Martin Olsen

DM507 Projekt 2012

Del I

19. marts 2012

FOTO: Colourbox

Indhold

Indledning

Dette projekt omhandler optælling af inversioner i en liste. En inversion kan beskrives som følger: Givet en liste L er parret $(L[i], L[j])$ en inversion hvis og kun hvis $i < j \wedge L[i] > L[j]$. Ingen har oplyst formået med optællingen, men hvad … Vi tæller løs alligevel med hjælp fra det forfærdelige sprog Java. **NB: I hele denne rapport benyttes nulbaseret indeksering.**

Opgave 1

Implementeringen blev fuldført som dokumenteret i appendikset.

Opgave 2

Implementeringen blev fuldført som dokumenteret i appendikset.

Opgave 3

Vi vil tælle antallet z af inversioner mellem de to delarrays `A[p..q]` og `A[q+1..r]` ved at tilføje en smule kode til `Merge`-proceduren. Bemærk, at de to delarrays kopieres over i to nye arrays, hhv. `L` og `R`, under tilføjelse af ∞ som sidste element i hvert array. Pseudokoden ser således ud:

```
1  Merge(A, p, q, r) {
2    Copy subarray A[p..q] to array L, adding infinity as last element
3    Copy subarray A[q+1..r] to array R, adding infinity as last element
4
5    i = 0
6    j = 0
7
8    for k = p to r
9      if L[i] <= R[j]
10       A[k] = L[i]
11       i = i + 1
12     else
13       A[k] = R[j]
14       j = j + 1
15 }
```

Ideen er at tælle antallet af inversioner, et givet element i `L` eller `R` indgår i, hver gang vi, som en del af sorteringsprocessen, kopierer dette element tilbage til `A`. På denne måde vil hvert element i de to arrays (undtagen de sidste, som er ∞, og som ikke kopieres over), blive behandlet præcis en gang. Det er værd at notere, at når vi tæller antallet af inversioner, et givet element indgår i, er det kun de af inversionerne, der stammer fra elementer, der ikke allerede er kopieret tilbage til `A`, vi skal tælle med; en inversion mellem det givne element og et element, der allerede er kopieret over, vil allerede være talt med en gang. Hvis vi kopierer et element fra `L` over, vil dette element være mindre end eller lig samtlige de elementer i `R`, der endnu ikke er kopieret tilbage til `A`. Da elementet fra `L` samtidig vil have et lavere indeks end nogen af disse, kan elementet, jf. definitionen på en inversion, ikke indgå i en inversion med noget sådant element i `R`. Vi skal altså intet foretage os, hvis vi kopierer et element fra `L` tilbage. Kopierer vi derimod et element fra `R` tilbage, vil dette element være mindre end samtlige elementer fra `L`, der ikke er kopieret tilbage til `A`. Bemærk, at elementet vil være skarpt mindre end samtlige førnævnte elementer. Dette skyldes, at ud af to elementer med samme størrelse, vil det altid være elementet fra `L`, der først tilbagekopieres. Når der tilbagekopieres et element fra `R`, vil alle elementer af samme størrelse i `L` altså allerede være blevet kopieret. Da elementet fra

R vil være mindre end alle de endnu ikke tilbagekopierede elementer i L, men samtidig have et højere indeks, må elementet indgå i inversioner med samtlige disse elementer. Disse er der på et givet tidspunkt netop $p - q - i + 1$ af; det mindste element fra L, der ikke er kopieret tilbage til A må have indeks i, mens det andensidste element i L (idet ∞ ikke skal tælles med), må have indeks L.length-2. Vi får således, at $L.length - i - 1$ angiver antallet af elementer i delarrayet L[i..L.length-2]. Eftersom samtlige elementer fra delarrayet A[p..q] blev kopieret over i L, hvorefter elementet ∞ blev tilføjet, må længden af L være $q - p + 2$, hvorfor førnævnte antal af elementer fås til $q - p - i + 1$. Til sidst akkumuleres inversionerne i variablen zCounter, hvorefter værdien af denne returneres. Pseudokoden ser ud som følger:

```
1  MergeNInvCount(A, p, q, r) {
2    Copy subarray A[p..q] to array L, adding infinity as last element
3    Copy subarray A[q+1..r] to array R, adding infinity as last element
4
5    i = 0
6    j = 0
7
8    zCounter = 0
9
10   for k = p to r
11     if L[i] <= R[j]
12       A[k] = L[i]
13       i = i + 1
14     else
15       zCounter = zCounter + q - p - i + 1
16       A[k] = R[j]
17       j = j + 1
18
19   return zCounter
20 }
```

Opgave 4

Beviset for algoritmens korrekthed forudsætter, at $p \leq q \leq r$. Følgende invariant anvendes til at vise, at antallet af inversioner udregnes korrekt:

Efter hver iteration af forløkken i linje 10, vil zCounter være lig z fratrukket antallet af inversioner mellem de to delarrays L[i..L.length-2] og R[j..R.length-2].

Efter iteration 0, dvs. før nogen iterationer har fundet sted, er invarianten tydeligvis sand; idet $i = j = 0$, får vi de to delarrays L[0..L.length-2] og R[0..R.length-2]. Disse svarer blot til L og R, de to elementer med værdien ∞ fraregnet, og da z netop var antallet af inversioner mellem disse, må resultatet blive 0. Værdien for zCounter er altså i dette tilfælde korrekt. Vi viser nu, at hvis invarianten er sand efter iteration $n - 1$, vil den også være sand efter iteration n. Iteration n har to mulige udfald: Enten vil betingelsen L[i] <= R[j] i linje 11 være sand, og linje 12 og 13 vil blive udført, eller også vil betingelsen være falsk, og linje 15-17 vil udføres. Hvis betingelsen er sand, vil L[i] være mindre end eller lig R[j] og ditto for alle elementer i R med højere indeks end j. Derfor kan L[i] ikke indgå i en inversion med noget element i R[j..R.length-2]. Vi kan derfor trygt inkrementere i i linje 13, så elementet L[i] forsvinder fra L[i..L.length-2], uden at tælle zCounter op. Havde tælleren den rigtige værdi før iterationen, vil den også have den rigtige værdi efter. Er betingelsen i linje 11 derimod ikke sand, vil R[j] være mindre end L[i] og ditto for alle elementer i L med indeks højere end i. R[j] vil derfor indgå i inversioner med samtlige

elementer i `L[i..L.length-2]`. Når vi fjerner `R[j]` fra `R[j..R.length-2]` ved at inkrementere `j`, skal vi derfor sørge for, at inversionerne mellem `R[j]` og elementerne i `L[i..L.length-2]` ikke længere fratrækkes `zCounter`, dvs. `zCounter` skal tælles op med antallet af disse inversioner. Dette sker i linje 15 efter fremgangsmåden beskrevet i forrige afsnit. Igen har vi, at hvis tælleren havde den rigtige værdi før iterationen, vil den også have den rigtige værdi efter. Løkken terminerer, når $k = r + 1$, altså efter netop $r - p + 1$ gennemløb. Da antallet af elementer i `L` er $q - p + 2$, og antallet i `R` er $r - q + 1$, bliver det samlede antal elementer i de to delarrays $r - p + 3$. Da de to elementer med værdien ∞ er størst, vil det være disse, der ikke behandles. Når løkken terminerer, må `L[i]` og `R[j]` derfor begge have værdien ∞. Disse værdier findes på indekserne `L.length-1` og `R.length-1`. Fra invarianten ved vi, at `zCounter` nu er z fratrukket antallet af inversioner mellem de to delarrays `L[L.length-1..L.length-2]` og `R[R.length-1..R.length-2]`. Da venstresiden for begge disse er større end højresiden, må de være tomme. Der vil derfor ingen inversioner være mellem dem, og `zCounter` får derfor, helt korrekt, værdien z, som var det totale antal inversioner mellem de to delarrays. Vi kan altså konkludere, at værdien returneret i linje 19, er korrekt.

`MergeNInvCount` indeholder en vis konstant mængde arbejde samt en løkke, også indeholdende konstant arbejde, der gennemløbes $n = r - p + 1$ gange. Antallet af gennemløb svarer netop til antallet af elementer i de to delarrays `A[p..q]` og `A[q+1..r]`, så det totale arbejde som funktion f af inputstørrelsen n er givet ved $f(n) = cn + k = \Theta(n)$. Algoritmen kører altså i linear tid.

Opgave 5

Algoritmen `MergeSortNInvCount` baseres på merge sort og den udvidede `Merge`-procedure, `MergeNInvCount`. Som input lader vi algoritmen tage et array `A` samt to indekser `p` og `r` angivende første hhv. sidste element i den del af arrayet, algoritmen skal tælle inversioner i. Som med den klassiske `Merge`-procedure vil vi lade algoritmen opdele `A[p..r]` i to lige store dele (plus/minus et element, hvis længden er ulige). Antallet af inversioner i `A[p..r]` må så bestå af antallet af inversioner i de to delarrays samt antallet af inversioner hen over midten. Det er værd at bemærke, at dette antal ikke ændrer sig, selvom de to delarrays sorteres. Dette er let at indse; en sortering ændrer ikke på, hvilket element i en inversion, der har det laveste indeks, idet ingen af elementerne kan krydse grænsen mellem de to delarrays. Da en sortering ydermere ikke ændrer elementernes værdi, må antallet af inversioner hen over midten være det samme som før sorteringen. Ud over antallet af inversioner over midten, er det også nødvendigt at vide hvor mange inversioner, der er i hvert af de to delarrays. Antallet af disse kan findes med to rekursive kald af `MergeSortNInvCount` med indekserne på første og sidste element i det relevante delarray som input. Til sidst er blot tilbage at addere værdierne returneret af disse kald med værdien returneret af `MergeNInvCount` og derefter returnere summen. Kalder vi summen `invs`, kommer pseudokoden til at se ud som følger:

```
1  MergeSortNInvCount(A, p, r) {
2    invs = 0
3
4    if p < r
5      q = floor((p+r) / 2)
6      invs = invs + MergeSortNInvCount(A, p, q)
7      invs = invs + MergeSortNInvCount(A, q+1, r)
8      invs = invs + MergeNInvCount(A, p, q, r)
9
10   return invs
11 }
```

Opgave 6

Beviset for korrektheden af den ovenfor beskrevne algoritme føres ved induktion over antallet $n = r - p + 1$ af elementer i `A`, der behandles af algoritmen. Vi vil benævne dette antal inputstørrelsen. Det vil i beviset forudsættes, at $p \leq r$. Vi viser først, at algoritmen kører korrekt for basistilfældet $n = 1 \Rightarrow p = r$. Idet linje 5-8 i dette tilfælde ikke vil udføres, vil værdien af `invs`, der i linje 2 blev sat til 0, ikke ændres. Det vil derfor være denne værdi, der bliver returneret i linje 10, hvilket er korrekt, da delarrays med længder på 1 ingen inversioner indeholder. Bemærk, at argumentet også viser, at algoritmen terminerer for $n = 1$. Vi viser nu, at algoritmen ligeledes kører korrekt for alle $n > 1$. Antag, at algoritmen kører korrekt for inputstørrelserne $n \in \{1,2,3,\dots,k-1\}$. Vi skal så vise, at algoritmen også kører korrekt for $n = k$. Vi ser, at linje 5-8 i dette tilfælde vil blive udført, da $n > 1 \Rightarrow p < r$. I linje 5 sættes $q = \left\lfloor \frac{p+r}{2} \right\rfloor$. Dermed bliver inputstørrelsen ξ_k for det i linje 6, af instansen med inputstørrelsen k, udførte rekursive kald $\xi_k = q - p + 1 = \left\lfloor \frac{p+r}{2} \right\rfloor - p + 1$. Idet vi har, at $k = r - p + 1$, får vi, at

$$\frac{k}{2} = \frac{r-p+1}{2} = \frac{p+r}{2} - p + 1 - \frac{1}{2} \leq \xi_k \leq \frac{p+r}{2} - p + 1 = \frac{r-p+2}{2} = \frac{k+1}{2}.$$

Da vi desuden ved, at $k > 1 \Leftrightarrow k + 1 < 2k$, må vi ligeledes have, at

$$0 < \frac{k}{2} \leq \xi_k \leq \frac{k+1}{2} < k.$$

Vi når altså aldrig ned på inputstørrelser, der ikke dækkes af basistilfældet. For det rekursive kald i linje 7 har vi inputstørrelsen $\chi_k = r - q = r - \left\lfloor \frac{p+r}{2} \right\rfloor$. Vi får så følgende:

$$\frac{k-1}{2} = \frac{r-p}{2} = r - \frac{p+r}{2} \leq \chi_k \leq r - \frac{p+r}{2} + \frac{1}{2} = \frac{r-p+1}{2} = \frac{k}{2}.$$

Hvis $k > 1$, får vi desuden, at

$$0 < \frac{k-1}{2} \leq \chi_k \leq \frac{k}{2} < k.$$

Igen ses det, at inputstørrelserne vil være dækket af basistilfældet. Jf. induktionsantagelsen returnerer de rekursive kald i linje 6 og 7 altså korrekte værdier. Ud over dette sorterer de også de to delarrays, de opererer på, men som argumenteret for tidligere, ændrer dette ikke ved antallet af inversioner over midten. Dette antal gøres i næste linje op af `MergeNInvCount`, hvis korrekthed allerede er bevist. Da de tre kald i linje 6-8 returnerer hhv. antallet af inversioner i de to delarrays `A[p..q]` og `A[q+1..r]` samt antallet af inversioner mellem disse, og eftersom disse tal akkumuleres i `invs`, der returneres i linje 10, må den returnerede værdi være korrekt, såfremt algoritmen terminerer. At dette er tilfældet ses let af ovenstående ligninger, idet længden af de delarrays, der behandles i hvert rekursivt kald, falder, indtil basistilfældet nås.

Vi anvender mastersætningen til at bestemme køretiden. Rekursionsligningen er nøjagtig identisk med ditto for merge sort: $T(n) = 2T\left(\frac{n}{2}\right) + cn$. Vi har to rekursive kald af (omtrentlig) halv størrelse samt en mængde lineart arbejde fra `MergeNInvCount`. Ifølge mastersætningen gælder, at givet

rekursionsligningen $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ vil $f(n) = \Theta\left(n^{\log_b a}\right) \Rightarrow T(n) = \Theta\left(n^{\log_b a} \lg n\right)$. I vores tilfælde er $a = b = 2$ og $f(n) = cn$, hvorfor ovenstående netop er tilfældet, idet vi har, at $f(n) = cn = \Theta(n) = \Theta\left(n^{\log_2 2}\right)$. Dermed fås, at $T(n) = \Theta(n \lg n)$.

Opgave 7

Implementeringen blev fuldført som dokumenteret i appendikset.

FOTO: Sony Music Entertainment

Appendiks

Kildekode til `SimpleInv.java`

```
1  import java.io.*;
2  import java.util.*;
3
4  public class SimpleInv {
5    public static void main(String[] args) throws FileNotFoundException {
6      Scanner sc = new Scanner(new File(args[0]));
7      List<Integer> intArray = new ArrayList<Integer>();
8      int zCounter = 0; // number of inversions
9
10     /* convert input file to list of integers */
11     while(sc.hasNextInt()) {
12       intArray.add(sc.nextInt());
13     }
14
15     /* count number of inversions */
16     for(int i = 0; i < intArray.size() - 1; i++) {
17       for(int j = i + 1; j < intArray.size(); j++) {
18         if(intArray.get(i) > intArray.get(j)) {
19           zCounter++;
20         }
21       }
22     }
23
24     System.out.println(zCounter);
25   }
26 }
```

Kildekode til `MergeSort.java`

```
1  import java.io.*;
2  import java.util.*;
3
4  public class MergeSort {
5    private static void Merge(List<Integer> A, int p, int q, int r) {
6      int n1 = q - p + 1; // number of elements in A[p..q]
7      int n2 = r - q; // number of elements in A[q+1..r]
8      int[] L = new int[n1 + 1]; // copy of A[p..q] with last element being
   infinity
9      int[] R = new int[n2 + 1]; // copy of A[q+1..r] with last element being
   infinity
10
11     /* copy A[p..q] to subarray L[0..n1-1] */
12     for(int i = 0; i < n1; i++) {
13       L[i] = A.get(p + i);
14     }
15
16     /* copy A[q+1..r] to subarray R[0..n2-1] */
17     for(int j = 0; j < n2; j++) {
18       R[j] = A.get(q + j + 1);
19     }
20
21     /* add infinity as last element of L and R */
22     L[n1] = Integer.MAX_VALUE;
23     R[n2] = Integer.MAX_VALUE;
24
25     int i = 0;
26     int j = 0;
```

```
27
28      /* merge L and R into A */
29      for(int k = p; k <= r; k++) {
30        if(L[i] <= R[j]) {
31          A.set(k, L[i]);
32          i++;
33        } else {
34          A.set(k, R[j]);
35          j++;
36        }
37      }
38    }
39
40    private static void MergeSort(List<Integer> A, int p, int r) {
41      if(p < r) {
42        int q = (p+r) / 2;
43        MergeSort(A, p, q);
44        MergeSort(A, q+1, r);
45        Merge(A, p, q, r);
46      }
47    }
48
49    public static void main(String[] args) throws FileNotFoundException {
50      Scanner sc = new Scanner(new File(args[0]));
51      List<Integer> intArray = new ArrayList<Integer>();
52
53      /* convert input file to list of integers */
54      while(sc.hasNextInt()) {
55        intArray.add(sc.nextInt());
56      }
57
58      MergeSort(intArray, 0, intArray.size() - 1);
59
60      /* build string of sorted numbers separated by whitespace */
61      StringBuilder sb = new StringBuilder();
62      for (int i : intArray) {
63        sb.append(i);
64        sb.append(" ");
65      }
66
67      System.out.println(sb.toString().trim());
68    }
69  }
```

Kildekode til `FastInv.java`

```
1   import java.io.*;
2   import java.util.*;
3
4   public class FastInv {
5     private static int MergeNInvCount(List<Integer> A, int p, int q, int r) {
6       int n1 = q - p + 1; // number of elements in A[p..q]
7       int n2 = r - q; // number of elements in A[q+1..r]
8       int[] L = new int[n1 + 1]; // copy of A[p..q] with last element being
    infinity
9       int[] R = new int[n2 + 1]; // copy of A[q+1..r] with last element being
    infinity
10
11      /* copy A[p..q] to subarray L[0..n1-1] */
12      for(int i = 0; i < n1; i++) {
13        L[i] = A.get(p + i);
14      }
15
16      /* copy A[q+1..r] to subarray R[0..n2-1] */
```

```
17      for(int j = 0; j < n2; j++) {
18        R[j] = A.get(q + j + 1);
19      }
20
21      /* add infinity as last element of L and R */
22      L[n1] = Integer.MAX_VALUE;
23      R[n2] = Integer.MAX_VALUE;
24
25      int i = 0;
26      int j = 0;
27
28      int zCounter = 0; // number of inversions
29
30      /* merge L and R into A and count number of inversions from L to R */
31      for(int k = p; k <= r; k++) {
32        if(L[i] <= R[j]) {
33          A.set(k, L[i]);
34          i++;
35        } else {
36          zCounter += q - p - i + 1;
37          A.set(k, R[j]);
38          j++;
39        }
40      }
41
42      return zCounter;
43    }
44
45    private static int MergeSortNInvCount(List<Integer> A, int p, int r) {
46      int invs = 0;
47
48      if(p < r) {
49        int q = (p+r) / 2;
50        invs += MergeSortNInvCount(A, p, q);
51        invs += MergeSortNInvCount(A, q+1, r);
52        invs += MergeNInvCount(A, p, q, r);
53      }
54
55      return invs;
56    }
57
58    public static void main(String[] args) throws FileNotFoundException {
59      Scanner sc = new Scanner(new File(args[0]));
60      List<Integer> intArray = new ArrayList<Integer>();
61
62      /* convert input file to list of integers */
63      while(sc.hasNextInt()) {
64        intArray.add(sc.nextInt());
65      }
66
67      int invs = MergeSortNInvCount(intArray, 0, intArray.size() - 1);
68
69      System.out.println(invs);
70    }
71  }
```