**Algoritmi şi structuri de date (I). Seminar 9:** Aplicaţii ale tehnicii reducerii. Analiza complexităţii algoritmilor recursivi.

**Problema 1** Să se genereze toate şirurile cu $n$ elemente din $\{0,1\}$. De exemplu, pentru $n=3$ sunt opt şiruri binare: $(0,0,0)$, $(0,0,1)$, $(0,1,0)$, $(0,1,1)$, $(1,0,0)$, $(1,0,1)$, $(1,1,0)$, $(1,1,1)$.

*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 ($p[k] \leftarrow 0$ şi $p[k] \leftarrow 1$). 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 2** 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:

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

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 $\Theta(n^3 \lg p)$.

**Problema 3** *Căutare poziţie inserare.* Se consideră un tablou $a[1..n]$ ordonat crescător şi $v$ o valoare. Să se determine, folosind un algoritm din $\mathcal{O}(\lg n)$, poziţia unde poate fi inserată valoarea $v$ în tabloul $a$ astfel încât acesta să rămână ordonat crescător.

*Rezolvare.*

O variantă de algoritm, bazată pe ideea de la căutarea binară, este:

```
cautare_pozitie(a[1..n], v)
li ← 1; ls ← n
if (v ≤ a[li]) return(li) endif
if (v ≥ a[ls]) return(ls + 1) endif
while (ls > li)
    m ← (li + ls)/2
    if (a[m] == v) then return(m + 1) endif
    if (v < a[m]) then ls ← m − 1
        else li ← m + 1
    endif
endwhile
if (v ≤ a[li]) return(li)
if (v > a[ls]) return(ls + 1)
```

Corectitudinea poate fi demonstrată folosind ca proprietate invariantă faptul că $a[li-1] \leq v \leq a[ls+1]$ (în ipoteza că se presupune formal că $a[0] = -\infty$ şi $a[n+1] = \infty$). Intrucât structura algoritmului este similară algoritmului de căutare binară ordinul de complexitate este $\mathcal{O}(\lg n)$.

O altă variantă a algoritmului este:

```
cautare_pozitie(a[1..n], v)
li ← 1; ls ← n
while (li ≤ ls)
    if (v ≤ a[li]) return(li) endif
    if (v ≥ a[ls]) return(ls + 1) endif
    m ← (li + ls)/2
    if (a[m] == v) then return(m + 1) endif
    if (v < a[m]) then ls ← m − 1
        else li ← m + 1
    endif
endwhile
return(li)
```

Şi în acest caz $a[li-1] \leq v \leq a[ls+1]$ este proprietate invariantă astfel că la ieşirea din ciclu (când $li = ls + 1$) are loc $a[li-1] \leq v \leq a[li]$. Prin urmare la ieşirea din ciclu trebuie returnată valoarea lui $li$.

Folosind acest algoritm de căutare binară a poziţiei de inserţie algoritmul de sortare prin inserţie poate fi transformat în:

```
insertie_binara(a[1..n])
for i ← 2, n do
    aux ← a[i]
    poz ←cautare_pozitie(a[1..i − 1],v)
    for j ← i − 1, poz, −1 do a[j + 1] ← a[j] endfor
    a[poz] ← aux
endfor
return a[1..n]
```

Din punctul de vedere al numărului de comparaţii efectuate algoritmul de inserţie binară are complexitatea $\mathcal{O}(n \lg n)$. Din punctul de vedere al numărului de deplasări de elemente efectuate algoritmul de sortare prin inserţie binară aparţine lui $\mathcal{O}(n^2)$.

**Problema 4** *Metoda bisecţiei.* Fie $f : [a, b] \to R$ o funcţie continuă având proprietăţile: (i) $f(a)f(b) < 0$; (ii) există un unic $x^*$ cu proprietatea că $f(x^*) = 0$. Să se aproximeze $x^*$ cu precizia $\epsilon > 0$.

*Rezolvare.* A determina pe $x^*$ cu precizia $\epsilon$ înseamnă a identifica un interval de lungime $\epsilon$ care conţine pe $x^*$ sau chiar un interval de lungime $2\epsilon$ dacă se consideră ca aproximare a lui $x^*$ mijlocul intervalului. Se poate aplica exact aceeaşi strategie ca la căutarea binară ţinându-se cont că $x^*$ se află în intervalul pentru care funcţia $f$ are valori de semne opuse în extremităţi.

```
bisectie(a,b,ε)
li ← a; ls ← b
repeat
   m ← (li + ls)DIV2
   if f(m) == 0 then return m endif
   if f(m) * f(li) < 0 then ls ← m
      else li ← m
   endif
until |ls − li| < 2ε
return (li + ls)DIV2
```

Complexitatea este determinată de dimensiunea intervalului $[a, b]$ şi de precizia dorită a aproximării, $\epsilon$. Notând $n = (b - a)/\epsilon$ şi observând că structura algoritmului este similară celui de la căutarea binară rezultă că algoritmul are complexitatea $\mathcal{O}(\lg n)$. Condiţia $f(m) == 0$ ar putea fi înlocuită cu o condiţie de tip $|f(m)| < \delta$ cu $\delta$ o valoare suficient de mică.

**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.

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
```

5. Descrieţi o variantă recursivă a algoritmului de căutare binară a unei poziţii de inserare a unei valori într-un tablou ordonat astfel încât acesta să rămână ordonat (problema 3 de mai sus).

6. *Căutare ternară.* Tehnica căutării binare poate fi extinsă prin divizarea unui subşir $a[li..ls]$ în trei subşiruri $a[li..m1]$, $a[m1+1..m2]$, $a[m2+1..ls]$, unde $m1 = li + \lfloor (ls - li)/3 \rfloor$ iar $m2 = li + 2\lfloor (ls - li)/3 \rfloor$. Care este ordinul de complexitate al algoritmului de căutare ternară descris mai jos?

```
cautare_ternara (a[1..n], v)
li ← 1; ls ← n;
while (li <= ls)
    m1 ← li + (ls − li)DIV3; m2 ← li + 2 ∗ (ls − li)DIV3
    if(x[m1] == v) then return(m1) endif
    if(x[m2] == v) then return(m2)endif
    if(v < x[m1]) then ls ← m1 − 1
    else if (v < x[m2])then li ← m1 + 1;ls ← m2 − 1;
       else li ← m2 + 1 endif
    endif endwhile
return(-1)
```