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

Problema 1 *Căutare eficientă a poziţiei de inserare într-un tablou ordonat.* 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. Să se analizeze ce efect are utilizarea acestui algoritm de căutare a poziţiei de inserţie în cazul în care este utilizat în cadrul algoritmului de sortare prin inserţie.

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 2 *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 $\epsilon > 0$. Stabiliţi ordinul de complexitate al algoritmului propus.

Rezolvare. A determina pe x^* cu precizia ϵ înseamnă a identifica un interval de lungime ϵ care conţine pe x^* sau chiar un interval de lungime 2ϵ 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, ϵ. Notând $n = (b-a)/\epsilon$ şi observând că structura algoritmului este similară celui de la căutarea binară rezultă că algoritmul are complexitatea $\mathcal{O}(\lg n)$. Având în vedere că în cazul operării cu valori de tip real relaţia de egalitate poate să nu fie satisfăcută exact, condiţia $f(m) == 0$ ar putea fi înlocuită cu o condiţie de tip $|f(m)| < \delta$ cu δ o valoare suficient de mică (de exemplu 10^{-6}.

Problema 3 *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 $m2 = li + 2\lfloor (ls - li)/3 \rfloor$. Descrieţi în pseudocod algoritmul de căutare ternară şi stabiliţi ordinul de complexitate.

Rezolvare. Ideea de bază este ca valoarea căutată să fie comparată cu $m1$ şi $m2$ şi să se continue căutarea în unul dintre cele 3 subtablouri: $x[li..m1 - 1]$, $x[m1 + 1, m2 - 1$, $x[m2 + 1, ls - 1]$.

```
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(a[m1] == v) then return(m1) endif
    if(a[m2] == v) then return(m2) endif
    if(v < a[m1]) then ls = m1 − 1
    else if (v < a[m2])then li = m1 + 1;ls = m2 − 1;
        else li = m2 + 1 endif
    endif endwhile
return(-1)
```

Considerând ca operaţii dominante comparaîile efectuate asupra elementelor din tablou, relaţia de recurenţă corespunzătoare timpului de execuţie satisface:

$$T(n) = \begin{cases} 0, & n = 0 \\ T(n/3) + c, & n > 0 \end{cases}$$

Prin dezvoltarea relaţiei de recurenţă, într-o manieră similară celei de la căutarea binară se obţine $T(n) \in \mathcal{O}(log_3 n) = \mathcal{O}(log n)$.

Probleme suplimentare

1. Se consideră un caroiaj de dimensiuni $2^k \times 2^k$ (pentru $k = 3$ se obţine o tablă de şah clasică) în care unul dintre pătrăţele este marcat. Propuneţi o strategie bazată pe tehnica divizării care acoperă întreaga tablă (cu excepţia pătrăţelului marcat) cu piese constând din 3 pătrăţele aranjate în forma de L (indiferent de orientare).

   Indicaţie. Se aplică tehnica divizării: se imparte pătratul în 4 pătrate de aceeaşi dimensiune şi se plasează o piesă astfel încât să ocupe un pătrăţel din fiecare dintre cele 3 pătrate care sunt complet libere. Ideea poate fi implementată folosind un algoritm recursiv care primeşte ca parametri indicii colţului din stânga jos, colţului din dreapta sus şi indicii pătrăţelului deja ocupat. La fiecare apel recursiv se va poziţiona o piesa prin completarea pătrăţelelor pe care le ocupă cu numărul de ordine al piesei. O variantă de implementare a acestei strategii (în ipoteza că `C` şi `nr` sunt variabile globale) este:

```
def triomino(i1,i2,j1,j2,ip,jp):
    global nr
    if (i1<i2) and (j1<j2):
        imij=(i1+i2)//2
        jmij=(j1+j2)//2
        nr=nr+1
        if (ip<=imij) and (jp<=jmij):
            C[imij][jmij+1]=nr
            C[imij+1][jmij]=nr
            C[imij+1][jmij+1]=nr
            triomino(i1,imij,j1,jmij,ip,jp)
            triomino(i1,imij,jmij+1,j2,imij,jmij+1)
            triomino(imij+1,i2,j1,jmij,imij+1,jmij)
            triomino(imij+1,i2,jmij+1,j2,imij+1,jmij+1)
        if (ip<=imij) and (jp>jmij):
            C[imij][jmij]=nr
            C[imij+1][jmij]=nr
            C[imij+1][jmij+1]=nr
            triomino(i1,imij,j1,jmij,imij,jmij)
            triomino(i1,imij,jmij+1,j2,ip,jp)
            triomino(imij+1,i2,j1,jmij,imij+1,jmij)
            triomino(imij+1,i2,jmij+1,j2,imij+1,jmij+1)
        if (ip>imij) and (jp<=jmij):
            C[imij][jmij]=nr
            C[imij][jmij+1]=nr
            C[imij+1][jmij+1]=nr
            triomino(i1,imij,j1,jmij,imij,jmij)
            triomino(i1,imij,jmij+1,j2,imij,jmij+1)
            triomino(imij+1,i2,j1,jmij,ip,jp)
            triomino(imij+1,i2,jmij+1,j2,imij+1,jmij+1)
        if (ip>imij) and (jp>jmij):
            C[imij][jmij]=nr
            C[imij][jmij+1]=nr
            C[imij+1][jmij]=nr
            triomino(i1,imij,j1,jmij,imij,jmij)
            triomino(i1,imij,jmij+1,j2,imij,jmij+1)
            triomino(imij+1,i2,j1,jmij,imij+1,jmij)
            triomino(imij+1,i2,jmij+1,j2,ip,jp)
```

   In cazul unei grile 8×8 în care celula ocupată este pe linia 2 şi coloana 3 se obţine:

```
[[3,  3,  4,  4,  8,  8,  9,  9],
 [3,  2,  2,  4,  8,  7,  7,  9],
 [5,  2,  6,  0,  10, 10, 7,  11],
 [5,  5,  6,  6,  1,  10, 11, 11],
 [13, 13, 14, 1,  1,  18, 19, 19],
 [13, 12, 14, 14, 18, 18, 17, 19],
 [15, 12, 12, 16, 20, 17, 17, 21],
 [15, 15, 16, 16, 20, 20, 21, 21]]
```

2. Propuneţi un algoritm de complexitate medie $\mathcal{O}(n \log n)$ pentru a verifica dacă elementele unui tablou sunt distincte sau nu.

   Indicaţie. Se ordonează crescător tabloul folosind un algoritm de complexitate $\mathcal{O}(n \log n)$, după care se parcurge tabloul ordonat şi dacă se identifică două elemente vecine identice se poate decide că tabloul nu conţine elemente distincte.

3. Se consideră un tablou ordonat crescător $a[1..n]$. Presupunem că tabloul a a fost transformat printr-o deplasare circulară la dreapta cu $k \geq 1$ poziţii. De exemplu, dacă $a = [2, 3, 6, 8, 9, 11, 14]$ iar $k = 3$ tabloul transformat este $a = [9, 11, 14, 2, 3, 6, 8]$.
   a) In ipoteza că valoarea k este cunoscută propuneţi un algoritm de complexitate $\mathcal{O}(1)$ care determină cea mai mare valoare din a.
   b) In ipoteza că valoarea k este necunoscută propuneţi un algoritm de complexitate $\mathcal{O}(\log n)$ care determină cea mai mare valoare din a.

   Indicaţie. (a) Tabloul fiind ordonat crescător, iniţial valoarea maximă se află pe poziţia n, iar după k deplasări se va afla pe poziţia k. (b) Se identifică poziţia p cu proprietatea că $x[p] > x[p+1]$ folosind ideea de la căutarea binară (după determinarea poziţiei m a elementului din mijloc se continuă căutarea în subtabloul caracterizat prin faptul că valoarea elementului din extremitatea stângă este mai mare decât valoarea elementului din extremitatea dreaptă. Un exemplu de implementare este:

```
def poz(a, s, d):
    if s<d:
        m=(s+d)//2
        if a[m]>a[m+1]:
            return m
        if a[s]>a[m]:
            return poz(a,s,m)
        else:
            return poz(a,m+1,d)
```

4. Se consideră un tablou cu numere naturale nenule distincte $a[1..n]$ ordonat crescător. Propuneţi un algoritm de complexitate $\mathcal{O}(\log n)$ care verifică dacă există $i \in \{1, \ldots, n\}$ cu proprietatea că $a[i] = i$.

   Indicaţie. Se aplică ideea de la căutarea binară: după determinarea mijlocului m se compară $a[m]$ cu m. Dacă $a[m] = m$ a fost găsit un element cu proprietatea cerută. Dacă $a[m] \neq m$ atunci se continuă căutarea în $a[m+1..d]$ (întrucât elementele sunt distincte nu e posibil ca $a[m] < m$, deci $a[m]$ trebuie să fie mai mare decât m prin urmare căutarea trebuie continuată doar în jumătatea din partea dreaptă a tabloului.