow-Error”. Jag försökte att hantera detta genom att sätta en mycket högre Stack Size men trots det så gick det inte att slutföra, och Quicksort får därmed uteslutas som alternativ i detta specifika användningsscenario.

I den sista, Almost Sorted-listan, så blir vinnaren i linje med förväntan. Det vill säga Insertionsort som trots sin tillhörighet till de elementära sorteringsalgoritmerna, presterar på topp i listor som är nästan helt sorterade.

Efter att ha gjort mina undersökningar och sedan implementerat och testat algoritmerna har jag några korta slutsatser. Bubblesort och Selectionsort är mycket bra ställen att börja på, och de fungerar förhållandevis bra på små datamängder. De skalar dock mycket dåligt och blir snabbt mycket sämre på större datamängder än andra algoritmer. De har heller inte någon fördel på Almost sorted-listor. Med en liten optimering så kan man göra en early termination (tidigt avslut) på gränsfall där listor är helt sorterade redan.

Insertionsort är en algoritm där det börjar bli intressant, den är enkel att förstå och implementera, den är en del av de tre elementära sorteringsalgoritmerna. Men den har en vederhäftig nytta då den faktiskt över presterar på Almost-sorted listor och fungerar bra på små listor. Något överraskande var också att den presterade så pass väl på Large Array i mina tester, vilket breddar dess lämplighet över ett större spann avseende datamängden.

Mergesort var vinnare i 3 av 4 av mina tester, endast trumfad av Insertionsort på Almost Sorted-listan. I synnerhet var det i de större listorna som Mergesort visade sig vara överlägsen, där skalbarhet blir allt viktigare. Till skillnad från Quicksort så levde Mergesort upp till förväntningarna. Quicksort hade i min implementation problem med att exekvera på Medium Array och Large Array. Misstankar fanns kring att Quicksort inte kunde dra nytta av caching på samma sätt som de andra sorteringsalgoritmerna. På Large Array så blev problemen så allvarliga att Quicksort inte ens kunde slutföra utan kraschade på grund av Stack Overflow.

Sammanfattningsvis så har jag genom detta arbete fått upp ögonen för de svåra avvägningar som en utvecklare måste ta. I mitt fall och i detta arbete så lät Quicksort som ”den bästa algoritmen” inledningsvis då den hade både lägre minneskomplexitet O(Log N) än Mergesort men bättre tidskomplexitet O(N Log N) än de elementära algoritmerna. Efter testresultaten så framstod Mergesort som en bättre algoritm, speciellt på de större datamängderna, den är också stabil till skillnad från Quicksort som inte är det. (Stabil algoritm, som i att

sorteringsalgoritmen bevarar ordningen på lika element)

Liknande avvägningar som jag behövt göra ställs utvecklare på Google inför dagligen, som fram tills ganska nyligen uppdaterat sin sorteringsalgoritm i Google Chrome till en hybridalgoritm som är stabil.

Webbkurs, Andrei Neagoie, Inhämtat: 2023 januari
https://www.udemy.com/course/master-the-coding-interview-data-structures-algorithms/

Diagram över algoritmers Time complexity och Space complexity, Inhämtat: 2023 januari
https://www.bigocheatsheet.com/

BILAGOR

Excelblad med all insamlad Data: Github.com/j-development/Sorting Algorithms.xlsx

Repository till sorteringsalgoritmerna: Github.com/j-development/