

Programmering för språkteknologer II

OH-serie: Sökning och sortering



UPPSALA
UNIVERSITET

Mats Dahllöf

Institutionen för lingvistik och filologi
September 2009

1

Programmering för språkteknologer II — HT 2009 (Mats Dahllöf)

Sökning och sortering

- **Sökning:** lokalisera objekt i samlingar.
Finns ett visst värde? I så fall: var?
- **Sortering:** placera objekten i en samling i en viss ordning.
(Om en samling är sorterad blir sökning effektivare. Jfr t.ex. med ett bibliotek.)
- Extremt viktiga problem. Olika algoritmer för båda.

2

Programmering för språkteknologer II — HT 2009 (Mats Dahllöf)

Algoritm

- "en systematisk procedur som i ett ändligt antal steg anger hur man utför en beräkning eller löser ett givet problem" (NE)
- Man talar ofta om algoritmer på en abstraktionsnivå som tillåter olika sätt att implementera dem.

3

Programmering för språkteknologer II — HT 2009 (Mats Dahllöf)

Linjär sökning

- Extremt enkel algoritm.
- Samlingen behöver ej vara sorterad. Låt oss ta fält, som är den typ av samling man brukar titta på. (Varför?)
- Man går igenom dess platser från början till slut (eller till träff).
- Tidsödande. Hur?

4

Programmering för språkteknologer II — HT 2009 (Mats Dahllöf)

Kodexempel (en metod för linjär sökning)

```
Söka int-värde i fält av int (int []).  
public static int linSearch(int[] arr, int x) {  
    for (int i=0; i < arr.length; i++) {  
        if (arr[i]==x) {  
            return i;    // index för första träff  
        }  
    };  
    return -1;    // för "ingen träff"  
}                // (-1 omöjligt index)
```

5

Programmering för språkteknologer II — HT 2009 (Mats Dahllöf)

Linjär sökning: komplexitet (förenklat)

- Varje jämförelse har en viss tidskostnad.
- Man måste titta igenom hela samlingen för att veta att sökt objekt inte finns.
- Om objektet finns och givet vissa antaganden om slumpmässighet, så kan man tänka sig att medelvärdet för antalet jämförelser är lika med samlingens halva längd.

6

Programmering för språkteknologer II — HT 2009 (Mats Dahllöf)

Linjär komplexitet

Om en algoritms tidsåtgång T betar sig på det här sättet, så är den av linjär komplexitet.

- G en konstant grundkostnad.
 n inputs längd.
 k en kostnad per värde/jämförelse.
- $T = G + n * k$.
- Linjär sökning är ett exempel på en algoritm med linjär komplexitet.

7

Programmering för språkteknologer II — HT 2009 (Mats Dahllöf)

Binär sökning (lätt förenklat)

- Om vi har en ordnad del av ett fält kan vi eliminera halva delen för varje jämförelse med sökt värde.
- Vi tittar på mittvärdets storlek och ser om sökt värde finns före eller efter.
- Vi kan då söka igenom t.ex. ett fält med 1024 värden med 11 jämförelser. Antal kvar efter varje mittvärdesjämförelse (\rightarrow): 1024 \rightarrow 512 \rightarrow 256 \rightarrow 128 \rightarrow 64 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1 \rightarrow svar.
- Stor vinst jämfört med linjär sökning (vid större fält)!

8

Exempel sök 29 (target

Jämförelse: `arr[mid] > target` (mid "mittvärdet").

Ja: uteslut "högerdel" inkl. `arr[mid]`.

Nej: uteslut "vänsterdel" exkl. `arr[mid]`.

1 3 5 7 9 10 11 13 17 18 21 23 25 27 29 31 (nej)

1 3 5 7 9 10 11 13 17 18 21 23 25 27 29 31 (nej)

1 3 5 7 9 10 11 13 17 18 21 23 25 27 29 31 (nej)

1 3 5 7 9 10 11 13 17 18 21 23 25 27 29 31 (ja)

1 3 5 7 9 10 11 13 17 18 21 23 25 27 29 31

rätt, returnera index 14

9

Exempel (sök 12)

1 3 5 7 9 10 11 13 17 18 21 23 25 27 29 31

1 3 5 7 9 10 11 13 17 18 21 23 25 27 29 31

1 3 5 7 9 10 11 13 17 18 21 23 25 27 29 31

1 3 5 7 9 10 11 13 17 18 21 23 25 27 29 31

1 3 5 7 9 10 11 13 17 18 21 23 25 27 29 31

ingen träff, returnera -1 (för "ingen träff")

10

Logaritmisk komplexitet

Om en algoritms tidsåtgång T betar sig på det här sättet, så är den av logaritmisk komplexitet.

- G en konstant grundkostnad. n inputs längd.
 k en kostnad per jämförelse.
- $T = G + m * k$, om $n = 2^m$. ($T = G + 2^m \log n * k$).
Varje fördubbling av indatas längd ger en konstant tidkostnad. (Jfr. linjär: fördubbling av indata ger fördubblad kostnad.)

11

Olika instanser av samma problem

Binärsök i givet fält 12 mellan index 0 och 15:

1 3 5 7 9 10 11 13 17 18 21 23 25 27 29 31

Binärsök i givet fält 12 mellan index 0 och 7:

1 3 5 7 9 10 11 13 17 18 21 23 25 27 29 31

Binärsök i givet fält 12 mellan index 4 och 7:

1 3 5 7 9 10 11 13 17 18 21 23 25 27 29 31

...

Binärsök i givet fält 12 mellan index 6 och 8:

1 3 5 7 9 10 11 13 17 18 21 23 25 27 29 31

12

Kodexempel: binär sökning

- Skall visa två metoder att söka efter ett `int`-målvärde i ett `int[]`-fält mellan två index.
- Rekursiv implementation: Anropar samma metod (fast med snävare intervall).
- Icke-rekursiv implementation med `while`-upprepning.
- Lab: Ni skall parametriskera så att det fungerar för alla "ordningsbara" objekttyper.

13

Kodexempel: binär sökning, rekursiv

```
public static int binSearch(int[] arr, int target,
                           int low, int high) {
    if (high - low > 0) {
        int mid = (low+high+1)/2;
        if (arr[mid] > target) {
            return binSearch(arr, target, low, mid-1);
        } else {
            return binSearch(arr, target, mid, high);
        }
    } else {
        if (arr[low] == target) {return low;}
        else {return -1;}}
}
```

14

Kodexempel: binär sökning, while-upprepning

```
public static int binSearch2(int[] arr, int target,
                             int low, int high) {
    while (high - low > 0) {
        int mid = (low+high+1)/2;
        if (arr[mid] > target) {high = mid-1;}
        else {low = mid;}}
    if (arr[low] == target) {return low;}
    else {return -1;}}
```

15

Generisering/parametriskering av detta

- Bör avse objekttyper med en ordningsrelation.
- Standardmetod: `int compareTo(T o)`
- Gränssnittet `Comparable<T>` kräver att denna metod finns.

Man kan ställa krav på en parameter till en generisk klass, t.ex. så här:

```
public class MyClass<T extends Comparable<T>> {...}
```

16

Sortering: urvalssortering ("naiv" algoritm)

- Princip: Gå igenom index från minsta till största och se till att minsta värdet hamnar på första platsen, näst minsta värde hamnar på andra platsen, o.s.v.
- Byt plats på (swap) jämförda värden så fort ett efterföljande värde är mindre än det på aktuellt index. (En variant som ger mest kompakt kod.)

Kodexempel: urvalssortering (av int []-fält)

```
public static void selsort(int[] arr) {
    for (int i = 0; i < arr.length-1; i++) {
        for (int j = i+1; j < arr.length; j++) {
            if (arr[i] > arr[j]) {
                swap(arr,i,j);
            }
        }
    }
}
```

Kodexempel: swap

```
public static void swap(int[] arr, int x, int y) {
    int temp = arr[x];
    arr[x] = arr[y];
    arr[y] = temp;
}
```

Ett exempel på urvalssortering

Indata int[] arr = {57,44,91,32,57};
 Jämförelser, indexpar (jämförda värden) och konsekvens (swap eller ingenting).
 0-1 (57-44): swap 1-3 (57-44): swap
 0-2 (44-91) 1-4 (44-57)
 0-3 (44-32): swap 2-3 (91-57): swap
 0-4 (32-57) 2-4 (57-57)
 1-2 (57-91) 3-4 (91-57): swap

Beteende hos urvalssortering

J/S: jämförelse/swap.

	[j = 0]	j = 1	j = 2	j = 3	j = 4
i = 0	-	J/S	J/S	J/S	J/S
i = 1	-	-	J/S	J/S	J/S
i = 2	-	-	-	J/S	J/S
i = 3	-	-	-	-	J/S
[i = 4]	-	-	-	-	-

Komplexitet hos urvalssortering

- Fält av längden n ger $1 + 2 + 3 + \dots + (n - 1)$ st jämförelse+swap (Swap varje gång i värstafallet. Hur ser det ut?) T.ex $1 + 2 + 3 + 4 = 10$ i exemplet med längden 5.
- Funktionen kallas triangelantal (kolla Wikipedia!) $1 + 2 + 3 + \dots + m = (m^2 + m)/2$.
- När n blir allt större så blir n^2 -komponenten relativt sett större än n -komponenten. Man talar därför om en kvadratisk komplexitet här. (Tänk "triangel är halvkvadrat".)

Smartare algoritm: "Quicksort"

- Välj ut ett pivotvärde, som bör ligga nära mitten ordningsmässigt bland de värden som skall sorteras.
- Dela (partitionera) sedan i två högar: värden mindre, respektive värden större än pivoten.
- Partitionera sedan de två högarna på samma sätt.
- (Typ, om vi väljer bokstaver smart: Lägg alla böcker A-L i en hög; M-Ö i en annan. Dela sedan i A-F och G-L samt M-R och S-Ö. O.s.v.)

Välja pivot

- Måste vara en billig beräkning.
- (För att räkna ut verkliga mittobjektet i samlingen måste man ju sortera den.)
- En lösning (givet ett fält som skall sorteras):
 Ta medianen av värdena på första, mittersta och sista index. Median: det mittersta i storleksordning.

```
int pivot = median(arr[low],
                    arr[(low+high+1)/2],
                    arr[high]);
```

Välja pivot, exempel

- `int[] arr = {44,57,91,32,12,93};`, vi ser på hela, från index 0 till 5.
- `int pivot = median(arr[0],arr[3],arr[5]);` blir samma som `median(44,32,93);`
Alltså 44. Råkar vara ett mycket bra val (men det blir det inte alltid).

25

Partitionering

Partitionera fält mellan index `low` och `high` (inklusive).

```
public static int partition(int[] arr,
                           int low,
                           int high) {
    int pivot = median(arr[low],
                       arr[(low+high+1)/2],
                       arr[high]);
```

Forts.

26

Partitionering, forts.

```
while (low < high) {
    while (arr[low] < pivot) {low++;};
    while (arr[high] > pivot) {high--};
    swap(arr,low,high); // som tidigare
};
return low; // index med pivot som värde
           // hamnar mellan partitionerna
}
```

Obs! Förutsätter unika värden i fältet.

27

Quicksort mellan två index, rekursiv

```
public static void qsort(int arr[],
                        int low, int high) {
    if (high-low < 2) { // EN ELLER TVÅ VÄRDEN
        if (arr[low] > arr[high]) {
            swap(arr,low,high);};}
    else { // TRE ELLER FLER VÄRDEN
        int pivotIdx = partition(arr, low, high);
        qsort(arr, low, pivotIdx); // HÖG 1
        qsort(arr, pivotIdx+1, high); } } // HÖG 2
    // OBS! PIVOT I HÖG 1.
```

28

Quicksort mellan två index, icke-rekursiv I

```
public static void qsort2(int[] arr,
                        int low, int high) {
    int[] lo = new int[500]; // som stack
    int[] hi = new int[500]; // som stack
    int il = 1; // antal bitar kvar att sortera
    lo[il]=low; hi[il]=high; // intervall nr il
```

[FORTSÄTTER med en while-SLINGA]

Obs! 500 maxdjup för stack. Ingen säkerhetskontroll.

29

Quicksort mellan två index, icke-rekursiv II

```
while (il > 0) {
    if (hi[il]-lo[il] < 2) { // EN ELLER TVÅ VÄRDEN
        if (arr[lo[il]] > arr[hi[il]]) {
            swap(arr,lo[il],hi[il]); };
        il--; } // BIT NR il KLAR
    else { // d.v.s MINST TRE
```

[SE NÄSTA SIDA]

```
} } }
```

30

Quicksort mellan två index, icke-rekursiv III

```
while (il > 0) {
    [FÖREGÅENDE SIDA]
    else { // MINST TRE; inte (hi[il]-lo[il] < 2)
        int pivotIdx = partition(arr,lo[il],hi[il]);
        lo[il+1] = lo[il]; // NY HÖG il+1 (LÄGRE)
        hi[il+1] = pivotIdx;
        lo[il] = pivotIdx+1; // DEN HÖGRE SKRIVS ÖVER
        // DEN GAMLAS INDEX.
        il++; // hi[il] SOM TIDIGARE
    } } } // il++: EN BIT TILLKOM
```

31

Sammanfattning: fyra algoritmer

- Sökning:
 - linjär sökning
 - binär sökning: halvera sökrymden för varje jämförelse
- Sortering:
 - placera ett värde i taget rätt: urvalssortering
 - Quicksort: *försök* halvera mängden värden att sortera för varje iteration

32