

Föreläsning 11

Datastrukturer (DAT037)

Fredrik Lindblad¹

4 december 2017

¹Slides skapade av Nils Anders Danielsson har använts som utgångspunkt. Se <http://www.cse.chalmers.se/edu/year/2015/course/DAT037>

Innehåll

- ▶ sortering
 - ▶ Urvalssortering
 - ▶ Insättningssortering
 - ▶ Mergesort
 - ▶ Quicksort
 - ▶ Heapsort

Sortering

Sortering

- ▶ Inte en datastruktur, men ...
- ▶ Vanligt förekommande uppgift att sortera listor. Välstuderat. Finns i varje standardbibliotek.
- ▶ *stabil* sortering = element som är lika byter inte plats
- ▶ *in-place* sortering = konstant extra minnesanvändning (sorting sker på plats)
- ▶ Generisk sorteringsmetod i Java: använd `Comparable` eller `Comparator`.

Enkla sorteringsalgoritmer

Löser problemet genom att ta ett element (ett par) i taget.

- ▶ Urvalssortering (selection sort)
- ▶ Insättningsortering (insertion sort)
- ▶ Bubble sort

Urvals- sortering

Urvalssortering

Välj minsta bland resterande element och sätt sist i växande lista. Denna sorterade lista innehåller de i minsta elementen.

```
public static void selectionSort(int[] x) {  
    for (int i = 0; i < x.length - 1; i++) {  
        int minidx = i, minval = x[i];  
        for (int j = i + 1; j < x.length; j++) {  
            if (x[j] < minval) {  
                minval = x[j]; minidx = j;  
            }  
        }  
        x[minidx] = x[i]; x[i] = minval;  
    }  
}
```

Urvalssortering

- ▶ Ej stabil
- ▶ In-place
- ▶ Tidskomplexitet: $O(n^2)$
- ▶ Antal skrivningar i arrayen: $O(n)$

Insättnings- sortering

Insättningsortering

Sätt in varje element på rätt plats i växande lista. Denna lista är sorterad och innehåller de i första elementen.

```
public static void insertionSort(int[] x) {
    for (int i = 1; i < x.length; i++) {
        int j, tmp = x[i];
        for (j = i; j > 0 && x[j-1] > tmp; j--) {
            x[j] = x[j-1];
        }
        x[j] = tmp;
    }
}
```

Insättningsortering

- ▶ Stabil
- ▶ In-place
- ▶ (Värstafalls)komplexitet: $O(n^2)$
- ▶ Bästafallskomplexitet: $O(n)$, vid sorterad lista.
- ▶ Antal skrivningar i arrayen (i värsta fall): $O(n^2)$

Tidskomplexitet

Insättningsortering och andra enkla algoritmer (som jämför element parvis) har värstafallskomplexiteten $O(n^2)$. Men det går att sortera mer effektivt.

Divide and conquer

Divide and conquer

- ▶ Algoritmteknik: söndra och härska (divide and conquer).
- ▶ Om man skapar två delproblem av halv storlek på linjär tid, och dessutom slår ihop delresultaten på linjär tid, så får man en $O(n \log n)$ -algoritm.

Mergesort

Mergesort

- ▶ Dela listan i två halvor.
- ▶ Sortera varje halva för sig genom rekursivt anrop.
- ▶ Slå ihop (merge) de två halvorna.

Mergesort

```
public static void mergeSort(int[] x) {
    int[] tmp = new int[x.length];
    msort(x, tmp, 0, x.length - 1);
}

private static void msort(int[] x, int[] tmp,
                          int l, int r) {
    if (r <= l) return;
    int m = l + (r - l) / 2;
    msort(x, tmp, l, m);
    msort(x, tmp, m + 1, r);
    merge(x, tmp, l, m+1, r);
}
```

Mergesort

```
private static void merge(int[] x, int[] tmp,
                          int l1, int l2, int r2) {
    int r1 = l2 - 1, i = l1;

    while (l1 <= r1 && l2 <= r2) {
        if (x[l1] <= x[l2]) tmp[i++] = x[l1++];
            else tmp[i++] = x[l2++];
    }
    while (l1 <= r1)
        tmp[i++] = x[l1++];
    while (l2 <= r2)
        tmp[i++] = x[l2++];
    for (int j = l1; j <= r2; j++) {
        x[j] = tmp[j];
    }
}
```

Mergesort

Värstafallstidskomplexitet (om \leq tar konstant tid):

- ▶ merge: Linjär i listornas längd: $O(|xs| + |ys|)$.
- ▶ mergeSort:

$$T(0) = 1$$

$$T(1) = 1$$

$$T(n) = n + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right), \text{ om } n \geq 2$$

Mergesort

Anta att $n = 2^k$, $k \in \mathbb{N}$.

$$\begin{aligned}T(n) &= n + 2T(n/2) \\&= n + 2(n/2 + 2T(n/4)) \\&= 2n + 4T(n/4) \\&= 2n + 4(n/4 + 2T(n/8)) \\&= 3n + 8T(n/8) \\&= \dots \\&= \log_2 n \cdot n + nT(n/n) \\&= n \log_2 n + n \\&= \Theta(n \log n)\end{aligned}$$

Mergesort

- ▶ Stabil
- ▶ Ej in-place
- ▶ Man kan undvika att kopiera från tmp efter varje merge genom att alternera de två arrayerna.
- ▶ Mergesort kan också implementeras bottom-up. Börja med att sortera $n/2$ par av listor av längden 1, sedan $n/4$ par av listor av längden 2, osv.

Quicksort

Quicksort

- ▶ Också divide and conquer
- ▶ Jobbet utförs före rekursiva anropen istället för efter.
- ▶ Välj ett element, pivotelementet. Idealt är detta medianen.
- ▶ Fördela alla element så att de mindre än pivot kommer först och de större än pivot sist.
- ▶ Sortera delen med mindre element för sig och delen med större för sig.

Val av pivotelement / Partitioneringsstrategi

- ▶ Viktigt välja pivotelementet p på ett bra sätt, de två delarna ska helst vara ungefär lika stora.
- ▶ Rekommenderade val:
 - ▶ Slumpmässigt element. (Kan ta tid att få fram.)
 - ▶ Medianen av tre element:
första, sista och ett av de mittersta.
- ▶ Att ta första eller sista element som pivot är inte bra. Blir långsamt för sorterade eller nästan sorterade listor.
- ▶ Bör även ha bra strategi för element lika med p .

Quicksort

```
public static void quickSort(int[] x) {
    qsort(x, 0, x.length - 1);
}
private static void qsort(int[] x, int l, int r) {
    if (r <= l) return;
    int pidx = pickPivot(x, l, r);
    swap(x, pidx, r);
    int i = split(x, l, r - 1, x[r]);
    swap(x, i, r);
    qsort(x, l, i - 1);
    qsort(x, i + 1, r);
}
private static void swap(int[] x, int i, int j) {
    int tmp = x[i]; x[i] = x[j]; x[j] = tmp;
}
```

Quicksort

```
private static int split(int[] x, int l, int r,
                        int piv) {
    // returns first idx with bigger element
    for (;;) {
        while (l < r && x[l] < piv) l++;
        while (l < r && x[r] > piv) r--;
        if (l == r) break;
        swap(x, l++, r--);
        if (l == r + 1) return l;
    }
    return x[l] < piv ? l + 1 : l;
}

private static int pickPivot(int[] x, int l, int r) {
    // returns idx of selected pivot
    return l + (r - l) / 2; // simply picks middle elt
}
```

Tidskomplexitet:

- ▶ I värsta fallet (bara dåliga val av p): $\Theta(n^2)$.
- ▶ Bara bra val av p :
 $O(n \log n)$.
- ▶ Medelkomplexitet (med bra implementation):
 $O(n \log n)$.

Quicksort

- ▶ Svårt att få stabil. (Blir typisk ej in-place om man gör den stabil.)
- ▶ In-place (bortsett från utrymmet på anropsstacken som går åt för de rekursiva anropen).
- ▶ Mindre minnesomflyttning men fler jämförelser än Mergesort. Detta kan avgöra vilken man ska välja.

Heapsort

Prioritetskösortering

- ▶ Enkel sorteringsalgoritm: Sätt in varje element, kör delete-min tills kön är tom.
- ▶ $O(n \log n)$ om insert och delete-min har tidskomplexiteten $O(\log n)$.

Heapsort

- ▶ Variant av den enkla prioritetskösorтерingsalgoritmen.
- ▶ *In-place*: Kräver $O(1)$ extra minne.
- ▶ Ej stabil.

Heapsort

- ▶ Bygg binär maxheap in-place med `build-heap` (med roten på position 0).
- ▶ Kör `delete-max` upprepade gånger. Lägga största elementet sist i arrayen, nästa element näst sist, och så vidare.