laborator 8

Sortări II

Conţinut

- Sortarea prin interclasare (merge sort)
- Sortarea rapidă (quick sort)
- Sortarea rapidă aleatoare (randomized quick sort)
- Sortarea Shell (shell sort)
- Sortarea cu ansamble (heap sort)

Referinţe

- **R. Ceterchi.** *Materiale de curs*, Anul universitar 2012-2013
- **http://laborator.wikispaces.com/**, *Temele 7, 8*
- **Wikipedia - the free encyclopedia**, *Shellsort*, **http://en.wikipedia.org/wiki/Shellsort**. Accesat în noiembrie, 2012
- **N. Wirth**, *Algorithms and Data Structures*, 1985
- **R. Sedgewick**, *Analysis of Shellsort and Related Algorithms*

Sortarea prin interclasare (merge sort)

- Complexitate $O(n \log n)$

Apelată inițial pentru sortarea unui vector A de dimensiune n prin: SORTEAZĂ–PRIN–INTERCLASARE($A[1..n]$).

►► SORTEAZĂ–PRIN–INTERCLASARE($A[p..u]$)

```
if p < u then
   q ← ⌊(p+u)/2⌋
   SORTEAZĂ–PRIN–INTERCLASARE(A[p..q])
   SORTEAZĂ–PRIN–INTERCLASARE(A[q+1..u])
   A[p..u] ← INTERCLASEAZĂ(A[p..q],A[q+1..u])
endif
```

Aceasta este doar una din variantele de algoritmi pentru interclasare.

►► INTERCLASEAZĂ($A[p1..u1]$, $A[p2..u2]$)

```
ALOCĂ (u1 − p1) + (u2 − p2) SPAŢIU PENTRU DESTINAŢIA L
i ← 1
while (p1 ≤ u1) and (p2 ≤ u2) do
   if A[p1] ≤ A[p2] then
      L[i] ← A[p1]
      p1 ← p1 + 1
   else
      L[i] ← A[p2]
      p2 ← p2 + 1
   endif
   i ← i+1
endwhile
if p1 > u1 then
   for k ← p2 to u2 do
      L[i] ← A[k]
      i ← i+1
   endfor
else
   for k ← p1 to u1 do
      L[i] ← A[k]
      i ← i+1
   endfor
endif
return L
```

Sortarea Shell (shell sort)

Algoritmul a fost propus de D.L. Shell şi este bazat pe metoda prin inserţie directă. Performanţa sa este îmbunătăţită, deoarece face comparaţii între chei mai distanţate din vector.

Este o sortare prin inserţie cu micşorarea incrementului.

Se sortează mereu *prin inserţie* componentele din vector aflate la distanţă h una de cealaltă. Distanţa h se numeşte *increment* şi la fiecare iteraţie el scade, până când devine 0 şi algoritmul se termină.

Prin urmare, putem folosi algoritmul de sortare prin inserţie pentru implementarea sortării Shell.

►► SORTEAZĂ–SHELL($A[1..n]$)

```
h ← INCREMENTUL–INIŢIAL(n)
while (h > 0)
  for k ← 1 to h do
    i ← k+h
    while i ≤ n do
      cheie ← A[i]
      j ← i-h
      while (j > 0) and (A[j] > cheie) do
        A[j+h] ← A[j]
        j ← j-h
      endwhile
      A[j+h] ← cheie
      i ← i + h
    endwhile
  endfor
  pas ← pas + 1
  h ← INCREMENTUL–URMĂTOR(h)
endwhile
```

În funcţie de secvenţa de incremenţi aleasă se vor implementa procedurile INCREMENTUL–INIŢIAL(N) şi INCREMENTUL–URMĂTOR(H).

Dacă dorim să folosim secvenţa de incremenţi propusă iniţial de Shell, obţinută prin divizarea repetată a lungimii vectorului la 2:

$$\left\lfloor \frac{n}{2} \right\rfloor, \left\lfloor \frac{n}{4} \right\rfloor, \ldots, 64, 32, 16, 8, 4, 2, 1$$

atunci procedura arată în modul următor.

Pentru această secvenţă de incremenţi, timpul de execuţie ar fi pătratic, pentru că elementele din poziţii impare nu sunt comparate cu elemente din poziţii pare decât atunci când incrementul este 1.

INCREMENTUL–INIŢIAL(n)

```
return ⌊n/2⌋
```

INCREMENTUL–URMĂTOR(h)

```
return h/2
```

Sortarea rapidă (quick sort)

- Complexitate $O(n \log n)$

►► SORTEAZĂ–RAPID($A[p..u]$)

```
1. if p < u then
2.   pivot ← PARTIŢIE(A[p..u])
3.   SORTEAZĂ–RAPID(A[p..pivot])
4.   SORTEAZĂ–RAPID(A[pivot+1..u])
5. endif
```

►► PARTIŢIE($A[p..u]$)

```
x ← A[p]
i ← p-1
j ← u+1
while true do
  repeat
    j ← j-1
  until A[j] ≤ x
  repeat
    i ← i+1
  until A[i] ≥ x
  if i < j then
    INTERSCHIMBĂ(A[i], A[j])
  else
    return j
  endif
endwhile
```

Puteţi implementa în C++ un ciclu ***repeat*** *…* ***until*** *(condiţie) printr-un ciclu* ***do*** *…* ***while*** *(!condiţie) (s-a negat condiţia ciclului* ***repeat*** *-* ***until*** *).*

Sortarea rapidă aleatoare (randomized quick sort)

- Complexitate $O(n \log n)$

În procedura SORTEAZĂ–RAPID($A[p..u]$) se va apela PARTIŢIE–ALEATOARE$A[p..u]$ la linia 2.

►► PARTIŢIE–ALEATOARE($A[p..u]$)

```
i ← RANDOM(p,u)
INTERSCHIMBĂ(A[p], A[i])
return Partiţie(A[p..u])
```

În C++, puteţi obţine un număr aleator în intervalul $[a,b]$ prin: `a + int ((b-a+1) * rand() / (RAND_MAX + 1.0));`

Sortarea cu ansamble (heap sort)

- Complexitate $O(n \log n)$

Algoritmul din această secţiune sortează crescător un vector `A[1..n]`. Sortarea cu ansamble lucrează pe un vector care iniţial este transformat într-un max-ansamblu. Astfel, rădăcina `A[1]` este elementul maxim din vector şi poate fi interschimbată cu elementul de la ultima poziţie, `n`. La fiecare iteraţie dimensiunea porţiunii de sortat din vectorul `A` scade cu 1 prin excluderea poziţiei pe care a fost dus maximul. Pentru ca subvectorul rezultat în urma interschimbării şi a micşorării să rămână un ansamblu, se apelează procedura de asamblare.

Atenţie dacă implementaţi ansamblul folosind structuri, atunci operaţiile din liniile 4–6 se fac pe vectorul din structura ansamblului, iar la sfârşit puteţi copia vectorul acesta în vectorul A.

▶▶ SORTEAZĂ–CU–ANSAMBLE(A, n)

```
1.  CREEAZĂ–ANSAMBLU(A,n)
2.  CONSTRUIEŞTE–ANSAMBLU(A,n)
3.  for i ← n to 2 do
4.    INTERSCHIMBĂ(A[1],A[i])
5.    sizeA ← sizeA-1
6.    ASAMBLEAZĂ(A,1)
7.  endfor
```

Alternativ, putem utiliza varianta care foloseşte procedura de decapitare a ansamblului.

Observaţia de mai sus rămâne valabilă (pentru liniile 4–5).

▶▶ SORTEAZĂ–CU–ANSAMBLE–2(A, n)

```
1.  CREEAZĂ–ANSAMBLU(A,n)
2.  CONSTRUIEŞTE–ANSAMBLU–2(A,n)
3.  for i ← n to 2 do
4.    x ← DECAPITEAZĂ–ANSAMBLU(A)
5.    A[i] ← x
6.  endfor
```

1. **(1p)** Să se implementeze metoda de ordonare merge sort (prin interclasare).
2. **(2p)** Să se implementeze algoritmul randomized quick sort, în care alegerea pivotului se face aleator.
3. **(2p)** Să se optimizeze algoritmul quick sort, folosind următoarea tehnică: subşirurile de dimensiune ≤ 11 elemente se sortează cu inserţia directă.
4. **(2p)** Să se optimizeze algoritmul de bază al metodei de sortare merge sort (prin interclasare) prin utilizarea inserţiei directe la sortarea subşirurilor mici (mai mici de 10 elemente).
5. **(2p)** Fie două secvenţe sortate care împart acelaşi tablou şi sunt poziţionate astfel: prima crescând urmată de cealaltă descrescând, sau prima descrescând urmată de cealaltă crescând (secvenţă bitonică). Se cere să să se sorteze prin interclasare (merge sort) tabloul respectiv.
6. **(2p)** Să se implementeze algoritmul shell sort.
7. **(2p)** Dat un numar natural n, reprezentând restul pe care o persoană trebuie să îl primească după efectuarea unei plăţi, să se spună care este numărul minim de bancnote utilizate pentru plata restului. Presupunem că există k tipuri de bancnote, cu valori $b_1; b_2; ..., b_k$. Datele de intrare se citesc din fişierul de intrare `input.txt`. Exemplu:

   $n = 1242$; $k = 4$; $b_1 = 90$; $b_2 = 25$; $b_3 = 6$; $b_4 = 3$

   Restul poate fi plătit ca $1242 = 10 \times b_1 + 12 \times b_2 + 5 \times b_3 + 4 \times b_4$. Soluţia nu este unică.
8. **(2p)** Un hotel este foarte faimos pentru sala sa de conferinţe. Acesta a primit n cereri de tipul $[s; f)$ de a închiria sala de conferinţe în intervalul de timp cuprins între s (inclusiv) şi f (exclusiv). Pentru ca fiecare închiriere aduce proprietarilor hotelului un venit fix, aceştia ar dori să onoreze cât mai multe cereri. Sala nu poate fi închiriată la două persoane în acelaşi timp. Să se spuna care este numărul maxim de cereri care pot fi onorate şi care sunt acestea.

   Datele de intrare se citesc din fişierul `input.txt` astfel: pe primul rând se află n, iar pe următoarele n rânduri câte o pereche `s f`, reprezentând intervalul $[s; f)$. Exemplu:

   ```
   7
   0  2
   3  7
   4  7
   9  11
   7  10
   1  5
   6  8
   ```

   Numărul maxim de cereri este 3, iar cererile pot fi $[0; 2)$; $[3; 7)$; $[7; 10)$.
9. **(10ps)** Spunem că o tablă de şah de $2^k \times 2^k$ pătrate este defectă, dacă unul din cele 2^{2^k} pătrate lipseşte. Problema vă cere să acoperiţi o astfel de tablă cu tromino-uri (prima figură), astfel încât oricare două tromino-uri să nu se suprapună, ele nu acoperă pătratul lipsă, dar acoperă toate celelalte pătrate. Sugestii de implementare:

   (a) o acoperire a unei table $m \times m$ se poate reprezenta printr-o matrice `Tabla[m][m]`, unde `Tabla[i][j]` indică numărul trominoului cu care este acoperit pătratul `(i; j)`.

   (b) Funcţia recursivă ce construieşte soluţia poate fi de forma: `Acopera(rt,ct,rd, cd,latura)`, unde:

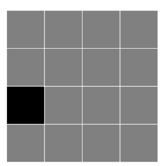
   i. `rt, ct` reprezintă rândul şi coloana pătratului din colţul stânga sus al porţiunii pătratice de tablă ce trebuie acoperită;

   ii. `rd, cd` reprezintă rândul şi coloana pătratului lipsă;

   iii. `latura` reprezintă latura porţiunii pătratice de tablă ce trebuie acoperită.

   Tromino-uri: 

   O tablă de şah defectă a cărei dimensiune este $2^2 \times 2^2$:

■ Termen de predare: **Săptămâna 12 (17–21 decembrie 2012) inclusiv.**

■ Detalii: **Studenţii pot obţine un maxim de 25 puncte. Problemele 1-2 sunt obligatorii. Problemele 3–8 sunt suplimentare.Problema 9 este facultativă, iar termenul de predare pentru ea este săptămâna 11 (10–14 decembrie). Un singur student poate rezolva problema facultativă.**