Metoda Divide et Impera

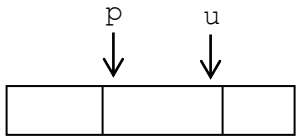
Metoda *Divide et Impera* ("desparte şi stăpâneşte") constă în împărţirea repetată a unei probleme de dimensiuni mari în mai multe subprobleme *de acelaşi tip*, urmată de rezolvarea acestora şi combinarea rezultatelor.

Este evident caracterul recursiv al acestei metode.

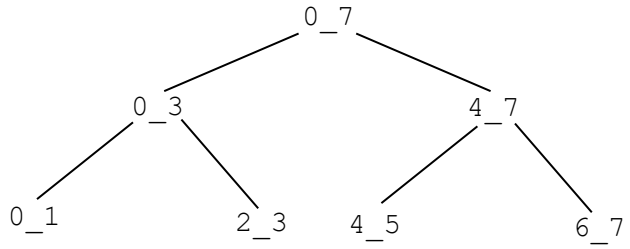
• Schema generală

Descriem schema generală pentru cazul în care aplicăm metoda pentru o prelucrare oarecare asupra elementelor unui vector. Funcţia `DivImp` întoarce rezultatul prelucrării asupra unei subsecvenţe $a_p, \ldots a_u$ şi va fi apelată prin `DivImp(0,n-1)`.

```
function DivImp(p,u)
   if u-p<ε
   then 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;
```



Exemplu. Urmărim ce se întâmplă pentru 8 elemente:



Putem spune că:

> *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)*.

Exemple:

- Calculul maximului elementelor unui vector;
- Parcurgerile în preordine, inordine şi postordine ale unui arbore binar;
- Sortarea folosind arbori de sortare.

• Sortare prin interclasare

Fie $a=(a_0, \ldots, a_{n-1})$ vectorul care trebuie ordonat crescător.

Este aplicată întocmai, ca mai sus, metoda Divide et Impera.

```
procedure Inter(p,m,u)
   k1←p; k2←m+1; k3←p;
   while k1≤m & k2≤u
      if ak1<ak2 then bk3←ak1; k1←k1+1;
                 else bk3←ak2 ;k2←k2+1
      k3←k3+1
   if k1>m       { au fost epuizate elementele primei subsecvenţe }
   then for i=k2,u
           bk3←ai; k3←k3+1
   else          { au fost epuizate elementelei primei subsecvenţe }
      for i=k1,m
           bk3←ai; k3←k3+1
   for i=p,u
      ai←bi
end
```

	p				m	m+1				u
a =	1	3	7	8	10	2	4	5	11	12

b =	1	2	3	4	5	7	8	10	11	12

Timpul de calcul este de ordinul $O(u-p)$, adică liniar în lungimea secvenţei.

În programul principal apelul `SortInter(0,n-1)`, unde:

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

Calculăm în continuare timpul de executare $T(n)$, unde $T(n)$ se poate scrie:

- t_0 (constant), pentru $n=1$;
- $2T(n/2)+an$, pentru $n>1$, unde a este o constantă:.

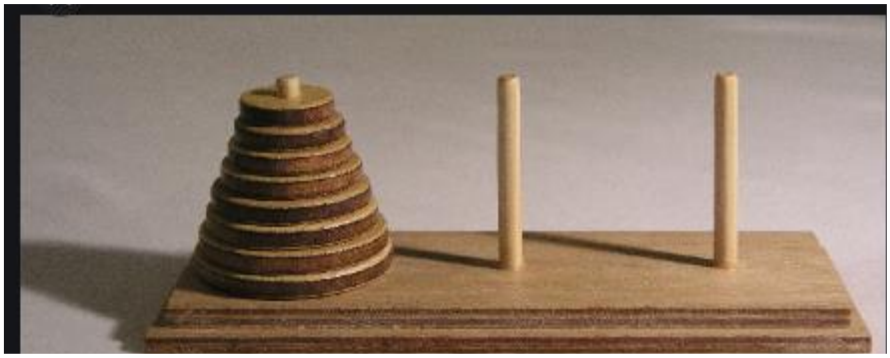
Presupunem că $n=2^k$. Atunci:

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

Rezultă că $T(n)=0(n\cdot \log n)$. Se poate demonstra că acest timp este optim.

- **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`. Numărul de mutări va fi este 2^n-1.

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 :

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

H(2;1,2) = H(1;1,3)  (1,2)  H(1;3,2)
         = (1,3)  (1,2)  (3,2)
H(3;1,2) = H(2;1,3)  (1,2)  H(2;3,2)
         = (1,2) (1,3) (2,3) (1,2) (3,1) (3,2) (1,2)
```

- **Căutarea binară**

Se consideră vectorul $a=(a_1,\ldots,a_n)$ ordonat crescător şi o valoare `x`. Se cere să se determine dacă `x` apare printre componentele vectorului. *Este un exemplu pentru cazul în care problema se reduce la o singură subproblemă.*

Vom adăuga $a_0=-\infty$, $a_{n+1}=+\infty$. Căutăm perechea `(b,i)` dată de:

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

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⌋
      case ai>x : u ← i-1
           ai=x : write(true,i); stop
           ai<x : p ← i+1
   write(false,p)
end
```

0	1	2	3	4	5	6	7	p	u	i
-∞	**2**	**4**	**7**	**9**	**11**	**15**	+∞	1	6	3
-∞	2	4	7	**9**	**11**	**15**	+∞	4	6	5
				9						

x=9
(true,9)

0	1	2	3	4	5	6	7	p	u	i
-∞	**2**	**4**	**7**	**9**	**11**	**15**	+∞	1	6	3
-∞	**2**	**4**	7	9	11	15	+∞	1	2	1
-∞	2	**4**	7	9	11	15	+∞	2	2	2
								2	1	

x=3
(false,2)

0	1	2	3	4	5	6	7	p	u	i
-∞	**2**	**4**	**7**	**9**	**11**	**15**	+∞	1	6	3
-∞	2	4	7	**9**	**11**	**15**	+∞	4	6	5
-∞	2	4	7	9	11	**15**	+∞	6	6	6
								7	6	

x=17
(false,7)

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←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←p-1`, iar dacă $x>a_i$ atunci `p←u+1`; în ambele cazuri se părăseşte ciclul `while` şi tipăreşte un rezultat corect.

Timpul necesitat de acest algoritm este evident $T(n)=O(\log n)$.

- **Metoda Quicksort**

Prezentăm încă o metodă de sortare a unui vector $a=(a_0,...,a_{n-1})$. 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.

Pentru (**3**,8,1,4,7,2) o poziţionare corectă este de exemplu: (2,1,**3**,4,8,7).

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(0,n-1)`, unde procedura `QuickSort` are forma:

```
procedure QuickSort(p,u)
   if p>=u
   then
   else k ← poz(p,u); QuickSort(p,k-1); QuickSort(k+1,u)
end
```

Funcţia `poz` lucrează astfel:

```
function poz(p,u)
   i←p; j←u; ii←0; jj←-1
   while i<j
      if ai<aj
      then
      else ai↔aj; (ii,jj)←(-jj,-ii)     (*)
      i←i+ii; j←j+jj
   return i
end
```

Să urmărim cum decurg calculele pentru secvenţa:
$(a_4,...,a_{11})=(4,3,2,5,0,1,9,7)$, unde iniţial `(ii,jj)=(0,-1) >> (1,0)`

4	3	2	5	0	1	9	7
1	3	2	5	0	4	9	7
1	3	2	4	0	5	9	7
1	3	2	0	4	5	9	7

Observaţie. 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. În acest caz timpul este de ordinul $O(n^2)$.

Trecem la calculul timpul *mediu* de executare al algoritmului Quicksort, dat de:

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

deoarece:

- în cazul cel mai defavorabil a_1 se compară cu celelalte n-1 elemente;
- a_1 poate fi poziţionat pe oricare dintre poziţiile $k=1,2,\ldots,n$; 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.

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

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

Scăzând cele două relaţii obţinem:

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

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

Împart cu $n(n+1)$:

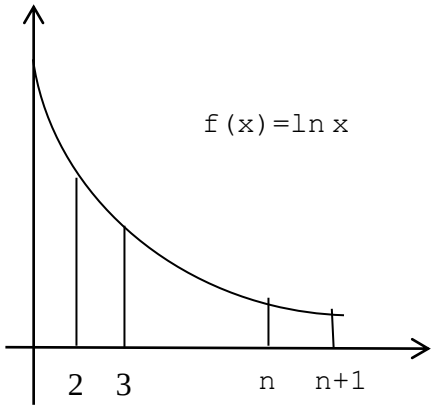
$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

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

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

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

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



Prin adunarea relaţiilor de mai sus obţinem:

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

Cum suma ultimilor doi termeni este negativă, rezultă:

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

(am folosit o inegalitate bazată pe sumele Riemann pentru funcţia $f(x)=\ln x$).

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

Arbori de decizie

Arborii de decizie sunt arbori binari stricţi ce reprezintă o vizualizare a algoritmilor de sortare bazaţi pe comparări.

Pentru fiecare nod intern:

- eticheta sa este $i?j$, semnificând compararea lui a_i cu a_j;
- descendentul său stâng corespunde situaţiei $a_i \le a_j$;
- descendentul său drept corespunde situaţiei $a_i > a_j$.

Fiecare frunză corespunde unei permutări σ a lui $\{1,2,...,n\}$, cu semnificaţia:
$a_{\sigma(1)} \le a_{\sigma(2)} \le \ldots \le a_{\sigma(n)}$,
relaţii deduse din semnificaţia muchiilor ce leagă rădăcina de frunză.

Deducem că numărul maxim de comparări necesare pentru sortare este egal cu înălţimea arborelui de decizie asociat algoritmului.

Teoremă. Orice algoritm de sortare bazat pe comparări are timpul de executare de un ordin cel puţin egal cu $O(n \cdot \log n)$.

Este suficient să demonstrăm că orice arbore de decizie are înălţimea de ordin cel puţin egal cu $O(n \cdot \log n)$.

Considerăm un arbore de decizie oarecare şi fie h înălţimea sa.

Cum un arbore binar strict de înălţime h are cel mult 2^h frunze şi cum numărul frunzelor este $n!$, rezultă că $n! \le 2^h$.

Folosind inegalitatea $n! > (n/e)^n$, obţinem:
$2^h > (n/e)^n$
adică $h > n \cdot \log_2(n/e)$
de unde rezultă $h = O(n \cdot \log n)$.

Alte probleme rezolvate prin metoda Divide et Impera

- **Numărarea inversiunilor**

Fie dat tabloul de numere a de lungime n.

Prin *inversiune* înţelem o pereche (i,j) cu $i<j$ şi $a_i > a_j$. Ne propunem să determinăm numărul inversiunilor din tablou.

Vom folosi metoda Divide et Impera.

Presupunem că am numărat inversiunile din $(a_0,\ldots,a_m)$ şi cele din $(a_{m+1},\ldots,a_{n-1})$, unde m este indicele de la mijloc; fie ele $n1$ şi $n3$.

Rămâne să comparăm perechiile (i,j) cu $i \le m < j$ pentru a calcula costul fazei de combinare; fie ele în număr de $n2$.

Mai observăm că în faza preliminară $(a_0,\ldots,a_m)$ respectiv $(a_{m+1},\ldots,a_{n-1})$ pot să fi suferit orice permutare. Alegem o permutare convenabilă, care se dovedeşte a fi sortarea, prin care ajungem la $a_0 \le \ldots \le a_m$ şi $a_{m+1} \le \ldots \le a_{n-1}$.

Sortarea este convenabilă deoarece prin interclasare este suficient să pornim cu $n2=0$ şi *ori de câte ori un element din stânga este trecut în* b, *să incrementăm pe* $n2$ *cu numărul elementelor din dreapta care au trecut în* b.

Rezultatul este `n1+n2+n3`.

Avantajul constă în faptul că faza de combinare necesită un timp liniar, ceea ce face ca timpul total să fie de ordinul $O(n \cdot \log n)$.

- **Determinarea celor mai apropiate puncte**

Fiind date `n` puncte $P_i(x_i, y_i)$, $i=0,\ldots,n-1$ din plan, ce cere să se determine o pereche de puncte aflate la distanţă (euclidiană) minimă. Fie ***P*** mulţimea acestor puncte.

Pentru simplificare, vom presupune că nu există două puncte cu aceeaşi abscisă sau aceeaşi ordonată.

Într-o primă etapă vom determina în timp $O(n \cdot \log n)$:

- $\boldsymbol{P_x}$ = lista punctelor în ordinea crescătoare a absciselor;
- $\boldsymbol{P_y}$ = lista punctelor în ordinea crescătoare a ordonatelor.

Fie ***L*** linia verticală care trece prin mediana lui ***P*** , împărţind ***P*** în ***S*** şi ***D*** ("jumătatea" stângă, respectiv dreaptă) a lui ***P***. Deci $\boldsymbol{P = S \cup D}$.

Conform strategiei Divide et Impera, calculăm distanţa minimă dintre două puncte din ***S***, apoi distanţa minimă dintre două puncte din ***D***; fie δ cea mai mică dintre ele.

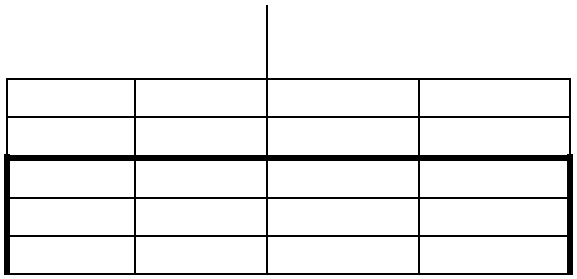
Pentru faza de combinare, considerăm banda punctelor din plan aflate la distanţă cel mult δ de ***L***, deoarece ne interesează perechi de puncte cu unul în ***S*** şi celălat în ***D***. Fie ***B*** mulţimea punctelor din ***P*** aflate în această bandă şi $\boldsymbol{B_y}$ lista punctelor din ***B*** cu ordonatele în ordine crescătoare (evident, abscisele nu respectă neapărat această ordine). $\boldsymbol{B_y}$ poate fi determinată în timp liniar prin eliminarea din $\boldsymbol{P_y}$ a punctelor care nu aparţin benzii.

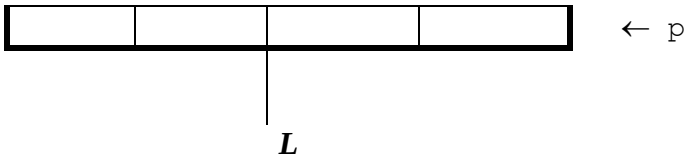
Împărţim banda în pătrate de latură $\delta/2$.

Observăm că în niciunul dintre aceste pătrate nu există două puncte din ***B***, deoarece s-ar afla ambele în ***S*** sau ambele în ***D*** şi ar fi la distanţă mai mică decât δ, ceea ce contrazice alegerea lui δ.

Parcurgem (în timp liniar) lista $\boldsymbol{B_y}$ în ordinea crescătoare a ordonatelor. Pentru primul punct (fie el `p`), doar unul dintre cel mult următoarele 15 puncte din listă poate candida la a fi la distanţă mai mică decât δ (cele din dreptunghiul haşurat din figură); valoarea minimă (iniţial δ) este eventual actualizată. Se continuă la fel pentru celelate puncte din lista $\boldsymbol{B_y}$.

Rezultă că timpul necesar pentru etapa de combinare este liniar, deci timpul total este de ordinul $O(n \cdot \log n)$.




← p
L