Hemuppgift 6- Introduktion till Datalogi

Peter Boström

2009-02-24 MIN FÖDELSEDAG <3

Innehåll

1 Quicksort

1.1 Testkod

Testkoden är modulärt uppbyggd och bygger på att listor skapas, kopieras och sorteras av flera olika sorteringsalgoritmer, både javas quicksort, egen insertion sort och diverse egna quicksortvarianter. Sedan *assertas* att alla sorterar korrekt, annars spottas felmeddelande ut med vilken algoritm som misslyckades.

Test görs på följande olika listor

- Tom lista.
- Lista med ett element.
- Lista med två element i ordning.
- Lista med två felsorterade element.
- Lista med tre element.
- Stor lista (4096 element) med slumpmässiga tal.
- Stor sorterad lista.
- Stor felsorterad lista.
- Stor lista där alla element är lika.

1.2 Timer-test

Till tidtagartesten användes Javas originalimplementation som referens för tidtagning. Likt testfallen så görs ett antal identiska listor som sorteras av varje enskild algoritm. Detta tas även tid på och jämförs med Javas quicksorts prestanda på samma lista. För att göra rimliga tidtagningar används ganska stora listor (1 000 000 element), förutom ett fåtal slumplistor med beräkningar på 1 000, 10 000 och 100 000 element resp. En miljon element verkade mer intressant, därför gjordes flera slumpmässiga listor med en miljon element för upprepade beräkningar.

Algoritm/Test	qsort (Java)	qsort	qsort(rand)	qsort+ins sort	qsort+ins sort (rand)
Sorted	300 ms	444 ms (1.48)	634 ms (2.11)	145 ms (0.48)	229 ms (0.76)
Reverse	297 ms	324 ms (1.09)	659 ms (2.11)	124 ms (0.42)	181 ms (0.61)
Same	31 ms	398 ms (12.8)	475 ms (15.32)	209 ms (6.74)	227 ms (7.3)
Rand (1000)	0 ms	0 ms	0 ms	0 ms	0 ms
Rand (10k)	3 ms	3 ms (1)	5 ms (1.67)	2 ms (0.67)	2 ms (0.67)
Rand (100k)	39 ms	66 ms (1.69)	96 ms (2.46)	29 ms (0.74)	53 ms (1.36)
Rand (1M)	437 ms	497 ms (1.14)	698 ms (1.60)	340 ms (0.78)	376 ms (0.86)

1.3 Brytpunkt till insertion sort

Brytpunkten till insertion sort beräknades genom att beräkna ratio mellan javas quicksort och den egna implementation som använde sig av insertion sort. Sedan testade jag att sätta övre gränsen för hur stora listor fick vara och fortsätta sorteras med quicksort. Diverse sorteringar med en miljon element gjordes för att säkerställa hur effektiv den var i förhållande till javas egna implementation.

Jag började på 100 element vilket var långt över quicksort utan insertion sort i körtid. Sedan minskade jag till 50, 25 och valde att testa diverse värden runt 10-20. Slutligen landade jag på 16 element som övre gräns till att påbörja insertion sort.

1.4 Reflektion

Intressant är att slumpmässigt värde på pivot inte är värt på vanliga quicksort-implementationen men inte har större förluster i implementationen med insertion sort. Trolig förklaring för detta är att pseudoslumpade tal helt enkelt inte är värt det på mindre listor, som istället i andra fallet tas med insertion sort. Förmodligen skulle man kunna hitta en brytpunkt där det slutar vara värt att beräkna slump-pivot.

Intressant är även att pseudomedian från tre slumpade index inte verkade vara värt det i någon av fallen för listor i denna storlek. Test gjordes tidigare men togs inte med senare i arbetet då förlusterna från tre slumpade tal helt enkelt var för stora.

Pivot som vänstervärde i sorterad lista blir för komplex och orsakar Stack overflow. Det var tänkt att ha med detta som jämförelsevärde, men det skippades av komplexitetsskäl.

Javas quicksort-implementation var tokigt överlägsen på listor med samma element.

I övrigt så fick den stryk med minst c:a 80% för större listor, och mer på sorterade eller felsorterade, vilket känns kul.

A Källkod

InsertionSort.java QuicksortInsertion.java QuicksortInsertionRandom.java Quicksort.java QuicksortJava.java QuicksortRandomPivot.java Sorting.java SortingTest.java SortingTimer.java Stopwatch.java

A.1 InsertionSort.java

```
/**
 * @author lemming
 * Implementation av Insertion sort.
 *
 * Sorterar monster (Se éPokdex öfr referens av vilka monster som sorteras).
 */
public class InsertionSort implements Sorting{

    /**
     * Sortera en array av intar
     * @param v array att sortera
     */
    public void sort(int[] v){
        for(int i = 1; i < v.length; ++i){
            if(v[i-1]>v[i]){
                int tmp = v[i];

                int j = i;

                while(j > 0 && v[j-1] > tmp){
                    v[j] = v[--j];
                }
                v[j] = tmp;
            }
        }
    }
}
```

A.2 QuicksortInsertion.java

```
/**
 * @author lemming
 * Quicksortvariant som byter till insertion sort öfr ätillrckligt åsm
    listor
 */
public class QuicksortInsertion extends Quicksort{
    /**
     * Quicksort av lista som ävxlar till insertionsort öfr mindre listor
     */
    public void qsort(int[] v, int first, int last){
        if(first >= last)
            return;

        // äVxla till insertion sort, enda skillnaden i quicksort-varianten
        if(last-first <= 16){
            insertionSort(v, first, last);
            return;
        }

```

```
19          // Annars äfortstt som vanligt
20          int mid = partition(v, first, last, getPivot(v, first, last));
21
22          qsort(v, first, mid);
23          qsort(v, mid+1, last);
24      }
25
26      private void insertionSort(int[] v, int first, int last){
27          for(int i = first+1; i < last+1; ++i){
28              if(v[i-1]>v[i]){
29                  int tmp = v[i];
30
31                  int j = i;
32
33                  while(j > 0 && v[j-1] > tmp){
34                      v[j] = v[--j];
35                  }
36                  v[j] = tmp;
37              }
38          }
39      }
40  }
```

A.3 QuicksortInsertionRandom.java

```
1   import java.util.Random;
2
3   /**
4    *
5    */
6
7   /**
8    * @author lemming
9    * Quicksort med insertionsort och slumpad pivot. Ingen pseudomedian.
10   */
11  public class QuicksortInsertionRandom extends QuicksortInsertion{
12      static Random rand = new Random();
13      /**
14       * Generate a pseudorandom pivot from the list
15       */
16      protected int getPivot(int[] v, int first, int last){
17          return v[rand.nextInt(last-first+1)+first];
18      }
19  }
```

A.4 Quicksort.java

```
1   /**
2    * @author lemming
3    * Egen implementation av Quicksort
4    */
5   public class Quicksort implements Sorting{
6       /**
7        * Sortera en array
8        * @param v array att sortera
9        */
10      public void sort(int[] v){
```

```
11          qsort(v, 0, v.length-1);
12      }
13
14      /**
15       * Sorterar alla element mellan first, last i en array med quicksort.
16       * @param v array
17       * @param first öfrsta elementet
18       * @param last  sista elementet
19       */
20      public void qsort(int[] v, int first, int last){
21          // Listan åinnehller äfrre än 2 element och öbehver inte sorteras
22          if(first >= last)
23              return;
24
25          // sortera upp listan i åtv delar, en som är östrre än eller lika
                med pivot, och en mindre
26          int mid = partition(v, first, last, getPivot(v, first, last));
27
28          //Sortera dessa mindre delar
29          qsort(v, first, mid);
30          qsort(v, mid+1, last);
31      }
32
33      /**
34       * Sorterar en del av en lista.
35       * @param v Lista att sortera i
36       * @param first öFrsta elementet att sortera
37       * @param last Sista elementet i listan som ska sorteras
38       * @param pivot Pivot som äanvnds vid sortering
39       * @return index till mitten av listan
40       */
41      public int partition(int[] v, int first, int last, int pivot){
42          // äanvnder sig av åtv index som ölper mot mitten
43          int low = first, high = last;
44
45          //åS älnge hela listan inte är ågenomgdd
46          while(low<=high){
47              // Hitta öfrsta elementet som är östrre än eller lika med pivot
48              while(v[low] < pivot)
49                  low++;
50              // Hitta sista elementet som är mindre än eller lika med pivot
51              while(v[high] > pivot)
52                  high--;
53              // Avbryt om alla element redan är hittade
54              if(low>high)
55                  break;
56
57              //Dessa åtv element ska byta plats
58              int tmp = v[low];
59              v[low] = v[high];
60              v[high] = tmp;
61              low++;
62              high--;
63          }
64          // Returnera index till mitten
65          return low-1;
```

```
66      }
67
68      protected int getPivot(int[] v, int first, int last){
69          return v[first + (last-first)/2];
70      }
71  }
```

A.5 QuicksortJava.java

```
1  import java.util.Arrays;
2
3  /**
4   * @author lemming
5   * Wrapper till javas Arrays.sort
6   */
7  public class QuicksortJava implements Sorting {
8      public void sort(int[] v){
9          Arrays.sort(v);
10     }
11 }
```

A.6 QuicksortRandomPivot.java

```
1  import java.util.Random;
2
3  /**
4   * @author lemming
5   * Quicksort med slumpad pivot
6   */
7  public class QuicksortRandomPivot extends Quicksort{
8      private static Random rand = new Random();
9
10     /**
11      * Genererar slumpad pivot åfrn listan.
12      */
13     protected int getPivot(int[] v, int first, int last){
14         return v[rand.nextInt(last-first+1)+first];
15     }
16 }
```

A.7 Sorting.java

```
1  /**
2   * @author lemming
3   * Interface öfr algoritmer som kan sortera arrayer av int.
4   */
5  public interface Sorting {
6      public void sort(int[] v);
7  }
```

A.8 SortingTest.java

```
1  import junit.framework.TestCase;
2  import java.util.Arrays;
3  import java.util.Random;
4
5  /**
```

```
 * @author lemming
 * Testkod öfr flera sorteringsalgoritmer
 */
public class SortingTest extends TestCase {

    // Antalet sorterande algoritmer
    private static final int algs = 6;

    // Sorterande algoritmer
    private Sorting sorters[] = {
        new QuicksortJava(),
        new Quicksort(),
        new QuicksortRandomPivot(),
        new QuicksortInsertion(),
        new QuicksortInsertionRandom(),
        new InsertionSort()};

    // Namn åp sorterande algoritmer, öfr debuggingä–åndaml (tm)
    private String sortingName[] =
        {"Java Quicksort implementation",
        "Quicksort",
        "Quicksort with random pivot",
        "Quicksort with insertion sort",
        "Quicksort with insertion sort and random pivot",
        "Insertion sort"};

    // Listor av tal, en öfr varje algoritm
    private int numbers[][] = new int[algs][];

    private static Random rand = new Random();

    /**
     * Tests how the algorithms handle the empty list
     */
    public void testEmptyList(){
        numbers[0] = new int[0];

        Test();
    }

    /**
     * Tests how the algorithms handle a list with a single element.
     */
    public void testSingleElementList(){

        numbers[0] = new int[1];
        numbers[0][0] = 42;

        Test();
    }

    /**
     * Tests how the algorithms handle a list with two sorted elements.
     */
    public void testTwoAscendingElements(){
        numbers[0] = new int[2];
```

```
62          numbers[0][0] = 5;
63          numbers[0][1] = 10;
64
65          Test();
66      }
67
68      /**
69       * Tests how the algorithms handle a list with three elements.
70       */
71      public void testThreeElements(){
72          numbers[0] = new int[3];
73          numbers[0][0] = 10;
74          numbers[0][1] = 5;
75          numbers[0][2] = 7;
76
77          Test();
78      }
79
80      /**
81       * Tests how the algorithms handle a list with two unsorted elements.
82       */
83      public void testTwoDescendingElements(){
84          numbers[0] = new int[2];
85          numbers[0][0] = 10;
86          numbers[0][1] = 5;
87
88          Test();
89      }
90
91      /**
92       * Tests how the algorithms handle a large list with pseudorandom
             elements.
93       */
94      public void testRandomList(){
95          numbers[0] = new int[4096];
96
97          for(int i = 0; i < 4096; ++i)
98              numbers[0][i] = rand.nextInt();
99
100         Test();
101     }
102
103     /**
104      * Tests how the algorithms handle an already sorted list
105      */
106     public void testAscendingList(){
107
108         numbers[0] = new int[4096];
109
110         for(int i = 0; i < 4096; ++i)
111             numbers[0][i] = i;
112
113         Test();
114     }
115
116     /**
```

```
117         * Tests how the algorithms handle a reversely sorted list
118         */
119        public void testDescendingList(){
120            numbers[0] = new int[4096];
121
122            for(int i = 0; i < 4096; ++i)
123                numbers[0][i] = 4096-i;
124
125            Test();
126        }
127
128        /**
129         * Tests how the algorithms handle a list with equal elements
130         */
131        public void testSameList(){
132            numbers[0] = new int[4096];
133
134            for(int i = 0; i < 4096; ++i)
135                numbers[0][i] = 42;
136
137            Test();
138        }
139
140        /**
141         * Runs test on the list.
142         */
143        private void Test(){
144            // Copy list for all the algorithms
145            copyArrays();
146
147            // Let all algorithms sort their own list
148            sort();
149
150            // Assert that all arrays are equal, it means that they're
151            // sorted, as both Java's implementation and the insertion
152            // sort is assumed to work, or atleast differ in case either
153            // give the wrong answer
154            equalArrays();
155        }
156
157        /**
158         * Runs sort for each algorithm on their list
159         */
160        private void sort(){
161
162            for(int i = 0; i < algs; i++)
163                sorters[i].sort(numbers[i]);
164        }
165
166        /**
167         * Asserts equality between the arrays, to be run after sorting
168         */
169        private void equalArrays(){
170            for(int i = 1; i < algs; i++)
171                assertTrue(sortingName[i], Arrays.equals(numbers[0],
                        numbers[i]));
```

```
172     }
173
174     /**
175      * Copies the first list to the others, to give equal lists for test
             cases
176      */
177     private void copyArrays(){
178         for(int i = 1; i < algs; i++){
179             numbers[i] = new int[numbers[0].length];
180             System.arraycopy(numbers[0], 0, numbers[i], 0,
                    numbers[0].length);
181         }
182     }
183 }
```

A.9 SortingTimer.java

```
1  import java.util.Random;
2
3  /**
4   * @author lemming
5   * Timer class for different sorting algorithms, program induces JIT,
6   * performs different test cases and print out the results and how
7   * they compare to the Java implementation Arrays.sort(v);
8   */
9  public class SortingTimer {
10     // Antalet algoritmer som finns att testa med
11     private static final int algs = 5;
12
13     // Sorteringsalgoritmer
14     private static Sorting sorters[] = {
15         new QuicksortJava(),
16         new Quicksort(),
17         new QuicksortRandomPivot(),
18         new QuicksortInsertion(),
19         new QuicksortInsertionRandom()/*,
20         new InsertionSort()*/};
21
22     // Namn öfr algoritmerna, öfr att identifiera tidtagningarna
23     private static String sortingName[] =
24         {"Java Quicksort implementation",
25         "Quicksort",
26         "Quicksort with random pivot",
27         "Quicksort with insertion sort",
28         "Quicksort with insertion sort and random pivot"/*,
29         "Insertion sort"()*/};
30
31     // Listor öfr talen som ska sorteras, en lista per algoritm
32     private static int numbers[][][] = new int[algs][];
33
34     // Slumpgenerator
35     private static Random rand = new Random();
36
37     /**
38      * Kopierar öfrsta listan till resten av listorna.
39      */
```

```
40      private static void copyArrays(){
41          for(int i = 1; i < algs; i++){
42              numbers[i] = new int[numbers[0].length];
43              System.arraycopy(numbers[0], 0, numbers[i], 0,
                    numbers[0].length);
44          }
45      }
46
47      /**
48       * Sorteringsprogrammet, skapar olika listtyper och ser till att de
             testas.
49       * @param args Unused
50       */
51      public static void main(String[] args){
52
53          // öKr alla sorteringsalgoritmer åp stora slumpade listor öfr att
                tvinga fram JIT
54          System.out.println("Inducing JIT compilation through repeatedly
                calling sort on random elements:");
55
56          for(int i = 0; i < 5; ++i){
57
58              numbers[0] = new int[1000000];
59
60              for(int j = 0; j < 1000000; ++j){
61                  numbers[0][j] = rand.nextInt();
62              }
63
64              copyArrays();
65              sort();
66          }
67
68          // JIT klar, starta testning
69          System.out.println("\nStarting test!\n");
70
71          // äRttsorterad lista [0, 1, ..., n-1]
72          System.out.println("\nSorted list, 1000000 elements:\n");
73          numbers[0] = new int[1000000];
74
75          for(int j = 0; j < 1000000; ++j){
76              numbers[0][j] = j;
77          }
78
79          copyArrays();
80          sort();
81
82          // Felsorterad lista [n, n-1, ..., 1]
83          System.out.println("\nReversely sorted list, 1000000 elements:\n");
84          numbers[0] = new int[1000000];
85
86          for(int j = 0; j < 1000000; ++j){
87              numbers[0][j] = 1000000-j;
88          }
89
90          copyArrays();
91          sort();
```

```
92
93          // Blandad lista som ser ut som åtv ihopslagna listor.
94          // v[0, 2...] ökar medan v[1, 3...] minskar
95          System.out.println("\nMixed sorted list, 1000000 elements:\n");
96          numbers[0] = new int[1000000];
97
98          for(int j = 0; j < 1000000; ++j){
99              if((j & 1) == 0) numbers[0][j] = j;
100             else numbers[0][j] = 1000000 - j;
101         }
102
103         copyArrays();
104         sort();
105
106         // Lista med ekvivalenta element
107         System.out.println("\nSame value, 1000000 elements:\n");
108         numbers[0] = new int[1000000];
109
110         for(int j = 0; j < 1000000; ++j){
111             numbers[0][j] = 42;
112         }
113
114         copyArrays();
115         sort();
116
117         // Lista med åtv olika ävrden
118         System.out.println("\nEttnoll, 1000000 elements:\n");
119         numbers[0] = new int[1000000];
120
121         for(int j = 0; j < 1000000; ++j){
122             numbers[0][j] = rand.nextInt(2);
123         }
124
125         copyArrays();
126         sort();
127
128         //Lista med 10 olika ävrden [0, 1, ... 9]
129         System.out.println("\nMOD 10, 1000000 elements:\n");
130         numbers[0] = new int[1000000];
131
132         for(int j = 0; j < 1000000; ++j){
133             numbers[0][j] = rand.nextInt(10);
134         }
135
136         copyArrays();
137         sort();
138
139         // Lista med 1000 olika ävrden
140         System.out.println("\nMOD 1000, 1000000 elements:\n");
141         numbers[0] = new int[1000000];
142
143         for(int j = 0; j < 1000000; ++j){
144             numbers[0][j] = rand.nextInt(1000);
145         }
146
147         copyArrays();
```

```
148         sort();
149
150         // Slumpad lista, 1 000 element
151         System.out.println("\nRandom list, 1 000 elements:\n");
152         numbers[0] = new int[1000];
153
154         for(int j = 0; j < 1000; ++j){
155             numbers[0][j] = rand.nextInt();
156         }
157
158         copyArrays();
159         sort();
160
161         // Slumpad lista 10 000 element
162         System.out.println("\nRandom list, 10 000 elements:\n");
163         numbers[0] = new int[10000];
164
165         for(int j = 0; j < 10000; ++j){
166             numbers[0][j] = rand.nextInt();
167         }
168
169         copyArrays();
170         sort();
171
172         // Slumpad lista 100 000 element
173         System.out.println("\nRandom list, 100 000 elements:\n");
174         numbers[0] = new int[100000];
175
176         for(int j = 0; j < 100000; ++j){
177             numbers[0][j] = rand.nextInt();
178         }
179
180         copyArrays();
181         sort();
182
183         // Slumpad lista 1 000 000 element
184         System.out.println("\nRandom list, 1 000 000 elements:\n");
185         numbers[0] = new int[1000000];
186
187         for(int j = 0; j < 1000000; ++j){
188             numbers[0][j] = rand.nextInt();
189         }
190
191         copyArrays();
192         sort();
193     }
194
195     /**
196      * Sorterar varje med varje algoritm, skriver ut tid och hur de ääjmfr
            sig
197      * med javas Arrays.sort() (algoritm #0)
198      */
199     private static void sort(){
200         Stopwatch s = new Stopwatch();
201
202         s.reset();
```

```
203             s.start();
204             sorters[0].sort(numbers[0]);
205             s.stop();
206             System.out.println(sortingName[0] + ": " + s.toString());
207 
208             long time = s.getTime();
209 
210             for(int j = 1; j < algs; ++j){
211                 s.reset();
212                 s.start();
213                 sorters[j].sort(numbers[j]);
214                 s.stop();
215                 String ratio = (time == 0) ? "?" :
                        Double.toString((((double)s.getTime())/time));
216                 System.out.println(sortingName[j] + ": " + s.toString() + " -
                        " + ratio);
217             }
218         }
219 }
```