
Algoritmi și structuri de date (I). Seminar 9: Aplicații ale tehnicii reducerii. Analiza complexității algoritmilor recursivi.

Problema 1 Să se genereze toate șirurile cu n elemente din $\{0, 1\}$. De exemplu, pentru $n = 3$ sunt opt șiruri 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 șirul valorilor egale cu 0. În 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 afișat șirul (1, 1, ..., 1)
until r == 1
```

cu algoritmul de incrementare descris prin:

```
inc( p[1..n])
// se începe incrementarea de la cifra cea mai puțin 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 operația dominantă este împărțirea la 2, pentru fiecare dintre cei 2^n vectori se efectuează cel puțin o operație și cel mult n . Deci algoritmul aparține 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 împărțiri (la celelalte apeluri se efectuează mai puține). Pe de altă parte algoritmul de incrementare poate fi rescris fără a folosi operații de împărțire (vezi Probleme suplimentare).

O altă variantă, bazată pe tehnica reducerii, este:

```

generare( $k$ )
if  $k == 1$  then
     $p[1] \leftarrow 0$ 
    write ( $p[1..n]$ )
     $p[1] \leftarrow 1$ 
    write ( $p[1..n]$ )
else
     $p[k] \leftarrow 0$ 
    generare( $k - 1$ )
     $p[k] \leftarrow 1$ 
    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 funcției se completează poziția corespunzătoare lui k cu 0, respectiv 1 după care se apelează recursiv algoritmul pentru a genera toate subșirurile binare cu $k - 1$ elemente.

Pentru a determina ordinul de complexitate al algoritmului notăm cu $T(n)$ numărul de atribuiri efectuate ($p[k] \leftarrow 0$ și $p[k] \leftarrow 1$). Acesta va satisface relația de recurență:

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

Pentru rezolvarea relației de recurență se aplică metoda substituției 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 relațiilor și reducerea termenilor asemenea se obține $T(n) = 2(1 + 2 + \dots + 2^{n-2} + 2^{n-1}) = 2(2^n - 1) \in \Theta(2^n)$.

Problema 2 Să se calculeze A^p unde A este o matrice $n \times n$ și 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 și B specificate ca parametri de intrare.

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

```

putere1( $A[1..n, 1..n], p$ )
 $P[1..n, 1..n] \leftarrow A[1..n, 1..n]$ 
for  $i \leftarrow 2, p$  do  $P \leftarrow$  produs( $P, A$ )
endfor

```

Pentru a analiza complexitatea considerăm că dimensiunea problemei este determinată de perechea (n, p) și că operația dominantă este cea de înmulțire efectuată în cadrul algoritmului **produs**. Este ușor de stabilit că la fiecare apel se efectuează n^3 operații de înmulțire astfel că numărul total de înmulțiri efectuate de către algoritmul **putere1** este $T(n, p) = (p - 1)n^3 \in \Theta(pn^3)$.

Aplicând tehnica reducerii se obține algoritmul:

```

putere2(real  $A[1..n, 1..n]$ , integer  $p$ )
if  $p == 1$  then return  $A[1..n, 1..n]$ 
else if  $p \text{MOD} 2 == 0$  then
     $B[1..n, 1..n] \leftarrow$  putere2( $A, p/2$ )
    return produs( $B[1..n, 1..n], B[1..n, 1..n]$ )
else
     $B[1..n, 1..n] \leftarrow$  putere2( $A, (p - 1)/2$ )
     $B[1..n, 1..n] \leftarrow$  produs( $B[1..n, 1..n], B[1..n, 1..n]$ )
     $B[1..n, 1..n] \leftarrow$  produs( $B[1..n, 1..n], A[1..n, 1..n]$ )
    return  $B[1..n, 1..n]$ 
endif
endif

```

Numărul de înmulțiri efectuate în cadrul algoritmului satisface relația de recurență:

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

Considerăm cazul particular $p = 2^k$ și aplicăm metoda substituției 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 relațiilor de mai sus se obține $T(n, p) = n^3 \lg p$ pentru $p = 2^k$. Întrucât $T(n, p)$ este crescătoare și $n^3 \lg p$ este o funcție netedă rezultatul poate fi extins și 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 poziție inserare.* Se consideră un tablou $a[1..n]$ ordonat crescător și v o valoare. Să se determine, folosind un algoritm din $\mathcal{O}(\lg n)$, poziția 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 $a[0] = -\infty$ și $a[n + 1] = \infty$). 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)

```

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

Folosind acest algoritm de căutare binară a poziției de inserție algoritmul de sortare prin inserție 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 comparații efectuate algoritmul de inserție binară are complexitatea $\mathcal{O}(n \lg n)$. Din punctul de vedere al numărului de deplasări de elemente efectuate algoritmul de sortare prin inserție binară aparține lui $\mathcal{O}(n^2)$.

Problema 4 *Metoda biseției.* Fie $f : [a, b] \rightarrow R$ o funcție continuă având proprietățile: (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 ϵ înseamnă a identifica un interval de lungime ϵ care conține pe x^* sau chiar un interval de lungime 2ϵ dacă se consideră ca aproximare a lui x^* mijlocul intervalului. Se poate aplica exact aceeași strategie ca la căutarea binară ținându-se cont că x^* se află în intervalul pentru care funcția f are valori de semne opuse în extremități.

```

bisectie( $a, b, \epsilon$ )
 $li \leftarrow a; ls \leftarrow b$ 
repeat
   $m \leftarrow (li + ls) \text{DIV} 2$ 
  if  $f(m) == 0$  then return  $m$  endif
  if  $f(m) * f(li) < 0$  then  $ls \leftarrow m$ 
    else  $li \leftarrow m$ 
  endif
until  $|ls - li| < 2\epsilon$ 
return  $(li + ls) \text{DIV} 2$ 

```

Complexitatea este determinată de dimensiunea intervalului $[a, b]$ și de precizia dorită a aproximării, ϵ . Notând $n = (b - a)/\epsilon$ și observând că structura algoritmului este similară celui de la căutarea binară rezultă că algoritmul are complexitatea $\mathcal{O}(\lg n)$. Condiția $f(m) == 0$ ar putea fi înlocuită cu o condiție de tip $|f(m)| < \delta$ cu δ 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 operații de împărțire.
Indicație. Se parcurge tabloul cu cifrele binare de la cifra cea mai puțin semnificativă către cea mai și se pune pe 1 prima cifră egală cu 0 întâlnită.
2. Să se genereze toate cele 2^n submulțimi ale unei mulțimi cu n elemente.
Indicație. O submulțime poate fi reprezentată prin tabelul indicatorilor de prezență care este un șir binar cu n elemente.
3. Folosind tehnica divizării algoritmul de înmulțire 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 înmulțirii a două matrici pătratice de dimensiune $n \times n$. Consultați resurse web (folosind "Strassen algorithm" drept cheie de căutare) unde este descris algoritmul și încercați să îl implementați.
4. Stabiliți ce afișează algoritmul de mai jos atunci când este apelat pentru $k = n$ (în ipoteza ca lucrează asupra unui tablou global $a[1..n]$ inițializat astfel încât $a[i] = i$) și stabiliți 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 k MOD 2 == 1 then a[1] ↔ a[k]
        else a[i] ↔ a[k]
    endif
endif
endfor
endif

```

5. Descrieți o variantă recursivă a algoritmului de căutare binară a unei poziții 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 subșir $a[li..ls]$ în trei subșiruri $a[li..m1]$, $a[m1 + 1..m2]$, $a[m2 + 1..ls]$, 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)

```
