Sortare

29 martie 2005

Sortarea. Generalităţi

Sortarea = aranjarea unei liste de obiecte după o relaţie de ordine dată (ex.: $\leq$ pentru numere, ordine lexicografică pt. şiruri, etc.)
– una din clasele cele mai fundamentale şi studiate de algoritmi
[D. Knuth - Tratat de programare a calculatoarelor. Vol. 3: Sortare şi căutare]
– sortare *internă* (în memorie) sau *externă* (folosind fişiere)

Principalele operaţii la sortare: *compararea* şi interschimbarea
– vom studia complexitatea algoritmilor (nr. de operaţii 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 şi cel mediu
– vom raţiona despre corectitudinea algoritmilor folosind invarianţi

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 şi `RAND_MAX`
`void srand(unsigned seed);`
setează starea iniţială pentru generatorul de numere pseudoaleatoare

OBS: în absenţa apelului la `srand`, funcţia `rand` va repeta aceeaşi secvenţă generată pentru fiecare rulare
– se poate iniţializa generatorul în funcţie 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 şi la acea adresă.

```
const int N=100; const int MAX=1000; int i, a[N];
srand((int)time(NULL)); /* iniţializează generatorul */
for (i = 0; i < N; i++) a[i] = rand() % MAX; /* între 0 şi 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` comparaţii
– total: $\sum_{i=1}^{N-1} N - i$ = N-1 + N-2 + ... + 1 = N(N-1)/2 = $O(N^2)$
– cele mai multe interschimbări: pentru şirul iniţial sortat invers

Invariant: după iteraţia `i` ($1 \leq$ `i`$\leq$ `N`), primele `i` elemente din tablou sunt cele mai mici, şi sunt ordonate
Pasul inductiv: în iteraţia `i` cel mai mic element dintre `a[i-1]` .. `a[N-1]` e adus pe poziţia `i` (`a[i-1]`, începând cu 0)

Exerciţiu: Modificaţi bubblesort aşa încât:
– să parcurgă tabloul alternativ în ambele direcţii
– să se încheie atunci când nu se mai produce nici o modificare

Sortarea prin selecţie

– 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 + ... + 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);
– prin 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 algoritmii mai simpli

În C: implementat ca funcție standard de bibliotecă:

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