En jämförelse mellan 8 välkända sorteringsalgoritmer

- Tillämpade på listor med nummervärden

Kurs: Examensarbete
Klass: .Net Syss3
Termin och år: 2021 4e termin
Författare: Fredrik Molén,
Denizhan Örgün,
Elchin Jabbari,
Meles Desbele,
Firas Hanna
Kursansvarig lärare: George Hanna

Sammanfattning

Studiens syfte var att undersöka och jämföra 8 välkända sorteringsalgoritmer. Det vi ville undersöka var att hitta de bästa och mest effektiva sorteringsalgoritmer som är viktiga i en programmerares värld. Samt att hitta de olika styrkorna och svagheterna med dessa olika sorteringsalgoritmer och vart de passar in bäst. De sorterings algoritmer vi valde var, Bucket sort, Insertion sort, Merge sort, Quick sort, Timsort , Bubble sort, Heap sort och slutligen Tree sort.

Metoden som användes var kvalitativ och materialet inhämtas genom intervjuer, artiklar på nätet och ett program som skrevs i C# som heter BenchmarkDot. Resultatet analyseras med hjälp av tidigare forskning och program.

Resultaten visade att de snabbaste av sorterings algoritmer var Timsort, Quicksort, Merge Sort och Bucketsort. Samt så visade det sig att Bucketsort var snabbast när det gällde hantering av större mängder av data och Timsort var snabbast vi hantering av mindre mänger av data. Med de resultaten vi har fått så kan man anta att de sämre presterande algoritmerna är oanvändbara fast att alla sorterings algoritmer har sina egna fördelar och nackdelar vilket gör det väldigt svårt att få obsoleta sorterings algoritmer.

Vårt benchmark-resultat visade sig att algoritmerna är väldigt nära varandra i tid vid mindre datamängder med Bubblesort som stack ut mest. Vid större datamängder så visade det sig att Bubblesort sjönk med 133.8 ms där vi jämförde med Bucketsort som hade hastigheten av 5 promille av tiden. Den näst segaste sorteringsalgoritmen vid större datamängder var Insertionsort som var helt oanvändbar vid större datamängder på 10,000.

Insertionsort var också väldigt snabb vid mindre mängder av data hantering fast blev exponentiellt segare desto mer data den behöver hantera. Mergesort visade sig vara ganska nära hastigheterna av de snabbare sorteringsalgoritmerna.

Innehållsförteckning

Historien

Själva ordet "algoritm" kommer från namnet på forskaren Abu Abdullah Muhammad ibn Musa al-Khwarizmi. Omkring 825 skrev han en uppsats där han först beskrev det decimaltalssystem som uppfanns i Indien. Tyvärr har bokens arabiska original inte överlevt. Al-Khwarizmi formulerade beräkningsreglerna i det nya systemet och var förmodligen den första som använde siffran 0 för att beteckna en saknad position i beteckningen av ett nummer (dess indiska namn översattes av araberna som as-sifr eller helt enkelt sifr, därav orden "nummer" och "chiffer"). Omkring samma tid började andra arabiska forskare använda indiska nummer. Under första hälften av 1100-talet trängde al-Khwarizmi-boken i latinsk översättning in i Europa. Översättaren, vars namn inte har kommit till oss, gav det namnet Algoritmi. På arabiska kallades boken Kitab al-jabr wal-muqabala ("The Book of Addition and Subtraction"). Från bokens ursprungliga titel kommer ordet Algebra.
Således ser vi att det latinska namnet på den centralasiatiska forskaren ingick i bokens titel, och idag tvivlar ingen på att ordet "algoritm" kom in på europeiska språk just på grund av detta arbete. Frågan om dess betydelse har dock orsakat hård kontrovers under lång tid. Under århundradena har ordets ursprung fått en mängd förklaringar.

Ovan nämnda översättning av al-Khwarizmi verk blev den första svalan, och under de närmaste århundradena verkade många andra verk ägnas åt samma fråga - lärde konsten att räkna med hjälp av siffror. De hade alla ordet algoritmi eller algorismi i sina namn.

1 Uppdragsbeskrivning/problemformulering

Vår grupp har valt att studera och jämföra 8 antal sorteringsalgoritmer. Vi ska göra en kort beskrivning och lite historia bakom varje del och sedan jämföra alla dessa med varandra. Jämförelsen ska göras med ett verktyg som heter BenchmarkDot där all kod är skriven i C#.

Anledningen till att vi har valt det här området är för vi är alla är intresserade av algoritmer samt så känner vi att det är ett ämne som är värt att undersöka och den kunskapen som det ger en är väldigt bra att ha i arbetslivet som en utvecklare. Problemet vi valde var att ha olika stora listor med siffervärden där vi ska använda oss av olika sorteringsalgoritmer för att få fram den mest effektiva sorteringsalgoritmen beroende på situation och storlek etc på listorna.

Vi valde att avgränsa oss till sorterings-algoritmerna (insertion sort, selection sort, merge sort, bubble sort, quick sort, bucket sort, tree sort och timsort). Vi gjorde marknadsundersökningar, intervjuer och research på internet för att få fram de mest använda och vanliga sorteringsalgoritmer.

Vi kommer även hålla vårt scope inom entrådiga sorteringsalgoritmer. Flertrådiga implementationer är ett intressant område där man kan göra väldigt stora tidsvinster, men vi ville hålla oss inom en rimlig avgränsning.

2 Examensarbetets mål

Enligt Wikipedia-webbplatsen om sorterade algoritmer finns det över 30 algoritmer. Vi har valt de mest populära och använda algoritmerna från listan över jämförande sorter och icke-jämförande sorter.

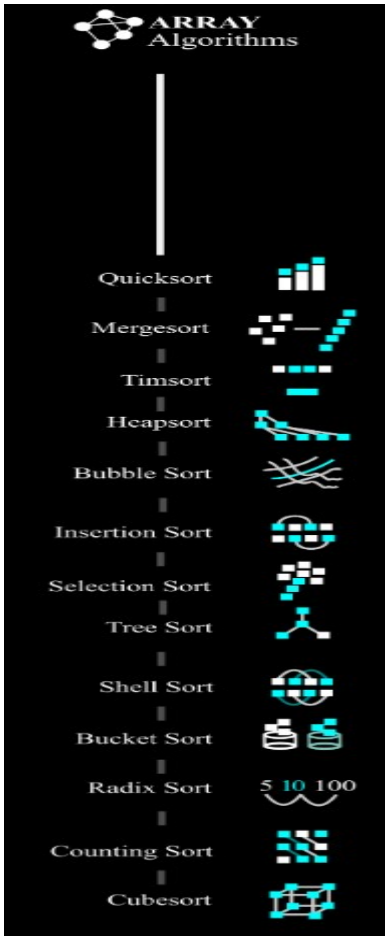
(Sorting Algorithm, Wikipedia, 21 maj 2021)

Valda algoritmer visas inte bara på Wikipedia-webbplatsen, utan visas också i många läroböcker och olika webbplatser om sorteringsalgoritmer.

Syftet med denna uppsats är att hjälpa läsaren att förstå rätt val av algoritmer. Vilka är de snabbaste, lättaste, användbara eller mindre i storlek än andra. Algoritmer är början för att lösa matematiska problem. Inom IT kan algoritmer ses som själva grunden för all inlärningsmaskinkod, datastruktur, skriva olika program samt att lösa de minsta data problemen.

Vår grupparbetets mål är att utforska och visa resultat på hur dom olika valda algoritmer implementeras och att ha en övergripande förståelse om deras tid och storleks komplexitet. Eftersom vi tror att god hastighet, bra minnesanvändning och flexibilitet i algoritmen är mycket viktigt för att välja en specifik algoritm metod för ett visst syfte. Så, uppsatsen fokuserar främst på implementering, hastighets komplexitet och storleks-komplexitet av 8 olika populära algoritmer. Vi vill även ha en klar bild över vilka fördelar och nackdelar de har, hur de skiljer sig från varandra under körtiden och hur deras effektivitet ser ut med liten och stor data storlek.

Vi vill gå in på djup över varje sorteringsalgoritm och beskriva deras egenskaper och användningsområden och sedan testa dem med vårt benchmark program för att få ett konkret resultat att jämföra med det teoretiska. Med andra ord, vilken sorteringsalgoritm som är bra och vilka som är mest användbara



Figur 1. En lista vi bl.a för att välja sorteringsalgoritm. Är från en väldigt känd sida för sorteringsalgoritmer (Cheat Sheet Poster, www.bigocheatsheet.com)

3 Nulägesbeskrivning

Idag finns det en god förståelse bland annat mellan utvecklare att algoritmer är osynliga för allmänheten och deras inflytande kommer att öka i framtiden. Eftersom i dagens teknologi algoritmer syftar på att optimera nästan allt. Internat, online-sökning, online-dejting och andra webbplatser skulle inte fungera utan algoritmer. Användningen av algoritmer i dagens banksystem är också viktigt och det hade varit nästan omöjligt att köra banksystemen utan algoritmer på grund av datamängden som banker använder. Det finns ju olika algoritm metoder som man använder för att bryta ner problem eller task. Men idag finns det 8 olika algoritmer som är jättepopulära och användbara. Dessa är Timsort, Merge sort, Quicksort, Insertionsort, Bucketsort, Treesort, Heapsort och Bubblesort.

4. Metodbeskrivning

Först och främst behövde vi undersöka vilka algoritmer som är populärast och använda. Vår metod var att söka på internet och samla in data på popularitet inom området. Därefter har vi gjort en marknadsundersökning där vi har ställt ett visst antal frågor berörande detta. Sist har vi också gjort intervjuer med lite mer ingående frågeställningar.

Sedan har vi behövt samla in information om dessa algoritmer, och där har vi mestadels hållit oss till internet och lite böcker som vi använt för att få lite mer ingående information.

Det sista vi har gjort är att använda oss av ett benchmark-bibliotek och sedan skrivit test-koden (https://github.com/mol1987/algorithm-benchmarks) i C# där vi sedan kör det här benchmark-verktyget som genererar en hop av värdefull data som vi sedan kan använda för att jämföra och få väldigt intressant information.

5. Resultatredovisning

1. Bucket sort
2. Insertion sort
3. Merge sort
4. Quick sort
5. Timsort
6. Bubble sort
7. Heap sort
8. Tree sort

5.1 Bucket Sort Algoritm

Sorteringstyp: Distribution Sort

Storlekskomplexitet: O(n+k)

In place: Nej

Stabilitet: Stabil

Adaptivitet: Adaptiv

Beskrivning

För Bucket Sort måste du dela in elementen i ingångsdata-arrayen i k-block (korgar). Vidare sorteras var och en av dessa block antingen genom en annan sort eller rekursiv med samma delnings metod. Efter sorteringen i varje block skrivs datan till matrisen i ordningen för uppdelningen i block. Man bör komma ihåg att denna sortering endast fungerar om uppdelningen i block utförs på ett sådant sätt att elementen i varje nästa block är större än det föregående.

Bucket Sort bryts ned mycket när det inte finns många lika elementer (de flesta elementer hamnar i en korg). Därför bör denna typ av sortering användas när det är hög sannolikhet att siffrorna sällan upprepas (till exempel en sekvens av slumpmässiga nummer).

Algoritmen

Om ingångselementen följer en enhetlig fördelningslag är den matematiska förväntningen på tiden för bucketsorteringsalgoritmen linjär. Detta är möjligt på grund av vissa antaganden om indata. Pocket-sortering förutsätter att ingångsdata fördelas jämnt över segmentet [0, 1).

Idén med algoritmen är att dela upp segmentet [0, 1] i N lika stora segment (bucket) och dela N ingångsvärden i dessa fickor. Eftersom ingångs numren är jämnt fördelade antas det att ett litet antal siffror faller i varje ficka. Sedan sorteras siffrorna i fickorna sekventiellt. En sorterad matris erhålls genom att sekventiellt lista elementen i varje ficka.

Hur fungerar Bucket Sort?

Vi sprider siffrorna i korgarna, sedan sprider vi dem i mindre korgar i varje korg, och så vidare tills det bara finns identiska element på någon nivå i korgen. Sedan från sådana korgar av lägsta nivå är det lätt att återställa matrisen i ett ordnat tillstånd.

Låt oss förklara med ett specifikt exempel. Låt oss säga att vi har en oordnad matris. Vi har till exempel en matris som innehåller siffror från 1 till 8.

Vi delar upp dessa siffror i två grupper: siffror från 1 till 4 faller i en grupp och siffrorna från 5 till 8 faller i den andra. Sedan fördelar vi siffrorna i den första korgen i två korgar: i siffrorna 1 och 2, och i de andra 3 och 4. Vi distribuerar också dessa korgar mellan korgarna där det redan finns siffror av samma storlek. Vi tillämpar en liknande rekursion på den stora korgen som innehåller siffror från 5 till 8.

Efter det från små korgar, som var och en innehåller samma nummer, returnerar vi elementen till huvudmatrisen i prioritetsordning.

Sortering av hinkar i denna form är inte särskilt användbar i praktiken, men den visar som referens hur all distribution sorterar i allmänhet.

Figur 2. Visuell illustration hur en bucket sort fungerar (Bucket sort, Programiz.com)

Fördelar:

Tillhör de snabba klassen av algoritmer med linjär tidsserie O(n).

Nackdelar:

Det bryts ned mycket med ett stort antal få utmärkta element eller på den misslyckade funktionen att få korg numret efter elementets innehåll.

Komplexitet

Värsta fallkomplexitet: O (n²)

Bucket Sort är användbart när data fördelas jämnt i den.

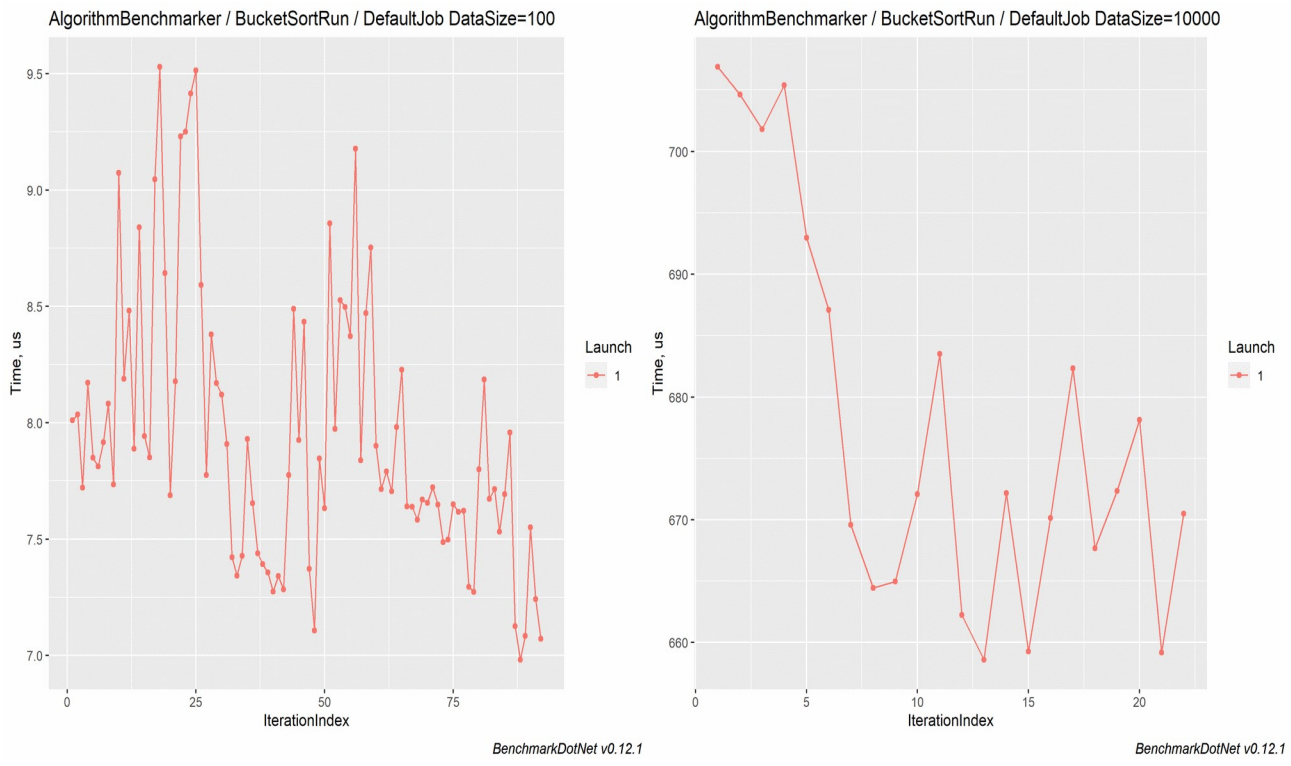
Om ingångsdata har nycklar som liknar eller är nära varandra flyttas de till en korg. Det visar sig att vissa korgar kommer att ha mer data än andra korgar. Det värsta fallet inträffar när alla eller nästan alla elementer flyttas i en korg.

Bästa fallkomplexitet: O (n + k)

Det bästa fallet är när artiklarna är i rätt ordning från början. Eller om artiklarna fördelas jämnt i korgar, det vill säga varje korg har samma antal artiklar. Bästa fallets komplexitet O(n + k).

Genomsnittlig fallkomplexitet: O (n + k)

Genomsnittlig fallkomplexitet uppstår när elementer distribueras slumpade över alla korgar. I detta fall är den genomsnittliga sorteringstiden lika med O (n+k)



Figur 3, visar tiderna på varje enskild körning av Bucketsort-algoritmen producerad av vårt test-program. Y-axeln är tiden i mikrosekunder. X-axeln är vilket nummer av körning. Vänstra bilden är vid en datamängd på 100 och högra är vid en datamängd på 10,000.

5.2 Insertion Sort

Sorteringstyp: Insertion Sort

Storlekskomplexitet: O(1)

In place: Ja

Stabilitet: Ostabil

Adaptivitet: Adaptiv

Beskrivning

Insertion sort är en sorteringsalgoritm baserad på platsjämförelse. Insertion Sort används ofta i datavetenskapliga studier och är vanligtvis en av de första sorteringsalgoritmerna som introducerades för studenterna. Det är intuitivt och enkelt att implementera, men den är väldigt långsam med stora matriser och används därför nästan aldrig för att sortera riktig data.

Insertion Sort är en stabil och på plats algoritm som bara fungerar riktigt bra för nästan sorterade eller vid mindre matriser.

När du sorterar efter insatser behöver du inte ha hela matrisen i förväg innan du sorterar. Algoritmen kan ta emot ett element i taget under sorterings processen. Detta är väldigt bekvämt om vi behöver lägga till fler element under sorteringen, algoritmen infogar ett nytt element av elementet på rätt plats utan att utföra sorteringen av hela arrayen.

Hur fungerar Insertion Sort?

Matrisen är uppdelad i en "sorterad" undergrupp och en "osorterad" undergrupp. I början innehåller den sorterade subarrayen bara det första elementet i vår ursprungliga array.

Det första elementet i den osorterade matrisen utvärderas så att vi kan infoga den på sin plats i den sorterade undergruppen.
Insättning utförs genom att flytta alla element som är större än det nya elementet en position åt höger.
Detta fortsätter tills hela vårt sortiment är sorterat.
Tänk dock på att när vi säger att ett element är större eller mindre än ett annat element betyder det inte nödvändigtvis större eller mindre heltal.

Vi kan definiera orden "mer" och "mindre" som vi vill när vi använder anpassade objekt. Exempelvis kan punkt A vara "större" än punkt B om den ligger längre bort från koordinatsystemets centrum.

Vi markerar den sorterade undermatrisen med djärva siffror och använder följande array för att illustrera algoritmen:

8, 5, 3, 11, 9

Det första steget skulle vara att lägga till 8 i vårt sorterade underarray.

***8**, 5, 3, 11, 9*

Låt oss nu titta på det första osorterade objektet - 5. Vi lagrar det här värdet i en separat variabel, till exempel aktuell, för skydd. 5 är mindre än 8. Vi flyttar 8 ett ställe till höger och skriver effektivt över 5 som tidigare lagrats där:

***8**, 3, 11, 9 (nuvarande = 5)*

5 är det minsta av alla element i vår sorterade subarray, så vi sätter in den i första position:

***8**, **5**, 3, 11, 9*

Därefter tittar vi på nummer 3. Vi lagrar detta värde nuvarande. 3 är mindre än 8, så vi flyttar 8 till höger och gör detsamma med 5.

***5**, **8**, 11, 9 (nuvarande = 3)*

Återigen stötte vi på ett element som var mindre än hela vårt sorterade underarray, så vi satte det i första position:

***3**, **5**, **8**, 11, 9*

11 är större än vårt högsta element i den sorterade undergruppen och därför större än något av elementen till vänster om 8. Så vi går bara vidare till nästa element:

***3**, **5**, **8**, **11**, 9*

9 är mindre än 11, så vi flyttar 11 till höger:

3, 5, 8, 11 (nuvarande = 9)

Men 9 är större än 8, så vi sätter bara in 9 direkt efter 8.

3, 5, 8, 11, 9

Fördelar:

Insert Sort är en online-algoritm som fungerar mycket bra med nästan sorterad data. Den är på plats och stabil. Den fungerar bra med andra sorteringsalgoritmer för att förbättra deras prestanda när de hanterar data med låg räckvidd.

Nackdelar: Tidskomplexitet för sorteringen är O (n ^ 2).

Komplexitet

Värsta fallkomplexitet: O (n^2)

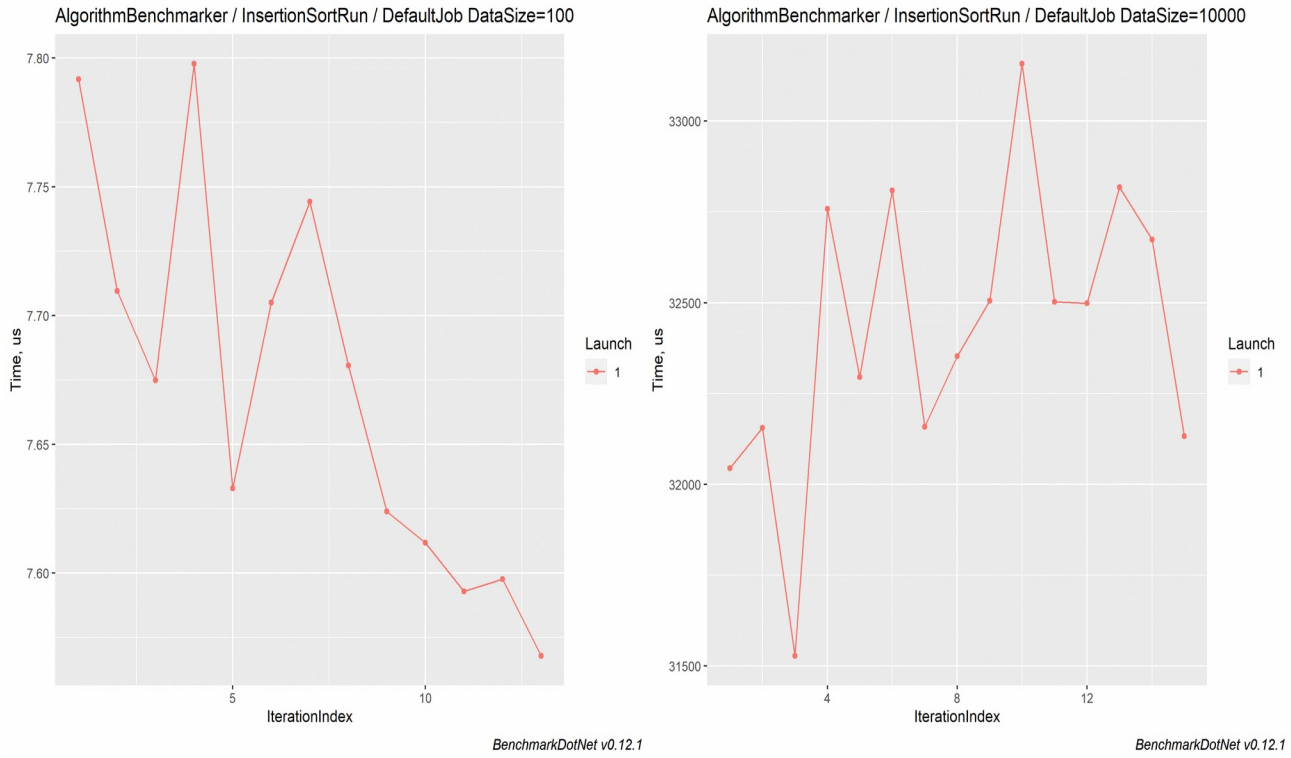
I värsta fall så är listan som skall sorteras en omvänd sorterad lista. Eftersom sista elementet behövs bytas *n-1* gånger och sedan det näst sista elementet behövs bytas minst *n-2* gånger.

Bästa fallkomplexitet: O (n)

I bästa fall när det är en redan sorterad lista så behöver algoritmen bara traversera *n* gånger och 0 utbyten av element.

Genomsnittlig fallkomplexitet: O (n^2)

Genomsnittlig tid är precis som värsta fall.



Figur 4, visar tiderna på varje enskild körning av Insertionsort-algoritmen producerad av vårt test-program. Y-axeln är tiden i mikrosekunder. X-axeln är vilket nummer av körning. Vänstra bilden är vid en datamängd på 100 och högra är vid en datamängd på 10,000.

5.3 Merge Sort Algorithm

Sorteringstyp: Merge sort

Storlekskomplexitet: O(n)

In place: Nej

Stabilitet: Stabil

Adaptivitet: Icke adaptiv

Beskrivning

Merge sort är ett bra exempel på hur man kan lösa stora problem genom att dela upp det i deluppgifter. Algoritmen i sig är mycket effektiv och samtidigt mycket lätt att förstå.

Stegen för algoritmen är som följer:

1) Dela upp en grupp i små bitar av samma storlek. Arrayen delas rekursivt tills vi har separata element.

2) Slå ihop delar av de "intilliggande matriserna" i en med sortering. Återigen är sortering ganska intressant: eftersom vi slår ihop redan sorterade matriser vill vi få en sorterad matris som ett resultat.

Hur fungerar Merge Sort?

Så, låt oss säga att vi har två matriser:

1. Pekare pekar på det första elementet i båda matriserna. Den minsta av dem väljs
2. I matrisen med det minsta numret flyttas pekaren till nästa element. Vi upprepar punkt 1.
3. Om en av matriserna har slut på element överför vi elementen i en annan array till vår sorterade array (dränerar resten)
4. Upprepa tills undergrupperna tar slut.

Så vi börjar med att dela upp arrayen i två identiska delar och det finns två till, och så vidare, tills mini-arrays av 1 element finns kvar från arrayen.

Först delas hela matrisen upp tills det inte blir någon delning:

[17, 46, 4, 21, 11, 7]

[17, 46, 4] [21, 11, 7]

[17, 46] [4, 21] [11, 7]

[17] [46] [4] [21] [11] [7]

Sedan slås de mindre listorna ihop i en ny lista i sorterad ordning.

[17] [46] [4] [21] [11] [7]

[17, 46] [4, 21] [7, 11]

[4, 17, 46] [7, 11, 21]

[4, 7, 11, 17, 21, 46]

Fördelar:

Den är snabbare för stora listor, då den inte går igenom hela listan flera gånger.

Körtiden är jämn under hela loppet, kan utföra olika delar i med likadana tider på motsvarande steg.

Nackdelar:

Långsammare i småa arbeten i jämförelsen med andra algoritmer.

Behöver lite mer utrymme för att spara i en underlista.

Komplexitet

Värsta fallkomplexitet: O (n log(n))

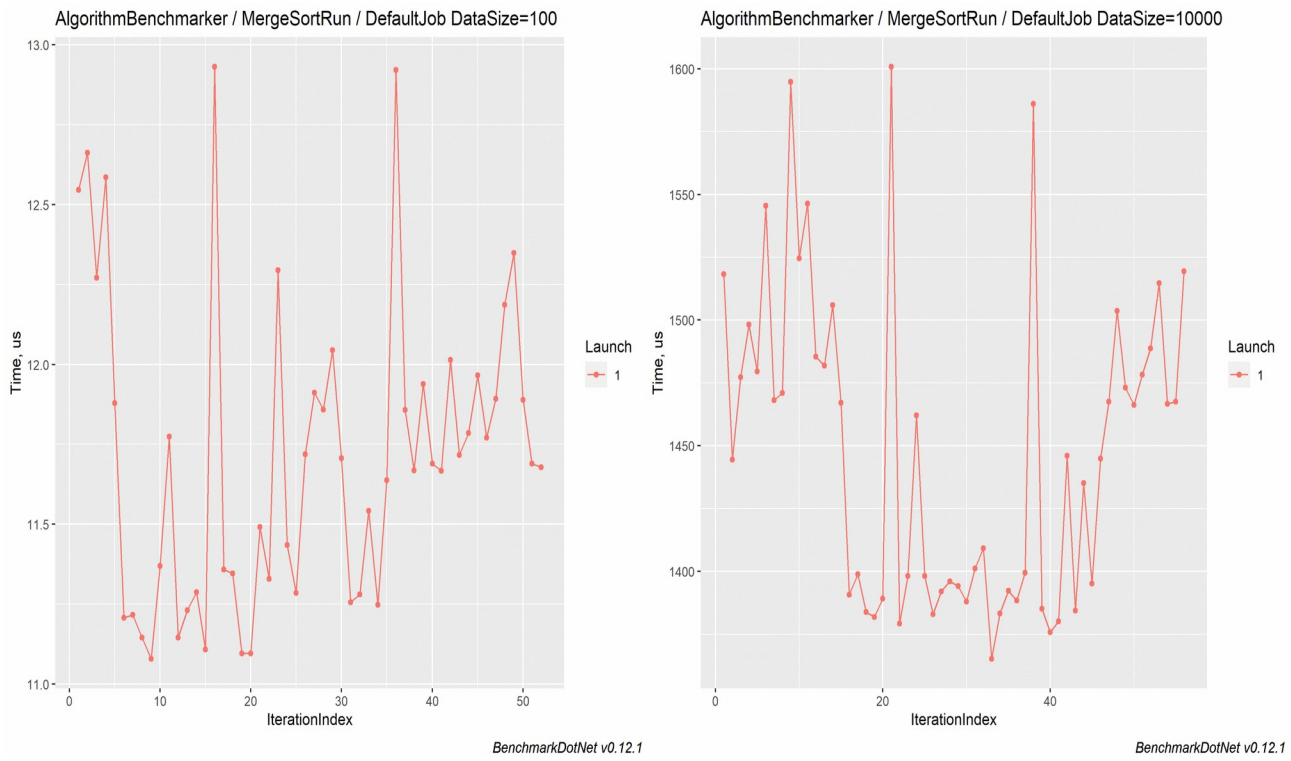
När algoritmen väl slår ihop de delade listorna så jämför de bådas första element och den lägsta läggs till output-listan. Då är det värsta fallet när varje ihopslagning av listorna att ett värde är kvar i den motsatta listan. Vilket då betyder att ingen jämförelse var undviken. Det här händer när två av det högsta värdena är i varje motsatt lista. Merge Sort behöver då kontinuerligt jämföra värden i varje motsatt lista tills de största värdena är jämförda.

Bästa fallkomplexitet: O (n log(n))

Om varje underlistas högsta värde är större än ett första elementet i en motsatt underlista så är det bästa fall.

Genomsnittlig fallkomplexitet: O (n log(n))

En randomiserad lista.



Figur 5, visar tiderna på varje enskild körning av Mergesort-algoritmen producerad av vårt test-program. Y-axeln är tiden i mikrosekunder. X-axeln är vilket nummer av körning. Vänstra bilden är vid en datamängd på 100 och högra är vid en datamängd på 10,000.

5.4 Quicksort Algoritm

Sorteringstyp: Exchange sort

Storlekskomplexitet: O(1)

In place:Ja

Stabilitet: Ostabil

Adaptivitet: Icke adaptiv

Beskrivning

Quick sort är en förbättrad sorteringsmetod baserad på utbytesprincipen. Quick sort är den mest effektiva av alla algoritmer för direkt sortering av stora mängder data. Den förbättrade algoritmen räknas dock som den mest kända metoden för att sortera matriser. Den har så lysande egenskaper att dess uppfinnare Tony Hoare döpte den till Quick sort.

Hur fungerar det?

För att uppnå störst effektivitet så börjar Quick sort med att välja ett slumpat värde från matrisen till ett så kallad (PIVOT) värde, detta värde används för att jämföra matrisens element.
Matrisens element delas upp efter jämföranden med PIVOT till två olika högar, högre än PIVOT och lägre än PIVOT, när detta steg är klart så ligger PIVOT värden i sin rätta plats i matrisen.
Nästa steg blir då att välja ett nytt PIVOT värde på en av dom osorterade högarna, därefter upprepas föregående steg om igen tills att PIVOT är på sin rätta plats, vilket resulterar till att den nya PIVOT också ligger på rätt plats i matrisen (som kan vara högre eller mindre än den föregående PIVOT).
Detta två nämnande steg upprepas tills hela matrisen är delad till mindre delar, där alla PIVOT värden är på sin rätta plats, vilket betyder att hela matrisen nu är sorterat.
Bild 1 Quicksort

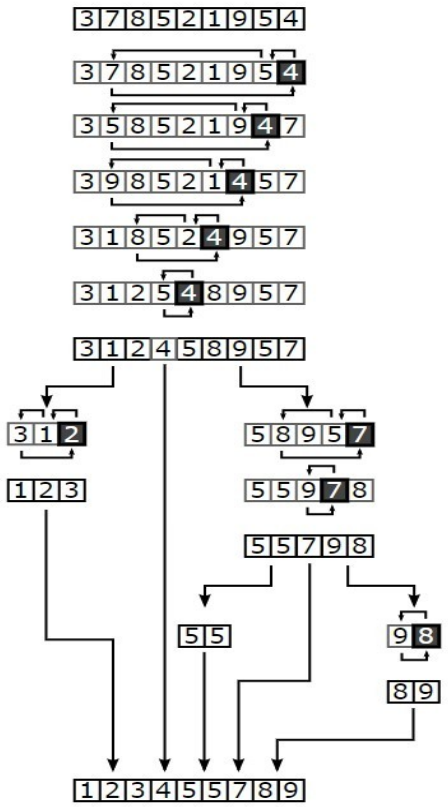


Bild källa: (2016), Algorithm, Tillgängligt: https://en.wikipedia.org/wiki/Quicksort

Figur 6, exemple av hur quicksort funkar

Fördelar:

Det används både för små och stora datamängder.

Quick sort använder inte mycket cache minne, då den har en referensplats när den används, också har den kort inne-loop.

Nackdelar:

Värsta fall komplexitet är samma som bubble sort och selection sort.
svårt att fixa vid implementations fel vilket kan resultera till att den presterar dåligt.

Komplexitet

Värsta fallkomplexitet: O (n^2)

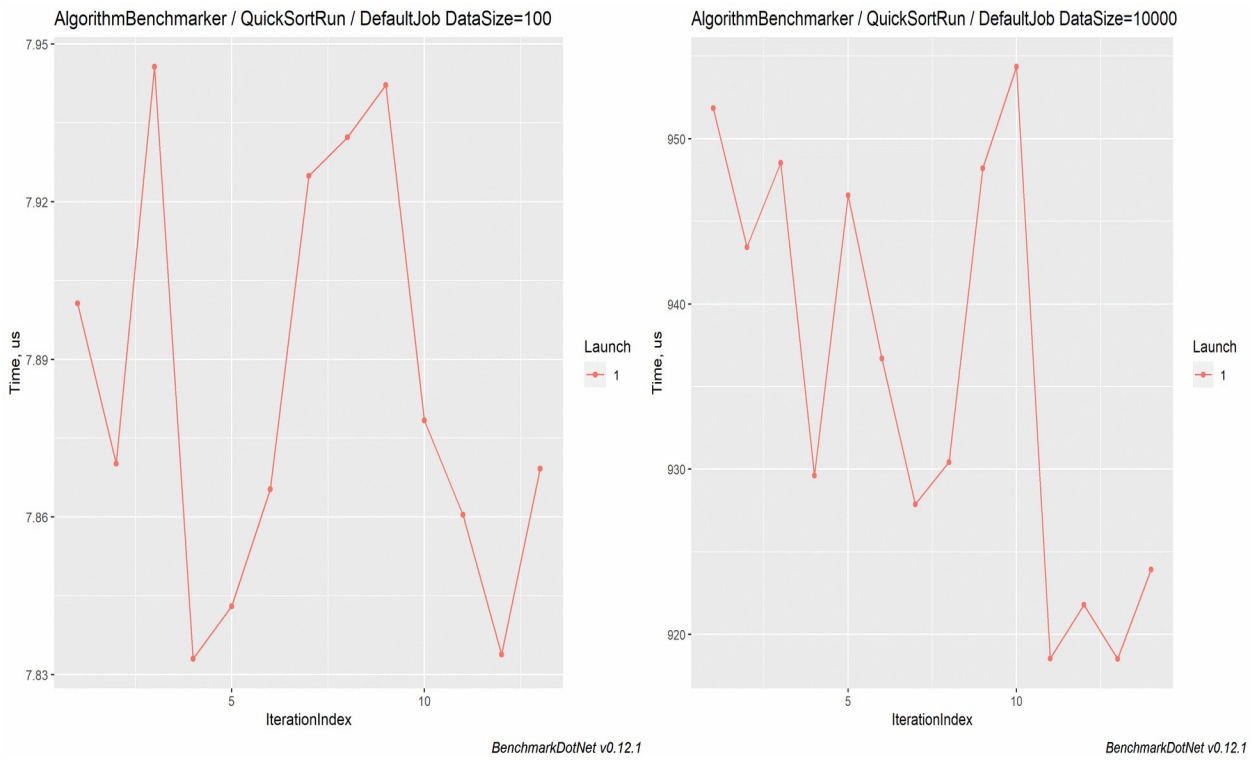
När det lägsta talet eller högsta talet är valt som pivot-värde så blir det bara en underlista och då kan ej en uppdelning av värden på pivot-värdet hända. Detta värsta fall kan undvikas lätt genom att välja median-värdet som pivot-värde.

Bästa fallkomplexitet: O (n log(n))

Bästa fall inträffar när varje pivot-steg väljer medianen i en lista.

Genomsnittlig fallkomplexitet: O (n log(n))

Precis som bästa fall.



Figur 7, visar tiderna på varje enskild körning av Quicksort-algoritmen producerad av vårt test-program. Y-axeln är tiden i mikrosekunder. X-axeln är vilket nummer av körning. Vänstra bilden är vid en datamängd på 100 och högra är vid en datamängd på 10,000.

5.5 Timsort algoritm

Storlekskomplexitet: O(n)

In place: Kan vara

Stabilitet: Stabil

Adaptivitet: Adaptiv

Beskrivning

Timsort introducerades för första gången i 2002 av Tim Peters och är tillämpligt för praktiska användning på verkliga data (Bauermeister, 2019).

Timsort implementeras med kombinationen av Merge sort och Insertion sort tillsammans med viss intern logik för att optimera manipuleringen av stor datastorlek (Brandon,2018) och(Bauermeister, 2019).

Implementering av Timsort sker i tre olika steg (Bauermeister, 2019). Dessa är binära sök, Insertion och merge sort. Hur hjälper binära sök för att implementera Timsort?

Binär sökningar används för att implementera Insertion sort. Insertion sort löper genom hela arrayens lista och kollar om det finns ett objekt eller element som inte är i ordning och flyttar den till rätt position. Insertion sort som används i Tim sort är jättesnabb för en array lista som är redan är sorterad eller för en mindre array lista. Till exempel om array listan som ska sorteras är mindre än 64 data storlek, så använder Timsort Insertion sort för att implementera sorteringen. I detta fall har Timsort en snabb körtid komplexitet som är formulerat i O(n). O (n) kallas ”Big O notation”. Om array listan som ska sorteras innehåller mer än 64 data storlek så kommer Timsort implementeras med kombination av båda Insertion sort och Merge sort. Detta implementeras genom att dela upp arrayen i block som heter ”run” och sedan sorterar man dessa körningar med Insertion sort en efter en och slår ihop dessa körningar med hjälp av en kombinerings funktion som används i Merge sort (Bauermeister, 2019). Timsort är jätte starkt , stabil , adaptiv och en väldigt effektiv algoritm. Oftast används Timsort i Python.

Hur den funkar

Ett exempel hur Timsort sorterar för data storlek som är mindre än 64 visas i tabell 1. Vad den gör är att den letar efter i hela arrayen för elementen som är inte är placerade i rätt ordning och flyttar de sedan till höger. Till exempel 34 i tabell 1 är placerat fel . Så 34 flyttar till höger efter 10

[34, 10, 64, 51, 32, 21]	No. shifted to right
[34, 10, 64, 51, 32, 21]	Nothing
[10, 34, 64, 51, 32, 21]	**34**
[10, 34, 64, 51, 32, 21]	Nothing
[10, 34, 51, 64, 32, 21]	**64**
[10, 32, 34, 51, 64, 21]	**34, 51, 64**
[10, 21, 32, 34, 51, 64]	**32, 34, 51, 64**

Tabell 1. Sortering av Timsort för mindre än 64 array storlek (brandon,2018)

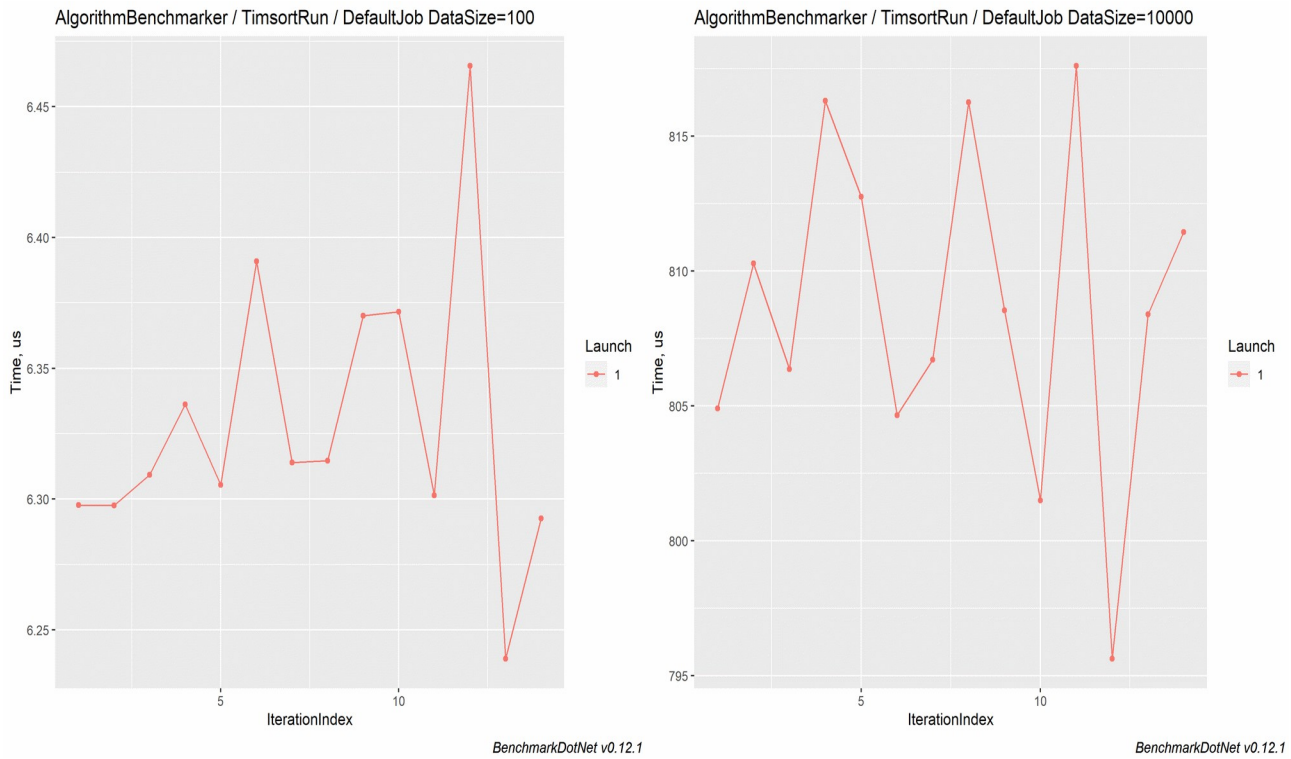
Precis som de andra algoritmer , Timsort har också tre olika tidkomplexiter, Dessa är bästa-fall komplexitet, västa-fall komplexitet och genomsnittlig-fall komplexitet.

En grej som Timsort gör bra är dess stabilitet och adaptivitet. Precis some i Merge sort och Quick sort, Timsort är också stabil och har O(nlogn) tid komplexitet i både västra och genomsnittligt fall komplexitet (Scheiwe, 2018). Men om det redan finns sorterade nummer i arrayen som ska sorteras , blir dress tidskomplexitet mindre än det värsta fallkompexitet (O(n).

I värsta fall komplexitet kräver Timsort tillfälligt lagringsutrymme (storage space) men i bästa fall komplexitet kräver det bara en liten konstant mängd utrymme. Vi vet att hastighet är den viktigaste kriterien man tar upp när man väljer en sorteringsmetod. Man vill ju veta att hur snabb den är och hur beroendet ändras på antalet element som ska sorteras. Ibland kan en sorterings algoritm vara snabb med mindre antal elementet och långsammare vid större data storlekar. Så, för att veta tid komplexiteten för Timsort, har vi genomfört ett test med ett verktyg som heter BenchmarkDot och vi har fått en medelhastighet får både 100 och 10,000 random data storlek.

Medelhastigheten som vi har fått från benchmark var 0.00044 millisekunder för 100 data storlek och 0.595 millisekunder för 10,000 data storlek. Vilket innebär att det var den lägsta medel hastigheten som vi har fått än de andra algoritm metoder. Så vi kan säga att Timsort är den snabbaste algoritmen med lite och mer data storlek av de andra populära algoritmer.

Fig 8 visar de maximala och minimum hastigheten av Timsort för båda 100 och 10000 data storlek.



Figur 8, visar tiderna på varje enskild körning av Timsort-algoritmen producerad av vårt test-program. Y-axeln är tiden i mikrosekunder. X-axeln är vilket nummer av körning. Vänstra bilden är vid en datamängd på 100 och högra är vid en datamängd på 10,000.

Fördelar och Nackdelar av Tim sort

Fördelar:

Några av Timsorts fördelar är att den är snabb och stabil. Timsort är också adaptiv vilket innebär att den har möjligheten att göra sorteringar på både små och stora arrayer (Valdarrama, 2012).

Nackdelar:

Timsort är komplex och kräver mycket kod på grund av att Timsort berör både Insertion och Merge sort. Så komplexitet är en grej som man kan räknar som en nackdel (Valdarrama, 2012).

Tid och Storlek komplexite av Timsort

Precis som de andra typ av algoritmer, Timsort också inträffar tre olika tid komplexite och storlek komplexitet.

Storlek komplexitet : O(n)

Timsort har storlek komplexitet som representerat med O(n). Detta kan betyder att Timsort kräver temporary lagringsutrymme under körtiden när input längden ökar. Så det betyder att komplexiteten är linjär.

Bästa -fall komplexitet: O(n)

Detta fall komplexitet inträffar om array listan som ska sorteras är redan sorterad. Så, komplexitet tid för Timsort blir det O(n). Detta betyder att körtiden (run time) ökar linjärt med input längden. Vilket innebar att hastigheten att sortera under bästa fall komplexitet är jättesnabb.

Värsta - fall komplexitet: O(nlog(n))

Eftersom Timsort är en hybrid algoritm, komplexitet tid som inträffar under en osorterad array lista är värsta fall komplexitet och värsta fall komplexitet för Timsort blir O(nlog(n). Detta betyder att körtiden (run time) ökar linjärt medan inputen ökar exponentiellt. Så, hastigheten att sortera blir inte så snabb som bästa fall komplexitet.

Genomsnittlig fall komplexitet: O(nlog(n))

Timsorts tid komplexitet under genomsnittlig fall komplexitet funkar precis som Värsta fall komplexitet. Så, komplexitet tid för genomsnittlig fall komplexitet är O(nlog(n)).

5.6 Bubble Sort Algorithm

Bubble sort är också en av de populäraste algoritm metoderna som jämför intilliggande elementerna från vänster till höger. Om elementerna sitter i fel ordning gör Bubble sort en jämförelse och flyttar elementen till rätt ordering. De högre elementen flyttar till höger och lägre elementen flyttar till vänster sida. Låt oss säga att vi har en array med [5,3,8,2] arraylist som ska soteras. Bubble sort tar de första två intilliggande siffror 5 och 3 och det gör en jämförelse mellan 5 och 3. 5 är högre än 3. Så, 5 ska flyttas till höger och 3 flyttar till vänster. Samma process fortsätter tills alla elementer sorterad i rätt ordning.

Exchange sort är en sorterings typ av bubble sort(Arora, Tamta, & Kumar, 2012). Eftersom bubble sort involverar det upprepade jämförelse och vid behov utbyte av intilliggande elementer. Detta är en typ av Exchange sort sorterings typ. Adaptivity, storlek komplexitet, sorterings typ och stabilitet är några av egenskaperna som bubble sort har. Bubble sort är en stabil och adaptiv algoritm metod.

Fördelar och Nackdelar av Bubble sort

Varje algoritm metoder har egna fördelar och nackdelar. Precis som de andra algoritm metoder, bubble sort har också både fördelar och nackdelar.

Fördelar:

Det är enkel att använda. Det kräver inte mycket kod och det är lätt att läsa. Bubble sort byter elementen för att sortera utan ytterligare tillfällig lagring vilket innebär att storleks komplexitet är minimal(dwivedi & C. Jain, 2014).

Nackdelar:

Bubble sort är extremt ineffektivt vid större antal siffror och detta är på grund av komplexitet tid. I värsta och genomsnittligt fall komplexitet kräver bubble sort O(n2). Så ineffektivitet med stor antal data storlek är en nackdel av bubble sort. Detta har redan prövats med 1 million data storlek med hjälp av benchmark verktyget. Det tar jättelång tid(ca 8-22 timmar) för att sortera 1 million random data storlek.

Tid och Storlek Komplexitet av Bubble sort

Bubble sort kan också inträffar tre olika komplexitet tid. Dessa är bästa, värsta och genomsnittligt fallkompexitet.

Storlek fallkomplexitet: O (1)

Precis som visar i tabell 2, storlek komplexitet av bubble sort är O(1) (Woltmann,2020). Detta innebar att storleket är konstant eller kräver inte extra utrymme under körtid när oavsett input längden.

Värsta fallkomplexitet: O (n^2)

Värsta fall komplexitet i bubble sort händer när de givna listor placeras helt motsatt ordning (Hinrichs, 2015). Låt oss anta att vi har n array storlek som ska sorteras och alla möjliga n - 1-passeringar av algoritmen kommer att utföras innan de är färdiga . I värsta fall jämför vi varje siffra med varannan siffra. Så, enligt (Woltmann, 2020) det värsta fall tidskomplexitet för bubble sort är O($\mathbf{n^2}$).

Bästa fallkomplexitet: O (n)

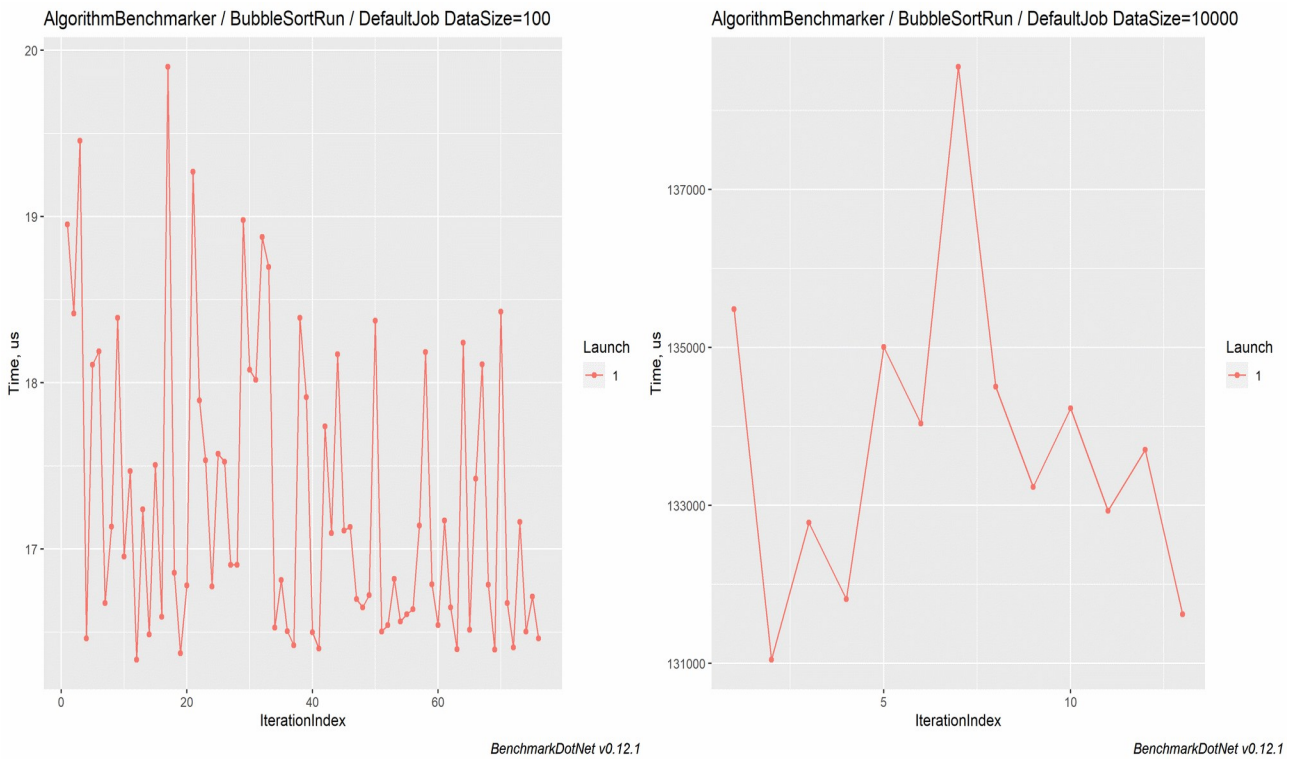
Bästa fall komplexitet är ett andra scenario där en viss matris eller lista över objekt är redan sorterad i ordning. I bästa fall komplexitet görs första iteration för att verifiera fallet, men inga element behöver bytas ut och iterationen kommer avslutas omedelbart (Hinrichs, 2015). Låt oss anta att vi har en given n array lista som kommer att sorteras, den första iterationen i bästa fall kommer att vara n-1-jämförelser och därefter avslutas jämförelsen. Så, tidskomplexitet för bubble sort i bästa fall är O(n) (Woltmann, 2020).

Genomsnittlig fallkomplexitet: O (n^2)

Genomsnittligt fall komplexitet för bubble sort funkar lika med värsta fall komplexitet. För mer detaljer av tidskomplexitet, space komplexitet och egenskaper av bubble sort(se tabell 2).

Algorithm	Time best case	Time avg. case	Time worst case	Space	Stable
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	No

Tabell 2. visar tidskomplexitet, storlek komplexitet och stabilitet av olika algoritmer (Woltmann,2020)



Figur 9, visar tiderna på varje enskild körning av Bubblesort-algoritmen producerad av vårt test-program. Y-axeln är tiden i mikrosekunder. X-axeln är vilket nummer av körning. Vänstra bilden är vid en datamängd på 100 och högra är vid en datamängd på 10,000.

5.7 HeapSort Algoritmer

Sorteringstyp: Selection sort

Storlekskomplexitet: O(1)

In place: Ja

Stabilitet: Ostabil

Adaptivitet: Icke adaptiv

Beskrivning:

Heap sort är en sorteringstyp som använder sig av binära träd för att sortera element. Istället för att skapa noder så skapar heap sorten ett binärt sökträd som sedan ordrar ihop positioneringen av elementen i en array av sig själv genom att plocka ut det minsta elementet i heapen och sedan lägga in den in i arrayen för att få en sorterad array.

Författaren förklarar att heapsort har två egenskaper som kallas: Shape property och Heap property. (Interviewbit, Heap Sort Algorithm). Tillgängligt**:**https://www.interviewbit.com/tutorial/heap-sort-algorithm/

Shape property: Är ett komplett binärt träd vilket betyder att alla nivåer som finns i trädet är fyllda. Alltså ska varje nod förutom slut noderna ha två “barn”, endast vid detta laget så kallas heapen för ett komplett binärt träd.

Heap property: I en heap property så ska alla noder vara lika med, större än eller mindre än varje “barn” nod. Alltså betyder detta att om förälder noden är större än barn noden så kallas heapen för en “max - heap”, och om barn noden är större än förälder noden så kallas heapen för en “min - heap”

Hur den funkar:

Det finns två faser som ingår i sorteringen av element som heapen använder sig av.
Först och främst så skapas heapen genom att sortera alla element som finns inuti arrayen genom att flytta alla element till rätt position och genom att göra detta så skapas ett binärt träd där det finns vänstra “barnen” och de högra “barnen”.
Efter att heapen har sorterat elementen så börjar den med att flytta ut det minsta talet ur arrayen och lägger in den i heapen, sedan fortsätter heapen med samma process tills att hela arrayen är sorterad.
När ett element tas ut ur heapen så måste maximum index värdet i heapen minskas med 1 så att antalet max-heaparna minskar medans antalet av min-heaparna ökar, detta fortsätter tills listan är sorterad.

Fördelar:

Heap sorten är väldigt effektiv om man jämför den med andra sorteringsalgoritmer, t.ex. så brukar andra sorteringsalgoritmer bli segare och segare desto mer siffror och värden sorteringsalgoritmen får. Alltså betyder det att heap sorten är optimal när det gäller att sortera större listor.

En annan fördel är att minneshanteringen i heap sorten samarbetar med garbage collectorn för att frigöra minne som används av objekten.

Nackdelar:

Nackdelen med heap sort att den är segare än andra sorteringsalgoritmer.

HeapSort Tid komplexitet

Enligt Woltmann (2020) Heapsort långsammare än Quicksort och Merge Sort för slumpmässigt fördelade indata. För sorterad data är heapsort åtta till nio gånger långsammare än quicksort och två gånger långsammare än Merge Sort. Resultat från benchmark verktyg visar också att heapsort är långsammare än quicksort och mergesort(se tabell 1 för mer jämförelse). Effektivitet och zero kräv av extra minne är några några av fördelarna med heapsort. Samtidigt, heapsort är mindre snabbare än Merge sort och andra sorting algoritmer. Heapsort är också instabil med genomsnittlig, bästa fall och värsta fall tidskomplexitet.

Komplexitet

Värsta fallkomplexitet: O (n log(n))

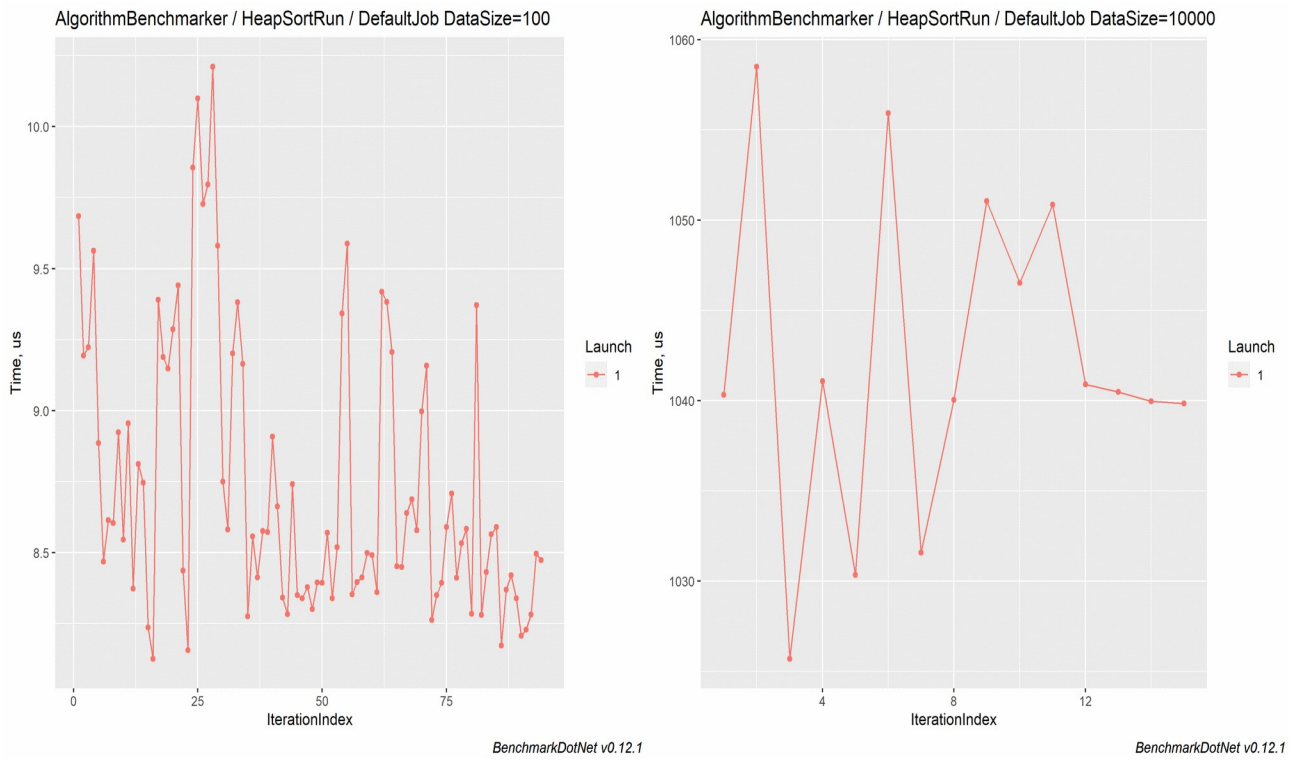
I steget när man tar bort högsta/lägsta värden från heapen så är värsta fall när listan kontinuerligt behöver göra som till en heap. När alla värden är olika så inträffar detta.

Bästa fallkomplexitet: O (n)

Om det är en lista med lika värden är det bästa fall. Där steget med att göra om listan till en heap helt undvikes.

Genomsnittlig fallkomplexitet: O (n log(n))

Lika med värsta fall.



Figur 10, visar tiderna på varje enskild körning av Heapsort-algoritmen producerad av vårt test-program. Y-axeln är tiden i mikrosekunder. X-axeln är vilket nummer av körning. Vänstra bilden är vid en datamängd på 100 och högra är vid en datamängd på 10,000.

5.8 Tree sort

Beskrivning:

En tree sort bygger ett eget binärt sökträd där elementen blir sorterade och därefter korsar trädet så att elementen kommer ut i en sorterad ordning. Tree sort används oftast som en online sorteringsalgoritm och elementen kan bli återhämtade i en sorterad ordning genom att göra en så kallad in order hämtning från det binära sök trädet. Eftersom att tree sort används online så är alltid elementen man lagt in i en sorterad ordning.

Tree sort motsvarar quicksort eftersom att båda rekursivt delar upp elementen baserat på deras vinkel men eftersom att quicksort har lägre omkostnader så har tree sort några fördelar jämfört med quicksort. Samt så kan man använda tree sort för engångsanvändning.

Hur det funkar:

En tree sort fungerar genom att lagra element i ett binärt sökträd. Efter att elementen har lagrats i sökträdet så visas elementen i ordning genom att göra en "inorder traversal" vilket betyder att de element som har ett lägre värde visas först och de element med högre värden visas sist.

Fördelar:

En fördel med tree sort är att vi kan göra ändringar väldigt enkelt. Sorteringen i en tree sort algoritm är lika snabb som att sortera i en quick sort algoritm.

Nackdelar:

Nackdelen med tree sort är att det värsta utfallet händer när elementen i en array redan är sorterade.

Komplexitet

Värsta fallkomplexitet: O (n^2)

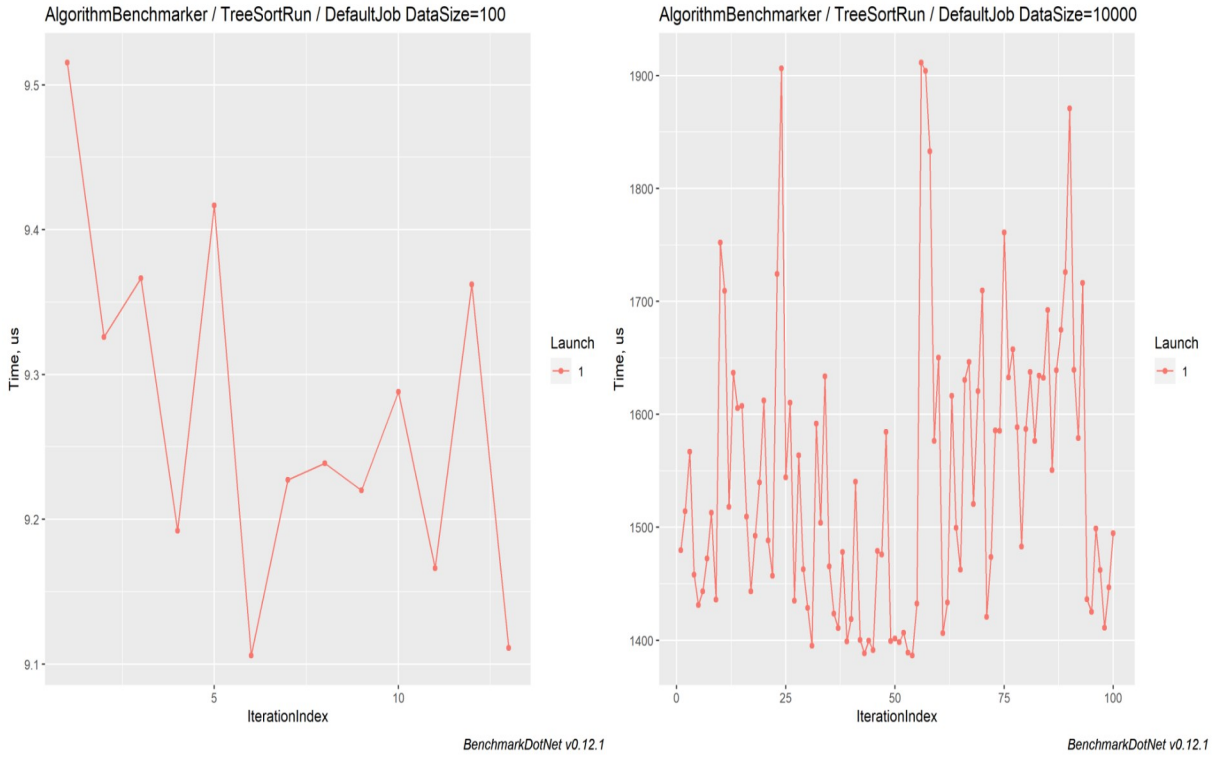
Värsta fall inträffar när listan är sorterad och då skapar ett obalanserat binärt träd med O(n) höjd. Då behövs det O(n^2) tid för att lägga värdena i en lista.

Bästa fallkomplexitet: O (n log(n))

När det binära trädet är balanserat inträffar bästa fall.

Genomsnittlig fallkomplexitet: O (n log(n))

Är detsamma som bästa fall.



Figur 11, visar tiderna på varje enskild körning av Treesort-algoritmen producerad av vårt test-program. Y-axeln är tiden i mikrosekunder. X-axeln är vilket nummer av körning. Vänstra bilden är vid en datamängd på 100 och högra är vid en datamängd på 10,000.

6. Analys och slutsatser

Resultatet av vår undersökning visar att Timsort, Quicksort, Mergesort och Bucketsort är väldigt snabba sorteringsalgoritmer där bucket sort är snabbast vid större datamängder och timsort är snabbast vid små. Insertionsort är också en väldigt snabb algoritm vid små datamängder men blir väldigt slö ju större det är. Insertion Sort används av hybrid algoritmen Timsort tillsammans med mergesort på ett optimalt sätt för att få en väldigt snabb algoritm.

Med resultaten kan man felaktigt anta att de sämre presterande algoritmerna, Bubblesort och insertionsort är obsoleta. Men tiderna avslöjar inte hela sanningen, och det finns fördelar och nackdelar med alla algoritmer.

Bubble Sort är den simplaste sorteringsalgoritmen vilket gör den icke användbar i många områden, men kan gott användas i ställen där tid inte är ett problem och vid enkla lösningar.
Bucket Sort är den snabbaste algoritmen, men har ett värsta fall komplexitet på O(n^2) och använder data utöver det utgivna för att sortera listan.
Insertionsort är en väldigt snabb algoritm vid små datamängder och är simpel att implementera.
Mergesort håller sig nära många av de snabbare algoritmerna och har fördelen att den är väldigt snabb på data som är nästan sorterad.
Quicksort är en av de mest populära algoritmerna, väldigt snabb, fast har nackdelarna att den är iterativ. Därmed kräver den extra stack space. Har ett värsta fall komplexitet på O(n^2).
Timsort väldigt snabb och adaptiv algoritm, men är väldigt komplicerad.
Treesort är en online algoritm som kan sortera medans den får data. Är även väldigt flexibel med olika typer av datastrukturer.
Heapsort är inte den snabbaste men har en garanterad värsta fall på O(n logn) och har fördelen att om den väl blir avbruten, så är listan fortfarande intakt och partiellt sorterad.

Tycker vårt mål med att jämföra dessa populära algoritmer har uppnåtts och vi har fått en bra bild över vilka sorteringsalgoritmer som är optimala att använda och när.
Om du vill sortera en mindre lista och inte behöver tänka på performance kan du plocka fram en bubble sort. I motsatta fall om du vill ha en high performance algoritm med riktiga och verkliga situationer, kan du välja Timsort för dess adaptiva och snabba sortering.

Benchmark-resultat

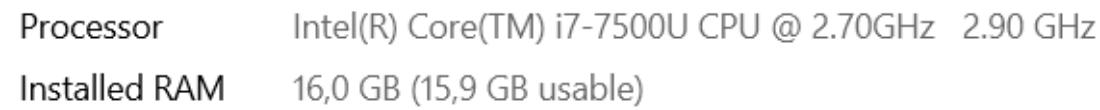
Om man läser resultatet av benchmark-testet (Figur 12) så ser man att vid lägre datamängder är de flesta sorteringsalgoritmer väldigt nära varandra i tid. Med Bubblesort som en utstickare. Så vid lägre datamängder är de flesta sorteringsalgoritmer adekvata. Vid större datamängder börjar man skönja större skillnader där Bubblesort är ett extremt fall med hela 133.8 ms, där jämförelsevis den snabbaste algoritmen Bucketsort är 5 promille av den tiden! Insertionsort är även där en väldigt slö sorteringsalgoritm vid större datamängder och är därmed oanvändbar vid sådana storlekar. Det följer också dess genomsnittliga tidskomplexitet O(n^2) väldigt bra där man kan se att vid högre datamängder ökar deras tid avsevärt. Bucketsort är överlägset snabbast och även där så följer det dess teoretiska genomsnittliga tidskomplexitet O(n + k) som står sig mot resten av de snabba sorteringsalgoritmerna som är på O(n log n).

När vi tittar på algoritmernas användning av allokerade minnesresurser(se figur 12) ser man att sorteringsalgoritmer med storlekskomplexitet O(n) - Bucketsort, Mergesort, Timsort, Treesort - har en avsevärt högre minnesanvändning gentemot algoritmer med storlekskomplexiteten O(1) - Bubblesort,

Heapsort, Insertionsort, Quicksort. Alla de senare algoritmer är även in place och använder då inga externa data strukturer, utan sköter sorteringen i den tilldelade listan. Vill man då ha en snabb algoritm och samtidigt har minnesrestriktioner är Heapsort och Quicksort ett bra alternativ.

Method	DataSize	Mean [ms]	Error [ms]	StdDev [ms]	Median [ms]	Ratio	RatioSD	Gen 0	Gen 1	Gen 2	Allocated [KB]
BubbleSortRun	100	0.0173 ms	0.0003 ms	0.0009 ms	0.0169 ms	1.00	0.00	0.3052	-	-	0.69 KB
BucketSortRun	100	0.0079 ms	0.0002 ms	0.0006 ms	0.0078 ms	0.47	0.04	3.4485	-	-	7.07 KB
HeapSortRun	100	0.0087 ms	0.0002 ms	0.0005 ms	0.0086 ms	0.51	0.03	0.3204	-	-	0.69 KB
InsertionSortRun	100	0.0077 ms	0.0001 ms	0.0001 ms	0.0077 ms	0.44	0.02	0.3357	-	-	0.69 KB
MergeSortRun	100	0.0117 ms	0.0002 ms	0.0005 ms	0.0117 ms	0.67	0.04	10.6812	-	-	21.83 KB
QuickSortRun	100	0.0079 ms	0.0000 ms	0.0000 ms	0.0079 ms	0.45	0.02	0.3204	-	-	0.69 KB
TimsortRun	100	0.0063 ms	0.0001 ms	0.0001 ms	0.0063 ms	0.36	0.02	0.7858	-	-	1.61 KB
TreeSortRun	100	0.0093 ms	0.0001 ms	0.0001 ms	0.0092 ms	0.53	0.03	2.2430	-	-	4.59 KB
BubbleSortRun	10000	133.7638 ms	2.3450 ms	1.9582 ms	133.7019 ms	1.000	0.00	-	-	-	39.69 KB
BucketSortRun	10000	0.6775 ms	0.0129 ms	0.0159 ms	0.6721 ms	0.005	0.00	128.9063	55.6641	-	678.92 KB
HeapSortRun	10000	1.0422 ms	0.0098 ms	0.0092 ms	1.0405 ms	0.008	0.00	15.6250	-	-	39.36 KB
InsertionSortRun	10000	32.4261 ms	0.4265 ms	0.3989 ms	32.4986 ms	0.242	0.01	-	-	-	39.36 KB
MergeSortRun	10000	1.4475 ms	0.0288 ms	0.0620 ms	1.4454 ms	0.011	0.00	1441.4063	-	-	2945.7 KB
QuickSortRun	10000	0.9357 ms	0.0145 ms	0.0129 ms	0.9336 ms	0.007	0.00	17.5781	-	-	39.36 KB
TimsortRun	10000	0.8087 ms	0.0069 ms	0.0061 ms	0.8085 ms	0.006	0.00	190.4297	-	-	391.24 KB
TreeSortRun	10000	1.5449 ms	0.0431 ms	0.1272 ms	1.5136 ms	0.011	0.00	105.4688	50.7813	-	429.98 KB

Figur 12. Resultat från vårt benchmark-test. Första sektionen är körningar av algoritmerna med en datamängd på 100 och den andra med en datamängd på 10,000. Mean [ms] är medelvärdet i mikrosekunder. Error vet ej. StDev är standard avvikelsen mellan varje körning. Median är medianen av alla körningar. Ratio är jämförelse med Bubble Sort som basvärde. RatioSD vet ej. Värdet i Gen 0, Gen 1 och Gen 2 visar hur många gånger per 1000 operationer garbage collectorn kördes. Allocated [KB] är hur mycket algoritmen använde i minne anvisat i kilobyte.



Figur 13. Datorn som Benchmark-testet kördes på.

Marknadsundersökning och intervjuer

Vi ändrade riktning efter en tid in i vårt arbete och började fokusera lite snävare på området inom algoritmer. Där vi valde, efter vi insåg hur brett det skulle vara att endast fokusera på sorteringsalgoritmer. Problem var då att marknadsundersökningen och intervjuerna - även om de var givande - så gav de ej hjälp för vilka sorteringsalgoritmer vi skulle fokusera på. Vi valde att då fokusera på att söka runt på nätet efter vad som var populärt och intressant.

7. Rekommendationer

Vårt mål var inte att komma med några direkta lösningar utan mera att få en överblick över populära sorteringsalgoritmer och
en jämförelse dem emellan. Men om det finns en väldigt bra och allmänt användbar sorteringsalgoritm så är det hybrid algoritmen Timsort.
Timsort är en av dem snabbaste, samtidigt som den har väldigt många bra egenskaper som t.ex. att den är adaptiv och stabil. Det är inte konstigt att den har blivit en av dem mera populära sorteringsalgoritmerna där ute.

8. Källförteckning

Bucket Sort:
Bucket sort algorithm. Tillgängligt: https://www.programiz.com/dsa/bucket-sort

Bubble Sort:

Arora, N., Tamta, V. K., & Kumar, S. (2012). A Novel Sorting Algorithm and Comparison with Bubble sort and Insertion sort. International Journal of Computer Applications (0975 – 8887), 1-2. https://research.ijcaonline.org/volume45/number1/pxc3878940.pdf

https://www.w3resource.com/c-programming-exercises/searching-and-sorting/c-search-and-sorting-exercise-3.php

Woltmann , S. (2020). Bubble Sort – Algorithm, Source Code, Time Complexity. HappyCoders. https://www.happycoders.eu/algorithms/sorting-algorithms/

Hinrichs, L. R. (2015). Sorting Algorithms and Run-Time Complexity. http://www.austinmohr.com/15spring4980/HinrichsPaper.pdf

dwivedi , R., & C. Jain, D. (2014). A Comparative Study on Different Types of Sorting Algorithms (On the Basis of C and Java). *International Journal of Computer Science & Engineering Technology (IJCSET)*, 1-4. http://www.ijcset.com/docs/IJCSET14-05-08-026.pdf

Tree sort:

Beskrivning av tree sort: Mauger Bradley, Study, Tree Sort,
Tillgängligt:
https://study.com/academy/lesson/using-trees-for-sorting-benefits-disadvantages.html
Hur fungerar en tree sort: GeeksforGeeks, 2017, Tillgängligt: https://www.youtube.com/watch?v=n2MLjGeK7qA, [Sedd: 2021]

Fördelar och nackdelar:
Mauger Bradley, Study, Advantages and Disadvantages or Organization,
Tillgängligt:
https://study.com/academy/lesson/using-trees-for-sorting-benefits-disadvantages.html

Heap Sort:

Beskrivning av heap sort
Interviewbit, Heap Sort Algorithm, Tillgängligt**:**https://www.interviewbit.com/tutorial/heap-sort-algorithm/

Hur fungerar en heap sort:
Interviewbit, Heap Sort Algorithm, Tillgängligt**:**https://www.interviewbit.com/tutorial/heap-sort-algorithm/
Fördelar och nackdelar: Interviewcake, Quick reference Strengths/Weaknesses, Tillgängligt: https://www.interviewcake.com/concept/java/heapsort

Woltmann , S. (2020). Heapsort – Algorithm, Source Code, Time Complexity. HappyCoders.

Insertion Sort:

Data structure and algorithms - Insertion sort. Tillgängligt: https://www.tutorialspoint.com/data_structures_algorithms/insertion_sort_algorithm.htm

Merge Sort:

Data structure - Merge sort algorithm. Tillgängligt: https://www.tutorialspoint.com/data_structures_algorithms/merge_sort_algorithm
Merge sort
Merge Sort - GeeksforGeeks

Quick Sort:

Data structure and algorithm - Quick sort. Tillgängligt: https://www.tutorialspoint.com/data_structures_algorithms/quick_sort_algorithm.htm
bild 1 källa:
https://upload.wikimedia.org/wikipedia/commons/a/af/Quicksort-diagram.svg

Timsort:

Brandon. (2018). *Timsort — the fastest sorting algorithm you've never heard of.* https://hackernoon.com/timsort-the-fastest-sorting-algorithm-youve-never-heard-of-36b28417f399

Scheiwe, R. (2018). *The Case for Timsort.*https://medium.com/@rscheiwe/the-case-for-timsort-349d5ce1e414

Bauermeister, R. (2019). *Understanding Timsort.*https. https://medium.com/@rylanbauermeister/understanding-timsort-191c758a42f3

Valdarrama, S. (2012). *Sorting Algorithms in Python.* Real Python. Sorting Algorithms in Python – Real Python

Sök och sortera algoritmer:

Hamrin , N., & Runebjörk, N. (2017). Examining Sorting Algorithm Performance Under System Load. Stockholm: Tekniska högskola.

Wandy, J. (2018). The Advantages & Disadvantages of Sorting Algorithms. sciencing.com.

C# sharp searching and sorting algorithm. Exercises: Bubble sort (2020). Tillgängligt:

https://www.w3resource.com/csharp-exercises/searching-and-sorting-algorithm/searching-and-sorting-algorithm-exercise-3.php

Heap Sort:

Heapsort - algorithm, source code, time complexity (2020). Tillgängligt: https://www.happycoders.eu/algorithms/heapsort/

9. Bilagor

Här listar ni bilagor med nummer så att läsaren lätt kan hitta respektive kan följa hänvisningar till dessa bilagor i den löpande texten.

1 Marknadsundersökning