EDAA35 Labb 5 Rapport

André Frisk

Mars 2021

1 Sammanfattning

Laborationens syfte är att få studenten att undersöka ett programspråks exekveringstid för att sortera Linked List, som i detta fall skrivs i Java. Med hjälp av scriptspråket R ska mätningar tas ut från java-programmet och jämföra om Javas Collections.sort() är snabbare och mer effektiv än en egenskriven sorteringsmetod som i detta fall är mergeSort. Utifrån resultatet kan det konstateras att Collections.sort() är den operation som är bäst att använda, detta kan bero på att den är mer optimerad för sortering än studentens egenskrivna metod.

2 Introduktion

Under denna laborationen ska studenten sätta sig in i hur man mäter exekveringstid och jämföra det med hjälp av programspråket R. Studenten ska skriva två java program där man använder javas egenimplementerade sorteringsfunktion och ett program där studenten själv skriver sorteringfunktion för att sedan jämföra och ta reda på med vilken funktion som är bäst att använda inom sortering. Målet är främst att lära sig hantera R i jämförande syften där olika programvarusystem ska testas samt att kunna dra slutsatser utifrån datan av dessa två program med hjälp av tester och grafer.

3 Bakgrund

De två java-programmen som ska skrivas ska använda sig utav två olika sorteringsfunktioner. Den ena som finns i javas standardbibliotek Collections.sort() och den andra ska vara skriven enligt egenvald sorteringsfunktion. Collections.sort() är en implementation av en stabil, adaptiv och iterativ mergeSort som har tidskomplexiteten O(nlog(n)) när listan är delvis sorterad, tidskomplexiteten O(n) när listan är nästintill sorterad vilket gör att denna sorterinsfunktion är väldigt snabb och felsäker [4]. Tidskomplexitet är ett mått på tid som görs genom att estimera tidskostnaden för de elementära operationer som krävs i en algoritm [2]. Sorteringsfunktionen som skrivs själv är den som ska testas

mot Collections.sort() och där finns många exempel kring vilka sorteringsfunktioner som kan användas. Exempelvis från Björn Regnells "Introduktion till programmering med Scala" berättas om olika sorterinsfunktioner som kan användas under kapitel 12 [3], som t.ex. insertionSort och selecionSort som båda har tidskomplexiteten O(n^2) vilket är i princip den tidskomplexiteten som ej är optimal då den har väldigt lång exekveringstid vid många indata element. För att försöka skriva en bra sorteringsfunktion måste därför en tidskomplexitet på O(nlog(n)) eller O(n) användas för att få bättre resultat än Collections.sort().

Under laborationen ska JIT undersökas. JIT står för "Just-In-Time"-kompilering vilket betyder att JIT kompilatorn körs efter att programmet har startats och kompilerar koden (bytekod) på vägen vilket oftast gör att programmet körs snabbare [1]. En traditionell kompilator kompilerar all kod till maskinkod innan programmet körs.

Senare i rapporten kommer konfidensintervall och T-test att beskriva resultaten. Konfidensintervall innebär av att detta intervall är att det verkliga medelvärdet av den bakomliggande fördelningen med en sannolikhet enligt konfidensgraden ligger inom konfidensintervallet [1]. T-test är ett test där man testar om det finns en signifikant skillnad mellan de underliggande medelvärdena mellan två talserier [1]

4 Frågeställning

Under laborationen ska två frågeställningar besvaras som rör sig om JIT och sorteringstiden:

- Hur lång tid tar sorteringsfunktionen Collections.sort() i jämförelse med den egenskrivna sorteringsfunktionen att exekvera.
- Vad är skillnaden om man kör med JIT-kompilering på dessa två sorteringsfunktioner?
- Blir det någon skillnad mellan olika indata som tillhandahålls i labben?

5 Metod

5.0.1 Förberedelseuppgift

Förberedelseuppgiften är uppdelad i två delar. Där den första delen baseras på att skriva ett java-program där tiden ska mätas mellan när Collection.sort() startas och avslutas och ta skillnaden mellan start- och sluttid. Programmet ska kunna köras N-gånger som användaren bestämmer värdet på N när programmet kallas i terminalen med en textfil med indata och ett skrivet namn för resultatfilen. Alla tider läggs till i en LinkedList<Integer>. Tiden det tar för sorteringen skrivs in i en resultatfil.

Detta sker även för den andra delen där studenten ska skriva en egen sorteringsfunktion och mäta dess tid. I denna rapport skrevs en mergeSort till

förberedelseuppgiften som har en tidskomplexitet på O(nlog(n)). Se koden till mergeSort i figur 1.

```
private static LinkedList<Integer> sorter(LinkedList<Integer> temp) {
    if (temp.size() <= 1) return temp;

    int mid = temp.size()/2;
    LinkedList<Integer> r = new LinkedList<~>(temp.subList(mid, temp.size()));
    LinkedList<Integer> l = new LinkedList<~>(temp.subList(0, mid));
    r = sorter(r); l = sorter(l);

    return mergeSort(r,l);
}

private static LinkedList<Integer> mergeSort(LinkedList<Integer> r, LinkedList<Integer> l) {
    LinkedList<Integer> resultSorted = new LinkedList<~>();
    while(r.size() > 0 && l.size() > 0){
        if(r.get(0) < l.get(0)) resultSorted.add(r.remove( index: 0));
        else resultSorted.add(l.remove( index: 0));
    }
    resultSorted.addAll(r);
    resultSorted.addAll(l);
    return resultSorted;
}
```

Figure 1: Koden för sorterinsfunktionen mergeSort.

5.0.2 Laboration

Denna del av laborationen sker i flera mindre steg där kod skrivs i R för att utföra dessa instruktioner. Se figur 2 längre ner för koden till laborationen (notera att viss kod kan ha blivit utbytt till en senare uppgift då man skulle utveckla koden successivt under laborationen). Om inget nämns ska filen data1.txt användas som tilldelades under laborationen.

1. Använd förberedelseuppgiften med Collections.sort() där exekveringstiden mäts med N = 600 från terminalen. Plotta dessa värdena där x-axeln ska vara numret för mätning och y-axeln är exekveringstiden. Bestäm var gränsen mellan insvängningsförloppet och jämviktsläget ligger (insvängningsförloppet - intervallet där grafen inte är planad, jämviktsläget - intervallet efter insvängningsförloppet där grafen är planad och jämn efter varje exekvering [1]).

2. Utöka scriptet i R så att ett medelvärde räknas ut efter varje gång programmet körs, detta ska göras N gånger. Detta ska även köras några

gånger i R som t.ex. 10 gånger med en while-loop. Medelvärdet av exekveringstiden ska räknas utifrån jämviktsområdet vilket man får reda på genom grafen i föregående instruktion. När java-programmet har exekverats en sista gång ska medelvärdet av medelvärdena räknas ut och därefter ta reda på konfidensintervallet (en skattning av osäkerheten av parametrarna man skattar). Jämför konfidensintervallet om man startar java-programmet 10 och 100 gånger.

3. Testa att avaktivera JIT-komilatorn och jämför hur långt tid det tar. Plotta en graf för detta. Bestäm även här medelvärdet och konfidensintervallet för exekveringen.

4. Kör java-programmet med den egenskrivna sorteringsfunktionen och ta reda på medelvärdet och konfidensintervallet. Sätt igång JIT-kompilatorn igen!

5. Jämför nu datan som samlats genom testerna med Collection.sort() programmet och programmet med den egenskrivna sorteringsfunktionen med hur stor skillnad det är i jämförelse med exekveringstiderna. Jämför det med t.ex. ett t-test.

6. Gör samma jämförelse som i förra instruktionen fast använd data2.txt filen som tillhandahålls i laborationen. Förklara eventuella skillnader.

```
confidenceInterval <- source("https://fileadmin.cs.lth.se/cs/Education/EDAA35/R_resources.R")$value

plotresult <- function(file, start = 1)
{
  data <- read.csv(file)
  data <- data[start:nrow(data),]
  plot(data, type = "l")
}

Lab <- c()
for(i in 1:10){
  system("java -cp C:/IntelliJ-Java/Labb5-EDAA35/src/ Lab C:/IntelliJ-Java/Labb5-EDAA35/src/data2.txt result1.txt 600")
  data <- read.csv("result1.txt", sep = " ")
  data <- data[360:nrow(data),]
  Lab <- append(Lab, mean(data$Time))
}
print(mean(Lab))
print(confidenceInterval(Lab))


LabListSorter <- c()
for(i in 1:10){
  system("java -cp C:/IntelliJ-Java/Labb5-EDAA35/src/ ListSorter C:/IntelliJ-Java/Labb5-EDAA35/src/data2.txt result1.txt 600")
  dataListSorter <- read.csv("result1.txt", sep = " ")
  dataListSorter <- dataListSorter[360:nrow(dataListSorter),]
  LabListSorter <- append(LabListSorter, mean(dataListSorter$Time))
}
print(mean(LabListSorter))
print(confidenceInterval(LabListSorter))
print(t.test(LabListSorter, Lab))
```

Figure 2: Koden för laborationen i R

6 Resultat

Notera att all form av tid uttryckt nedan är i nanosekunder. För att få fram medelvärdet i jämviktsområdet så måste man ange ett startnummer i R-scriptet när man ska börja räkna tiderna från javaprogrammet. Enligt figur 3 har insvängningsförloppet stoppats när N $>$ 360 vilket gör att jämviktsområdet börjar precis efter (Collections.sort() programmet).

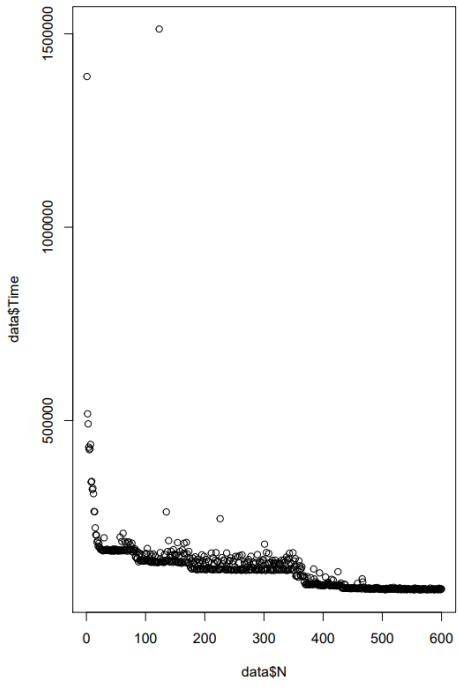

Figure 3: Graf för N = 600 gånger, data$Time - tiden för exekverings uttryckt i nanosekunder, data$N - antal exekveringar.

I följande tabell visas medelvärdet och konfidensintervallet för när Collections.sort() programmet kördts 600 gånger i en loop av 10 gånger med data1.txt som indata.

Iterationer (R)	Medelvärde	Konfidensintervall (Lower)	Konfidensintervall (Upper)
10	190466,6	166142,2	214791,0
100	177168,9	170157,5	184180,2

Table 1: Medelvärde & konfidensintervall för 10 & 100 iterationer.

En graf för att visa hur JIT-kompilering påverkar exekveringstiden om man kör utan JIT skulle produceras och visas i figur 4 samt en tabell som visar medelvärdet och konfidensintervallet utifrån denna data med data1.txt.

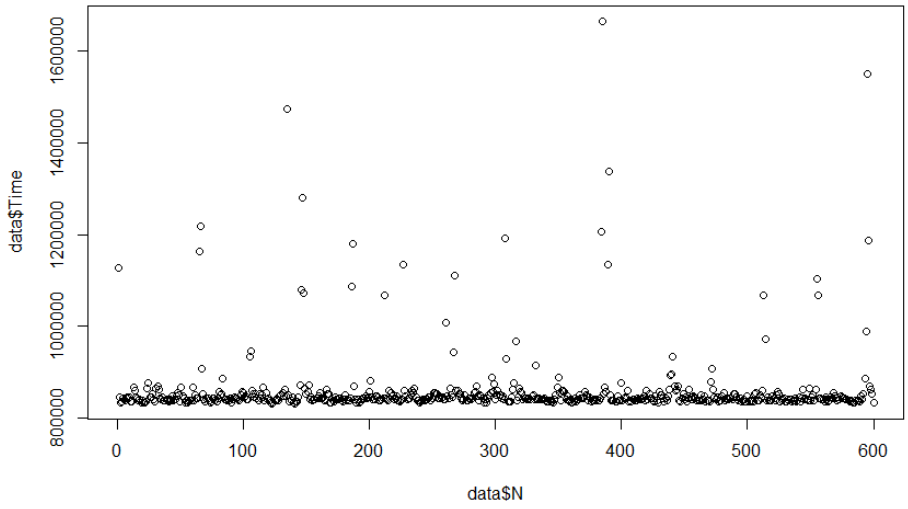

Figure 4: Graf för N = 600 gånger när JIT är avstängd, data$Time - tiden för exekverings i nanosekunder, data$N - antal exekveringar

Iterationer (R)	Medelvärde	Konfidensintervall (Lower)	Konfidensintervall (Upper)
10	2249157	2161710	2336603

Table 2: Medelvärde & konfidensintervall för 10 iterationer utan JIT

Data för den egenskrivna sorteringsfunktionen (mergeSort i detta fall) samlades in med JIT-kompilatorn på samt data1.txt som indata. Se tabell 3.

Iterationer (R)	Medelvärde	Konfidensintervall (Lower)	Konfidensintervall (Upper)
10	508958,7	474893,5	543023,8

Table 3: Medelvärde & konfidensintervall för 10 iterationer med den egenskrivna sorteringsfunktionen.

Jämförelse av data som samlats genom testerna med Collection.sort() programmet och programmet med den egenskrivna sorteringsfunktionen ska göras med hur stor skillnad det är i jämförelse med avseende på exekveringstiderna. Detta gjordes med både data1.txt och data2.txt som indata (två separata fall). Se t-testerna nedan (Lab är Collections.sort() programmet och LabListSorter är den egenskrivna).

```
        Welch Two Sample t-test

data:  LabListSorter and Lab
t = 22, df = 16.749, p-value = 8.563e-14
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 268877.9 325991.7
sample estimates:
mean of x mean of y
 461211.3  163776.5
```

Figure 5: T-test för data1.txt som indata

```
        Welch Two Sample t-test

data:  LabListSorter and Lab
t = 14.721, df = 11.109, p-value = 1.235e-08
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 319431.9 431589.3
sample estimates:
mean of x mean of y
 501565.6  126055.0
```

Figure 6: T-test för data2.txt som indata

7 Diskussion

Mycket kring laborationen är baserat på tolkning av resultatet och eventuellt fellästa parametrar kan ge stora skillnader kring resultatet. Till exempel vid

början på laborationen skulle en avläsning på jämviktsområdet göras för att kunna läsa data utifrån detta område då det är där programmet stabiliserat sig som mest med JIT-kompilering. Då programmet som skrevs i R inte riktigt pekade ut vart jämviktsområdet ligger utan avläsning skedde i grafen (se figur 3). Där N = 360 tycktes jämviktsområdet börja på vilket ger ungefär 240 mätningar till statistiken, om fel N har lästs av här förloras väldigt många mätningar som påverkar resultatet som gavs i resultatsektionen.

JIT-kompilering var en av frågeställningarna som var att undersöka utifall någon skillnad uppstod att köra med och utan JIT-kompilering. I figur 3 och figur 4 syns en markant skillnad på exekveringstider. Där i figur 3 med JIT-kompilering ligger snittet på exekveringstiden på 20 000 nanosekunder medans i figur 4 utan JIT-kompilering visar ett snitt på 50 000 nanosekunder. Dock finns flera mätningar som visar tider mellan 100 000 och 160 000 nanosekunder. JIT-kompilering gör om koden till maskinkod vid exekvering vilket gör att koden kommer att köras snabbare än med utan JIT- Utan JIT körs koden med bytekod via Javas kompilator och körs långsammare. Eftersom att koden körs utan JIT lär sig inte datorn om programmet och memoriserar steg så det hamnar i jämviktsområde utan den kör om programmet varje gång som om det vore på nytt från maskinkoden.

Under laborationen tillhandahålls två filer som ska agera indata, data1.txt och data2.txt. Dessa filer innehåller samma information där data2.txt är en aning sorterad med datan medans data1.txt har sina värden slumpvalt utsatta i filen. Detta ska teoretiskt sätt göra att sorteringsfunktionerna, både Collections.sort() och den egenskrivna ska ge en lägre exekveringstid då sorterad indata är lättare att sortera än osorterad. Med hjälp av dessa två filer ska en skillnad tas ut ur t-testerna genom att undersöka testet med till exempel medelvärde och konfidensintervall. Dessa tester visas i figur 5 (data1.txt som indata) och figur 6 (data2.txt som indata). Då konfidensintervallet baseras på en 95% signifikansnivå ska p-value vara mindre än 0,05 och båda tester visar ett avsevärt mindre värde än 0,05. Därför kan man dra slutsatsen att det finns en signifikant skillnad mellan medelvärdena i talserierna Lab och LabListSorter. Om konfidensintervallet och medelvärdet undersöks i båda figurer syns det en stor skillnad kring talserierna. Utifrån t-testet och tabellerna visar det sig att Collections.sort() programmet (talserien Lab) har bäst exekveringstid som gör den till det självklara valet vid sortering.

Något som nämndes under redovisningen av vår data vi samlats in är att istället för att göra LinkedList<Integer> temp = new LinkedList<Integer>(list); när listan skulle kopieras så användes LinkedList<Integer> temp = list.clone(); i java-koden. Dock är java-bibliotekets List metod clone() shallow vilket betyder att den inte skapar en äkta kopia av listan utan är sets av pointerns till minnes allokeringen. Enligt labbhandledaren ska detta inte ge någon inverkan på resultatet i denna laborationen. Det kan ändå va värt att nämna utifall något resultat visar sig vara utstickande.

8 Slutsatser

8.0.1 Jämför hur lång tid sorteringsfunktionen Collections.sort() tar med den egenskrivna sorteringsfunktionen.

Enligt t-testerna visas en signifikant skllnad där Collections.sort() är det självklara valet kring tiden det tar att sortera listorna. Medelvärdet och föregående data visar även att Collections.sort() programmet presterar bäst inom mängden tid.

8.0.2 Vad är skillnaden om man kör utan JIT-kompilering på dessa två sorteringsfunktioner kontra med JIT-kompilering?

Skillnaden blir att exekveringstiden ökar drastiskt om man kör programmet utan JIT-kompilering pga att bytekod tar mycket längre tid att läsas och exekveras. Dessutom om man kör utan JIT-kompilering så planas grafen aldrig ut och ett jämviktsområde kan ej avläsas utifrån figur 4.

8.0.3 Blir det någon skillnad mellan olika indata som tillhandahålls i labben?

Utifrån t-testerna sorteras Collection.sort() programmet lite snabbare med data2.txt än med data1.txt medans den egenskrivna sorteringsfunktionen tar längre tid. Detta kan visa att den egenskrivna sorteringsfunktionen är dåligt kodad. Collections.sort() kan även hantera någorlunda sorterad indata annorlunda och snabbare.

9 Fortsatt arbete

Något som hade kunnat vara fortsatt arbete på laborationen är att försöka skriva en bättre sorteringsfunktion då det visade sig den var sämre än Collections.sort(). Målet hade varit att försöka sänka tidskomplexiteten och få sorteringen att bli smidigare när det exekveras så exekveringstiden sjunker mot nivån på Collections.sort(). En sorteringsfunktion man hade försökt kunna skriva själv är TimSort [2]. TimSort är ursprungligen från programspråket Python och fungerar liknande som Collections.sort() där TimSort är en kombination utav insertionSort och mergeSort. TimSort har en tidkomplexitet på O(nlog(n)) och används av java.Arrays.sort vilket tyder på en bra sorteringsfunktion, dock är den rätt lång och svår att implementera.

References

[1] Martin Höst. *EDAA35 Utvärdering av programvarusystem.* Institutionen för Datorvetenskap, Lunds Tekniska Högskola, 2021.

[2] Elliot B. Koffman and Paul A. T. Wolfgang. *Data Structures: Abstraction and Design Using Java, Third Edition*. Wiley, Temple University, 2016.

[3] Björn Regnell. *Introduktion till programmering med Scala*. Institutionen för Datorvetenskap, Lunds Tekniska Högskola, 2020.

[4] Wikipedia. Time complexity.