Algoritmi şi structuri de date (I). Seminar 8: Algoritmi de generare a permutărilor. Aplicaţii ale tehnicii reducerii. Analiza complexităţii algoritmilor recursivi.

Problema 1 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 2 *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]
```

Problema 3 Să se genereze toate şirurile cu n elemente din $\{0, 1\}$. De exemplu, pentru $n = 3$ sunt opt şiruri binare: $(0, 0, 0)$, $(0, 0, 1)$, $(0, 1, 0)$, $(0, 1, 1)$, $(1, 0, 0)$, $(1, 0, 1)$, $(1, 1, 0)$, $(1, 1, 1)$. Stabiliţi ordinul de complexitate al algoritmului propus.

Rezolvare. O primă variantă de rezolvare constă în numărarea în baza doi pornind de la şirul valorilor egale cu 0. In ipoteza că $p[1..n]$ este o variabilă globală, algoritmul poate fi descris prin:

```
generare( n)
for i = 1, n do p[i] = 0 endfor
repeat
    write p[1..n]
    r =inc(p[1..n])
// r este reportul corespunzător celei mai semnificative cifre
// dacă este 1 înseamnă că a fost deja afişat şirul (1, 1, ..., 1)
until r == 1
```

cu algoritmul de incrementare descris prin:

```
inc( p[1..n])
// se începe incrementarea de la cifra cea mai puţin semnificativă
s = p[n] + 1
p[n] = sMOD2
r = sDIV2 // r reprezintă reportul
i = n − 1
while (r > 0) AND (i >= 1) do
    s = p[i] + r;
    p[i] = sMOD2;
    r = sDIV2;
    i = i − 1
endwhile
return r
```

Considerând că dimensiunea problemei este n iar operaţia dominantă este împărţirea la 2, pentru fiecare dintre cei 2^n vectori se efectuează cel puţin o operaţie şi cel mult n. Deci algoritmul aparţine lui $\Omega(2^n)$ respectiv $\mathcal{O}(n2^n)$. Trebuie remarcat însă că marginea $n2^n$ este largă întrucât doar la ultimul apel al algoritmului `inc` se efectuează n împărţiri (la celelalte apeluri se efectuează mai puţine). Pe de altă parte algoritmul de incrementare poate fi rescris fără a folosi operaţii de împărţire (vezi Probleme suplimentare).

O altă variantă, bazată pe tehnica reducerii, este:

```
generare(k)
if k == 1 then
    p[1] = 0
    write (p[1..n])
    p[1] = 1
    write (p[1..n])
else
    p[k] = 0
    generare(k − 1)
    p[k] = 1
    generare(k − 1)
endif
```

Algoritmul de mai sus se apelează pentru $k = n$ (`generare`(n)) presupunând că $p[1..n]$ este o variabilă globală. La fiecare apel al funcţiei se completează poziţia corespunzătoare lui k cu 0, respectiv 1 după care se apelează recursiv algoritmul pentru a genera toate subşirurile binare cu $k-1$ elemente.

Pentru a determina ordinul de complexitate al algoritmului notăm cu $T(n)$ numărul de atribuiri efectuate ($p[k] = 0$ şi $p[k] = 1$). Acesta va satisface relaţia de recurenţă:

$$T(n) = \begin{cases} 2 & n = 1 \\ 2T(n-1) + 2 & n > 1 \end{cases}$$

Pentru rezolvarea relaţiei de recurenţă se aplică metoda substituţiei inverse:

$$\begin{array}{l|l} T(n) = 2T(n-1) + 2 & \cdot 1 \\ T(n-1) = 2T(n-2) + 2 & \cdot 2 \\ \vdots & \\ T(2) = 2T(1) + 2 & \cdot 2^{n-2} \\ T(1) = 2 & \cdot 2^{n-1} \end{array}$$

Prin însumarea relaţiilor şi reducerea termenilor asemenea se obţine $T(n) = 2(1 + 2 + \ldots + 2^{n-2} + 2^{n-1}) = 2(2^n - 1) \in \Theta(2^n)$.

Problema 4 Să se calculeze A^p unde A este o matrice $n \times n$ şi p este o valoare naturală mai mare decât 1. Să se analizeze complexitatea algoritmului propus.

Rezolvare. Presupunem că `produs`$(A[1..n, 1..n], B[1..n, 1..n])$ este un algoritm care returnează produsul matricilor A şi B specificate ca parametri de intrare.

O primă variantă de calcul a lui A^p se bazează pe metoda forţei brute şi conduce la un algoritm de forma:

```
putere1(A[1..n, 1..n], p)
P[1..n, 1..n] = A[1..n, 1..n]
for i = 2, p do P =produs(P, A)
endfor
```

Pentru a analiza complexitatea considerăm că dimensiunea problemei este determinată de perechea (n, p) şi că operaţia dominantă este cea de înmulţire efectuată în cadrul algoritmului `produs`. Este uşor de stabilit că la fiecare apel se efectuează n^3 operaţii de înmulţire astfel că numărul total de înmulţiri efectuate de către algoritmul `putere1` este $T(n, p) = (p-1)n^3 \in \Theta(pn^3)$.

Aplicând tehnica reducerii se obţine algoritmul:

```
putere2(real A[1..n, 1..n], integer p)
if p == 1 then return A[1..n, 1..n]
else if pMOD2 == 0 then
        B[1..n, 1..n] =putere2(A, p/2)
        return produs(B[1..n, 1..n], B[1..n, 1..n])
    else
        B[1..n, 1..n] = putere2(A, (p − 1)/2)
        B[1..n, 1..n] = produs(B[1..n, 1..n], B[1..n, 1..n])
        B[1..n, 1..n] = produs(B[1..n, 1..n], A[1..n, 1..n])
        return B[1..n, 1..n]
    endif
endif
```

Numărul de înmulţiri efectuate în cadrul algoritmului satisface relaţia de recurenţă:

$$T(n,p) = \begin{cases} 0 & p = 1 \\ T(n, p/2) + n^3 & p \text{ par} \\ T(n, (p-1)/2) + 2n^3 & p \text{ impar} \end{cases}$$

Considerăm cazul particular $p = 2^k$ şi aplicăm metoda substituţiei inverse:

$$\begin{array}{l} T(n,p) = T(n,p/2) + n^3 \\ T(n,p/2) = T(n,p/4) + n^3 \\ \vdots \\ T(n,2) = T(1) + n^3 \\ T(n,1) = 0 \end{array}$$

Prin însumarea relaţiilor de mai sus se obţine $T(n,p) = n^3 \lg p$ pentru $p = 2^k$. Întrucât $T(n,p)$ este crescătoare şi $n^3 \lg p$ este o funcţie netedă rezultatul poate fi extins şi pentru p arbitrar. Prin urmare, în varianta bazată pe metoda reducerii, algoritmul de ridicare la putere a unei matrice este din $\Theta(n^3 \lg p)$.

Probleme suplimentare

1. Să se rescrie algoritmul de incrementare cu 1 a unui număr reprezentat în baza 2 fără a folosi operaţii de împărţire.
   Indicaţie. Se parcurge tabloul cu cifrele binare de la cifra cea mai puţin semnificativă către cea mai semnificativă şi se pune pe 1 prima cifră egală cu 0 întâlnită iar cifrele egale cu 1 întâlnite până la primul 0 se înlocuiesc cu 0. Algoritmul are ordinul de complexitate $\mathcal{O}(n)$.

2. Să se genereze toate cele 2^n submulţimi ale unei mulţimi cu n elemente.
   Indicaţie. O submulţime poate fi reprezentată prin tabelul indicatorilor de prezenţă care este un şir binar cu n elemente, prin urmare generarea tuturor submulţimilor unei mulţimi cu n elemente este echivalentă cu generarea tuturor şirurilor binare cu n elemente.

3. Folosind tehnica divizării algoritmul de înmulţire a două matrici poate fi reorganizat astfel încât ordinul de complexitate să fie redus. Un exemplu în acest sens este algoritmul lui Strassen al cărui ordin de complexitate este $\mathcal{O}(n^{2.7})$ în cazul înmulţirii a două matrici pătratice de dimensiune $n \times n$. Consultaţi resurse web (folosind "Strassen algorithm" drept cheie de căutare) unde este descris algoritmul şi încercaţi să îl implementaţi.

4. Stabiliţi ce afişează algoritmul de mai jos atunci când este apelat pentru $k = n$ (în ipoteza că lucrează asupra unui tablou global $a[1..n]$ iniţializat astfel încât $a[i] = i$) şi stabiliţi ordinul de complexitate al algoritmului.

```
alg(integer k)
if k == 1 then write a[1..n]
else for i = 1, k do
    alg(k − 1)
    if kMOD2 == 1 then a[1] ↔ a[k]
       else a[i] ↔ a[k]
    endif
endfor
endif
```

   Indicaţie. Considerând că a este variabilă globală, iar parametrul de apel reprezintă numărul de elemente din a, o variantă de implementare este:

```
def alg(k):
if k==1:
    print(a)
else:
    for i in range(k):
        alg(k-1)
        if (k%2 == 0):
            a[0],a[k-1]=a[k-1],a[0]
        else:
            a[i],a[k-1]=a[k-1],a[i]
```

   Executând algoritmul rezultă că acesta generează permutările de ordin n. Considerând dimensiunea problemei n şi interschimbarea ca operaţie dominantă relaţia de recurenţă corespunzătoare timpului de execuţie este:

$$T(n) = \begin{cases} 0 & n = 1 \\ n(T(n-1)+2) & n > 1 \end{cases}$$

iar prin aplicarea tehnicii substituţiei inverse se ajunge la:

$$\begin{array}{l|l} T(n) = nT(n-1) + 2n & \cdot 1 \\ T(n-1) = (n-1)T(n-2) + 2(n-1) & \cdot n \\ T(n-2) = (n-2)T(n-3) + 2(n-2) & \cdot n(n-1) \\ \vdots & \\ T(2) = 2T(1) + 2 \cdot 2 & \cdot n(n-1)\ldots 3 \\ T(1) = 0 & \end{array}$$

de unde, prin însumare obţinem $T(n) = 2(n+n(n-1)+\ldots n!)$ adică $T(n) = 2n!(\frac{1}{(n-1)!}+\frac{1}{(n-2)!}+\ldots 1) \in \Theta(n!)$.