Rapport: Inda Quicksort timing test

Inledning

Denna rapport handlar om 4 olika implementationer av sorteringsalgoritmen quicksort i java. De olika versionerna testkörs och tidsmäts. I slutet jämförs de även med Javas egna Arrays.sort.

Testdata

Alla algoritmer fick sortera likadana data. Den data jag valde att sortera var:

```
Data[] d = {
        new Data(0, 20, Data.Order.RANDOM),
        new Data(1, 20, Data.Order.RANDOM),
        new Data(n, 1000000, Data.Order.RANDOM),
        new Data(n, 1, Data.Order.RANDOM),
        new Data(n, 5, Data.Order.RANDOM),
        new Data(n, 1000000, Data.Order.ASCENDING),
        new Data(n, 1000000, Data.Order.DESCENDING),
    };
```

Detta innebär:
En vektor med 0 element i slumpad ordning och slumpat vald mellan 1-20.
En vekor med 1 element i slumpad ordning och slumpat vald mellan 1-20.
En vektor med 10 000 000 element i slumpad ordning och slumpat valda mellan 1-1 000 000.
En vektor med 10 000 000 element i slumpad ordning och slumpat valda mellan 1-1.
En vektor med 10 000 000 element i slumpad ordning och slumpat valda mellan 1-5.
En vektor med 10 000 000 element i stigande ordning och slumpat valda mellan 1-1 000 000.
En vektor med 10 000 000 element i fallande ordning och slumpat valda mellan 1-1 000 000.

Tidsmätningarna

Jag har tagit hänsyn till följande:

- Hur hanterar du problemet att det ibland tar extra lång tid de första gångerna man kör ett kodavsnitt?
- Hur hanterar du problemet att resultaten varierar mellan körningar?

Det första problemet hanterar jag genom att låta algoritmen sortera en likadan vektor en gång innan tidsmätningen startar.

Jag låter varje sorteringsalgoritm sortera lika data 5 gånger för att få resultat från alla fem gånger och se om det skiljde sig mycket. Varje tidsmätning görs också genom att sortera 2 lika vektorer efter varandra och sedan dela tiden på två. Då får jag ett snitt på hur lång tid det tar att sortera vektorerna.

Brytpunkten

Brytpunkten är alltså det antalet element som ska sorteras av insertionsort istället för quicksort.

För att testa vilket värde som var bäst för detta tal försökte jag först köra tester manuellt många gånger med många olika värden men fick väldigt nära resultat och lite varierande. P.g.a. detta så gjorde jag ett test där jag räknade ut summan av alla tider det tog för en sorteringsalgoritm och jämförde med det senast bästa. Detta gjorde att i slutet av körningen fick jag ett resultat på bäst värde i snitt.

Detta värde varierade lite mellan olika tester. Första gångerna testade jag från 1-100 som värden. Sedan såg det som bäst ut mellan 5-20. Genom att dra ihop intervallet mer och mer och efter många tester kom jag fram till en brytpunkt mellan 5-8 var bra, där 6 och 7 var mest förekommande.

Jag valde därför värdet 6 som min standardbrytpunkt.

Resultat

Detta är ett urdrag från en utskrift i testet av bästa brytpunkt:

Time to sort 1000000 from 1-1000000 DESCENDING: 5 ms

Time to sort 1000000 from 1-1000000 DESCENDING: 5 ms

Time to sort 1000000 from 1-1000000 DESCENDING: 5 ms

Sorted with Quicksort with insertionsort at 10 with random pivot

52	52	53	53	52
53	53	52	53	54

Best sorter2: 6 Best sorter4: 7

Där båda är quicksort med insertionsort. Sorter 2 väljer första elementet som pivot och Sorter 4 tar ett slumpat pivot.

Nedan följer utdrag ur mätningarna av tiden det tar för algoritmerna att sortera precis likadana listor.

Time to sort 10000000 from 1-1000000 RANDOM: 1204 ms

Time to sort 10000000 from 1-1 RANDOM: 16 ms

Time to sort 10000000 from 1-5 RANDOM: 132 ms

Time to sort 10000000 from 1-1000000 ASCENDING: 20 ms

Time to sort 10000000 from 1-1000000 DESCENDING: 30 ms

Sorted with Basic quicksort

Time to sort 10000000 from 1-1000000 RANDOM: 1204 ms

Time to sort 10000000 from 1-1 RANDOM: 16 ms

Time to sort 10000000 from 1-5 RANDOM: 130 ms

Time to sort 10000000 from 1-1000000 ASCENDING: 23 ms

Time to sort 10000000 from 1-1000000 DESCENDING: 28 ms

Sorted with Quicksort with insertionsort at 6

Time to sort 10000000 from 1-1000000 RANDOM: 1104 ms

Time to sort 10000000 from 1-1 RANDOM: 16 ms

Time to sort 10000000 from 1-5 RANDOM: 114 ms

Time to sort 10000000 from 1-1000000 ASCENDING: 20 ms

Time to sort 10000000 from 1-1000000 DESCENDING: 29 ms

Sorted with Basic quicksort with random pivot

Time to sort 10000000 from 1-1000000 RANDOM: 1308 ms

Time to sort 10000000 from 1-1 RANDOM: 16 ms

Time to sort 10000000 from 1-5 RANDOM: 119 ms

Time to sort 10000000 from 1-1000000 ASCENDING: 20 ms

Time to sort 10000000 from 1-1000000 DESCENDING: 28 ms

Sorted with Quicksort with insertionsort at 6 with random pivot

Time to sort 10000000 from 1-1000000 RANDOM: 1250 ms

Time to sort 10000000 from 1-1 RANDOM: 18 ms

Time to sort 10000000 from 1-5 RANDOM: 145 ms

Time to sort 10000000 from 1-1000000 ASCENDING: 214 ms

Time to sort 10000000 from 1-1000000 DESCENDING: 236 ms

Sorted with java Arrays.sort

I en tabell där vi kan jämföra värden med varandra:

List	10^7 from 1-10^6 RANDOM	10^7 from 1-1 RANDOM	10^7 from 1-5 RANDOM	10^7 from 1-10^6 ASCENDING	10^7 from 1-10^6 DESCENDING
Sorter1	1204	16	132	20	30
Sorter2	1204	16	130	23	28
Sorter3	1104	16	114	20	29
Sorter4	1308	16	119	20	28
Arrays.sort	1250	18	145	214	236

Sorter1: Vanlig quicksort med pivot alltid vald som första element.

Sorter2: Quicksort hybrid med Insertionsort. Startar insertionsort vid 7 element. Alltid första element som pivot

Sorter3: extendar Sorter1 och skillnaden är valet av pivot är slumpat tal i listan

Sorter4: extendar Sorter2 och skillnaden är valet av pivot är slumpat tal i listan.

Diskussion

Genom mina mätningar kan man se att hybriden med insertsionsort inte alltid är bättre än de som inte använder sig av insertionsort. Det är en intressant observation. Quicksort är ju väldigt snabb när man har flera liknande element i listan för dessa utesluts på linjär tid att de redan är sorterade. Men med många olika värden blir det fler element som inte är lika med pivot och man får många fler rekursioner.

När jag började testa märkte jag att listorna som redan var sorterade tog ca 51000 ms för de algoritmer som alltid tog första elementet som pivot. Detta p.g.a. att den får endast listan med element större än pivot hela tiden när den är stigande redan från början. Därför implementerade jag också att kolla om listan är stigande eller fallande och om den är fallande så vänder jag bara på den. Detta gav väldigt bra prestanda på de redan sorterade listorna.

Det är intressant att nästan alla mina algoritmer nästan alltid är snabbare än Arrays.sort. Vid de redan sorterade listorna använder jag ju inte quicksort algoritmen direkt, eftersom jag aldrig kommer fram dit. Jag vet inte hur Arrays.sort hanterar redan sorterade listor men den har något som gör att den hanterar dessa hyfsat snabbt ändå. Den algoritmen kanske kollar fler saker för att vara säker på

hur den ska ta an sorteringen innan den börjar sortera. Eller upptäcker att vissa bitar är sorterade och sedan sätter ihop dem. Det kan vara orsaken till att den tar längre tid i de fallen.

När jag gjorde mina mätningar ville jag också vara säker på att java runtime inte går in och förstör någon av mätningarna genom att göra något extrajobb under själva mätningen. Man kan köra ett javaprogram med vissa inparametrar. Jag körde exempelvis med -verbose:gc som skriver ut när garbage collector går in och gör jobb. Den körde vissa jobb ibland men det var alltid då klockan var stoppad vilket var bra. Jag testade även men parametern -XX:+Printcompilation. Den skriver ut när programmet kompileras och då även när programmet compilerar bytekoden till x86. Detta skedde inte heller under tiden klockan var startad i någon av mina tester vilket innebär att java inte förstörde min mätning.

Om jag inte hade förbättrat algoritmen med att först kolla om listan redan är sorterad så hade sorterade listor tagit väldigt lång tid att sortera igen, medans det blir några färre operationer för de listor som inte är sorterade. Men kostnaden att kolla detta tycker jag är värd p.g.a. att programmet inte blir superlångsamt om man försöker förstöra det genom att mata in redan sorterade listor.