

Sortare și căutare

8 ianuarie 2004

Sortarea. Generalități

Sortarea = aranjarea unei liste de obiecte după o relație de ordine dată (ex.: \leq pentru numere, ordine lexicografică pt. șiruri, etc.)

– una din clasele cele mai fundamentale și studiate de algoritmi

[D. Knuth - Tratat de programare a calculatoarelor. Vol. 3: Sortare și căutare]

– sortare *internă* (în memorie) sau *externă* (folosind fișiere)

Principalele operații la sortare: *compararea* și *interschimbarea*

– vom studia complexitatea algoritmilor (nr. de operații necesare)

$O(f(n))$ dacă timpul de rulare este $\leq cf(n)$, pt. $n > n_0$

$\Omega(f(n))$ dacă timpul de rulare este $\geq cf(n)$, pt. $n > n_0$

se studiază pentru cazul cel mai defavorabil și cel mediu

– vom raționa despre corectitudinea algoritmilor folosind invarianți

Înainte de a sorta ...

... să generăm un tablou de numere aleatoare pe care să le sortăm.

```
int rand(void); /* în stdlib.h */
```

generează număr pseudoaleator între 0 și RAND_MAX

```
void srand(unsigned seed);
```

setează starea inițială pentru generatorul de numere pseudoaleatoare

OBS: în absența apelului la `srand`, funcția `rand` va repeta aceeași secvență generată pentru fiecare rulare

– se poate inițializa generatorul în funcție de ceas (`time.h`):

```
time_t time(time_t *timer); (time_t e unsigned long)
```

ret. nr. de secunde trecute de la o dată origine (UNIX: 1 ian. 1970)

dacă param. pointer e nenul, valoarea se stochează și la acea adresă.

```
const int N=100; const int MAX=1000; int i, a[N];
```

```
srand((int)time(NULL)); /* inițializează generatorul */
```

```
for (i = 0; i < N; i++) a[i] = rand() % MAX; /* între 0 și MAX-1 */
```

Algoritmul Bubblesort

- interschimbăm elemente adiacente, pornind de la ultimul;
- cel mai mic element va ajunge până la începutul tabloului;
- se continuă pentru tabloul de lungime $N-1$ rămas

```
void swap (int *p, int *q) { /* o vom folosi peste tot */
    int aux;
    aux = *p; *p = *q; *q = aux;
}
```

```
void bubblesort(int *a, int N) {
    int i, j;
    for (i = 1; i < N; i++)
        for (j = N-1; j >= i; j--) /*
            if (a[j] < a[j-1]) swap(&a[j-1], &a[j]);
        }
}
```

Analiza lui Bubblesort

Complexitatea:

- bucla după j execută $N - i$ comparații
- total: $\sum_{i=1}^{N-1} N - i = N-1 + N-2 + \dots + 1 = N(N-1)/2 = O(N^2)$
- cele mai multe interschimbări: pentru șirul inițial sortat invers

Invariant: după iterația i ($1 \leq i \leq N$), primele i elemente din tablou sunt cele mai mici, și sunt ordonate

Pasul inductiv: în iterația i cel mai mic element dintre $a[i-1] \dots a[N-1]$ e adus pe poziția i ($a[i-1]$, începând cu 0)

Exercițiu: Modificați bubblesort așa încât:

- să parcurgă tabloul alternativ în ambele direcții
- să se încheie atunci când nu se mai produce nici o modificare

Sortarea prin selecție

- se selectează cel mai mic element
- se interschimbă cu primul element din tablou
- se continuă pentru tabloul de lungime $N-1$ rămas

```
void selectionsort (int *a, int N) {  
    int i, j, low;  
    for (i = 0; i < N; i++) {  
        for (low = j = i; ++j < N;)  
            if (a[j] < a[low]) low = j;  
        if (low != i) swap(&a[i], &a[low]);  
    }  
}
```

Analiza sortării prin selecție

Complexitatea: similar cu bubblesort

- bucla după j execută $N - i$ comparații
 - total: $\sum_{i=1}^{N-1} N - i = N-1 + N-2 + \dots + 1 = N(N-1)/2 = O(N^2)$
 - însă numărul de interschimbări de elemente: cel mult $N-1$
- ⇒ preferabil dacă dimensiunea elementelor este mare

Invariant: același ca la bubblesort: după iterația i ($1 \leq i \leq N$), primele i elemente din tablou sunt cele mai mici, și sunt ordonate

Algoritmul Quicksort

- dezvoltat de Hoare (1960); descompunere recursivă în probleme mai mici – se alege o valoare numită *pivot*, în mod ideal cât mai aproape de valoarea mediană a elementelor din tablou.
- se interschimbă elemente din tablou până se partiționează în două segmente, cu elemente mai mici și respectiv mai mari decât pivotul
- se apelează recursiv pentru cele două partiții (tablouri mai mici)

Găsirea pivotului: o euristică simplă (ex. mediana a 3 elemente)

```
int findpivot (int *a, int i, int j)
{
    int m, n, p;
    m = rand() % (j - i + 1);
    n = rand() % (j - i + 1);
    p = rand() % (j - i + 1);
    return (a[m] < a[n]) ? ((a[n] < a[p]) ? a[n] : max(a[m], a[p]))
        : ((a[m] < a[p]) ? a[m] : max(a[n], a[p]))
}
```


Quicksort: algoritmul propriu-zis

```
void partition (int *a, int i, int j, int *pl, int *pr) {
    int p, l = i, r = j;
    p = findpivot(a, i, j);
    while (1) {
        while (l <= j && a[l] <= p) l++;
        while (r >= i && a[r] >= p) r--;
        if (l < r) swap(&a[l], &a[r]);
        else break;
    }
    *pl = l; *pr = r;
}

void quicksort (int *a, int i, int j) {
    int l, r;
    partition(a, i, j, &l, &r);
    if (r > i) quicksort (a, i, r);
    if (l < j) quicksort (a, l, j);
}
```

Discuția algoritmului Quicksort

- timpul mediu de rulare este $O(n \log n)$ (limita teoretică inferioară pentru sortarea bazată pe comparații).
- $O(n^2)$ în cazul cel mai defavorabil de alegere a pivotului
- cu algoritm mai sofisticat pt. pivot, $O(n \log n)$ în toate cazurile
- numeroase variante pe lângă cea prezentată
- la invocarea recursivă, pentru subproblemele mici se folosește de regulă unul din algoritmi mai simpli

În C: implementat ca funcție standard de bibliotecă:

```
void qsort(void *base, size_t num, size_t size, int (*compar)(void *, void *));
```

Căutare: ghicire din nr. minim de întrebări

```
#include <stdio.h>
void main(void)
{
    unsigned m, lo = 0, hi = (1 << 10) - 1;
    printf("Gândiți-vă la un număr întreg între 0 și %d\n", hi);
    do {
        m = (lo + hi) >> 1;
        printf("Numărul e mai mare decât %d ? (d/n) ", m);
        if (tolower(getchar()) == 'd') lo = m+1; else hi = m;
        while (getchar() != '\n');
    } while (lo < hi);
    printf("Numărul este %d !\n", lo);
}
```

Invariant: $lo \leq nr_c\acute{a}utat \leq hi$

Altfel spus: găsim pe rând biții din scrierea binară a numărului căutat

Căutare binară: recursiv și nerecursiv

Căutăm cheia v în tablou sortat `int a[N]`; returnăm indicele

```

int bsrch(int v, int *a,
          int l, int r) {
    int m;
    while(l<r) {
        int m = (l+r)/2;
        if (v>a[m]) l=m+1;
        else r=m;
    }
    if (v==a[l]) return l;
    else return -1;
}

int bsrch_nr(int v, int *a,
             int l, int r) {
    int m;
    if (l<r) {
        int m = (l+r)/2;
        if (v>a[m])
            return bsrch(v, a, m+1, r);
        else
            return bsrch(v, a, l, m);
    } else if (v==a[l]) return l;
    else return -1;
}

```

```

void *bsearch(const void *key, const void *base, size_t nmemb,
             size_t size, int (*compar)(const void *, const void *));

```