Paradigma Divide et Impera

Prezentarea generală a metodei

Studii de caz simple

1. *Produs de n factori*
2. *Determinarea minimului dintre n numere*
3. *Turnurile din Hanoi*

Studii de caz complexe

1. *Algoritmul de căutare binară*
2. *Algoritmi de sortare proiectaţi prin metoda divide et impera:*
   - *Algoritmul de sortare prin interclasare*
   - *Algoritmul de sortare rapidă*
3. *Determinarea rădăcinilor unei ecuaţii*
4. *Evaluarea polinoamelor*

Studii de caz avansate

1. *Problema plierilor*
2. *Dreptunghi de arie maximă*
3. *Generarea fractalilor*
   - *Triunghiul lui Sierpinski*
   - *Problema pătratului*
4. *Algoritmul lui Strassen de înmulţire a marticilor*
5. *Algoritm de înmulţire a numerelor foarte mari*

Aplicaţii propuse

Concluzii

Prezentare generală

Paradigma "*divide et impera*" se poate aplica problemelor care permit descompunerea lor în subprograme independente. Numele metodei exprimă concis această modalitate de rezolvare (*divide et impera* se poate traduce prin *împarte şi stăpîneşte*) iar rezolvarea unei astfel de probleme solicită trei faze:

Divide problema într-un număr de subprobleme independente, similare preoblemei iniţiale, de dimensiuni din ce în ce mai mici; descompunerea subproblemelor se opreşte la atingerea unor dimensiuni care permit o rezolvare simplă;

Stăpâneşte subproblemele prin rezolvarea acestora în mod recursiv. Dacă dimensiunile acestora sînt suficient de mici, rezolvă problemele în mod uzual, nerecursiv;

Combină soluţiile tuturor subproblemelor în soluţia finală pentru problema iniţială.

- Modelul matematic:
  - P(n): problema de dimensiune n
  - baza
    - daca $n \le n_0$ atunci rezolva P prin metode elementare
  - divide-et-impera
    - ***divide*** P in a probleme $P_1(n_1),\ldots,P_a(n_a)$ cu $n_a \le n/b$, $b > 1$
    - ***rezolva*** $P_1(n_1)$, …, $P_a(n_a)$ in aceeaşi maniera şi obţine soluţiile S_1, …, S_a
    - ***asamblează*** S_1, ..,S_a pentru a obţine soluţia S a problemei P

Paradigma divide-et-impera: algoritm

Pentru aceaşti algoritmi există atît implementarea recursivă cît şi iterativă, cea mai flexibilă şi cea mai utilizată fiind varianta recursivă.

```
procedure DivideEtImperaR(P, n, S)
begin
     if (n <= n0)
     then determina S prin metode elementare
     else imparte P in P1, ..., Pa
          DivideEtImperaR(P1, S1)
          ......................
          DivideEtImperaR(Pa, Sa)
          Asambleaza(S1, ..., Sa, S)
end
```

Paradigma divide-et-impera: complexitate

Această paradigmă conduce în general la un timp de calcul polinomial.

Vom presupune că dimensiunea n_i a subproblemei P_i satisface $n_i \leq n/b$, unde $b > 1$, în acest fel pasul de divizare reduce o subproblemă la altele de dimensiuni mai mici, ceea ce sigură proprietatea de terminare a subprogramului recursiv. Divizarea problemei în subprobleme şi asamblarea soluţiilor necesită timpul $O(n^k)$. Complexitatea timp T(n) a algoritmului *Divide et impera* este dată de următoare relaţie de recurenţă:

$$T(n) = \begin{cases} O(1), & daca\ n \leq n_0 \\ a \cdot T(\frac{n}{b}) + O(n^k), & daca\ n > n_0 \end{cases}$$

Teoremă: Dacă $n > n_0$ atunci:

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{daca} \quad a > b^k, \\ O(n^k \log_b n) & \text{daca} \quad a = b^k, \\ O(n^k) & \text{daca} \quad a < b^k. \end{cases}$$

Algoritmii *divide et impera* au o bună comportare în timp, dacă dimensiunile subproblemelor în care se împarte subproblema sînt aproximativ egale. Dacă lipsesc fazele de combinare a soluţiilor, viteza acestor algoritmi este şi mai mare (ex. *căutarea binară*). În majoritatea cazurilor, descompunerile înjumătăţesc dimensiunea problemei, un exemplu tipic reprezentându-l sortarea prin interclasare.

Studii de caz simple

Pe parcursul capitolului pentru fiecare studiu de caz se va analiza problema urmărind aplicarea paradigmei[1].

Pentru a fi înţeleasă esenţa metodei vor fi expuse câteva exemple simple, prezentate din motive didactice.

1. Produs de n factori

Enunţ*: Să se calculeze produsul a n numere reale.*

Schema problemei:

- generalizare: *produs* (`a[p..q]`)
- baza: `p ≥ q`

Soluţie:

- divide-et-impera
  - divide: `m = [(p + q)/2]`
  - subprobleme: *produs*(`a[p..m]`), *produs*(`a[m+1..q]`)
  - asamblare: înmulţeşte rezultatele subsecvenţelor *produs*(`a[p..m]`) şi *produs*(`a[m+1..q]`)

Întrucât la un moment dat nu putem calcula decât produsul a doi factori, produsul va trebui descompus în *subproduse*. O posibilitate de descompunere ar fi să împărţim numărul factorilor în jumătate, şi să considerăm produsele factorilor din cele două jumătăţi, procedând apoi la fel în continuare cu fiecare jumătate până ajungem la produse de doi factori. Acest mod de descompunere este ilustrat în figura următoare:

[1] In Anexa sint prezentate implement[rile Pascal/C]nso`ite de comentarii \i parte grafic[ (acolo unde este necesar) pentru to`i algoritmii aborda`i

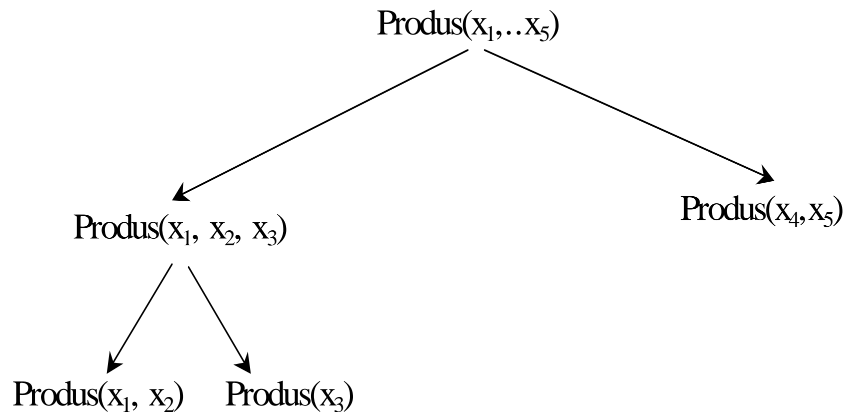


Fig. 1: Arborele descompunerii pentru un produs de n numere (n=5)

Produsul calculat conform acestei scheme de descompunere este redat în figura următoare; p_5 este produsul celor 5 factori, ordinea în care s-au calculat produsele este specificată de indicii produselor p_i.

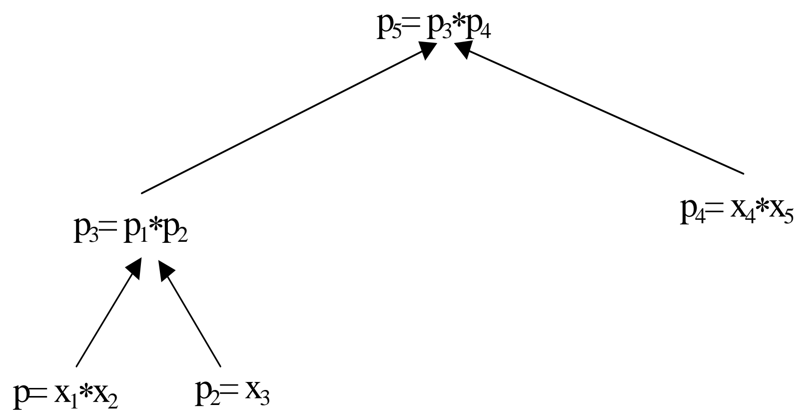


Fig. 2: Arborele soluţiei pentru un produs de n numere (n=5)

- complexitate: a = 2, b = 2, k = 1,T(n) = O(n)

2. Minim prin divide et impera

Enunţ: *Să se determine valoarea minimă dintr-un şir de n numere întregi*

Schema problemei:

- generalizare: *minim*(`a[p..q]`)
- baza: `p ≥ q`

Soluţie:

- divide-et-impera
  - divide: `m = [(p + q)/2]`
  - subprobleme: *minim*(`a[p..m]`), *minim*(`a[m+1..q]`)
  - asamblare: se combină soluţiile subproblemelor *minim*(`a[p..m]`) şi *minim*(`a[m+1..q]`)

"Divide et Impera" constă în împărţirea şirului de elemente în două subşiruri $\{a_1, a_2,..,a_m\}$, respectiv $\{a_{m+1},..,a_n\}$ unde m reprezintă mijlocul şirului.

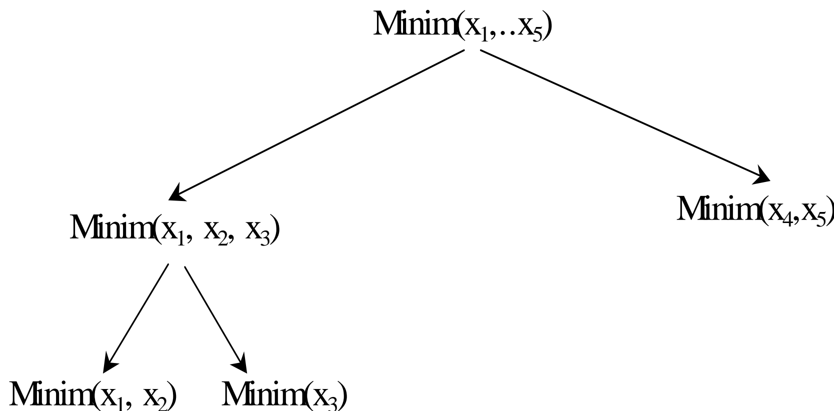


Fig. 3: Arborele descompunerii pentru determinarea minimului din n numere (n=5)

Astfel $min\{a_1,a_2,..,a_m\}=min\{min\{a_1, a_2,..,a_m\},min\{a_{m+1},..,a_n\}\}$.

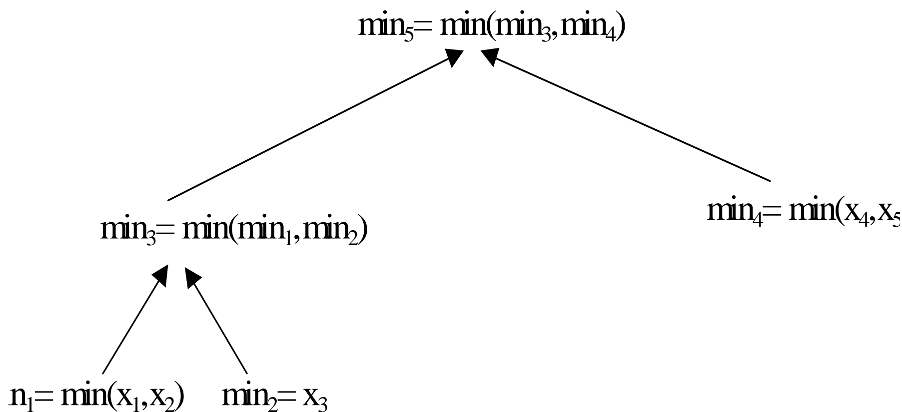


Fig. 4: Arborele soluţiei pentru minim din n numere (n=5)

- complexitate: T(n) = O(n)

3. Turnurile din Hanoi

***Enunţ**: Având un turn cu 8 discuri păstrate în ordinea descrescătoare a diametrelor pe una din cele trei tije, realizaţi mutarea acestui turn pe o altă tijă respectând următoarele reguli:*

- *se mută câte un singur disc;*
- *pe nici o tijă nu se poate aşeza un disc deasupra altuia de diametru mai mic;*
- *din cele trei tije, cea de-a doua reprezintă "tija auxiliară" putând fi folosită pentru păstrarea intermediară a discurilor.*

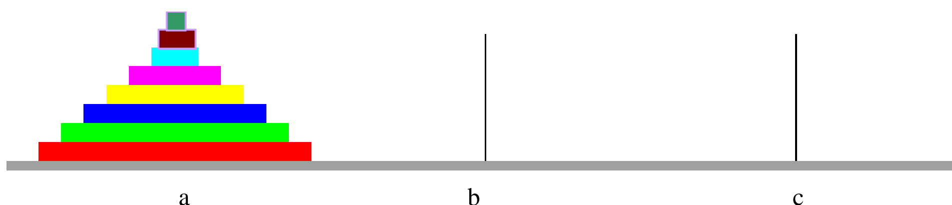


Fig. 5: Reprezentarea turnurilor din Hanoi (n=8)

Problema "Turnurilor din Hanoi" inventată de matematicianul francez Edouard Lucas in 1883 este inspirată din legendele orientale fiind un exemplu clasic în studierea paradigmei *divide et impera*.

Iniţial era un joc şi a fost găsit pentru prima dată printre scrierile ilustrului Mandarin FER-FER-TAM-TAM, publicate mai târziu din ordinul guvernului din China. Turnul din Hanoi era compus de 8 discuri de lemn, în ordinea descrescătoare a diametrelor iar în Japonia, în China şi Tonkin, acestea erau din porţelan. Jocul constă în demolarea turnului nivel cu nivel şi reconstituirea sa într-un loc apropiat conform regulii că *nu se poate aşeza un disc deasupra altuia de diametru mai mic*. Amuzant şi instructiv, uşor de învăţat şi jucat este foarte util în popularizarea problemei ştiinţifice.

Conform unei vechi *legende indiene*, brahmanii urmează fiecare de o lungă perioadă de timp paşii schimbării în Templul din Benares realizând mutarea turnului sacru al lui Brahma cu 64 discuri de aur împodobite cu diamante de Golconde după aceleaşi reguli, mutând un singur disc într-o zi.

În ziua când totul va fi gata stelele vor dispărea, turnul va cădea iar lumea întreagă se va sfârşi. Previziunea este destul de optimistă deoarece astăzi se cunoaşte că sunt necesare 2^{64}-1 mutări (zile) 18 446 744 073 709 551 615 mutări ceea ce înseamnă mai mult de *5 bilioane de secole!*

Schema problemei:

- generalizare: `TH(n), n>0`

Soluţie:

- divide-et-impera
  - subprobleme: `TH(n-1,a,c,b),TH(1),TH(n-1,c,b,a);`
  - asamblare: **nu există,** deoarece soluţia problemei iniţiale a fost obţinută simultan cu etapele de descompunere
- complexitate: $T(n) = O(2^n)$

Analiza complexităţii:

O mutare poate fi scrisă ca o pereche (i,j) cu $(i,j) \in \{1,2,3\}$ semnificând că se mută discul de pe tija i pe tija j. Vom nota H(m,i,j) şirul mutărilor necesare pentru a muta primele m discuri de pe tija i pe tija j, folosind şi tija de manevra k=6-i-j astfel că problema revine la a determina H(n,1,2). Mutările pot fi reprezentate prin etichetele unui arbore binar complet avînd n nivele, construit astfel:
-rădăcina e etichetată (1,2).
-fiecare vîrf aflat pe un nivel $<n$ şi etichetat (i,j) are descendent stâng nodul (i,k) iar descendent drept nodul (k,j).

Astfel sînt 2^n-1 mutări succesiunea corectă fiind obţinută prin parcurgerea arborelui în *inordine*.
Pentru exemplul din figură succesiunea celor 2^3-1 mutări este: (1,2), (1,3), (2,3), (1,2), (3,1), (3,2), (1,2).

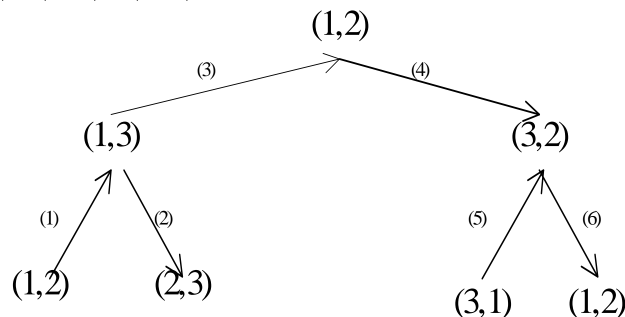


Fig.6. Parcurgerea în inordine a arborelui binar pentru mutările celor n discuri (n=3)

Studii de caz complexe

1. Căutarea binară

Problema regăsirii informaţiei stocate în memorie e o problemă foarte frecventă şi care poate solicita un timp mare atunci când numărul înregistrărilor este ridicat.

Enunţ: *Fiind dat un şir de n numere întregi, ordonat strict crescător, să se determine poziţia pe care se află în şir o anumită valoare x citită de la tastatură. În cazul în care valoarea nu se află în şir, să se specifice acest lucru printr-un mesaj.*

Schema problemei:

- generalizare: *caută* (`a[p..q],x`)
- baza: `p ≥ q`

Soluţie:

Observaţie: prin rezolvarea clasică (comparând pe rând valoarea căutată cu fiecare componentă a vectorului) în cazul unei căutări cu succes numărul mediu de comparaţii este n/2 iar în cazul unei căutări fără succes numărul de comparaţii este egal cu n. De aceea este mult mai eficient să se ordoneze elementele vectorului iar apoi să se aplice algoritmul următor.

- divide-et-impera
  - divide: `m = [(p + q)/2]`
  - subprobleme: *caută* (`a[p..m],x`), *caută* (`a[m+1..q],x`)
  - asamblare: nu există, deoarece soluţia unei subprobleme reprezintă soluţia problemei iniţiale

- complexitate: $T(n) = O(\log_2 n)$

Analiza complexităţii

Reprezentarea strategiei de căutare printr-un arbore de decizie binar (ilustrată în fig. 7) ajută la evaluarea performanţelor algoritmului. Decizia de continuare a procesului de căutare, adică fie oprirea căutării, fie căutarea pe ramura din stânga, fie căutarea pe ramura din dreapta a unui nod, depind de rezultatul comparaţiei numărului căutat cu numărul din şir

indicat de nodul în care ne aflăm. Evaluarea eficienţei acestui algoritm este dată de următoarea teoremă, care poate fi demonstrată prin inducţie după k.

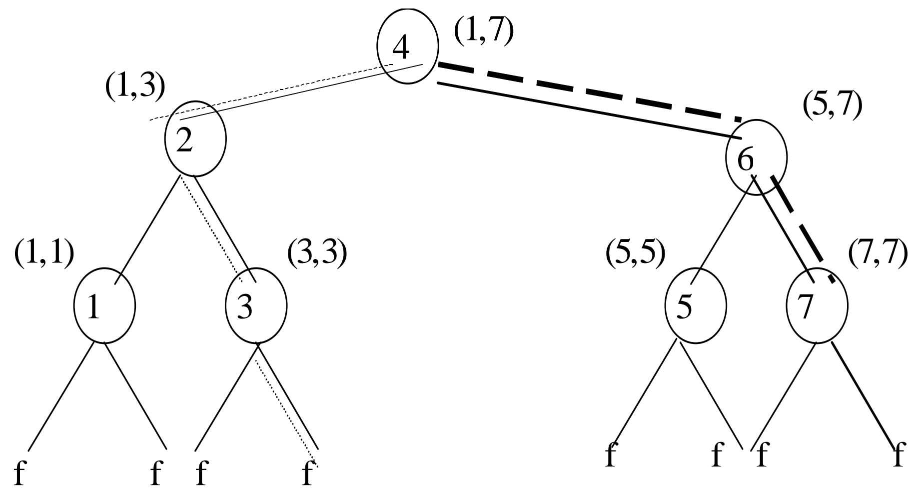


Fig.7 Arborele de căutare binară: n=7 şi valorile 3, 5, 12, 14, 15, 18, 24
Traseul din stânga ilustrează căutarea fără succes a numărului x=13 iar traseul din dreapta ilustrează căutarea cu succes a numărului x=24

Teoremă: *Dacă efectuăm o căutare într-un şir ordonat de n numere cu algoritmul de căutare binară şi dacă n verifică inegalităţile:*
$2^{k-1} \le n < 2^k$ *atunci:*
a) o căutare cu succes necesită cel mult k comparaţii;
b) o căutare fără succes necesită k-1 sau k comparaţii;
Logaritmând relaţia precedentă obţinem: $k-1 \le \log_2 n < k$, *deci* $k=[\log_2 n]+1$

De exemplu, dacă vrem să căutăm un număr printre 1000 de înregistrări cu algoritmul clasic de comparare ar trebui efectuate în medie 500 de operaţii în cazul unei căutări cu succes şi 1000 de căutări în cazul fără succes. Dacă insă înregistrările sunt sortate, aplicând teorema anterioară rezultă $k=[\log_2 1000]+1=10$, iar căutarea fără succes necesită 9 sau 10 comparaţii, obţinându-se astfel o importantă reducere a numărului de comparaţii deci şi a timpului de căutare.

2. Algoritmi de sortare

Sortarea unui şir de elemente este o operaţie foarte importantă şi pentru aceasta se pot folosi mai mulţi algoritmi. Metodele cunoscute de sortare "*Inserţie*", "*Selecţie*", "*Interschimbare*" au complexitatea $O(n^2)$ dar vom arăta că există algoritmi mai performanţi.

2.1. Sortare prin interclasare (Merge sort)

Strategia acestui algoritm de sortare se bazează pe împărţirea şirului iniţial în două subşiruri de lungimi aproximativ egale, sortarea crescătoare a ambelor subşiruri şi apoi interclasarea lor. Pentru cele două subşiruri se aplică aceeaşi metodă. Procesul de înjumătăţire continuă până când se obţin şiruri de lungime 1 (ordonate) sau 2, acestea din urmă tratându-se direct.

Schema problemei:

- generalizare: *sortează*(`a[p..q]`)
- baza: `p ≥ q`

Soluţie:

- divide-et-impera
  - divide: `m = [(p + q)/2]`
  - subprobleme:*sortează*(`a[p..m]`), *sortează*(`a[m+1..q]`)
  - asamblare: interclaseaza subsecvenţele sortate `a[p..m]` şi `a[m+1..q]`
- complexitate: T(n) = O(n log n)

Analiza complexităţii:

Pentru analiza complexităţii urmărim diagrama care descrie modul de sortare al unui şir de numere (valorile 16, 50, 25,14, 86, 35). Observăm că operaţia de interclasare necesită O(n) comparaţii pe fiecare nivel al diagramei, unde n reprezintă lungimea vectorului iniţial. Cum numărul de nivele este $\log_2 n$, concluzia este că acest algoritm are complexitatea O(nlogn), fiind una din cele mai eficiente metode de sortare.

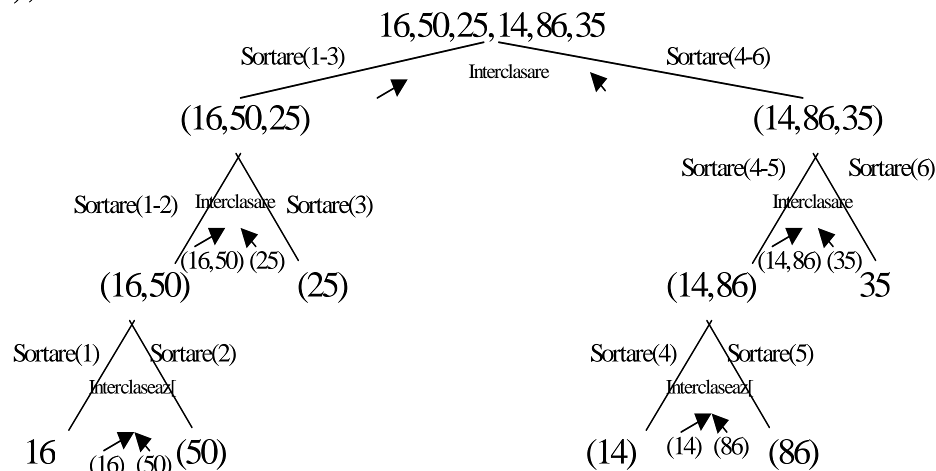


Fig. 8: Arborele soluţiei pentru sortarea prin interclasare a unui sir de n numere (n=5)

2.2. Sortare rapida (Quick sort)

Algoritmul Quicksort a fost realizat de C.A.R. Hoare în 1962, strategia prin care elementele tabloului sunt ordonate bazându-se pe paradigma *divide et impera*.

Schema problemei:

- generalizare: a[p..q]
- baza: $p \geq q$

Soluţie:

- divide-et-impera
  - divide: determină k intre p si q prin interschimbări astfel că după determinarea lui k avem:
    - $p \leq i \leq k \Rightarrow a[i] \leq a[k]$
    - $k < j \leq q \Rightarrow a[k] \leq a[j]$
  - subprobleme: a[p..k-1], a[k+1..q]
  - asamblare: **nu exista**;

Fazele algoritmului se exprimă astfel:

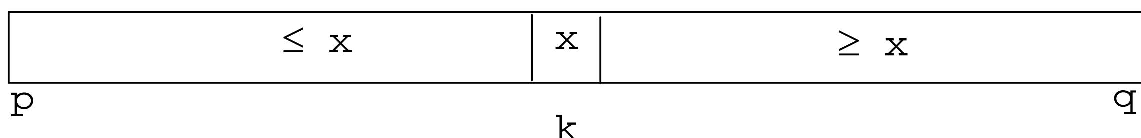
Pas 1. se selectează primul element a[p] care va fi numit pivot şi se realizează partiţionarea tabloului în două subtablouri care cuprind grupul elementelor cu valori mai mici decât ale pivotului (a[1]..a[poz-1]), respectiv, grupul elementelor cu valori mai mari sau egale cu ale pivotului (a[poz+1]…a[q]).

Împărţirea este realizată prin intermediul pivotului, el fiind plasat pe poziţia finală, **poz** ocupată în tabloul sortat, adică după elementele mai mici ca el şi înaintea grupului de elemente mai mari sau egale cu el.

Pas 2. sortarea celor două subtablouri obţinute reprezintă subproblemele în care a fost descompusă problema iniţială, ele fiind rezolvate prin descompuneri succesive până la obţinerea unor subtablouri de dimensiune 1 (cazul de bază).

Quick sort: partiţionare

- iniţial:
  - alege x din a[p..q] (ex. $x \leftarrow a[p]$)



⇨ `i ← p; j ← q`

- pasul curent:

⇨ daca `a[i] ≤ x` atunci `i ← i+1`

⇨ daca `a[j] ≥ x` atunci `j ← j-1`

⇨ daca `a[i] > x > a[j]` si `i < j` atunci

- `swap(a[i], a[j])`
- `i ← i+1`
- `j ← j-1`

pasul curent se execută atit timp cît `i ≤ j.`

Faza de combinare a soluţiilor subproblemelor lipseşte, deoarece în cadrul fiecăreia un element desemnat ca pivot este plasat pe poziţia finală în tabloul sortat, deci soluţia problemei iniţiale se construieşte simultan cu descompunerea ei în subprobleme.

- complexitate medie = O(n log n)

Analiza complexităţii

Pentru analiza complexităţii acestui algoritm se pleacă de la două situaţii, generate de poziţia pe care o ocupă pivotul în urma operaţiei de partiţionare. Astfel, **în cel mai favorabil caz**, pivotul este plasat în cadrul fiecărei subprobleme pe poziţia din mijloc a subtabloului, în această situaţie problema fiind descompusă în două subprobleme de aceeaşi dimensiune. Cum fiecare subtablou de **m** elemente necesită maxim m comparaţii, este clar că pentru un număr de m autoapeluri pentru subtablouri de dimensiune **n div m** vor fi necesare O(n) comparaţii. Cum există **$\log_2 n$** nivele, rezultă că algoritmul Quicksort are o complexitate **O(*n*log*n*)** în cazul cel mai favorabil.

Există însă situaţii când această complexitate este deteriorată: de exemplu un tablou care are deja elementele sortate-**cazul cel mai defavorabil**. La descompunerea problemei iniţiale se va obţine o singură subproblemă care urmăreşte sortarea unui tablou cu n-1 elemente. Deoarece numărul de nivele din recursivitate este n, complexitatea algoritmului degenerează către **$O(n^2)$**. Pentru a evita această deteriorare a complexităţii Knuth D.E. a propus efectuarea unei permutări aleatoare a tabloului iniţial, înaintea aplicării algoritmului Quicksort.

De asemenea trebuie reţinut că acest algoritm de sortare presupune alocarea implicită a unei stive (datorată recursivităţii) de lungime ce variază între **$\log_2 n$** şi **n**.

3. Determinarea rădăcinilor unei ecuaţii

Enunţ: *Se dă o funcţie continuă care are semne contrare în cele două capete ale intervalului [a,b], $F(a)\cdot F(b)<0$. Determinaţi o rădăcină a lui F din intervalul [a, b], cu eroarea ε.*

Schema problemei:

- generalizare: `[a..b]`
- baza: `a ≥ b`

Soluţie:

- divide-et-impera
  - divide: `m = [(a + b)/2[`
  - subprobleme: `a[a..m], a[m+1..b]`
  - asamblare: nu există

Notăm c mijlocul intervalului [a, b], adică $c=(a+b)/2$ şi definim $a_1=a$ şi $b_1=c$, dacă $f(c)>0$ respectiv $a_1=c$ şi $b_1=b$, dacă $f(c)<0$. Dacă $f(c)=0$ atunci soluţia ecuaţiei este c. În caz contrar, funcţia $f: [a_1, b_1]\rightarrow R$, satisface ipotezele dar lungimea intervalului de definiţie este redusă la jumătate. Metoda poate fi continuată prin alegerea mijlocului intervalului $[a_1, b_1]$ iar noul interval $[a_2, b_2]$ va fi definit exact în acelaşi mod ca cel precedent, procedeul continuând pentru intervalul $[a_n, b_n]$.

Dacă vrem să obţinem soluţia aproximativă a ecuaţiei $f(x)=0$ cu eroarea ε, va trebui să calculăm termenii şirului c_n până la prima valoare a lui n care satisface inegalitatea: $\frac{b-a}{2^n}<\varepsilon$, adică $n=[\ln((b-a)/\varepsilon)/\ln 2]+1$.

- complexitate: T(n) = O(n)

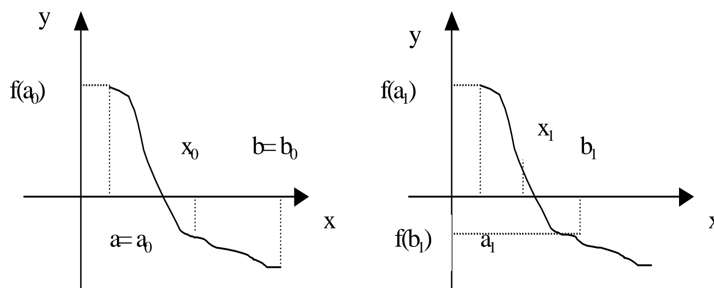


Fig. 9 Determinarea soluţiei unei ecuaţii prin metoda bisecţiei

4. Evaluarea polinoamelor

Enunţ: *Scrieţi un algoritm divide et impera care să calculeze valoarea unui polinom într-un punct.*

Schema problemei:

- generalizare: `a[p..q]`
- baza: `p ≥ q`

Soluţie:

- divide-et-impera
  - divide: `m = [n/2]`
  - subprobleme: `a[p..m], a[m+1..q]`
  - asamblare: determină valoarea finală prin combinare rezultatelor subsecvenţelor

Fie polinomul $P(x)=a_n x^n+..+a_1 x+a_0$. Vom calcula valoarea sa în punctul y folosind exprimarea $P(y)=Q(y)*y^{[n/2]}+T(y)$, urmând să evaluăm polinoamele Q şi T în x. Pentru Q şi T folosim o scriere analoagă, până când gradul lor este 0 sau 1.

- complexitate: $T(n) = O(n)$

Studii de caz avansate

1. Dreptunghi de arie maximă

Enunţ: *Se dau coordonatele vârfurilor stânga-jos şi dreapta-sus ale unui dreptunghi cu laturile paralele cu axele de coordonate. Se dau coordonatele a n puncte din plan. Să se determine un dreptunghi de arie maximă, cu laturile paralele cu axele, cuprins în dreptunghiul iniţial, cu proprietatea că punctele date nu se află în interiorul dreptunghiului.*

Schema problemei:

- generalizare: x_j,y_j,x_s,y_s (coordonatele dreptunghuilui), n, x[n],y[n], *dreptunghi[* i , x_j,y_j,x_s,y_s]

Soluţie:

- divide-et-impera
  - subprobleme:
    dreptunghi[i+1,x_j,y_j,x_s,y[i]],
    dreptunghi[i+1,x_j,y[i],x_s,y_s],
    dreptunghi[i+1,x_j,y_j,x[i]$_s$,y_s],
    dreptunghi[i+1, x[i],y_j,x_s,y_s].
  - asamblare: nu există.

Dacă toate punctele sunt exterioare dreptunghiului dat sau se află pe laturile sale, dreptunghiul căutat este chiar cel iniţial.
Dacă există un singur punct interior dreptunghiului, ducând prin punctul respectiv o paralelă la una din axele de coordonate se obţin alte două dreptunghiuri care nu mai conţin punctul interior (fig. 10).

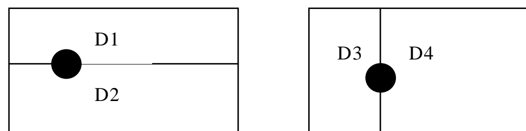


Fig.10 Exemplu pentru problema dreptunghiului

Din cele patru dreptunghiuri formate va trebui reţinut cel de arie maximă. Dacă există două puncte interioare dreptunghiului, al doilea punct ar putea fi punct interior la doar unul dintre dreptunghiurile D_1, D_2, respectiv D_3, D_4, regăsim astfel problema iniţială pentru dreptunghiurile D_1, D_2, D_3, D_4 şi punctul al doilea, toate acestea conducând la o abordare

divide et impera. Problema iniţială se descompune în patru subprobleme corespunzătoare celor patru dreptunghiuri formate prin trasarea celor două paralele prin primul punct interior al dreptunghiului. Fie acesta punctul *i* de coordonate (x_i,y_i). În continuare, punctele interioare vor putea fi numai punctele $j \in \{i+1,.., n\}$. Descompunerea continuă în acelaşi mod, până se ajunge la dreptunghiuri fără puncte interioare ale căror arii se calculează şi se reţine maximul. Observăm că nu este necesară o etapă de combinare a soluţiilor.

2. Problema plierilor

Enunţ: *Se consideră un vector de lungime n. definim* ***plierea*** *vectorului prin suprapunerea unei jumătăţi, numită* ***donatoare****, peste cealaltă jumătate, numită receptoare, cu precizarea că dacă numărul de elemente este impar, elementul din mijloc este eliminat. Prin pliere, elementele subvectorului obţinut vor avea numerotarea jumătăţii receptoare. Plierea se poate aplica în mod repetat, până când se ajunge la un subvector format dintr-un singur element, numit* ***element final****. Scrieţi un program care să afişeze toate elementele finale posibile şi să se figureze pe ecran pentru fiecare element final o succesiune de plieri.*

Schema problemei:

- generalizare: *pliază[*`p..q]`
- baza: `p ≥ q`

Soluţie:

- divide-et-impera
  - divide: `m = [n/2]`
  - subprobleme:
  dacă (q-p+1) mod 2=1 *atunci* Ls=(p+q)div 2-1 *altfel* Ls=(p+q) div 2; Ld=(p+q) div 2 +1;
  pliază[`p..Ls]`, *pliază[*`Ld..q]`
  - asamblare: nu există;

Pentru determinarea tuturor elementelor finale şi succesiunile de plieri corespunzătoare pentru un vector cu numerotarea elementelor p..q, vom utiliza o procedură *Pliaza(p,q)*. Dacă p=q, atunci vectorul este format dintr-un singur element, deci final. Dacă p<q, calculăm numărul de elemente din vector (q-p+1).

Dacă numărul de elemente din vector este impar, elementul din mijloc (*(p+q)div2*) este eleminat, plierea la stânga se face de la poziţia (p+q)*div*2 -1, iar plierea la dreapta de la poziţia (p+q) *div*2+1.

Dacă numărul de elemente din vector este par, plierea la stânga se poate face de la poziţia (p+q) div 2 iar plierea la dreapta de la poziţia (p+q)div2+1. Vom codifica plierea la stînga reţinând şirul mutărilor simbolul 'S' urmat de poziţia *Ls*, de la care se face plierea spre stânga, iar o pliere la dreapta prin simbolul 'D', urmat de poziţia *Ld* de la care se face plierea spre dreapta. Pentru a determina toate elementele finale din intervalul p..q, apelăm recursiv procedura *Pliaza* pentru prima jumătate a intervalului, precedând toate succesiunile de plieri cu o mutare spre stânga, apoi apelăm procedura *Pliaza* pentru cea de-a doua jumătate a intervalului, precedând toate succesiunile de plieri corespunzătoare de o pliere la dreapta.

Exemplu, pentru n=7, elementele finale şi plierile corespunzătoare sînt:

1: S3 S1
3: S3 D3
5: D5 S5
7: D5 D7

3. Fractali

Figuri ciudate şi aparent paradoxale, fractalii exercită o adevărată fascinaţie pentru matematicieni şi informaticieni.

Noţiunea de *fractal* a apărut ca urmare a studiului vieţii reale, în care informaţia genetică conţinută în nucleul unei celule se repetă la diferite scări.

Fractalii sunt forme caracterizate printr-o extremă fragmentare, neregularitate, asemănarea detaliului cu întregul indiferent de scara de observaţie, în ultimul timp aceştia jucând un rol vital în toate ramurile ştiinţelor moderne. Primii fractali au fost creaţi de matematicienii Waclaw Sierpinski, David Hilbert, George Cantor sub forma unor exerciţii abstracte.

Dincolo de spectaculosul imediat al reprezentărilor, fractalii şi-au găsit aplicaţii nu doar în programele de grafică ci şi în domenii cum ar fi compresia datelor sau criptografie.

***Iterarea pătratului**. Realizaţi un program recursiv care să afişeze pe ecran în mod grafic un fractal simplu obţinut prin iterarea unui pătrat. Latura unui pătrat se citeşte de la tastatură (0<L<260) iar iterarea se face pîna la obţinerea pătratelor de latură 1.*

Indicaţie: Paradigma *divide et impera* ne permite să desenăm mai întâi pătratele cu cele mai mici laturi şi apoi, pe rând pe celelalte. Pentru aceasta, desenul unui pătrat trebuie să se afle la sfârşitul procedurii recursive, executându-se după revenirea din activările ulterioare ale procedurii în care s-au desenat pătratele cu laturile mai mici decât latura pătratului curent.

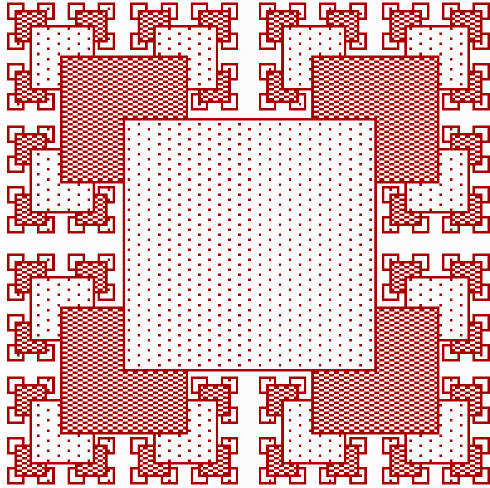


Fig. 11. Iterarea pătratului

Triunghiul lui Sierpinski: *Fie un triunghi iniţial care se împarte în 4 triunghiuri egale. Fiecare din cele 4 triunghiuri obţinute se împarte la rândul lui în 4 triunghiuri egale, procedeul continuând la infinit. Realizaţi un program recursiv care citind de la tastatură coordonatele vârfurilor triunghiului iniţial şi n- numărul iteraţiilor afişează în mod grafic un triunghi Sierpinski după n iteraţii.*

Indicaţie: Aplicînd paradigma *divide et impera,* pentru a putea genera fractalul vom determina mijloacele laturilor triunghiului, le vom uni prin linii, determinând astfel cele 4 triunghiuri incluse în triunghiul iniţial, apoi procedeul se repetă pentru fiecare din cele 4 triunghiuri nou obţinute.

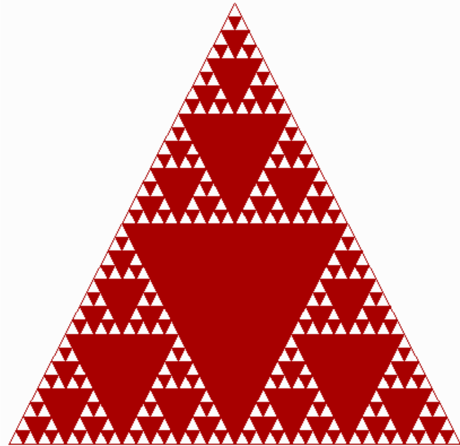


Fig. 12 Triunghiul lui Sierpinski

4. Înmulţirea matricelor

Pentru matricile A şi B de $n \times n$ elemente, dorim să obţinem matricea produs $C=AB$. Algoritmul clasic provine direct din definiţia înmulţirii a două matrici şi necesită n^3 înmulţiri şi $(n-1)n^2$ adunări scalare, astfel că timpul necesar pentru calcularea C este în $O(n^3)$. Intenţia este să găsim un algoritm de înmulţire matriceală al cărui timp să fie într-un ordin mai mic decît n^3. Pe de altă parte, este clar că $O(n^2)$ este o limită inferioară pentru orice algoritm de înmulţire matriceală, deoarece trebuie în mod necesar să parcurgem cele n^2 elemente ale lui C.

Paradigma *divide et impera* sugerează un alt mod de calcul al matricii C. Vom presupune că n este o putere a lui 2. Partiţionăm pe A şi B în cîte patru submatrici de $n/2 \times n/2$ elemente fiecare. Matricea produs C se poate calcula conform formulei pentru produsul matricilor de 2×2 elemente:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

unde $C_{11}=A_{11}B_{11}+A_{12}B_{21}$ $\quad C_{12}=A_{11}B_{12}+A_{12}B_{22}$

$C_{21}=A_{21}B_{11}+A_{22}B_{21}$ $\quad C_{22}=A_{21}B_{12}+A_{22}B_{22}$

Pînă aici nu am cîştigat nimic faţă de metoda clasică: este de dorit ca în calcularea submatricilor C să folosim mai puţine înmulţiri, chiar dacă prin aceasta mărim numărul de adunări (adunarea matricilor necesită un timp pătratic, în timp ce înmulţirea matricilor necesită un timp cubic). Astfel, în 1969, Strassen a descoperit o metodă de calculare a submatricilor C_{ij}, care pentru $n=2$ utilizează 7 înmulţiri şi 18 adunări şi scăderi. Pentru început se calculează şapte matrici de $n/2 \times n/2$ elemente:

$P=(A_{11}+A_{22})(B_{11}+B_{22})$

$Q=(A_{21}+A_{22})B_{11}$

$R=A_{11}(B_{12}-B_{22})$

$S=A_{22}(B_{21}-B_{11})$

$T=(A_{11}+A_{12})B_{22}$

$U=(A_{21}-A_{11})(B_{11}+B_{22})$

$V=(A_{12}-A_{22})(B_{21}+B_{22})$

Este uşor de verificat că matricea produs C se obţine astfel:

$C_{11}=P+S-T+V$ $\quad C_{12}=R+T$

$C_{21}=Q+S$ $\quad C_{22}=P+R-Q+U$

Timpul total pentru noul algoritm divide et impera este $t(n) \in 7t(n/2)+\Theta(n^{\lg 7})$. Cum $\lg 7<2.81$,rezultă că $t \in O(n^{2.81})$, algoritmul lui Strassen fiind mai eficient decît algoritmul clasic de înmulţire matriceală.

Metoda lui Strassen nu este unică: s-a demonstrat că există exact 36 de moduri diferite de calcul a submatricilor C_{ij}, fiecare din aceste metode utilizând 7 înmulţiri.

Practic, la calculator s-a putut observa că, pentru n≥40, algoritmul lui Strassen este mai eficient decât metoda clasică dar în schimb foloseşte memorie suplimentară.

5. Înmulţirea numerelor întregi mari

Pentru anumite aplicaţii, trebuie să lucrăm cu numere întregi foarte mari, care nu mai pot fi reprezentate prin tipurile standard existente. Astfel vom da un algoritm *divide et impera* pentru înmulţirea întregilor foarte mari.

Fie *u* şi *v* doi întregi foarte mari, fiecare de n cifre zecimale. Dacă $s=\lfloor n/2 \rfloor$, reprezentăm pe u şi v astfel:

$u=10^s w+x$, $v=10^s y+z$, unde $0 \le x < 10^s$, $0 \le z < 10^s$

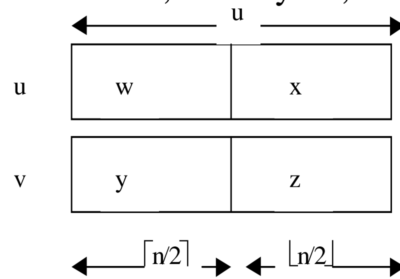


Fig. 13 Reprezentarea numerelor u şi v la înmulţirea întrgilor foarte mari

Întregii *w* şi *y* au câte $\lceil n/2 \rceil$ cifre, iar întregii x şi z au câte $\lfloor n/2 \rfloor$ cifre. Din relaţia $uv=10^{2s}wy+10^s(wz+xy)+xz$ obţinem următorul algoritm *divide et impera* de înmulţire a două numere întregi mari:

function înmulţire(u,v)
    n←cel mai mic întreg astfel încît u şi v să aibă fiecare n cifre
    ***if** n este mic **then** calculează în mod clasic produsul uv*
                ***return** produsul uv calculat*
    s←n div 2
    w←u div 10^s; x←u mod 10^s;
    y←v div 10^s; z←v mod 10^s;
return înmulţire(w,y)×10^{2s}+(înmulţire(w,z)+înmulţire(x,y))×10^s+înmulţire(x,z)

Deoarece înmulţirea întregilor mari este mult mai lentă decât adunarea, încercăm să reducem numărul înmulţirilor, chiar dacă prin aceasta mărim numărul adunărilor. Astfel, încercăm să calculăm

wy,wz+xy şi xz prin mai puţin de patru înmulţiri. Considerând produsul
$r=(w+x)(y+z)=wy+(wz+xy)+xz$
observăm că putem înlocui ultima linie din algoritm cu

r←înmulţ(w+x,y+z)
p←înmulţ(w,y); q←înmulţ(x,z);
return $10^{2s}p+10^{s}(r-p-q)+q$

Ca şi la metoda lui Strassen, datorită constantelor multiplicative implicate, acest algoritm este interesant în practică doar pentru valori mari ale lui n.

Aplicaţii propuse

Probleme pentru consolidarea cunoştinţelor şi lucrări de verificare

1. *Cel mai mare divizor comun a n numere.* Fie n valori naturale nenule a_1, $a_2,..,a_n$. determinaţi cel mai mare divizor comun al lor *cmmdc*($(a_1, a_2,..,a_n)$)
Indicaţie: Soluţia "Divide et Impera" constă în împărţirea şirului de elemente în două subşiruri $\{a_1, a_2,..,a_m\}$, respectiv $\{a_{m+1},..,a_n\}$ unde m rerpezintă mijlocul şirului.
Astfel: $cmmdc(a_1,a_2,..,a_m)=cmmdc(cmmdc(a_1,a_2,..,a_m),cmmdc(a_{m+1},..,a_n))$

2. *Min-Max*. Realizati un subprogram care folosind tehnica "Divide et impera" să determine simultan minimul şi maximul unui şir de numere

3. *Detaliu*. Avînd tabloul de numere întregi (3,2,5,4,7,6,8,2,12) care va fi configuraţia lui după efectuarea a trei partiţionări specifice algoritmului Quicksort?
4. *Calculul factorialului*. Fie n natural. Să se calculeze n! printr-un algoritm *Divide et Impera*

5. *Înmulţirea polinoamelor*: Fie P(x) şi Q(x) două polinoame de grad n, cu coeficienţi reali. Să se înmulţească cele două polinoame, utilizând metoda "Divide et Impera". Comparaţi această metodă de înmulţire a polinoamelor cu varianta "clasică"
Indicaţie: Vom grupa termenii celor două polinoame în felul următor:
$p(x)=p_1(x)+x^{N/2}p_2(x)$, $q(x)=q_1(x)+x^{N/2}q_2(x)$
Vom calcula următoarele produse parţiale, observând că se înmulţesc polinoame avînd gradul N/2:
$r_1(x)=p_1(x)* q_1(x)$, $r_2(x)=(p_1(x)+ p_2(x))*(q_1(x)+q_2(x))$, $r_3(x)=p_2(x)*q_2(x)$
Atunci polinomul produs se scrie:
$r(x)=r_1(x)+(r_2(x)-r_1(x)-r_3(x))*x^{N/2}+r_3(x)*x^N$
Deoarece înmulţirea polinoamelor necesită N^2 paşi, continuând algoritmul de mai sus, obţinem *n*logn* paşi.

6. *Problema selecţiei*: Fie a un tablou de n elemente şi k un număr întreg, $1\le k\le n$. Să se găsească elementul care se află pe locul k în şirul ordonat crescător, fără a efectua ordonarea [11].
Indicaţie: Folosim ideea *algoritmului quicksort* care aşează un element pe locul m, poziţia lui corectă în şirul ordonat crescător, astfel încât pe

primele m-1 poziţii se găsesc elemente mai mici sau egale decât a[*m*] iar după poziţia m se găsesc numai elemente mai mari decât a[*m*].

7. Problema punctului fix: Fie *a* un tablou ordonat crescător de *n* numere întregi distincte. Să se determine un indice m, ($1 \leq m \leq n$), cu a[m]=m, dacă este posibil [11].
Indicaţie: Folosim ideea de la căutarea binară.

Concluzii

Tipuri de greşeli ce pot interveni în implementarea paradigmei *divide et impera*

Greşelile care pot apărea la realizarea şi implementarea algoritmilor bazaţi pe paradigma *divide-et-impera* se pot impărţi în două categorii:

- greşeli care ţin de o strategie incorect proiectată;
- greşeli care apar în etapa de programare datorită implementării defectuoase a recursivităţii;

Pornind de la presupunerea că problema căreia i se caută soluţia acceptă rezolvare prin paradigma *divide et impera*, trebuie avute în vedere următoarele situaţii care conduc la greşeli de fond ale algoritmului:

- ***Descompunerea incorectă a problemei:***

exemplu: *sunt generate în general de obţinerea unor subprobleme care nu sunt disjuncte;*

- ***Identificarea greşită a dimensiunii problemei*** *care acceptă rezolvarea imediată (cazul de bază);*
- ***Parametrii*** *care indică dimensiunea subproblemelor nu tind către cazul de bază;*
- *programul fiind recursiv, dacă nu este prezentă* ***condiţia de oprire****, va conduce la întreruperea execuţiei programului şi afişarea mesajului de eroare: "Stack overflow" (depăşirea stivei);*
- ***Faza de combinare a soluţiilor subproblemelor*** *nu este prezentă, iar soluţia problemei iniţiale nu se construieşte simultan cu descompunerea sa şi nici nu se reprezintă soluţia vreunei subprobleme;*
- ***Proiectarea greşită a fazei de combinare a soluţiilor.***