Algoritmi şi structuri de date (I). Seminar 7: Variante ale algoritmilor de sortare. Generarea permutări şi submulţimi.

Problema 1 *Counting sort - sortare prin numărare.* Să considerăm problema sortării unui tablou $x[1..n]$ având elemente din $\{1, 2, \ldots, m\}$. Pentru acest caz particular poate fi utilizat un algoritm de complexitate liniară bazat pe următoarele idei:

- se construieşte tabelul $f[1..m]$ al frecvenţelor de apariţie a valorilor elementelor tabloului $x[1..n]$;
- se calculează frecvenţele cumulate asociate;
- se foloseşte tabelul frecvenţelor cumulate pentru a construi tabloul ordonat y.

Algoritmul poate fi descris prin:

```
countingsort(integer x[1..n],m)
integer i, f[1..m], y[1..n]
for i ← 1, m do f[i] ← 0 endfor
for i ← 1, n do f[x[i]] ← f[x[i]] + 1 endfor
for i ← 2, m do f[i] ← f[i − 1] + f[i] endfor
for i ← n, 1, −1 do
    y[f[x[i]]] ← x[i]
    f[x[i]] ← f[x[i]] − 1
endfor
for i ← 1, n do x[i] ← y[i] endfor
return x[1..n]
```

Dacă $m \in \mathcal{O}(n)$ atunci algoritmul de sortare prin numărare are complexitate liniară. Dacă însă m este mult mai mare decât n (de exemplu $m \in \mathcal{O}(n^2)$) atunci ordinul de complexitate al algoritmului de sortare este determinat de m. Atât timp cât ciclul de construire al tabloului y este descrescător (se porneşte cu contorul i de la n), algoritmul este stabil.

Problema 2 *Radix sort - sortare pe baza cifrelor.* Considerăm un tablou $x[1..n]$ având elemente numere naturale constituite din cel mult k cifre. Dacă $m = 10^k$ este semnificativ mai mare decât n atunci sortarea prin numărare nu este eficientă. Pe de altă parte, dacă k este mult mai mic decât n atunci un algoritm de complexitate $\mathcal{O}(kn)$ ar putea fi acceptabil. Un astfel de algoritm este cel bazat pe următoarea idee: se ordonează tabloul în raport cu cifra cea mai puţin semnificativă a fiecarui număr (folosind counting sort în ipoteza că mulţimea valorilor din tabloul de sortat este mulţimea cifrelor, $\{0, 1, \ldots, 9\}$) după care se sortează în raport cu cifra de rang imediat superior ş.a.m.d. până se ajunge la cifra cea mai semnificativă. Deci structura generală a algoritmului este:

```
radixsort(integer x[1..n], k)
integer i
for i ← 0, k − 1 do
    x[1..n] ← counting2(x[1..n], 9, i)
endfor
return x[1..n]
```

Algoritmul `counting2` este o adaptare a algoritmului de sortare prin numărare în raport cu cifrele de un anumit rang (i):

```
counting2(integer x[1..n],m,c )
integer i, j, f[0..m], y[1..n]
for i ← 0, m do f[i] ← 0 endfor
for i ← 1, n do
    j ← (x[i]DIV putere(10, c))MOD 10
    f[j] ← f[j] + 1
endfor
for i ← 1, m do f[i] ← f[i − 1] + f[i] endfor
for i ← n, 1, −1 do
    j ← (x[i]DIV putere(10, c))MOD 10
    y[f[j]] ← x[i]
    f[j] ← f[j] − 1
endfor
for i ← 1, n do x[i] ← y[i] endfor
return x[1..n]
```

Algoritmul `counting2` se apelează pentru $m = 9$ (care este valoare constantă în raport cu n) deci este de ordinul $\mathcal{O}(n)$. Pe de altă parte pentru ca algoritmul de sortare pe baza cifrelor să funcţioneze corect este necesar ca subalgoritmul de sortare prin numărare apelat să fie stabil.

Problema 3 (*Shell sorting - sortare prin inserţie cu pas variabil.*) Unul dintre dezavantajele sortării prin inserţie este faptul că la fiecare etapă un element al şirului se deplasează cu o singură poziţie. O variantă de reducere a numărului de operaţii efectuate este de a compara elemente aflate la o distanţă mai mare ($h \geq 1$) şi de a realiza deplasarea acestor elemente peste mai multe poziţii. De fapt tehnica se aplică în mod repetat pentru valori din ce în ce mai mici ale pasului h, asigurând h-sortarea şirului. Un şir $x[1..n]$ este considerat h-sortat dacă orice subşir $x[i_0], x[i_0 + h], x[i_0 + 2h]$... este sortat ($i_0 \in \{1, \ldots, h\}$). Aceasta este ideea algoritmului propus de Donald Shell în 1959 cunoscut sub numele "shell sort".

Elementul cheie al algoritmului îl reprezintă alegerea valorilor pasului h. Pentru alegeri adecvate ale secvenţei h_k se poate obţine un algoritm de complexitate $\mathcal{O}(n^{3/2})$ în loc de $\mathcal{O}(n^2)$ cum este în cazul algoritmului clasic de sortare prin inserţie.

Exemplu. Exemplificăm ideea algoritmului în cazul unui şir cu 15 elemente pentru următoarele valori ale pasului: $h = 13$, $h = 4$, $h = 1$ (care corespund unui şir h_k dat prin relaţia $h_k = 3h_{k-1} + 1$, $h_1 = 1$):

Etapa 1: pentru $h = 13$ se aplică algoritmul sortării prin inserţie subşirurilor $x[1], x[14]$ şi $x[2], x[15]$, singurele subşiruri cu elemente aflate la distanţa h care au mai mult de un element:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
14	8	10	7	9	12	5	15	2	13	6	1	4	3	11

şi se obţine

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	8	10	7	9	12	5	15	2	13	6	1	4	14	11

Etapa 2: pentru $h = 4$ se aplică algoritmul sortării prin inserţie succesiv subşirurilor: $x[1], x[5], x[9], x[13]$, $x[2], x[6], x[10], x[14], x[3], x[7], x[11], x[15], x[4], x[8], x[12]$. După prima subetapă (prelucrarea primului subşir) prin care se ordonează subşirul constituit din elementele marcate:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	8	10	7	9	12	5	15	2	13	6	1	4	14	11

se obţine:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	8	10	7	3	12	5	15	4	13	6	1	9	14	11

La a doua subetapă se aplică sortarea prin inserţie asupra subşirului constituit din elementele marcate:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	8	10	7	3	12	5	15	4	13	6	1	9	14	11

obţinându-se aceeaşi configuraţie (subşirul este deja ordonat crescător) din care se prelucrează acum subşirul constituit din elementele marcate:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	8	10	7	3	12	5	15	4	13	6	1	9	14	11

obţinându-se

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	8	5	7	3	12	6	15	4	13	10	1	9	14	11

Se aplică acum sortarea prin inserţie asupra subşirului constituit din elementele marcate:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	8	5	7	3	12	6	15	4	13	10	1	9	14	11

obţinându-se

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	8	5	1	3	12	6	7	4	13	10	15	9	14	11

Se aplică acum sortarea prin inserţie asupra întregului şir. Pentru acest exemplu aplicarea directă a sortării prin inserţie conduce la execuţia a 68 de comparaţii pe când aplicarea sortarii cu pas variabil de inserţie conduce la execuţia a 34 de comparaţii.

Prelucrarea specifică unei etape (o valoare a pasului de inserţie) poate fi descrisă prin:

```
insertie_pas(real x[1..n], integer h)
integer i, j
real aux
for i ← h + 1, n do
    aux ← x[i]
    j ← i − h
    while j ≥ 1 AND aux < x[j] do
        x[j + h] ← x[j]
        j ← j − h
    endwhile
    x[j + h] ← aux
endfor
return x[1..n]
```

Această prelucrare se efectuează pentru diferite valori ale lui h. Un exemplu de şir de valori pentru h (care s-a dovedit a conduce la o variantă eficientă a algoritmului) este cel bazat pe relaţia de recurenţă: $h_k = 3h_{k-1} + 1$, $h_1 = 1$. Structura generală a algoritmului va consta în:

```
shellsort(real x[1..n])
integer h
h ← 1
while h ≤ n do
    h ← 3 * h + 1
endwhile
repeat
    h ← hDIV3
    x[1..n] ← insertie_pas(x[1..n], h)
until h = 1
return x[1..n]
```

O altă variantă de alegere a pasului h, pentru care s-a demonstrat că algoritmul are complexitatea $\mathcal{O}(n\sqrt{n})$ este $h_k = 2^k - 1$.

Problema 4 (*Shaker sort.*) Sortarea prin interschimbarea elementelor vecine asigură, la fiecare pas al ciclului exterior, plasarea câte unui element (de exemplu maximul din subtabloul tratat la etapa respectivă) pe poziţia finală. O variantă ceva mai eficientă ar fi ca la fiecare etapă să se plaseze pe poziţiile finale câte două elemente (minimul respectiv maximul din subtabloul tratat la etapa respectivă). Pe de altă parte prin reţinerea indicelui ultimei interschimnări efectuate, atât la parcurgerea de la stânga la dreapta cât şi de la dreapta la stânga se poate limita regiunea analizată cu mai mult de o poziţie atât la stânga cât şi la dreapta (când tabloul conţine porţiuni deja sortate).

Structura algoritmului este:

```
shakersort(real x[1..n])
integer s, d, i, t
s ← 1; d ← n
repeat
    t ← 0
    for i ← s, d − 1 do
        if x[i] > x[i + 1] then x[i] ↔ x[i + 1]; t ← i endif
    endfor
    if t ≠ 0 then
        d ← t; t ← 0
        for i ← d, s + 1, −1 do
            if x[i] < x[i − 1] then x[i] ↔ x[i − 1]; t ← i endif
        endfor
        s ← t
    endif until t = 0 OR s = d
return x[1..n]
```

Problema 5 Să se genereze toate permutările de ordin n în ordine lexicografică (în ordinea crescătoare a valorii asociate permutării). Pentru $n = 3$ aceasta înseamnă generarea valorilor în ordinea: $(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$, $(3, 2, 1)$.

Rezolvare. Se porneşte de la permutarea identică $p[1..n]$, $p[i] = i, i = \overline{1,n}$ (care pentru $n = 3$ are asociată valoarea $123\ldots n$) şi se generează succesiv valoarea imediat următoare constituită din aceleaşi cifre. Pentru fiecare nouă permutare generată se parcurg etapele:

- identifică cel mai mare indice $i \in \{2, 3, \ldots, n\}$ cu proprietatea $p[i] > p[i-1]$;
- determină poziţia, k, a celei mai mici valori din $p[i..n]$ cu proprietatea că $p[k] > p[i-1]$;
- interschimbă $p[k]$ cu $p[i-1]$;
- inversează ordinea elementelor din subtabloul $p[i..n]$.

Structura generală a algoritmului:

```
perm(integer n)
integer i, kmin
for i ← 1, n do p[i] ← i endfor
write p[1..n]
i ← identific(p[1..n])
while i > 1 do
    kmin ←minim(p[i..n],i − 1)
    p[i − 1] ↔ p[kmin]
    p[i..n] ← inversare(p[i..n])
    write p[1..n]
    i ← identific(p[1..n])
endwhile
```

unde identific($p[1..n]$) determină cel mai mare indice i cu proprietatea că $x[i] > x[i-1]$, minim($p[i..n]$,$i-1$) determină indicele celui mai mic element din $p[i..n]$ care este mai mare decât $p[i-1]$ iar inversare($p[i..n]$) returnează elementele $p[i..n]$ în ordine inversă.

Pentru fiecare dintre cele $n! - 1$ permutări diferite de permutarea identică se efectuează un număr de operaţii de ordinul $\mathcal{O}(n)$. Ordinul de complexitate al algoritmului este $\mathcal{O}(n \cdot n!)$.

Problema 6 *Algoritmul Johnson-Trotter.* Să se genereze permutările de ordin n astfel încât fiecare permutare să fie obţinută din cea imediat anterioară prin interschimbarea a exact două elemente. Acest lucru se poate realiza prin algoritmul Johnson-Trotter ale cărui etape principale sunt:

- Se porneşte de la permutarea identică; asociază fiecărui element o un sens de parcurgere (iniţial toate elementele au asociat un sens de parcurgere înspre stânga).
- Determină elementul mobil maximal (un element este considerat mobil dacă sensul ataşat lui indică către o valoare adiacentă mai mică).
- Interschimbă elementul mobil cu elementul mai mic către care indică.
- Modifică direcţia tuturor elementelor mai mari decât elementul mobil.

Ultimele trei etape de mai sus se repetă până nu mai poate fi găsit un element mobil. În cazul permutărilor de ordin 3 algoritmul conduce la următoarea secvenţă de rezultate:

$\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{3}$
$\overleftarrow{1}\,\overleftarrow{3}\,\overleftarrow{2}$
$\overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{2}$
$\overrightarrow{3}\,\overleftarrow{2}\,\overleftarrow{1}$
$\overleftarrow{2}\,\overrightarrow{3}\,\overleftarrow{1}$
$\overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{3}$

Sensurile de parcurgere pot fi uşor modelate printr-un tablou $d[1..n]$ în care valoarea -1 corespunde orientării înspre stânga iar valoarea $+1$ corespunde orientării înspre dreapta. In aceste ipoteze algoritmul poate fi descris prin:

```
perm(integer n)
integer p[1..n], d[1..n], k, delta
for i ← 1, n do p[i] = i; d[i] = −1; endfor
k ← mobil (p[1..n], d[1..n])
while k > 0
    delta ← d[k]
    p[k] ↔ p[k + delta]
    d[k] ↔ d[k + delta]
    d[1..n] ←modific(p[1..n],d[1..n],k + delta)
    write p[1..n]
    k ← mobil(p[1..n], d[1..n])
endwhile
```

Subalgoritmul de determinare a elementului mobil este:

```
mobil(integer p[1..n], d[1..n])
integer i, k, j
k ← 0; i ← 1;
while k = 0 AND i ≤ n do
    if i + d[i] ≥ 1 AND i + d[i] ≤ n AND p[i] > p[i + d[i]] then k ← i
    else i ← i + 1
endwhile
for j ← i + 1, n do
    ifj + d[j] ≤ n AND p[j] > p[j + d[j]] AND p[j] > p[k] then k ← j
    endif
endfor
return k
```

Subalgoritmul de modificare a sensurilor asociate elementelor mai mari decât $p[k]$:

```
modific(integerp[1..n], d[1..n],k)
integer i
for i ← 1, n do
    if p[i] > p[k] then d[i] ← −d[i] endif
endfor
return d[1..n]
```

Probleme suplimentare

1. Propuneţi un algoritm care sortează crescător elementele de pe poziţiile impare ale unui tablou şi descrescător cele de pe poziţii pare.

2. Adaptaţi algoritmul de sortare prin numărare pentru ordonarea descrescătoare a unui tablou.

3. Se consideră un set de valori din $\{1, \ldots, m\}$. Preprocesaţi acest set folosind un algoritm de complexitate $\mathcal{O}(\max\{m, n\})$ astfel încât răspunsul la întrebarea "câte elemente se află în intervalul $[a, b]$, $a, b \in \{1, \ldots, m\}$ să poată fi obţinut în timp constant.

4. Propuneţi un algoritm de complexitate liniară pentru ordonarea crescătoare a unui tablou constituit din n valori întregi aparţinând mulţimii $\{0, 2, \ldots, n^2 - 1\}$. *Indicaţie.* Se vor interpreta valorile ca fiind reprezentate în baza n.

5. Se consideră un tablou constituit din triplete de trei valori întregi corespunzătoare unei date calendaristice. Propuneţi un algoritm de ordonare crescătoare după valoarea datei.