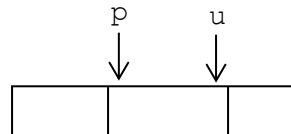


Metoda Divide et Impera

Metoda *Divide et Impera* ("desparte și stăpânește") constă în împărțirea repetată a unei probleme de dimensiuni mari în mai multe subprobleme **de același tip**, urmată de rezolvarea acestora și combinarea rezultatelor obținute pentru a determina rezultatul corespunzător problemei inițiale. Pentru fiecare subproblemă procedăm în același mod, cu excepția cazului în care dimensiunea ei este suficient de mică pentru a fi rezolvată direct. Este evident caracterul recursiv al acestei metode.

- **Schema generală**

```
function DivImp(p,u)
    if u-p<ε
        r ← Rezolva(p,u)
    else m ← Intermed(p,u);
        r1 ← DivImp(p,m);
        r2 ← DivImp(m+1,u);
        r ← Combina(r1,r2)
    return r
end;
```



Example:

- Calculul celui mai mare divizor comun al elementelor unui vector a;

```
function DivImp(p,u)
    if u-p<2
        r ← Cmmdc2(a[p],a[u])
    else m ← ⌊(p+u)/2⌋;
        r1 ← DivImp(p,m);
        r2 ← DivImp(m+1,u);
        r ← Cmmdc2(r1,r2)
    return r
end;
```

unde funcția $Cmmdc2(a,b)$ calculează cel mai mare divizor comun a două numere a și b

- Parcurgerile în preordine, inordine și postordine ale unui arbore binar;
- Sortarea folosind arbori de sortare.

Mai pe scurt, putem spune:

Metoda Divide et Impera constă în:

- construirea dinamică a unui arbore (prin împărțirea în subprobleme) urmată de
- parcurgerea în postordine a arborelui (prin asamblarea rezultatelor parțiale).

• Căutarea binară

Se consideră vectorul $a = (a_1, \dots, a_n)$ ordonat crescător și o valoare x . Se cere să se determine dacă x apare printre componentele vectorului.

Mai exact căutăm perechea (b, i) dată de:

- (true, i) dacă $a_i = x$;
- (false, i) dacă $a_{i-1} < x < a_i$,

unde $a_0 = -\infty$, $a_{n+1} = +\infty$.

Problema enunțată constituie un exemplu pentru cazul în care problema se reduce la o singură subproblemă.

Ținând cont de faptul că a este ordonat crescător, vom compara pe x cu elementul din "mijlocul" vectorului. Dacă avem egalitate, algoritmul se încheie; în caz contrar vom căuta fie în "jumătatea" din stânga, fie în cea din dreapta.

Deoarece problema se reduce la o singură subproblemă, nu mai este necesar să folosim recursivitatea.

Algoritmul este următorul:

```
procedure CautBin
  p ← 1; u ← n
  while p ≤ u
    i ← ⌊(p+u)/2⌋
    {pentru a evita depasiri este de preferat i ← p+⌊(u-p)/2⌋ }
    case ai>x : u ← i-1
      ai=x : write(true,i); stop
      ai<x : p ← i+1
  write(false,p)
end
```

Algoritmul necesită o mică analiză, legată de corectitudinea sa parțială. Mai precis, ne întrebăm: când se ajunge la $p > u$?

- pentru cel puțin 3 elemente : nu se poate ajunge la $p > u$;
- pentru 2 elemente, adică pentru $u = p+1$: se alege $i = p$. Dacă $x < a_i$, atunci $u \leftarrow p-1$. Se observă că se iese din ciclul while și $a_{i-1} < x < a_i = a_p$;
- pentru un element, adică $p = u$: se alege $i = p = u$. Dacă $x < a_i$ atunci $u \leftarrow p-1$, iar dacă $x > a_i$ atunci $p \leftarrow u+1$; în ambele cazuri se părăsește ciclul while și tipărește un rezultat corect.

Complexitate. Presupunem că $n = 2^k$. Atunci avem

$$T(n) = T(n/2) + c = \dots = T(n/2^k) + kc$$

Rezultă $T(n) = O(\log n)$

- **Sortare prin interclasare**

Fie $a = (a_1, \dots, a_n)$ vectorul care trebuie ordonat crescător.

Ideea este următoarea: împărțim vectorul în doi subvectori, ordonăm crescător fiecare subvector și asamblăm rezultatele prin *interclasare*. Se observă că este aplicată tocmai metoda Divide et Impera.

Programul principal urmează întocmai strategia Divide et Impera, deci se face apelul $\text{SortInter}(1, n)$, unde procedura recursivă SortInter are forma:

```
procedure SortInter(p, u)
  if p = u
  else m ← ⌊(p+u)/2⌋;
        SortInter(p, m);
        SortInter(m+1, u);
        Inter(p, m, u)
end
```

Presupunând că (a_p, \dots, a_m) și (a_{m+1}, \dots, a_u) sunt ordonați crescător, procedura de interclasare Inter va ordona crescător întreaga secvență (a_p, \dots, a_u) (**pentru implementare, vezi aplicația următoare**). Timpul de calcul al procedurii este de ordinul $O(u-p)$, adică liniar în lungimea secvenței analizate.

Complexitate. Calculăm în continuare **timpul de executare** al sortării prin interclasare $T(n)$, unde $T(n)$ se poate scrie:

- t_0 (constant), pentru $n=1$;
- $2T(n/2) + an$, pentru $n > 1$, unde a este o constantă: problema de dimensiune n s-a descompus în două subprobleme de dimensiune $n/2$, iar combinarea rezultatelor s-a făcut în timp liniar (prin interclasare).

Presupunem că $n=2^k$. Atunci:

$$\begin{aligned} T(n) &= T(2^k) = 2T(2^{k-1}) + a2^k = \\ &= 2[2T(2^{k-2}) + a2^{k-1}] + a2^k = 2^2T(2^{k-2}) + 2a2^k = \\ &= 2^2[T(2^{k-3}) + a2^{k-2}] + 2a2^k = 2^3T(2^{k-3}) + 3a2^k = \\ &\quad \cdot \quad \cdot \quad \cdot \\ &= 2^i T(2^{k-i}) + i a 2^k = \\ &\quad \cdot \quad \cdot \quad \cdot \\ &= 2^k T(1) + k a 2^k = n t_0 + a \cdot n \cdot \log_2 n. \end{aligned}$$

Rezultă că $T(n) = O(n \log n)$.

Mențiune. Se poate demonstra că acest timp este optim, folosind arbori de decizie.

Numărul de inversiuni

Problemă (aplicație). Se consideră un vector cu n elemente. Să de determine numărul de inversiuni din acest vector (numărul de perechi (i, j) cu proprietatea că $i < j$ și $a_i > a_j$)

Aplicații:

- ▶ Măsură a diferenței între două liste ordonate
- ▶ “Gradul de ordonare” al unui vector
- ▶ Probleme de analiză a clasificărilor (ranking)
 - Asemănarea între preferințele a doi utilizatori – sugestii de utilizatori cu preferințe similare
 - Asemănări dintre rezultatele întoarse de motoare diferite de căutare pentru aceeași cerere
 - collaborative filtering

Soluție. Un algoritm cu complexitatea $O(n^2)$ este evident. Putem însă rezolva problema printr-un algoritm cu complexitatea $O(n \lg n)$, numărând inversiunile în timpul sortării prin interclasare. Mai exact:

numărul de inversiuni dintr-un vector = numărul de inversiuni din subvectorul stâng (prima jumătate) + numărul de inversiuni din subvectorul drept + numărul de inversiuni (i, j) cu $i \leq m$ indice în subvectorul stâng și $j > m$ indice în subvectorul drept (am notat cu m indicele elementului din mijlocul vectorului).

Pentru a număra inversiunile (i, j) cu $i \leq m$ și $j > m$ observăm că la interclasare au loc următoarele:

1. când un element de pe poziția j cu $j > m$ este adăugat în vectorul rezultat, el este mai mic (doar) decât toate elementele din subvectorul stâng neadăugate încă în vectorul rezultat; astfel, **$a[j]$ determină $m - i + 1$ inversiuni cu elemente din subvectorul stâng**
2. când un element de pe poziția i cu $i \leq m$ este adăugat în vectorul rezultat, el este mai mare (doar) decât toate elementele din subvectorul drept adăugate deja la vectorul rezultat

Avem astfel două modalități de a număra inversiunile la interclasare.

Exemplu - vezi slideuri curs DI

• Metoda Quicksort (de știut, nediscutată la curs)

Prezentăm încă o metodă de sortare a unui vector $a = (a_1, \dots, a_n)$. Va fi aplicată tot metoda Divide et Impera. Și de această dată fiecare problemă va fi descompusă în două subprobleme mai mici de aceeași natură, dar nu va mai fi necesară combinarea (asamblarea) rezultatelor rezolvării subproblemelor.

Fie (a_p, \dots, a_u) secvența curentă care trebuie sortată. Vom *poziționa* pe a_p în secvența (a_p, \dots, a_u) , adică printr-o permutare a elementelor secvenței:

- $x=a_p$ va trece pe o poziție k ;
- toate elementele aflate la stânga poziției k vor fi mai mici decât x ;
- toate elementele aflate la dreapta poziției k vor fi mai mari decât x .

În acest mod a_p va apărea pe poziția sa finală, rămânând apoi să ordonăm crescător elementele aflate la stânga sa, precum și pe cele aflate la dreapta sa.

Fie poz funcția cu parametrii p, u care întoarce indicele k pe care va fi poziționat a_p în cadrul secvenței (a_p, \dots, a_u) .

Atunci sortarea se realizează prin apelul $\text{QuickSort}(1, n)$, unde procedura QuickSort are forma:

```
procedure QuickSort(p, u)
  if p = u
  else m ← poz(p, u); QuickSort(p, m-1); QuickSort(m+1, n)
end
```

Funcția poz lucrează astfel:

```
function poz(p, u)
  i ← p; j ← u; di ← 0; dj ← -1
  while i < j
    if  $a_i > a_j$ 
       $a_i \leftrightarrow a_j$ ;  $(di, dj) \leftarrow (-dj, -di)$       (*)
       $i \leftarrow i + di$ ;  $j \leftarrow j + dj$ 
  return i
end
```

unde prin \leftrightarrow am notat interschimbarea valorilor.

Să urmărim cum decurg calculele pentru secvența:

$(a_4, \dots, a_{11}) = (6, 3, 2, 5, 8, 1, 9, 7)$

- se compară 6 cu a_{11}, a_{10}, \dots până când găsim un element mai mic. Acesta este $a_9 = 1$. Se interschimbă 6 cu 1. Acum secvența este $(1, 3, 2, 5, 8, 6, 9, 7)$ și vom lucra în continuare pe subsecvența $(3, 2, 5, 8, 6)$, schimbând direcția de comparare conform (*);
- 6 va fi comparat succesiv cu 3, 2, ... până când găsim un element mai mare. Acesta este $a_8 = 8$. Se interschimbă 6 cu 8.

Se obține astfel $(1, 3, 2, 5, 6, 8, 9, 7)$, în care la stânga lui 6 apar valori mai mici, iar la dreapta lui 6 apar valori mai mari, deci l-am poziționat pe 6 pe poziția 8, valoare întoarsă de funcția poz .

Complexitate.

Cazul cel mai defavorabil pentru metoda Quicksort este cel în care vectorul este deja ordonat crescător: se compară a_1 cu a_2, \dots, a_n rezultând că el se află pe poziția finală, apoi se compară a_2 cu a_3, \dots, a_n rezultând că el se află pe poziția finală etc.

Trecem la calculul **timpului mediu** de executare al algoritmului Quicksort. Vom număra câte comparări se efectuează (componentele vectorului nu sunt neapărat numere, ci elemente dintr-o mulțime ordonată oarecare). Timpul mediu este dat de formulele:

$$\begin{cases} T(n) = n - 1 + \frac{1}{n} \sum_{k=1}^n [T(k-1) + T(n-k)] \\ T(1) = T(0) = 0 \end{cases}$$

deoarece:

- în cazul cel mai defavorabil a_1 se compară cu celelalte $n-1$ elemente;
- a_1 poate fi poziționat pe oricare dintre pozițiile $k=1, 2, \dots, n$; considerăm aceste cazuri echiprobabile;
- $T(k-1)$ este timpul (numărul de comparații) necesar ordonării elementelor aflate la stânga poziției k , iar $T(n-k)$ este timpul necesar ordonării elementelor aflate la dreapta poziției k .

$$nT(n) = n(n-1) + 2[T(0) + T(1) + \dots + T(n-1)]$$

$$(n-1)T(n-1) = (n-1)(n-2) + 2[T(0) + \dots + T(n-2)]$$

Scăzând cele două relații obținem:

$$nT(n) - (n-1)T(n-1) = 2(n-1) + 2T(n-1), \text{ deci:}$$

$$nT(n) = (n+1)T(n-1) + 2(n-1).$$

Împărțim cu $n(n+1)$:

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + 2\left(\frac{2}{n+1} - \frac{1}{n}\right)$$

$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + 2\left(\frac{2}{n} - \frac{1}{n-1}\right)$$

.....

$$\frac{T(2)}{3} = \frac{T(1)}{2} + 2\left(\frac{2}{3} - \frac{1}{2}\right)$$

Prin adunarea relațiilor de mai sus obținem:

$$\frac{T(n)}{n+1} = 2\left(\frac{1}{n+1} + \frac{1}{n} + \dots + \frac{1}{3}\right) + \frac{2}{n+1} - 1$$

Cum suma ultimilor doi termeni este negativă, rezultă:

$$\frac{T(n)}{n+1} \leq 2 \int_2^{n+1} \frac{1}{x} dx = 2 \ln x \Big|_2^{n+1} \leq 2 \cdot \ln(n+1)$$

(am folosit o inegalitate bazată pe sumele Riemann inferioare pentru funcția $f(x) = 1/x$).

Deci $T(n) = O(n \cdot \log n)$.

Variantă (cu rezultate bune în practică): Sortarea prin interclasare **cu pivot aleator**

procedure QuickSortR(p, u)

 if p = u

 else k ← pozRand(p, u); QuickSortR(p, k-1); QuickSortR(k+1, n)

end

Funcția pozRand lucrează astfel:

function pozRand(p, u)

 r ← random(p, u); a[r] ↔ a[p]; return poz(p, u)

end

Statistici de ordine

Problemă (aplicație). Dat un vector a de n numere și un indice k , $1 \leq k \leq n$, să se determine al k -lea cel mai mic element din vector.

Soluție. Cadru general: **Statistici de ordine**

A i -a statistică de ordine a unei mulțimi de n elemente este al i -lea cel mai mic element.

- **Minimul** = prima statistică de ordine
- **Maximul** = a n -a statistică de ordine
- **Mediana** = punctul de la jumătatea drumului unei mulțimi = o valoare v cu proprietatea că numărul de elemente din mulțime mai mici decât v este egal cu numărul de elemente din mulțime mai mari decât v . Dacă n este impar, atunci mediana este a $\lceil n/2 \rceil$ -a statistică de ordine, altfel, prin convenție mediana este media aritmetică dintre a $\lfloor n/2 \rfloor$ -a statistică și a $(\lfloor n/2 \rfloor + 1)$ -a statistică de ordine

Un algoritm optim de determinare a minimului sau maximului necesită $n - 1$ comparații, iar un algoritm de determinare simultană a minimului și maximului, $3\lceil n/2 \rceil - 2$ comparații (v. seminar).

Pentru a rezolva problema determinării celui de al k -ale minim (cele de a k -a statistică de ordine) folosim ideea de partiționare de la QuickSort

```
function kmin(p,u)
    m ← pozRand(p,u);
    if (m = k)      return am
    if (m < k)      return kmin(m+1,u)
    else           return kmin(p,m-1)
end
```

Exemplu - vezi slideuri curs DI

Observații. 1. Timpul mediu de executare al acestui algoritm este $O(n)$. Mai exact se poate demonstra prin inducție că pentru o constantă c suficient de mare avem $T(n) \leq cn$ (consultați slide-uri + Cormen pentru mai multe detalii)

2. Există algoritm cu complexitate $O(n)$ pentru determinarea celei de a i -a statistică de ordine.