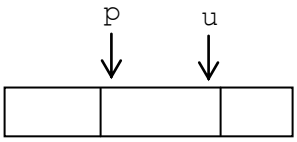
Metoda Divide et Impera

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

• Schema generală

Descriem schema generală pentru cazul în care aplicăm metoda pentru o prelucrare oarecare asupra elementelor unui vector. Funcţia `DivImp`, care întoarce rezultatul prelucrării asupra unei subsecvenţe $a_p, \ldots a_u$, va fi apelată prin `DivImp(1,n)`.

```
function DivImp(p,u)
   if u-p<ε
        r ← Prel(p,u)
   else m ← Interm(p,u);
        r1 ← DivImp(p,m);
        r2 ← DivImp(m+1,u);
        r ← Combin(r1,r2)
   return r
end;
```



unde:

- funcţia `Interm` întoarce un indice în intervalul `p..u` (de obicei `m=⌊(p+u)/2⌋`).
- funcţia `Prel` este capabilă să întoarcă rezultatul subsecvenţei `p..u`, dacă aceasta este suficient de mică;
- funcţia `Combin` întoarce rezultatul asamblării rezultatelor parţiale `r1` şi `r2`.

Exemple:

- 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 funcţia `Cmmdc2(a,b)` calculează cel mai mare divizor comun a două numere a şi b

- Parcurgerile în preordine, inordine şi 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 împărţirea în subprobleme) urmată de*
> *- parcurgerea în postordine a arborelui (prin asamblarea rezultatelor parţiale).*

• Problema turnurilor din Hanoi

Se consideră 3 tije. Iniţial pe tija 1 se află `n` discuri cu diametrele decrescătoare privind de la bază către vârf, iar pe tijele 2 şi 3 nu se află nici un disc. Se cere să se mute aceste discuri pe tija 2, ajutându-ne şi de tija 3, respectând condiţia ca în permanenţă pe orice tijă sub orice disc să se afle baza tijei sau un disc de diametru mai mare.

O *mutare* este notată prin `(i,j)` şi semnifică deplasarea discului din vârful tijei `i` deasupra discurilor aflate pe tija `j`. Se presupune că mutarea este corectă (vezi condiţia de mai sus).

Fie *H*`(m;i,j)` şirul de mutări prin care cele `m` discuri din vârful tijei `i` sunt mutate peste cele de pe tija `j`, folosind şi a treia tijă, al cărei număr este `k=6-i-j`. Problema constă în a determina *H*`(n;1,2)`.

Se observă că este satisfăcută relaţia:

H`(m;i,j) = `*H*`(m-1;i,k) (i,j) `*H*`(m-1;k,j)` (*)

respectându-se condiţia din enunţ. Deci problema pentru `m` discuri a fost redusă la două probleme pentru `m-1` discuri, al căror rezultat este asamblat conform (*).

Corespunzător, vom executa apelul `Hanoi(n,1,2)`, unde procedura `Hanoi` are forma:

```
procedure Hanoi(n,i,j)
   if n = 1
        write(i,j)
   else k←6-i-j;
        Hanoi(n-1,i,k); Hanoi(1,i,j); Hanoi(n-1,k,j)
end
```

Observaţie. Avem

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

Rezultă că numărul de mutări este 2^n-1.

- ## Căutarea binară

Se consideră vectorul $a=(a_1,...,a_n)$ ordonat crescător şi 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 enunţată constituie un exemplu pentru cazul în care problema se reduce la o singură subproblemă.

Ţinând 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 parţială. Mai precis, ne întrebăm: când se ajunge la $p>u$?

- pentru cel puţin 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` şi $a_{i-1}<x<a_i=a_p$;
- 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 părăseşte ciclul `while` şi tipăreşte un rezultat corect.

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

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

Rezultă $T(n) = O(\log n)$

Temă

1. http://infoarena.ro/problema/cautbin Se consideră vectorul $a=(a_1,...,a_n)$ ordonat crescător şi o valoare x. Se cere să afişeze prima poziţie şi ultima poziţie pe care x apare printre componentele vectorului; dacă x nu se află în vector, se va afişa -1.

2. (aplicaţie) Să se găsească (dacă există) cel mai mic numar natural *N* astfel încât *N!* să se termine în exact *p* cifre de *0*, $p \leq 10^8$ (http://infoarena.ro/problema/fact)

- ## Sortare prin interclasare

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

Ideea este următoarea: împărţim vectorul în doi subvectori, ordonăm crescător fiecare subvector şi 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 `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 $(a_p,...,a_m)$ şi $(a_{m+1},...,a_u)$ sunt ordonaţi crescător, procedura de interclasare `Inter` va ordona crescător întreaga secvenţă $(a_p,...,a_u)$ (**pentru implementare, vezi aplicația următoare**). Timpul de calcul al procedurii este de ordinul $O(u-p)$, adică liniar în lungimea secvenţei 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:

$$T(n) = T(2^k) = 2T(2^{k-1}) + a2^k =$$
$$= 2[2T(2^{k-2}) + a2^{k-1}] + a2^k = 2^2T(2^{k-2}) + 2a2^k =$$
$$= 2^2[T(2^{k-3}) + a2^{k-2}] + 2a2^k = 2^3T(2^{k-3}) + 3a2^k =$$
$$\cdots$$
$$= 2^iT(2^{k-i}) + ia2^k =$$
$$\cdots$$
$$= 2^kT(1) + ka2^k = nt_0 + a \cdot n \cdot \log_2 n.$$

Rezultă că $T(n)=0(n \cdot \log n)$.

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

Problemă (aplicaţie). 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 şi $a_i > a_j$)

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 ≤ 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 ≤ 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.
2. când un element de pe poziția i cu i ≤ 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.

Implementarea acestei aplicații se găsește în fișierul Inversiuni_1.java (varianta 1) și Inversiuni_2.java (varianta 2)

• Metoda Quicksort

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

Fie $(a_p,...,a_u)$ secvenţa curentă care trebuie sortată. Vom *poziţiona* pe a_p în secvenţa $(a_p,...,a_u)$, adică printr-o permutare a elementelor secvenţei:

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

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

Fie `poz` funcţia cu parametrii `p,u` care întoarce indicele `k` pe care va fi poziţionat a_p în cadrul secvenţei $(a_p,...,a_u)$.

Atunci sortarea se realizează prin apelul `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
```

Funcţia `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↔a_j; (di,dj)←(-dj,-di)     (*)
      i←i+di; j←j+dj
   return i
end
```

unde prin $\leftrightarrow$ am notat interschimbarea valorilor.

Să urmărim cum decurg calculele pentru secvenţa:
$(\mathbf{a_4},...,\mathbf{a_{11}})=(6,3,2,5,8,1,9,7)$

- se compară 6 cu $a_{11}, a_{10}, \ldots$ până când găsim un element mai mic. Acesta este $a_9 = 1$. Se interschimbă 6 cu 1. Acum secvenţa este (1,3,2,5,8,6,9,7) şi vom lucra în continuare pe subsecvenţa (3,2,5,8,6), schimbând direcţia 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 obţine 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 poziţionat pe 6 pe poziţia 8, valoare întoarsă de funcţia poz.

Complexitate.

Cazul cel mai defavorabil pentru metoda Quicksort este cel în care vectorul este deja ordonat crescător: se compară a_1 cu $a_2, \ldots, a_n$ rezultând că el se află pe poziţia finală, apoi se compară a_2 cu $a_3, \ldots, a_n$ rezultând că el se află pe poziţia 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 mulţime 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 poziţionat pe oricare dintre poziţiile k=1,2,...,n; considerăm aceste cazuri echiprobabile;
- T(k-1) este timpul (numărul de comparări) necesar ordonării elementelor aflate la stânga poziţiei k, iar T(n-k) este timpul necesar ordonării elementelor aflate la dreapta poziţiei k.

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

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

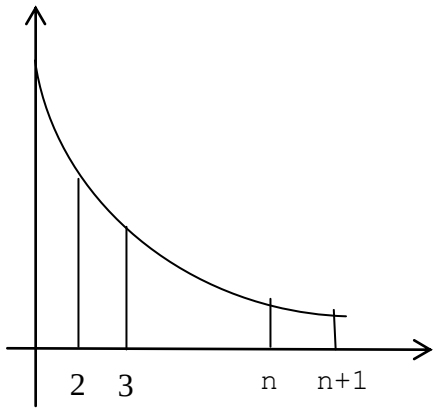
Scăzând cele două relaţii obţinem:

$nT(n)-(n-1)T(n-1) = 2(n-1)+ 2T(n-1)$, deci:

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

Împărţim cu $n(n+1)$:

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



Prin adunarea relaţiilor de mai sus obţinem:

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

Cum suma ultimilor doi termeni este negativă, rezultă:

$$\frac{T(n)}{n+1}\le 2\int_{2}^{n+1}\frac{1}{x}dx=2\ln x\Big|_{2}^{n+1}\le 2\cdot\ln(n+1)$$

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

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
```

Funcţia `pozRand` lucrează astfel:

```
function pozRand(p,u)
   r ← random(p,u)
   a[r] ↔ a[p]
   return poz(p,u)
end
```

O **implementare** a metodei Quicksort cu pivot aleator se găsește în fișierul QSortAleat.java

Problemă (aplicaţie). Dat un vector a de n numere şi un indice k, 1 ≤ k ≤ 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 (celei de a k-a statistică de ordine) folosim idea de partiționare de la QuickSort

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

Observaţie. 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 Cormen pentru mai multe detalii)

O **implementare** a acestui algoritm se găsește în fișierul AlKMinim.java

Problemă - Problema elementului majoritar Se dă un şir de *n* numere naturale. Se cere determinarea unui element care apare de cel puţin *[n/2]+1* ori în şir dacă există un astfel de element în şir. Acest element se numeşte element majoritar (http://infoarena.ro/problema-majoritatii-votului)

Soluție. Problema se poate rezolva prin diferite metode (determinarea, pentru fiecare element, a numărului apariţiilor acestora, eventual după o sortare prealabilă a şirului sau folosind tabele de dispersie), existând şi algoritmi de complexitate $O(n)$ pentru rezolvarea ei. O soluţie eficientă are la bază următoarea observaţie: un element majoritar se află în şirul sortat pe poziţia $n/2$. Astfel, este suficient să determinăm al $n/2$-lea minim folosind algoritmul anterior (complexitate medie $O(n)$)

- ## Mediana a doi vectori sortați[1]

Se dau doi vectori a și b de lungime n, cu elementele ordonate crescător. Să se determine mediana vectorului obținut prin interclasarea celor doi vectori.

Observație. Elementele vectorilor se consideră **numeroatate de la 0.**

Metoda 1 – Efectuăm procedura de interclasare până sunt completate `n+1` elemente în vectorul rezultat. **Complexitate**: `O(n)`

Metoda 2 – Comparăm medianele celor doi vectori. În funcție de rezultat, problema se reduce la aceeași problemă pentru subvectori ai vectorilor inițiali. Astfel, fie `m1` mediana vectorului `a` și `m2` mediana vectorului `b`

Observație. Mediana unui vector sortat cu n elemente se poate calcula în timp constant, fiind chiar elementul de pe poziția $\lfloor n/2 \rfloor$ (!! pozițiile sunt numerotate de la 0), dacă n este impar, sau media aritmetică a elemetelor de pe pozițiile $\lfloor n/2 \rfloor$-1 și $\lfloor n/2 \rfloor$, dacă n este par

- Dacă `m1 = m2` atunci această valoare este mediana
- Dacă `m1 > m2` atunci mediana se află în unul din subvectorii

  $a[0..\left\lfloor \frac{n}{2} \right\rfloor], b[\left\lfloor \frac{n-1}{2} \right\rfloor..n-1]$
- Dacă `m1 < m2` atunci mediana se află în unul din subvectorii

  $a[\left\lfloor \frac{n-1}{2} \right\rfloor..n-1], b[0..\left\lfloor \frac{n}{2} \right\rfloor]$

Dacă vectorii au cel mult două elemente, atunci problema se poate rezolva direct.

O implementare a acestei probleme în Java se găsește în fișierul Mediana.java. Metoda principală (unde se folosește Divide et Impera) este `calculMediana`, în care se folosește ideea amintită pentru subsecvențele `a[pa..ua],b[pb..ub]`.

```
double calculMediana(int pa, int ua,int pb, int ub){
   int n = ua-pa+1;
   if(n <= 2)
      return (Math.max(a[pa],b[pb])+Math.min(a[ua],b[ub]))/2.0;
   double m1 = mediana(a,pa,ua);//calculul direct al medianei
   double m2 = mediana(b,pb,ub);//calculul direct al medianei
   if(m1 == m2)  return m1;
   if (m1 > m2) return calculMediana(pa, pa+n/2, pb+(n-1)/2, ub);
   else return calculMediana(pa+(n-1)/2, ua, pb, pb+n/2);
 }
```

Complexitate: O(log n)

[1] ***Median of two sorted arrays***

Metoda 3 (suplimentar) – Pornim de la următoarea observație: putem testa în timp constant dacă elementul a[i] este mediana dorită. Mai exact, valoarea a[i] este mai mare decât i – 1 elemente din a. Pentru a fi mediană, trebuie să fie mai mare decât j = n – i – 1 elemente din b. Este suficient deci să testăm dacă b[j] ≤ a[i] ≤ b[j+1].
Dacă valoarea a[i] nu este mediana, în funcție de rezultatul comparării ei cu b[j] și b[j+1], putem decide dacă a[i] este mai mic sau mai mare decât mediana și căutarea medianei va continua în subvectorul corespunzător din a. Dacă mediana nu este găsită cu ajutorul elementelor din a, reluăm procedeul pentru elementele din b.

Incheiem cu menţiunea că metoda Divide et Impera are o largă aplicativitate în calculul paralel.