1. **RECURSIVITATE. DIVIDE ET IMPERA. CĂUTARE BINARĂ**
   1. **SCOPUL LUCRĂRII**

Lucrarea familiarizează studentul cu aplicarea metodei recursivităţii, atât în cazurile în care problema se prezintă într-o formă inerent (nativ) recursivă (cum ar fi funcţiile definite recursiv – ex. Funcţia Ackermann), cât şi în cazurile în care problema nu prezintă această caracteristică.

Includerea, în majoritatea aplicaţiilor, a unui contor al autoapelărilor funcţiei recursive subliniază limitările metodei.

Lucrarea îşi propune, de asemenea, să clarifice modul de aplicare în practică a metodei de programare “divde et impera” (de cele mai multe uri implementată printr-o funcţie recursivă) prin exerciţii pe vectori, dar şi în alte cazuri în care metoda poate fi aplicată.

Evident, este prezent şi cazul particular al căutării binare – cel mai eficient algoritm, lucrând în O[ln n].

* 1. **BREVIAR TEORETIC**
     1. **RECURSIVITATE**

Recursivitatea este un mecanism general de elaborare a algoritmilor. O funcţie se numeşte recursivă dacă ea se autoapelează, fie *direct* (în definiţia ei se face apel la ea însăşi), fie *indirect* (funcţia X apelează funcţia Y, care apelează funcţia X).

Recursivitatea a apărut din necesitati practice, date de transcrierea directă a formulelor matematice recursive. Apoi, acest mecanism a fost extins, fiind utilizat în elaborarea multor algoritmi.

De exemplu, definiţia recursivă a lui n! (factorialul lui n) este :



Functia recursivă se va scrie astfel:

*long factorial(int n)*

*{if(n==0) return 1;*

*else return factorial(n-1)\*n;}*

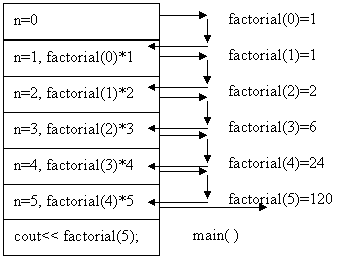
*void main()*

*{cout<<factorial(5) ;}*

Se observă ca funcţia factorial se autoapelează. Autoapelul se realizează prin instrucţiunea *return factorial(n-1)\*n.*

Recursivitatea utilizează segmentul de stivă pentru a memora datele programului ***apelant***, câtă vreme calculatorul rulează programul ***apelat***. De fiecare dată când o funcţie se autoapelează, se creează un nou nivel în segmentul de stivă.

Funcţionarea poate fi urmărită în figura de mai jos:



 Fiecare apel de funcţie lucrează cu datele aflate pe nivelul corespunzator acelui apel. La ieşirea din apelul unei funcţii, nivelul respectiv se eliberează şi datele aflate acolo se pierd. Există posibilitatea ca subprogramul să lucreze direct cu variabilele globale, dar în acest caz, subprogramul işi pierde independenţa.

Observaţii :

-        în cazul unui număr mare de autoapelări, există posibilitatea ca segmentul de stivă sa se ocupe total, caz în care programul se va termina cu eroarea STACK OVERFLOW. Aceasta se întamplă mai ales atunci când condiţia de terminare este pusă greşit şi subprogramul se apelează la nesfârşit.

-       pentru orice algoritm recursiv există unul iterativ, care rezolvă aceeaşi problemă.

-     mecanismul recursivităţii înlocuieste instrucţiunile repetitive.

-     datorită faptului că la fiecare autoapel se ocupă o zonă de memorie, recursivitatea este eficientă numai dacă numărul de autoapelări nu este prea mare, pentru a nu se ajunge la umplerea zonei de memorie alocată.

-        recursivitatea oferă avantajul unor soluţii mai clare pentru probleme şi a unei lungimi mai mici a programului. Ea prezintă însă dezavantajul unui timp mai mare de execuţie şi a unui spaţiu de memorie alocată mai mare. Este de preferat ca, atunci când programul recursiv poate fi transformat cu uşurinţă într-unul iterativ, să se facă apel la cel din urmă.

Este clar că ordinea executării instructiunilor va depinde de locul în care se întâlneşte apelul recursiv. Să studiem trei cazuri:

* ***Situaţia 1***

Pentru structura generică:

*tip\_func\_rec* ***sub*** *(listă de parametri){*

*{setul “****a****“de instrucţiuni;}*

*sub(parametri); // apelul recursiv*

*}*

se va executa setul *“****a****“* de instrucţiuni, în ordine “directă” (prima dată cele întâlnite la primul apel al funcţiei recursive ***sub***, ultima dată celor întâlnite la ultimul apel al funcţiei recursive ***sub***).

De ***exemplu***, pentru programul de mai jos

*#include <conio.h>*

*#include <iostream.h>*

*void recurs1(int n, int m){*

*cout<<"execut setul de instructiuni \"a\", pentru n="<<n<<endl;*

*if(n==m) cout<<"recursia s-a terminat!"<<endl;*

*else recurs1(n+1,m);*

*}*

*void main(){*

*clrscr();*

*int n,m;*

*cout<<"de la cat pornesc ? "; cin>>n;*

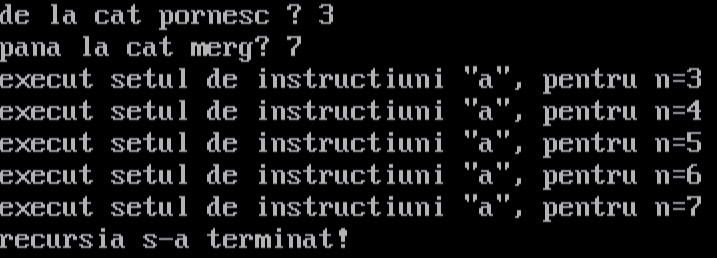
*cout<<"pana la cat merg? "; cin>>m;*

*recurs1(n,m);*

*getch();*

*}*

se vor obţine rezultatele:



* ***Situaţia 2:***

Pentru structura generică:

*tip\_func\_rec* ***sub*** *(listă de parametri){*

*sub(parametri); // apelul recursiv*

*{setul “****b****“de instrucţiuni;}*

*}*

se va executa setul *“****b****“* de instrucţiuni, în ordine “inversă” (prima dată cele întâlnite la ultimulul apel al funcţiei recursive ***sub***, ultima dată celor întâlnite la primul apel al funcţiei recursive ***sub***).

De ***exemplu***, pentru programul de mai jos

*#include <conio.h>*

*#include <iostream.h>*

*void recurs2(int n, int m){*

*if(n==m) cout<<"recursia s-a terminat!"<<endl;*

*else recurs2(n+1,m);*

*cout<<"execut setul de instructiuni \"b\", pentru n="<<n<<endl;*

*}*

*void main(){*

*clrscr();*

*int n,m;*

*cout<<"de la cat pornesc ? "; cin>>n;*

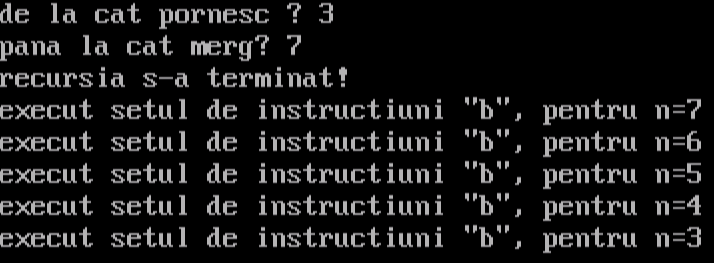
*cout<<"pana la cat merg? "; cin>>m;*

*recurs2(n,m);*

*getch();*

*}*

se vor obţine rezultatele:



* ***Situaţia 3 (combinaţie a celor două anterioare):***

fie structura generică:

*tip\_func\_rec* ***sub*** *(listă de parametri){*

*{setul “****a****” de instrucţiuni}*

*sub(parametri); // apelul recursiv*

*{setul “****b****“de instrucţiuni;}*

*}*

De ***exemplu***, pentru programul de mai jos

*#include <conio.h>*

*#include <iostream.h>*

*void recurs3(int n, int m){*

*cout<<"execut setul de instructiuni \"a\", pentru n="<<n<<endl;*

*if(n==m) cout<<"recursia s-a terminat!"<<endl;*

*else recurs3(n+1,m);*

*cout<<"execut setul de instructiuni \"b\", pentru n="<<n<<endl;*

*}*

*void main(){*

*clrscr();*

*int n,m;*

*cout<<"de la cat pornesc ? "; cin>>n;*

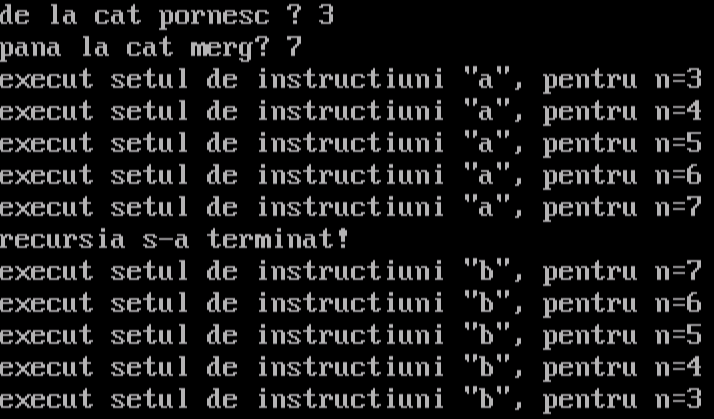
*cout<<"pana la cat merg? "; cin>>m;*

*recurs3(n,m);*

*getch();*

*}*

se vor obţine rezultatele:



Un ***exemplu practic*** pentru această din urmă situaţie în prezintă programul următor, care afişează în ordine inversă valorile citite de la tastatură (introducerea datelor se consideră încheiată când se introduce valoarea zero)

*#include <conio.h>*

*#include <iostream.h>*

*void recurs4(){*

*int n;*

*cout<<"n=? "; cin>>n;*

*if(n!=0) recurs4();*

*cout<<"n="<<n<<endl;*

*}*

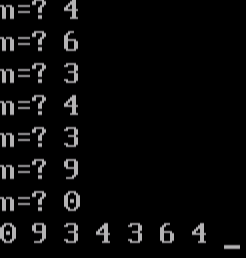
*void main(){*

*clrscr();*

*recurs4();*

*getch();*

*}*

**

* + 1. **DIVIDE ET IMPERA**

Metoda ***Divide et Impera*** (Imparte şi Stăpâneste) este o metodă de programare care se aplică problemelor care pot fi descompuse în subprobleme independente, similare problemei iniţiale, de dimensiuni mai mici, şi care pot fi rezolvate foarte uşor. Procesul se reia până când (în urma descompunerilor repetate) se ajunge la probleme care admit rezolvare imediată.

Nu toate problemele pot fi rezolvate prin utilizarea acestei tehnici. Numărul celor rezolvabile prin "divide et impera" este relativ mic, tocmai datorită cerinței ca problema să admită o descompunere repetată.

Divide et impera este o tehnică ce admite o implementare [recursivă](http://ro.wikipedia.org/wiki/Recursivitate). Principiul general prin care se elaborează algoritmi recursivi este: "ce se întâmplă la un nivel, se întâmplă la orice nivel" (având grijă să asigurăm condiţiile de terminare).

Un algoritm prin divide et impera se elaborează astfel: la un anumit nivel avem două posibilităţi:

1. s-a ajuns la o problemă care admite o rezolvare imediată (condiţia de terminare), caz în care se rezolvă şi se revine din apel;
2. nu s-a ajuns în situaţia de la punctul 1, caz în care problema curentă este descompusă în (două sau mai multe) subprobleme, pentru fiecare din ele urmând un apel recursiv al funcţiei, după care combinarea rezultatelor are loc fie pentru fiecare subproblemă, fie la final, înaintea revenirii din apel.

Aceasta metodă (tehnică) se poate implementa atât iterativ cât şi recursiv. Dat fiind că problemele se împart în subprobleme în mod recursiv, de obicei împărţirea se realizează până când şirul obţinut este de lungime 1, caz în care rezolvarea subproblemei este foarte uşoară.

De exemplu, fie un vector X=[x1, x2, x3, …**xi … xp… xj**, …xn] asupra căruia se aplică o prelucrare. Pentru orice secvenţă din vector delimitată de indecşii i si j, i<j există o valoare p, astfel încât prin prelucrarea secvenţelor :

xi, xi+1, xi+2, xi+3, …xp  si xp+1, xp+2, xp+3, …xj

se obţin soluţiile corespunzatoare celor două subşiruri care, prin compunere, conduc la obţinerea soluţiei prelucrării secvenţei:

xi, xi+1, xi+2, xi+3, …xj

* + 1. **CĂUTARE BINARĂ**

Un exemplu probleme de tip « Divide et Impera »  îl constituie căutarea binară: Se consideră un vector ordonat crescator cu N elemente şi o valoare K. Se cere să se scrie o funcţie care returnează poziţia elementului K în vectorul dat. În cazul în care K nu există în vector, se returnează -1. O soluţie ar fi căutarea secvenţială: se parcurge vectorul în mod normal, şi se localizează elementul K. Această soluţie are complexitatea O(N). Metoda optimă de rezolvare este cautarea binară. Aceasta se foloseşte de metoda divide et impera. Se împarte vectorul în jumătate, după care se continuă căutarea, în mod similar, în acea parte a vectorului unde se afla elementul K. Condiţia ca vectorul să fie sortat este esenţială. Complexitatea acestui algoritm este O(logN).

*#include <iostream.h>*

*#include <conio.h>*

*#include <fstream.h>*

*int v[1000], n, k;*

*int cautbin(int st, int dr) {*

*int m;*

*if( dr < st ) return -1;*

*m = (st + dr) / 2;*

*if(k == v[m]) return m;*

*if(k < v[m]) return cautbin(st, m-1);*

*return cautbin(m+1, dr);*

*}*

*int main() {*

*clrscr();*

*int i, poz;*

*ifstream fin("vectord.txt");*

*fin>>n;*

*for(i=1; i<=n; i++) fin>>v[i];*

*for(i=1; i<=n; i++) cout<<v[i]<<" ";*

*cout<<endl<<"ce valoare caut in vector? "; cin>>k;*

*poz=cautbin(1,n);*

*if(poz!=-1) cout<<"am gasit valoarea "<<k<<" pe pozitia "<<poz;*

*else cout<<"n-am gasit valoarea "<<k<<" in vector";*

*getch();*

*}*

* 1. **EXEMPLE**
     1. **RECURSIVITATE**

1. ***factorial recursiv***

*#include <conio.h>*

*#include <iostream.h>*

*unsigned long fact(int n){*

*if(n==1) return 1;*

*else return n\*fact(n-1);*

*}*

*void main(){*

*int n;*

*clrscr();*

*cout<<"n=";cin>>n;*

*cout<<n<<"!="<<fact(n);*

*getch();*

*}*

1. ***Cmmdc prin algoritmul lui Euclid implementat recursiv***

*#include<iostream.h>*

*#include<conio.h>*

*int euclid(int a, int b){*

*if(b==0) return a;*

*else euclid(b, a%b);*

*}*

*void main(){*

*clrscr();*

*int a, b, c;*

*cout<<"a=";cin>>a;*

*cout<<"b=";cin>>b;*

*c=euclid(a,b);*

*cout<<"cmmdc="<<c;*

*getch();*

*}*

1. ***Şirul lui Fibonacci (varianta recursivă)***

*#include <iostream>*

*using namespace std;*

*unsigned long f(int n){*

*if(n==0) return 0;*

*else if(n==1) return 1;*

*else return f(n-1)+f(n-2);*

*}*

*int main(){*

*int n, i;*

*cout<<"n=";cin>>n;*

*for(i=0; i<=n; i++)*

*cout<<"fibo("<<i<<") = "<<f(i)<<endl;*

*return 0;*

*}*

1. ***Trecerea unui număr din baza 10 în baza b (varianta recursivă)***

*#include <iostream>*

*using namespace std;*

*void baza(int n,int b){*

*int r;*

*char cr;*

*r=n%b;*

*if(r<10) cr=r+48; else cr=r+55;*

*if(n>=b) baza(n/b,b);*

*cout<<cr;*

*}*

*int main(){*

*int n,b;*

*cout<<"n=";cin>>n;*

*cout<<"b=";cin>>b;*

*baza(n,b);*

*return 0;*

*}*

Explicaţi utilizarea variabilei ***cr***, de tip ***char***!

1. ***Umplerea cu 1 a unei zone închise de valori de 0, delimitate de valori de 1***

*#include <conio.h>*

*#include <stdlib.h>*

*#include <iostream.h>*

*int nl,nc,i,j,l,c,a[100][100];*

*char r;*

*void umplu(int l,int c){*

*a[l][c]=1;*

*if ((l>1)&&(a[l-1][c]==0)) umplu(l-1,c);*

*if ((c>1)&&(a[l][c-1]==0)) umplu(l,c-1);*

*if ((c<nc)&&(a[l][c+1]==0)) umplu(l,c+1);*

*if ((l<nl)&&(a[l+1][c]==0)) umplu(l+1,c);*

*}*

*void main(void){*

*r='d';*

*clrscr();*

*cout<<'\333';*

*cout<<"linii=";cin>>nl;*

*cout<<"coloane=";cin>>nc;*

*randomize();*

*for (i=1;i<=nl;i++)*

*for (j=1;j<=nc;j++)*

*a[i][j]=random(2);*

*clrscr();*

*cout<<" ";*

*for (j=1;j<=nc;j++) cout<<j%10;cout<<endl<<endl;*

*for (i=1;i<=nl;i++) {*

*cout<<i%10<<" ";*

*for (j=1;j<=nc;j++) cout<<a[i][j];*

*cout<<endl;*

*}*

*cout<<endl;*

*while (r=='d') {*

*cout<<"linia=";cin>>l;*

*cout<<"coloana=";cin>>c;*

*umplu(l,c);*

*clrscr();*

*cout<<" ";*

*for (j=1;j<=nc;j++) cout<<j%10;cout<<endl<<endl;*

*for (i=1;i<=nl;i++) {*

*cout<<i%10<<" ";*

*for (j=1;j<=nc;j++) cout<<a[i][j];*

*cout<<endl;*

*}*

*cout<<endl;*

*cout<<"continuati?";r=getch();*

*cout<<endl;*

*}*

*}*

1. ***Ieşirea din labirint***

*#include <conio.h>*

*#include <iostream.h>*

*#include <fstream.h>*

*#include <stdio.h>*

*int a[50][50], nl, nc, xp, yp, xs, ys;*

*void afisare(){*

*int i, j, b, c;*

*for(i=0;i<=nl+1;i++){*

*for(j=0;j<=nc+1;j++)*

*if(a[i][j]==0) {b=15;textbackground(b);c=15;textcolor(c);cprintf(" ");}*

*else if(a[i][j]==2)*

*{b=15;textbackground(b);c=4;textcolor(c);cprintf("\*");}*

*else {b=4;textbackground(b);c=1;textcolor(c);cprintf("Û");}*

*cout<<endl;*

*}*

*getch();*

*cout<<endl;*

*}*

*void ies(int i, int j){*

*if(!a[i][j]){*

*a[i][j]=2;*

*if(i==xs && j==ys) afisare();*

*else{*

*ies(i-1,j);*

*ies(i,j+1);*

*ies(i+1,j);*

*ies(i,j-1);*

*}*

*a[i][j]=0;*

*}*

*}*

*void main(){*

*clrscr();*

*char numefis[20];*

*int i, j, k;*

*cout<<"numele fisierului de intrare = ? "; cin>>numefis;*

*ifstream fin(numefis);*

*fin>>nl>>nc;*

*fin>>xp>>yp>>xs>>ys;*

*for(i=1;i<=nl;i++){*

*for(j=1;j<=nc;j++)*

*fin>>a[i][j];*

*}*

*for(i=0;i<=nl+1;i++){*

*a[i][0]=1;*

*a[i][nc+1]=1;*

*}*

*for(j=0;j<=nc+1;j++){*

*a[0][j]=1;*

*a[nl+1][j]=1;*

*}*

*cout<<endl;*

*/\**

*cout<<"punctul de plecare:"<<endl;*

*cout<<"xp= ? "; cin>>xp;*

*cout<<"yp= ? "; cin>>yp;*

*cout<<"punctul de sosire:"<<endl;*

*cout<<"xs= ? "; cin>>xs;*

*cout<<"ys= ? "; cin>>ys;*

*\*/*

*ies(xp,yp);*

*getch();*

*}*

***Exemplu de fisier de intrare: maze1.in***

*11 11 7 3 4 10*

*0 0 0 0 1 0 0 0 0 0 0*

*0 1 0 1 1 1 0 1 1 0 1*

*0 1 0 0 0 0 0 0 1 0 1*

*1 1 0 1 1 1 0 1 0 0 1*

*0 1 0 1 0 1 0 1 0 1 1*

*0 1 0 1 0 1 0 1 0 0 0*

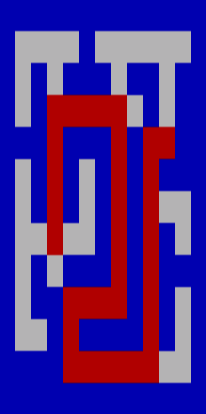
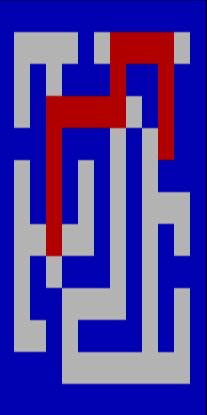
*0 0 0 0 0 1 0 1 0 1 0*

*0 1 0 1 1 1 0 1 0 1 1*

*0 1 1 0 0 0 0 1 0 1 0*

*0 0 1 0 1 1 1 1 0 1 0*

*1 1 1 0 0 0 0 0 0 0 0*



* + 1. **DIVIDE ET IMPERA. CĂUTARE BINARĂ**

1. ***Ghicirea unui număr***

*#include <fstream.h>*

*#include <conio.h>*

*void main(){*

*clrscr();*

*int x, y, z, rasp, gasit=0;*

*char fis[20];*

*cout<<"numele fisierului de intrare : ";cin>>fis;*

*ifstream f(fis);*

*f>>x>>y;*

*cout<<x<<" "<<y<<endl;*

*while((!gasit)&&(x<=y)){*

*z=(x+y)/2;*

*f>>rasp;*

*if(rasp) gasit=1;*

*else{*

*f>>rasp;*

*if(rasp)y=z-1;*

*else x=z+1;*

*}*

*}*

*if(gasit)cout<<z;*

*else cout<<"0";*

*getch();*

*f.close;*

*}*

1. ***Algoritmi de sortare-Merge sort***

Dată fiind o listă neordonată, o împărţim în două liste, de dimensiuni egale sau foarte apropriate, ordonăm cele două liste (folosind acelaşi algoritm) şi apoi efectuăm operaţia "merge", adică vom combina cele două liste ordonate într-una singură, obţinând astfel lista originală ordonată.

Operaţia principală care se efectuează este comparaţia dintre elementele listei.

*#include<iostream.h>*

*#include<conio.h>*

*int a[1000],n;*

*void interclas(int i,int m,int j)*

*{int b[1000];*

*int x=i;*

*int k=1;*

*int y=m+1;*

*while(x<=m && y<=j)*

*if(a[x]<a[y]) b[k++]=a[x++];*

*else b[k++]=a[y++];*

*while(x<=m) b[k++]=a[x++];*

*while(y<=j) b[k++]=a[y++];*

*int t=i;*

*for(k=1;k<=(j-i)+1;k++)*

*a[t++]=b[k];*

*}*

*int divimp(int i, int j)*

*{if(i<j) {int m=(i+j)/2;*

*divimp(i,m);*

*divimp(m+1,j);*

*interclas(i,m,j);}*

*}*

*void main()*

*{clrscr();*

*cout<<”n= ”; cin>>n;*

*for(int i=1;i<=n;i++)*

*{cout<<”a[”<<i<<”]= ”;*

*cin>>a[i];*

*}*

*divimp(1,n);*

*for(i=1;i<=n;i++)*

*cout<<a[i]<<” ”;*

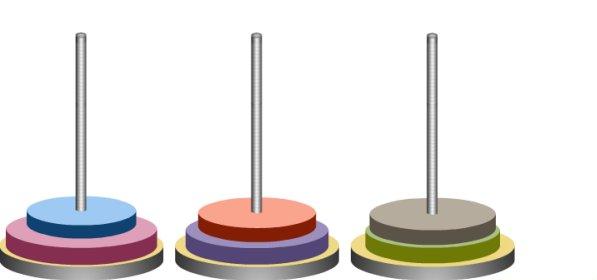
*getch();*

*}*

1. ***Turnurile din Hanoi***

Se dau trei tije verticale A, B şi C. Pe tija A se găsesc un număr ***n*** de discuri de diametre diferite, perforate la mijloc, aranjate în ordine descrescătoare a diametrelor discurilor, de la bază spre vârf. Celelalte tije sunt goale. Se cere să se găsească o strategie de mutare a tuturor discurilor de pe tija A pe tija B, respectând următoarele reguli:

* La un moment dat se va muta un singur disc (cel care se aflădeasupra celorlalte discuri pe o tijă).
* Un disc oate fi aşezat doar peste un alt disc având diametru mai mare decât al său, sau pe o tijă goală.

******

*#include <stdio.h>*

*#include <stdlib.h>*

*#include <conio.h>*

*#include <iostream.h>*

*void hanoi(int n,int a, int b, int c){*

*if (n==1) {cout<<"se muta discul 1 de pe tija "<<a<<" pe tija "<<b<<endl;*

*getch();return;}*

*hanoi(n-1,a,c,b);*

*cout<<"se muta discul "<<n<<" de pe tija "<<a<<" pe tija "<<b<<endl;*

*getch();*

*hanoi(n-1,c,b,a);*

*}*

*void main(void){*

*int n;*

*cout<<"n=";cin>>n;*

*hanoi(n,1,2,3);*

*}*

* 1. **STUDIU DE CAZ**

Până acum, ne-am ocupat mai mult de găsirea unui algoritm pentru a rezolva problemele pe care ni le-am propus (sau pe care ni le-au propus alţii …), fără a ne preocupa, în majoritatea cazurilor, de eficienţa acestor algoritmi. Situaţia este normală, la început, căci, pentru a îmbunătăţi un lucru, trebuie să ai ce îmbunătăţi!

În aplicaţiile reale, se pune adesea problema “ameliorării” soluţiilor găsite. Fără a încerca o abordare exhaustivă, să discutăm un caz concret:

Într-un fişier de pe disc, se găsesc două secvenţe de numere, ambele strict crescătoare. Astfel, pe prima linie se găsesc două numere naturale, ***n*** şi ***m***, reprezentând respectiv numărul de valori din secvenţa ***a*** şi din secvenţa ***b***. Pe linia a doua se găsesc n valori naturale, în ordine strict crescătoare, care constituie secvenţa ***a***, iar pe a treia linie ***m*** valori naturale, în ordine strict crescătoare, care constituie secvenţa ***b***. Se cere să se afişeze eventualele valori comune din cele două secvenţe şi numărul acestora.

În ***exemplul*** de mai jos, secvenţa ***a*** este constituită din 32 de valori, iar secvenţa ***b*** din 30 de valori.

32 30

4 11 27 33 42 53 57 65 81 92 108 121 175 198 204 236 275 299 313 350 491 503 507 508 512 519 606 610 611 615 702 717

8 12 16 22 28 31 36 49 57 88 100 108 203 275 300 313 314 317 322 383 391 412 453 491 493 496 503 506 519 729

Se observă că valorile 57, 108, 275, 313, 491, 503 şi 519 se regăsesc în ambele secvenţe.

Algoritmul de mai jos abordează “prin forţă brută” problema: se citesc din fişier valorile ***n*** şi ***m***, reprezentând lungimile celor două secvenţe, apoi cele două secvenţe sunt salvate în vectorii ***a[ ]*** şi ***b[ ]***. Cu un dublu for, fiecare element din vectorul ***b[ ]*** (primul for) este căutat în vectorul ***a[ ]*** (al doilea for). Programul ar putea arăta aşa:

*#include <conio.h>*

*#include <iostream.h>*

*#include <fstream.h>*

*void main(){*

*clrscr();*

*int n, m, i, j, a[200], b[200], nr=0;*

*char numefis[50];*

*cout<<"numele fisierului de intrare: "; cin>>numefis;*

*ifstream fin(numefis);*

*fin>>n>>m;*

*for(i=1; i<=n; i++) fin>>a[i];*

*cout<<endl;*

*cout<<"vectorul a:"<<endl;*

*for(i=1; i<=n; i++) cout<<a[i]<<" ";*

*cout<<endl;*

*for(j=1; j<=m; j++) fin>>b[j];*

*cout<<endl;*

*cout<<"vectorul b:"<<endl;*

*for(j=1; j<=m; j++) cout<<b[j]<<" ";*

*cout<<endl<<endl;*

*for(j=1; j<=m; j++){*

*for(i=1; i<=n; i++)*

*if(a[i]==b[j]){*

*nr++;*

*cout<<a[i]<<" ";*

*}*

*}*

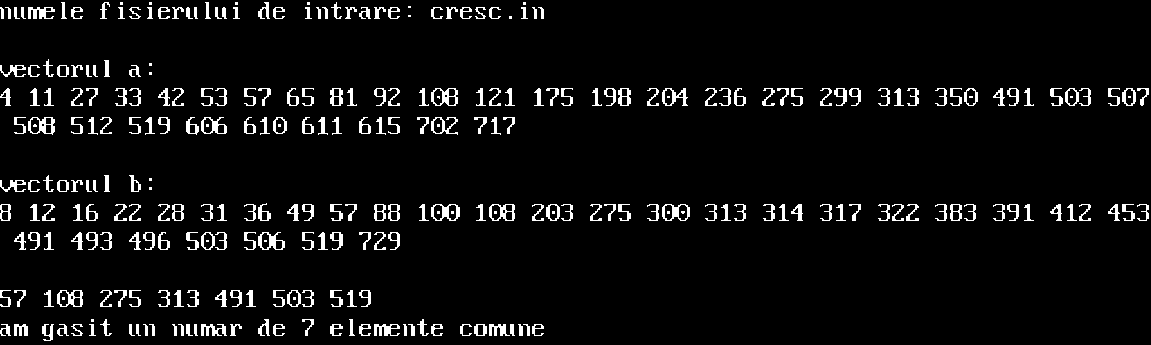
*cout<<endl;*

*cout<<"am gasit un numar de "<<nr<<" elemente comune"<<endl;*

*getch();*

*}*

Rulând acest program, se obţine rezultatul (corect!) de mai jos:

****

Prima problemă de “îmbunătăţire” are în vedere consumul de memorie RAM (să nu uităm că, în practică, secvenţele ar putea fi foarte lungi, conducând la consum exagerat de memorie): este oare strict necesar să reţinem ambele secvenţe în vectori? Se observă imediat că, datorită căutării repetate în secvenţa ***a***, aceasta ***trebuie*** reţinută într-un vector, pentru a evita citirea repetată din fişier a acestei secvenţe (lucru prohibitiv, care ar anula orice efort de ameliorare a algoritmului). Un număr din a doua secvenţă, însă, nu ne este de folos decât pe parcursul căutării lui în prima secvenţă, ceea ce înseamnă că, după citirea lui şi căutarea în prima secvenţă, poate fi “abandonat”, nemaifiind necesară reţinerea sa într-un vector. Am înlocuit astfe un întreg vector ( ***b[ ]*** ) cu o singură variabilă (***b***)! (***Întrebare***: puteam renunţa la a memora vectorul ***a[ ]*** , în locul lui memorând, în schimb, vectorul ***b[ ]*** ?)

Programul ar putea arăta aşa:

*#include <conio.h>*

*#include <iostream.h>*

*#include <fstream.h>*

*void main(){*

*clrscr();*

*int n, m, i, j, a[200], b, nr=0, nrcomp=0;*

*char numefis[50];*

*cout<<"numele fisierului de intrare: "; cin>>numefis;*

*ifstream fin(numefis);*

*fin>>n>>m;*

*for(i=1; i<=n; i++) fin>>a[i];*

*cout<<endl;*

*cout<<"vectorul a:"<<endl;*

*for(i=1; i<=n; i++) cout<<a[i]<<" ";*

*cout<<endl<<endl;*

*for(j=1; j<=m; j++){*

*fin>>b;*

*for(i=1; i<=n; i++){*

*nrcomp++;*

*if(a[i]==b){*

*nr++;*

*cout<<a[i]<<" ";*

*}*

*}*

*}*

*cout<<endl;*

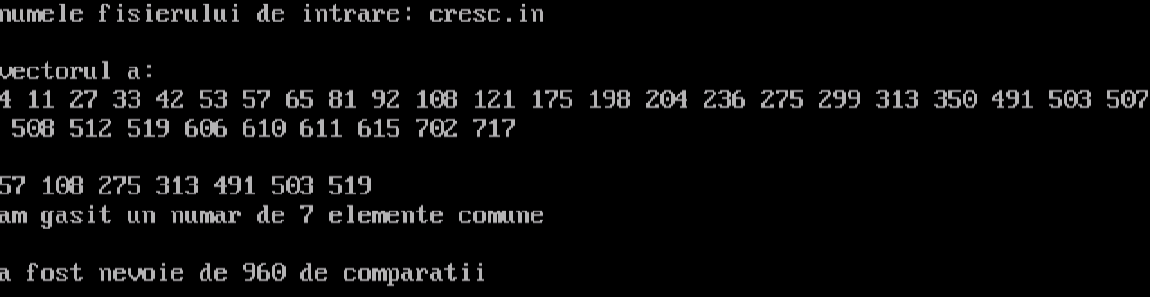
*cout<<"am gasit un numar de "<<nr<<" elemente comune"<<endl<<endl;*

*cout<<"a fost nevoie de "<<nrcomp<<" de comparatii";*

*getch();*

*}*

Rulând acest program, se obţine rezultatul de mai jos:

****

Au fost, în mod evident, necesare 32 x 30 = 960 de comparaţii între elemente.

N-am ţinut însă nicăieri seama, până acum, că cele două secvenţe sunt ambele strict crescătoare, reluând mereu de pe prima poziţe căutarea în secvenţa ***a***, pentru fiecare nouă valoare ***b***, când este clar că, dacă am găsit o “potrivire” pe o poziţie oarecare ***poz*** din vectorul ***a[ ]***, o alta nu poate să mai fie găsită decât pe o poziţie ulterioară. Este, deci, suficient să reţinem, de fiecare dată, poziţia ultimei “potriviri” şi să pornim căutarea următoare de la această poziţie mai departe.

Programul ar putea arăta acum aşa:

*#include <conio.h>*

*#include <iostream.h>*

*#include <fstream.h>*

*void main(){*

*clrscr();*

*int n, m, i, j, a[200], b, nr=0, nrcomp=0, poz=1;*

*char numefis[50];*

*cout<<"numele fisierului de intrare: "; cin>>numefis;*

*ifstream fin(numefis);*

*fin>>n>>m;*

*for(i=1; i<=n; i++) fin>>a[i];*

*cout<<endl;*

*cout<<"vectorul a:"<<endl;*

*for(i=1; i<=n; i++) cout<<a[i]<<" ";*

*cout<<endl<<endl;*

*for(j=1; j<=m; j++){*

*fin>>b;*

*for(i=poz; i<=n; i++){*

*nrcomp++;*

*if(a[i]==b){*

*poz=i;*

*nr++;*

*cout<<a[i]<<" ";*

*}*

*}*

*}*

*cout<<endl;*

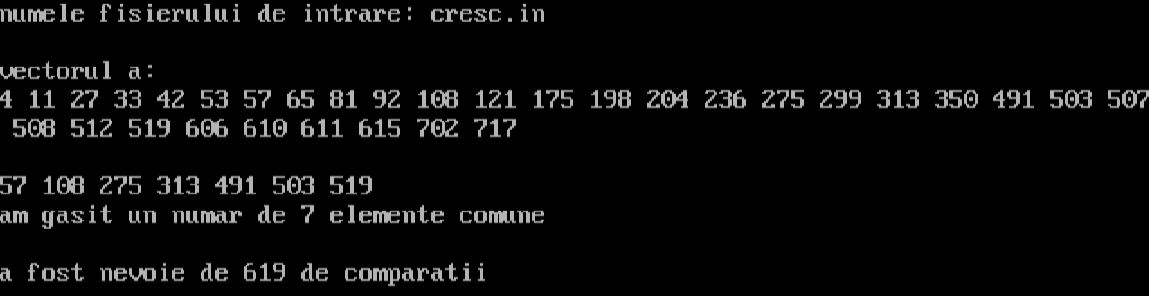
*cout<<"am gasit un numar de "<<nr<<" elemente comune"<<endl<<endl;*

*cout<<"a fost nevoie de "<<nrcomp<<" de comparatii";*

*getch();*

*}*

Rulând acest program, se obţine rezultatul de mai jos:

****

Observăm că n-a mai fost nevoie decât de 619 comparaţii!

Dar, tocmai am aflat în acest capitol că, în cazul vectorilor ordonaţi, avem o “armă” de căutare mult mai eficientă (de fapt, cea mai eficientă, care lucrează în O[ln n] ): căutarea binară. Aşa că s-o folosim!

*#include <conio.h>*

*#include <iostream.h>*

*#include <fstream.h>*

*int nrcomp=0;*

*int cautbin(int v[], int n, int k, int p, int u){*

*int poz=-1, m;*

*while(p<=u){*

*nrcomp++;*

*m=(p+u)/2;*

*if(k==v[m]) return m;*

*if(k<v[m]) u=m-1;*

*else p=m+1;*

*}*

*return poz;*

*}*

*void main(){*

*clrscr();*

*int n, m, i, j, a[200], b, nr=0, poz=1, gasit;*

*char numefis[50];*

*cout<<"numele fisierului de intrare: "; cin>>numefis;*

*ifstream fin(numefis);*

*fin>>n>>m;*

*for(i=1; i<=n; i++) fin>>a[i];*

*cout<<endl;*

*cout<<"vectorul a:"<<endl;*

*for(i=1; i<=n; i++) cout<<a[i]<<" ";*

*cout<<endl<<endl;*

*for(j=1; j<=m; j++){*

*fin>>b;*

*gasit=cautbin(a,n,b,1,n);*

*if(gasit!=-1){*

*nr++;*

*cout<<a[gasit]<<" ";*

*}*

*}*

*cout<<endl;*

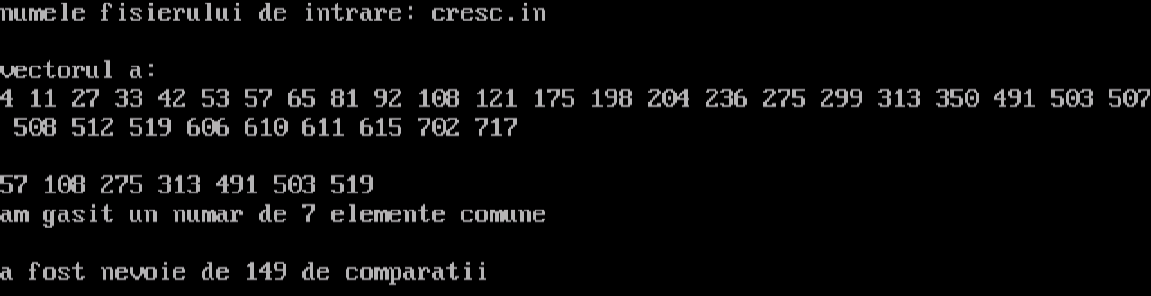
*cout<<"am gasit un numar de "<<nr<<" elemente comune"<<endl<<endl;*

*cout<<"a fost nevoie de "<<nrcomp<<" de comparatii";*

*getch();*

*}*

Rulând programul (ce cuprinde subprogramul recursiv ***cautbin***), se obţine rezultatul de mai jos:

****

A mai fost nevoie doar de 149 de comparaţii.

Combinând cu “perfecţionarea” de a porni căutarea de la poziţia ultimei “potriviri”, programul devine:

*#include <conio.h>*

*#include <iostream.h>*

*#include <fstream.h>*

*int nrcomp=0;*

*int cautbin(int v[], int n, int k, int p, int u){*

*int poz=-1, m;*

*while(p<=u){*

*nrcomp++;*

*m=(p+u)/2;*

*if(k==v[m]) return m;*

*if(k<v[m]) u=m-1;*

*else p=m+1;*

*}*

*return poz;*

*}*

*void main(){*

*clrscr();*

*int n, m, i, j, a[200], b, nr=0, poz=1, gasit;*

*char numefis[50];*

*cout<<"numele fisierului de intrare: "; cin>>numefis;*

*ifstream fin(numefis);*

*fin>>n>>m;*

*for(i=1; i<=n; i++) fin>>a[i];*

*cout<<endl;*

*cout<<"vectorul a:"<<endl;*

*for(i=1; i<=n; i++) cout<<a[i]<<" ";*

*cout<<endl<<endl;*

*for(j=1; j<=m; j++){*

*fin>>b;*

*gasit=cautbin(a,n,b,poz,n);*

*if(gasit!=-1){*

*poz=gasit;*

*nr++;*

*cout<<a[gasit]<<" ";*

*}*

*}*

*cout<<endl;*

*cout<<"am gasit un numar de "<<nr<<" elemente comune"<<endl<<endl;*

*cout<<"a fost nevoie de "<<nrcomp<<" de comparatii";*

*getch();*

*}*

Rulând acest program (ultimul din această discuţie!), se obţine rezultatul de mai jos:

****

A mai fost nevoie doar de 130 de comparaţii.

Ţineţi cont că am pornit de la doi vectori şi 960 de comparaţii!

**8.5 TEME DE LABORATOR**

1. Să se calculeze recursiv funcţia Ackerman, definită
2. Să se calculeze recursiv funcţia Ackerman, definită



pentru valori ale lui m de la 1 la 3 şi valori ale lui n de la 1 la 20

1. Pentru un număr ***n***, introdus de la tastarură, să se calculeze recursiv sumele primelor ***n*** termeni din
2. Să se calculeze recursiv primele ***n*** elemente din şirul lui Fibonacci, cu numărarea apelurilor necesare
3. Să se determine maximul (minimul) a ***n*** numere întregi, cu numărarea apelurilor necesare
4. Să se determine recursiv cel mai mare divizor comun a elementelor unui vector ***v*** cu ***n*** elemente
5. Să se caute o valoare într-un vector oarecare. Dacă se găseşte, se va afişa poziţia pe care s-a găsit, altfel se va afişa un mesaj.
6. Să se caute o valoare într-un vector ordonat descrescător