EDAA35
projekt

Niklas Hedström
dat15nhe@student.lu.se

Jakob Hök
dat15jh1@student.lu.se

Emil Wihlander
dat15ewi@student.lu.se

2015–09–23

Sammanfattning

I denna rapport undersöks exekveringstiden för tre olika sorteringsalgoritmer med samma värsta falls komplexitet vilket i detta fall är $O(n^2)$. Det används tre olika sorters indata med två olika storlekar, 500 tal och med 2000 tal där alla är heltal men det är olika sorterade från början. Den första är helst slumpgenerad, den andra är nästan helt sorterad och den sista är nästan omvänt sorterad. Alla algoritmerna körs 600 och ger ett medelvärde av de 600 gångerna ges och den processen görs 100 gånger för att sedan ta ut medelvärdet av de 100 medelvärdena. Sedan jämförs alla resultaten med varandra för att därefter vissa att *Bubblesort* är den som presterar bäst på alla testerna.

1 Inledning

Allt jämnt med att tekniken går framåt blir det viktigare för program att blir snabbare och då kan minsta nanosekund vara avgörande för om slutresultatet är bra eller dåligt. Ett sätt att få sitt program att jobba optimalt är att välja rätt sorts algoritm till rätt sorts arbete. Ett exempel kan vara att en sorteringsalgoritm kan vara bra på att sortera indata som är nästan helt sorterad men helt värdelös när det kommer till indata som är helt slumpgenererad, så det gäller att veta algoritmernas styrkor och svagheter när man väljer vilken algoritm man ska använda i programmet. Många gånger ska algoritmen köras flera tusen gånger och då kan valet av algoritm vara avgörande för slutresultatet, om en suboptimal algoritm körs en gång och gör en liten negativ differens i resultatet gör inte så stor skillnad men om den upprepas flera gånger blir den negativa differensen snabbt ett problem. Därför gäller det att man är noggrann när man väljer algoritm för att kunna spara viktiga nanosekunder i slutresultatet.

2 Bakgrund

Syftet med dessa tester som vi har gjort är för att kunna jämföra hur *Bubblesort*, *Insertionsort* och *Selectionsort* presterar i ett exekveringstidssammanhang för olika former av indata. Anledningen att valet föll på just dessa sorteringsalgoritmer då de har samma värsta falls tidskomplexitet. Den hypotes som vi satte upp i början av laborationen var att *Insertionsort* och *Bubblesort* skulle vara likvärdiga i många tester men att *Insertionsort* skulle vara oftare än inte vara snabbare än *Bubblesort*, till sist skulle *Selectionsort* vara den som tar längst tid varje gång då den har den värsta normal falls komplexiteten.[1]

3 Frågeställning

- I vilka fall av indata presterar respektive algoritm bäst?
- I vilka fall av indata presterar respektive algoritm sämst?
- Hur presterar algoritmerna i jämförelse med varandra för motsvarande indata?

4 Teori

4.1 Tidskomplexitet

Valet av algoritmer baseras på teorin kring tidskomplexitet. Det är ett mått på antalet iterationer en algoritm behöver gå igenom för att sortera en viss samlingen indata. Eftersom få algoritmer presterar lika bra oberoende hur samlingen indata ser ut delar man in tidskomplexitet i två kategorier, bästa och värsta fall där bästa fall är när algoritmen har som kortast körtid och där värsta fall var som längst körtid.

Det som denna rapport behandlar är värsta fall och mer specifikt algoritmer med $O(n^2)$ som värsta fall. $O(n^2)$ betyder att för n antal element i samlingen indata kommer algoritmen i värsta fall behöva iterera n^2 gånger. Värt att notera

är att detta inte är ett direkt mått på hur lång tid en algoritm tar då en iteration kan ta en obestämd mängd tid.

4.2 Bubblesort

Bubblesort har $O(n^2)$ som värsta fall och arbetar igenom listan genom att jämföra par och byta plats på dem om de ligger i fel ordning enda tills listan är sorterad.[2]

Algorithm 1 Psuedokod för Bubblesort (listan är 0-indexerad)

```
function bubbleSort(list A)
    while swapped do
        swapped ← false
        for i in 1 to A.length − 1 do
            if A[i] < A[i − 1] then
                swap A[i] and A[i − 1]
                swapped ← true
            end if
        end for
    end while
end function
```

4.3 Selectionsort

Selectionsort har $O(n^2)$ som värsta fall och arbetar igenom listan genom att successivt hitta det minsta osorterade elementet och lägga det sista bland de sorterade elementen enda till listan är sorterad.[3]

Algorithm 2 Psuedokod för Selectionsort (listan är 0-indexerad)

```
function selectionSort(list A)
    for i in 0 to A.length − 2 do
        min ← i
        for j in i + 1 to A.length − 1 do
            if A[min] > A[j] then
                min = j
            end if
        end for
        if min ≠ i then
            swap A[min] and A[i]
        end if
    end for
end function
```

4.4 Insertionsort

Insertionsort har $O(n^2)$ som värsta fall och arbetar igenom listan genom att successivt ta nästa element och stoppa in det på den rätta platsen bland de sorterade elementen.[4]

Algorithm 3 Psuedokod för Insertionsort (listan är 0-indexerad)

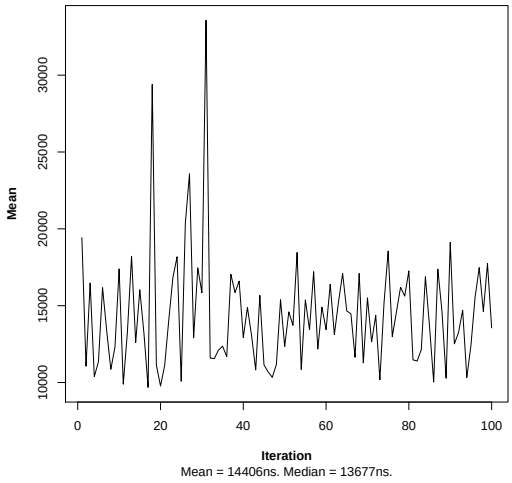
```
function insertionSort(list A)
    for i in 0 to A.length − 1 do
        tmp ← A[i]
        j ← i − 1
        while j ≥ 0 and A[j] > A[i] do
            A[j + 1] ← A[j]
            j ← j − 1
        end while
        A[j + 1] ← tmp
    end for
end function
```

5 Metodik

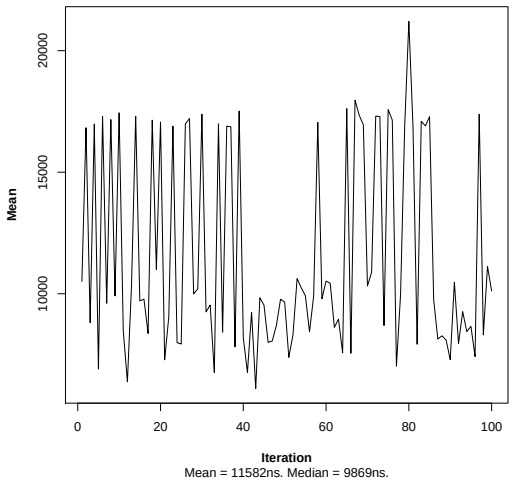
De program som användes för att utföra denna undersökning var *Eclipse* där alla sorterings algoritmer skrevs i java. Sedan för att kunna köra algoritmerna flera gånger och få ut ett medelvärde så skrevs två program i R med hjälp av *Rstudio*. För att få fram indata som skulle sorteras användes först för den slumpgenererade listan ett eget skrivet program i java som hette randomSorted.java som använde javas `Random.nextInt()` för att kunna få trovärdigt slumpgenerade tal. Sedan för att få fram de andra två formerna av indata användes liknande program som hette almostNotSortedIndata.java och almostSortedIndata.java och de skapar indata-filer med ett visst antal element (i detta fall 2000 och 500). Den itererar då 500 eller 2000 gånger i en `for`-loop. I almostSortedIndata.java skapas för varje iteration "i"ett slumptal som kan vara ett värde från `i` till `i+3`. Alltså om vi är på iterationen 345 så kommer det slumpas ett tal mellan 345-348 och lägga in det i en fil. I almostNotSorted.java är det istället `i` till `i+3`.

När alla förberedelseuppgifter var klara blev undersökningens upplägg sådant att de tre olika algoritmerna vilket var *Bubblesort*, *Selectionsort* och *Insertionsort* kördes 600 gånger och de gjordes på tre olika sorterade listor vilket var en slumpgenerade lista, en nästan helt sorterad lista och en nästan omvänt sorterad lista med två olika storlekar 500 och 2000 vilket i slutändan gjorde att det blev 18 olika mätningar. För varje gång en algoritm har körts 600 gånger så togs ett medelvärde av de körningarna och det gjordes 100 gånger för att kunna få ett medelvärde av medelvärdena. Att köra den 100 gånger gör att risken för outliers minskar vilket ger ett mer säkert resultat.

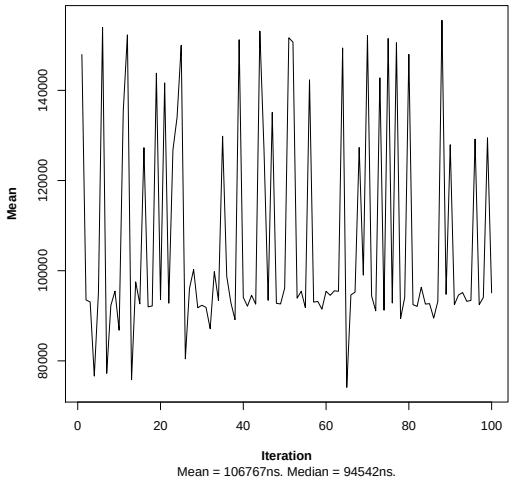
Efter varje körning skapades en plot av medelvärdena av medelvärdena och medelvärdet av dem räknades ut. I figur 1-3 ned kan man se exempel på plottar från mätningarna som gjordes med 500 element som var nästan sorterad från början.



Figur 1: Plot för medelvärdena av medelvärdena som körts 600 gånger med sorteringsalgoritmen *Insertionsort* på en lista med 500 element som var nästan helt sorterade.



Figur 2: Plot för medelvärdena av medelvärdena som körts 600 gånger med sorteringsalgoritmen *Bubblesort* på en lista med 500 element som var nästan helt sorterade.



Figur 3: Plot för medelvärdena av medelvärdena som körts 600 gånger med sorteringsalgoritmen *Selectionsort* på en lista med 500 element som var nästan helt sorterade.

6 Resultat

Här är en tabell över alla medelvärden och medianer som är framtagna från undersökningen med indata som hade 500 element i sig.

Medelvärde: mv, Median: md

antal: 500	Slumpgenererad	Nästan sorterad	Nästan omvänt sorterad
Bubblesort	mv: 3276ns md: 2544ns	mv: 11582ns md: 9869ns	mv: 4018ns md: 2464ns
Selectionsort	mv: 115834ns md: 131243ns	mv: 106767ns md: 94542ns	mv: 115260ns md: 97796ns
Insertionsort	mv: 9384ns md: 7283ns	mv: 14406ns md: 13677ns	mv: 6977ns md: 5141ns

Tabell 1: Tabell med medelvärde och medianer för alla tester som gjordes med 500 element.

antal: 2000	Slumpgenererad	Nästan sorterad	Nästan omvänt sorterad
Bubblesort	mv: 10177ns md: 9304ns	mv: 19265ns md: 19320ns	mv: 10992ns md: 11017ns
Selectionsort	mv: 1922456ns md: 1910580ns	mv: 1857603ns md: 1846435ns	mv: 1977647ns md: 18918116ns
Insertionsort	mv: 14040ns md: 13581ns	mv: 27678ns md: 27089ns	mv: 13828ns md: 13492ns

Tabell 2: Tabell med medelvärde och medianer för alla tester som gjordes med 2000 element.

7 Diskussion

När mätningarna genomfördes var webbläsaren uppe. Detta skulle kunna vara ett validitetshot då det tar extra resurser och kan därför medföra att exekveringstiden ökar. Dock gäller detta för samtliga mätningar vilket därför inte bör spela någon större roll. Majoriteten av mätningarna hade tydliga outliers, vilket syns tydligast i figur 1 där exekveringstiden ökade markant under två mätningar. Webbläsaren skulle kunna spela en roll i detta men då den var konstant uppe och inte användes under själva körningen, är det mer troligt att det är garbage collector som frigör utrymme i minnet vilket får exekveringstiden att öka. Värt att nämna är att outliers vi talar om är ett medelvärde av 600 körningar, alltså måste datorn just under dessa körningar vara extra belastad av externa faktorer.

Det man kan avgöra utifrån resultaten av mätningarna i vårt test är att *Bubblesort* presterar bäst på alla olika former av indata vilket kan tyckas vara konstigt då de flesta andra källor som jämför sorteringsalgoritmer visar på att *Insertionsort* skulle vara likvärdig eller bättre än *Bubblesort* på de flesta formerna av indata.

Den sorteringsalgoritm som presterar svagast på alla testerna är *Selectionsort* vilket inte är speciellt konstigt då *Selectionsorts* värsta falls komplexitet är den samma som *Selectionsorts* normal falls komplexitet.

Det man kan se att utifrån resultaten är att de olika formerna av indata inte har haft någon större inverkan på slutresultatet då de i förhållande till varandra ger ungefär samma resultat för alla de olika formerna av indata. Att det blir så kan bero på att *Insertionsort* samt *Bubblesort* har snarlika tillvägagångssätt att sortera en vektor. Jämförs *Selectionsorts* körningar på de olika indata (med samma mängd element) så syns det tydligt att sorteringsalgoritmen presterar likadant oberoende av vilken indata. Det är inte förvånansvärt då *Selectionsort* har bästa-fall-, medel-fall och värsta-fall-komplexitet $O(n^2)$.

8 Slutsats

För att återkoppla till själva forskningsfrågorna kan man utifrån tabell 1 och 2 se väldigt tydliga svar på frågorna. Den första frågan var ”I vilka fall av indata presterar respektive algoritm bäst?” man kan tydligt se att *Bubblesort* presterade bäst i alla tester som gjordes.

Den andra frågan var ”I vilka fall av indata presterar respektive algoritm sämst?” det går även i denna fråga se att *Selectionsort* presterade sämst i alla tester som utfördes.

Den sista frågan var ”Hur presterar algoritmerna i jämförelse med varandra för motsvarande indata?” här kan man se att i alla olika former av indata är den förhållandevis skillnaden mellan algoritmerna omkring den samma.

Referenser

[1] *Sorting Algorithm Animations*, http://www.sorting-algorithms.com/, (2016 April 27, 11:30)

[2] *Bubble sort*, (senast ändrad: 28 April 2016, 00:53), https://en.wikipedia.org/wiki/Bubble_sort, (2016 April 28, 19:30)

[3] *Selection sort*, (senast ändrad: 25 April 2016, 07:57) https://en.wikipedia.org/wiki/Selection_sort, (hämtad: 28 April 2016, 19:30)

[4] *Insertion sort*, (senast ändrad: 26 April 2016, 12:46) https://en.wikipedia.org/wiki/Insertion_sort, (hämtad: 28 April 2016, 19:30)