Desene recursive

O imagine sau un desen are un model recursiv (*recursive pattern*) dacă o anumită parte a sa este compusă din mai multe componente similare cu aceasta şi, la rândul lor, fiecare astfel de componentă este compusă din părţi similare cu ea, şi aşa mai departe. De exemplu, căpăţâna de conopidă/broccoli Romanesco[1]

Figura 1. *Romanesco broccoli*

are un model recursiv. La fel şi o frunză de cucută:

Figura 2. *Conium maculatum*

[1] vezi https://en.wikipedia.org/wiki/Romanesco_broccoli

Vom prezenta aici trei metode de trasare a unui desen recursiv utilizând un exemplu mult mai simplu decât imginile de mai sus, şi anume vom desena *pătratul lui Sierpiski*[2] , S , care nu este altceva decât varianta bidimensională a *mulţimii ternare a lui Cantor*[3], K, mai precis

$$S = K \times K.$$

Să definim mulţimea S în mod direct (vezi Figura 3). Pornim cu un pătrat P de latură ℓ pe care îl împărţim în 9 pătrate p de latură $\ell/3$, dispuse în trei rânduri şi trei coloane. Eliminăm interiorul pătratului central şi ne rămân 8 pătrate p. Pentru fiecare dintre acestea repetăm procededura de împărţire şi eliminare. Obţinem 8^2 pătrate de latură $\ell/3^2$. Repetăm la nesfârşit procedura. Punctele care rămân ne-eliminate formează, prin definiţie, mulţimea S.

Este clar că S, pătratul lui Sierpinski, este nevid: măcar punctele aflate pe laturile pătratelor centrale rămân ne-eliminate. Această mulţime are multe proprietăţi interesante care ţin de topologia lui R^2 şi de teoria măsurii, dar aici noi dorim numai să o desenăm în C#.

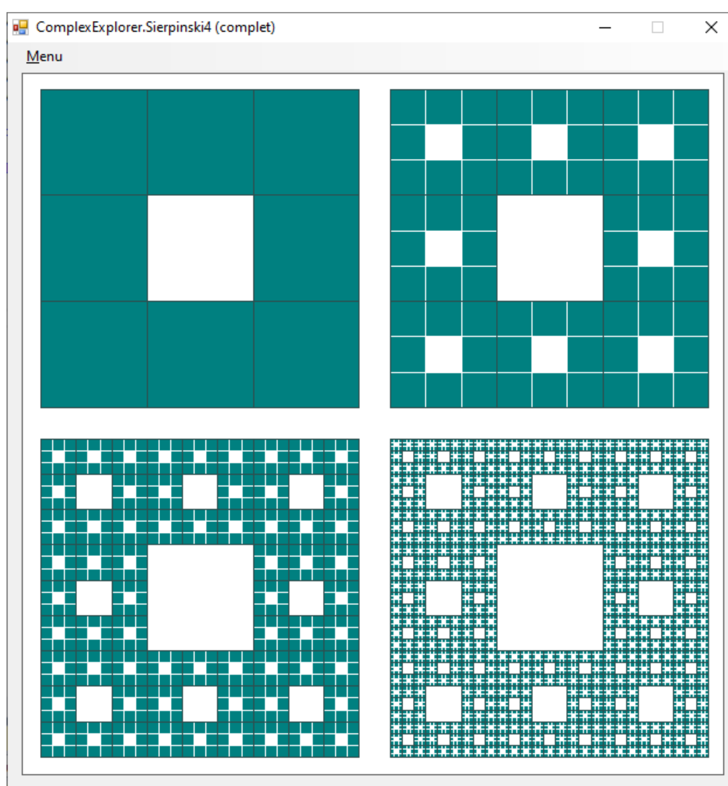


Figura 3. Pătratul lui Sierpinski

Vom folosi liste de pătrate, deci va trebui să definim o structură care să păstreze toate informaţiile necesare desenării unui pătrat. Obsrvăm că toate pătratele care apar în figură sunt cu laturile paralele cu axele de coordonate, iar pentru a trasa un astfel de pătrat este suficient să cunoaştem centrul şi lungimea laturii.

[2] Wacław Sierpiński (1882-1969), matematician polonez.
[3] vezi https://en.wikipedia.org/wiki/Cantor_set

Vom îngloba definiţia structurii şi a metodelor de trasare ale ei într-o clasă derivată din `ComplexForm`, pe care o vom moşteni mai departe când vom trasa efectiv desenul cu metoda `makeImage()`.

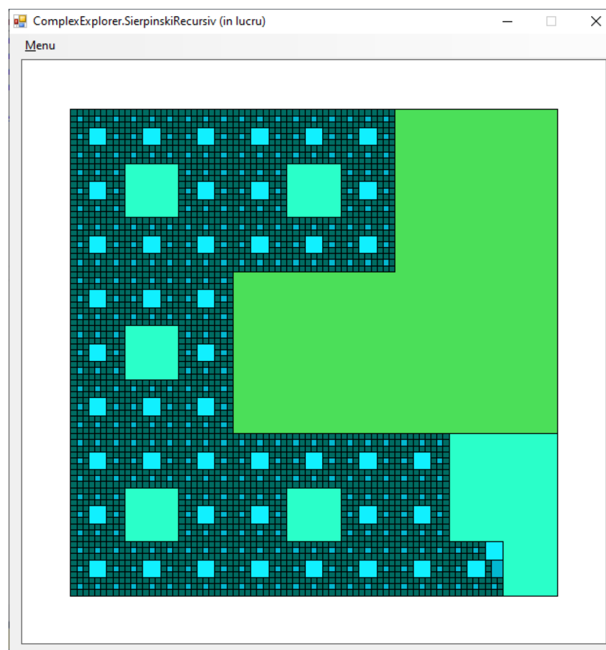
```
public class ComplexFormSierpinski : ComplexForm
{
    public static Complex i = Complex.setReIm(0, 1);
    public struct Patrat
    {   //patratul de centru q si de latura lat
        public Complex q;
        public double lat;
        public Patrat(Complex qq, double llat)
        {
            q = qq;
            lat = llat;
        }
    }
    public void deseneazaPatrat(Patrat P, Color col)
    {
        double r = P.lat / 2;
        int iimin = getI(P.q.Re - r), iimax = getI(P.q.Re + r);
        int jjmin = getJ(P.q.Im - r), jjmax = getI(P.q.Im + r);
        for (int ii = iimin; ii <= iimax; ii++)
            for (int jj = jjmin; jj <= jjmax; jj++)
            {
                setPixel(ii, jj, col);
            }
        setLine(iimin, jjmin, iimax, jjmin, PenColor);
        setLine(iimin, jjmax, iimax, jjmax, PenColor);
        setLine(iimin, jjmin, iimin, jjmax, PenColor);
        setLine(iimax, jjmin, iimax, jjmax, PenColor);
    }
    public void traseaza(List<Patrat> li, Color col)
    {
        foreach (Patrat p in li) deseneazaPatrat(p, col);
    }
}
```

Începem cu metoda cea mai uşor de programat şi cea mai grea de executat de calculator: funcţii recursive. Despre ineficienţa implementărilor cu funcţii recursive am mai discutat şi vom mai reveni.

Ce executăm pe un anumit nivel de autoapelare? Dacă am depăşit nivelul final nu avem nimic de făcut: închidem cu `return` apelul curent, dacă nu, trasăm pătratul curent, apoi îl împărţim în 9 părţi egale şi trecem la nivelul următor cu 8 autoapelări. Simplu:

```
public class SierpinskiRecursiv : ComplexFormSierpinski
    {
        void deseneazaRec(Patrat P, int niv)
        {
            if (niv <= 0) return;
            deseneazaPatrat(P, getColor(900 - 50 * niv));
            if (!resetScreen()) return;

            niv--;
            double x = P.q.Re, y = P.q.Im;
```

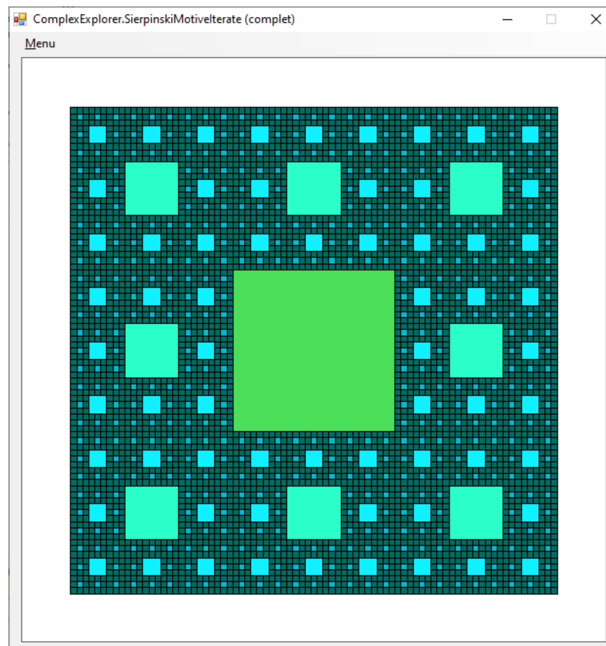
```
            double lat = P.lat / 3;
            deseneazaRec(new Patrat(new Complex(x - lat, y - lat), lat), niv);
            deseneazaRec(new Patrat(new Complex(x - lat, y), lat), niv);
            deseneazaRec(new Patrat(new Complex(x - lat, y + lat), lat), niv);
            deseneazaRec(new Patrat(new Complex(x, y - lat), lat), niv);

            deseneazaRec(new Patrat(new Complex(x, y + lat), lat), niv);
            deseneazaRec(new Patrat(new Complex(x + lat, y - lat), lat), niv);
            deseneazaRec(new Patrat(new Complex(x + lat, y), lat), niv);
            deseneazaRec(new Patrat(new Complex(x + lat, y + lat), lat), niv);
        }

        public override void makeImage()
        {
            setXminXmaxYminYmax(-0.1, 1.1, -0.1, 1.1);
            ScreenColor = Color.White;
            Patrat P = new Patrat((1 + i) / 2, 1.0);
            deseneazaRec(P, 5);
            resetScreen();


        }
    }
    /*******************************************/
```



Rezolvarea care urmează, prin metoda motivelor iterate, este cea mai bună: şi uşor de programat, şi uşor de executat. Singura dificultate care apare: trebuie să memorăm cumva pătratele generate la fiecare etapă şi pentru aceasta vom folosi, aşa cum am spus, liste generice.

În ce constă metoda: în etapa iniţială baza este un pătrat care va fi înlocuit cu motivul format din 8 pătrăţele care în etapa următoare vor deveni 8 baze. Prin urmare, în fiecare etapă, transformăm o listă înlocuind fiecare pătrat din listă cu alte 8 pătrate mai mici. La fel de simplu ca mai înainte, dar mai eficient:

```
public class SierpinskiMotiveIterate : ComplexFormSierpinski
    {
        void transforma(ref List<Patrat> li)
        {
            List<Patrat> rez = new List<Patrat>();
            foreach (Patrat P in li)
            {
                double x = P.q.Re, y = P.q.Im;
                double lat = P.lat / 3;
                rez.Add(new Patrat(new Complex(x - lat, y - lat), lat));
                rez.Add(new Patrat(new Complex(x - lat, y), lat));
                rez.Add(new Patrat(new Complex(x - lat, y + lat), lat));
                rez.Add(new Patrat(new Complex(x, y - lat), lat));

                rez.Add(new Patrat(new Complex(x, y + lat), lat));
                rez.Add(new Patrat(new Complex(x + lat, y - lat), lat));
                rez.Add(new Patrat(new Complex(x + lat, y), lat));
                rez.Add(new Patrat(new Complex(x + lat, y + lat), lat));
            }
            li = rez;
        }
        public override void makeImage()
        {
            setXminXmaxYminYmax(-0.1, 1.1, -0.1, 1.1);
            ScreenColor = Color.White;
            List<Patrat> fig = new List<Patrat>();
            fig.Add(new Patrat((1 + i) / 2, 1.0));
            traseaza(fig, getColor(650));
            for (int k = 1; k < 5; k++)
            {
                transforma(ref fig);
                traseaza(fig, getColor(650 + 50 * k));
                if (!resetScreen()) return;
            }
        }
    }
```



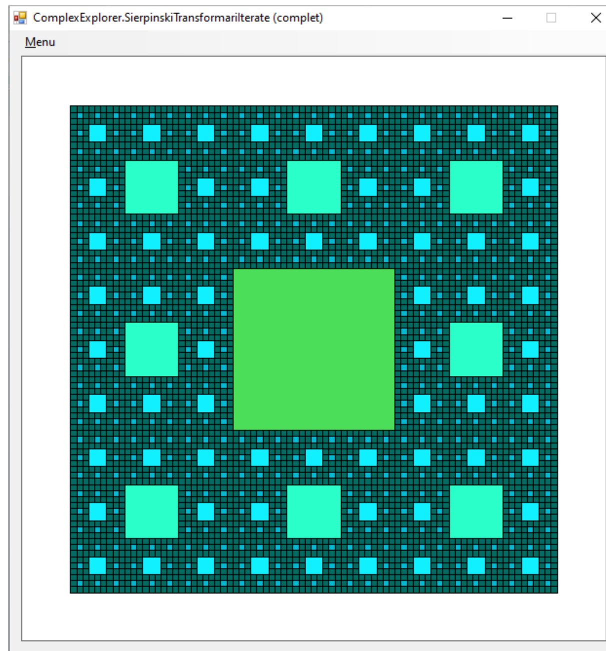
În sfârşit, ultima metodă, cea a transformărilor geometrice iterate, se deosebeşte de cele precedente prin modul de aplicare a similitudinilor: până acum în etapa curentă fiecare pătrat era înlocuit cu 8 pătrăţele, acum în etapa curentă vom înlocui întreaga figură S cu 8 figuri s, similare cu S prin 8 transformări geometrice stabilite de la bun început, şi anume cele care duc pătratul iniţial, $U = [0,1] \times [0,1]$, în cele 8 pătrăţele de latură $1/3$.

Dificultatea metodei constă numai în stabilirea acestor transformări geometrice. Să studiem codul:

```
public class SierpinskiTransformariIterate : ComplexFormSierpinski
    {

        Complex w1 = (1 + 1 * i) / 6;
        Complex w2 = (1 + 3 * i) / 6;
        Complex w3 = (1 + 5 * i) / 6;
        Complex w4 = (3 + 1 * i) / 6;
        Complex w5 = (3 + 3 * i) / 6;
        Complex w6 = (3 + 5 * i) / 6;
        Complex w7 = (5 + 1 * i) / 6;
        Complex w8 = (5 + 3 * i) / 6;
        Complex w9 = (5 + 5 * i) / 6;


        Patrat T1(Patrat P) { return new Patrat(w1 + (P.q - w5) / 3.0, P.lat / 3); }
        Patrat T2(Patrat P) { return new Patrat(w2 + (P.q - w5) / 3.0, P.lat / 3); }
        Patrat T3(Patrat P) { return new Patrat(w3 + (P.q - w5) / 3.0, P.lat / 3); }
        Patrat T4(Patrat P) { return new Patrat(w4 + (P.q - w5) / 3.0, P.lat / 3); }

        //lipseste T5
        Patrat T6(Patrat P) { return new Patrat(w6 + (P.q - w5) / 3.0, P.lat / 3); }
        Patrat T7(Patrat P) { return new Patrat(w7 + (P.q - w5) / 3.0, P.lat / 3); }
        Patrat T8(Patrat P) { return new Patrat(w8 + (P.q - w5) / 3.0, P.lat / 3); }
        Patrat T9(Patrat P) { return new Patrat(w9 + (P.q - w5) / 3.0, P.lat / 3); }

        void transforma(ref List<Patrat> li)
        {
            List<Patrat> rez = new List<Patrat>();
            foreach (Patrat P in li) rez.Add(T1(P));
            foreach (Patrat P in li) rez.Add(T2(P));
            foreach (Patrat P in li) rez.Add(T3(P));
            foreach (Patrat P in li) rez.Add(T4(P));
            //lipseste T5
            foreach (Patrat P in li) rez.Add(T6(P));
            foreach (Patrat P in li) rez.Add(T7(P));
            foreach (Patrat P in li) rez.Add(T8(P));
            foreach (Patrat P in li) rez.Add(T9(P));
            li = rez;
        }


        public override void makeImage()
        {
            setXminXmaxYminYmax(-0.1, 1.1, -0.1, 1.1);
            ScreenColor = Color.White;
            List<Patrat> fig = new List<Patrat>();
            fig.Add(new Patrat(w5, 1.0));
            traseaza(fig, getColor(650));
```

```
            for (int k = 1; k < 5; k++)
            {
                transforma(ref fig);
                traseaza(fig, getColor(650 + 50 * k));
                if (!resetScreen()) return;
            }
        }
    }
```

După cum se vede obţinem acelaşi rezultat:



Aici `Complex w5 = (3 + 3 * i) / 6;` este centrul pătratului iniţial, $U = [0,1] \times [0,1]$, iar celelale 8 `w`-uri sunt centrele celor 8 pătrăţele de latură $1/3$ din interiorul lui U.

Transformarea `Patrat T1(Patrat P)`,de exemplu, duce orice pătrat `P` din plan în pătratul obţinut după ce se aplică o omotetie de centru `w5` şi raport $1/3$, urmată de o translaţie de vector `w1- w5`, care mută centrul pătrăţelului rezultat în `w1`. În primul `foreach` din `transforma()`, aplicăm transformarea geometrică `T1` la toate pătratele din lista primită, altfel spus transformăm prin `T1` întreaga figură obţinută până în etapa curentă. Aplicăm apoi şi celelalte transformări `T`.

În cazul exemplului nostru, pătratul lui Sierpinski, această metodă pare mai dificilă decât cea a motivelor iterate, deoarece acolo, la aplicarea motivului am făcut doar câteva calcule foarte simple pentru a determina centrele celor 8 pătrăţele. Totuşi, în general, aplicarea motivului presupune utilizarea locală a unor transformări geometrice, vezi cazul curbei lui Koch, şi atunci cele două metode sunt comparabile ca dificultate.

În final, revin cu discuţia despre ineficienţa implementărior cu funcţii recursive. Deoarece problema este aceeaşi şi în C#, în C/C++, care sunt limbaje imperative executate pe o

singură stivă, adaug aici un curs mai vechi, care nu mai apare în programa actuală a disciplinei *Fundamentele programării* din anul I de studiu, dar al cărui conţinut este valabil în continuare.

Funcţii recursive în C/C++

(*Cursul* 9.*b din* 2017*, anul* I)

Antetul unei funcţii conţine toate informaţiile necesare compilatorului pentru a proiecta apelarea ei. De exemplu, în cazul funcţiei ***f*** cu antetul

```
int f(double x, double y){ ... }
```

la evaluarea unui apel de forma

```
a=f(1.0,2.0)
```

compilatorul ştie că mai întâi trebuie depuse pe stivă valorile actuale `1.0` şi `2.0` ale celor doi parametrii în două locaţii de tip ***double*** nou create, după care trebuie pus în execuţie codul funcţiei ***f*** printr-o instrucţiune ***call*** a asamblorului, iar la încetarea apelului rezultatul, lăsat de funcţie într-un registru al microprocesorului, trebuie atribuit lui ***a***. Când proiectează codul funcţiei ***f*** compilatorul deja a citit antetul acesteia, deci apelul de mai sus poate fi tradus în cod maşină chiar şi în cazul în care el apare chiar în corpul funcţiei ***f***. Intr-o astfel de situaţie spunem că ***f*** *se auto-apelează*, sau că ***f*** este o *funcţie recursivă*.

In programul următor avem două exemple de funcţii recursive, unul bun şi altul rău. Amândouă trec de faza de compilare, dar la rulare funcţia ***asaNu*** produce o eroare gravă de execuţie (depăşire de stivă), în timp ce ***asaDa*** are rezultatul afişat în comentariul final:

```
#include<iostream>
using namespace std;

void asaNu(char ch){
	cout << ch;
	asaNu(ch + 1);
	return;
}

void asaDa(char ch){
	if (ch > 'z') return;
	cout << ch;
	asaDa(ch + 1);
	return;
}

int main(void){
	asaDa('a');
	//asaNu('a');
	cout << endl;
	return 0;
}
/*
```

```
abcdefghijklmnopqrstuvwxyz
Press any key to continue . . .*/
```

In cazul funcţiei ***asaNu*** compilatorul ne dă avertismentul

```
warning C4717: 'asaNu' : recursive on all control paths, function will cause
runtime stack overflow
```

deoarece constată că nu a fost prevăzută nici o cale de *ieşire din recursivitate*, printr-o testare care să oprească auto-apelarea la nesfârşit a funcţiei. Dacă trecem peste acest avertisment şi lansăm în execuţie funcţia, stiva se va umple imediat iar programul va fi stopat de către sistemul de operare.

La proiectarea unei funcţii recursive prima grijă a programatorului este să asigure ieşirea din recursivitate, pe cât posibil chiar din prima instrucţiune (ca în cazul funcţiei ***asaDa***).

O funcţie ***f*** poate să fie recursivă şi fără să se auto-apeleze, atunci când face parte dintr-un şir de funcţii care se apelează ciclic, de exemplu când în corpul ei există un apel către o funcţie ***g*** care, la rândul ei o apelează pe ***f***. Şi în acestă situaţie trebuie asigurată, în primul rând, ieşirea din recursivitate.

Să observăm că orice buclă iterativă poate fi implementată cu o funcţie recursivă. De exemplu, următorul ***for***

```
	for (int i = imin; i < imax; i++){
		cout << i << endl;
	}
```

este echivalent cu apelul `pasCuPas(imin, imax)` al funcţiei auto-apelante

```
void	pasCuPas(int i, int imax){
	if (i >= imax) return;
	cout << i << endl;
	pasCuPas(i + 1, imax);
	return;
}
```

Avem aici două modalităţi distincte, una iterativă şi alta recursivă, pentru a produce acelaşi rezultat, dintre care prima este de preferat, deoarece se execută mai repede (reiterările cu ***for***, ***while*** sau ***do-while*** sunt mult mai rapide în C/C++ decât apelurile de funcţii) şi nu încarcă deloc stiva. In general, în limbajul C/C++ rezolvările iterative sunt mai eficiente decât cele recursive.

Iată o rezolvare cu funcţii recursive a problemei determinării valorii maxime a unui tablou, bazată pe observaţia că maximul unei mulţimi $\{a_i, a_{i+1}, \ldots a_j, a_{j+1}, \ldots a_k\}$ este egal cu cel mai mare dintre maximul submulţimii $\{a_i, a_{i+1}, \ldots a_j\}$ şi maximul submulţimii $\{a_{j+1}, \ldots a_k\}$:

```
#include<iostream>
using namespace std;
int maxim(int a[], int i, int k){
	if(i>=k) return a[k];
	int j=(i+k)/2;
```

```
        int maxStg=maxim(a,i,j);
        int maxDrp=maxim(a,j+1,k);
        return maxStg<maxDrp? maxDrp : maxStg;
}
int maximIterativ(int a[], int i, int k){
        int max=a[i];
        for(int j=i+1;j<=k;j++)
                max=(max>=a[j] ? max : a[j]);
        return max;
}
const int n=100000;
int a[n]={1,2,300,4,50,60,7,8,9,100};
int main(void){
        cout<<maxim(a,0,n-1)<<endl;
        cout<<maximIterativ(a,0,n-1)<<endl;
        return 0;
}
```

Am prezentat spre comparaţie şi rezolvarea iterativă clasică. Chiar dacă metoda recursivă de mai sus pare foarte eficientă, numărul total de comparaţii este în ambele cazuri de ordinul lui n, de unde urmează că vitezele de execuţie sunt comparabile. In cazul recursiv, deoarece la fiecare etapă se înjumătăţeşte dimensiunea problemei, nivelul maxim de auto-apelare este de ordinul lui $\log_2 n$; dar fiecare apel de pe nivelele intermediare generează două auto-apelări, în consecinţă numărul total de apeluri este de ordinul lui n.

Subliniem că orice metodă iterativă se bazează tot pe surprinderea unei relaţii de recursivitate a problemei rezolvate, de exemplu metoda implementată în funcţia ***maximIterativ*** următoare foloseşte în mod recurent faptul că maximul mulţimii $\{a_i, a_{i+1}, \ldots a_k\}$ este egal cu cel mai mare dintre a_k şi maximul submulţimii $\{a_i, a_{i+1}, \ldots a_{k-1}\}$.

Să studiem următoarea variantă recursivă a funcţiei ***maximIterativ*** :

```
#include<iostream>
#include <windows.h>
using namespace std;

int maxim(int a[], int i, int k){
        if(k<=i) return a[i];
        int max=maxim(a,i,k-1);
        return max>=a[k] ? max : a[k];
}
const int n=5000;//Depasire stiva!
int a[n]={1,2,300,4,50,60,7,8,9,100};
int main(void){
                    // bloc try-except special
                    // pentru a prinde erorile grave
                    // blocul standard C++ "try-catch" nu functioneaza aici
        __try{      // (Microsoft specific)
                cout<<maxim(a,0,n-1)<<endl;
        }
        __except (EXCEPTION_EXECUTE_HANDLER){
```

```
            if ( GetExceptionCode()==EXCEPTION_STACK_OVERFLOW )
                  cout << "Depasire stiva!"<< endl;
            else
                  cout << "Eroare la executie!"<< endl;
      }
      return 0;
}
```

Programul funcţionează pentru n=4000, dar pentru n=5000 avem depăşire de stivă. Spre deosebire de prima abordare cu funcţii recursive a maximului unui şir, în care dimensiunea problemei se înjumătăţea la fiecare auto-apelare, acum scade doar cu o unitate, deci nivelul maxim de auto-apelare este acum de ordinul lui n, şi după cum se observă 5000 de apeluri nerezolvate şi, prin urmare, stivuite unul peste altul au epuizat stiva, care în mod implicit are mărimea de numai 1 Mb.

Iată o aplicare şi mai dezastroasă a auto-apelării: se cere definirea funcţiei

```
double x(int n)
```

care returnează valoarea termenului x_n al şirului dat de relaţia de recurenţă $x_n=(x_{n-1}+2/x_{n-2})/2$ pentru n=2, 3, ..., cu x_0=1 şi x_1=10 fixaţi.

Programul

```
#include<iostream>
using namespace std;
double x0=1.0;
double x1=10.0;

double x(int n){
      if(n<=0) return x0;
      if(n==1) return x1;
      return (x(n-1)+2.0/x(n-2))/2.0;
}
int main(void){
      int n=30;
      //la n=50 executia se blocheaza
      cout<<x(n)<<endl;
      return 0;
}
```

funcţionează pentru n=30, dar la n=50 rămâne "atârnat în gol" şi trebuie oprit cu Task Manager. Este uşor de văzut că nivelul maxim de auto-apelare este ca şi în exemplul precedent de ordinul lui n, dar acum numărul total de apeluri este de ordinul lui 2^n. Volumul de ocupare a stivei fiind de ordin exponenţial, rezolvarea este inutilizabilă.

Ineficienţa metodei de mai sus provine din faptul că nu păstrează nici un rezultat intermediar. Pentru calculul lui x_4 este calculat x_3 şi x_2, dar la calcularea lui x_5 din x_4 şi x_3, termenul x_3 este calculat din nou. Comparaţi cu următoarea variantă iterativă:

```
#include<iostream>
using namespace std;
```

```
double x0=1.0;
double x1=2.0;

double x(int n){
	if(n==0) return x0;
	double  xTrec, xPrez=x0, xViit=x1;
	for(int i=2; i<=n; i++){
		xTrec=xPrez;
		xPrez=xViit;
		xViit=(xPrez+2.0/xTrec)/2.0;
	}
	return xViit;
}
int main(void){
	int n=300000000;
	cout<<x(n)<<endl;// 1.41421
	return 0;
}
```

Să analizăm acum calculul sumelor de forma $s_n=a_1 + a_2 + ... + a_n$. Metoda iterativă standard se bazează pe relaţia de recurenţă $s_n=s_{n-1}+a_n$, cu $s_0=0$, şi se implementează prin alocarea unei singure variabile iniţializată cu 0 la care adunăm pe rând termenii a_i pe măsură ce îi formăm.

Rezolvarea cu funcţii recursive se bazeză pe aceeaşi relaţie de recurenţă şi arată astfel:

```
#include<iostream>
#include <windows.h>
using namespace std;
double a(int n){ return 1.0/n; }
double  sumaIterativa(int n){
	double suma=0;
	for(int i=1;i<=n;i++) suma+=a(i); //adica: suma=suma+a(i);
	return suma;
}
double  suma(int n){	//metoda recursiva
	if (n==0) return 0;
	return suma(n-1) + a(n);
}
int main(void){
	int n=4700;
	cout<<"n="<<n<<endl;
	cout<<"suma iterativa = "<<sumaIterativa(n)<<endl;
	cout<<"suma recursiva = ";
	__try{
		cout<<suma(n)<<endl;
	}
	__except (EXCEPTION_EXECUTE_HANDLER){
		if ( GetExceptionCode()==EXCEPTION_STACK_OVERFLOW )
			cout << "Depasire stiva!"<< endl;
		else
			cout << "Eroare la executie!"<< endl;
	}
```

```
	return 0;
}

/*REZULTAT
n=4700
suma iterativa = 9.03264
suma recursiva = Depasire stiva!
Press any key to continue . . .*/
```

Dacă analizăm comportamentul stivei la apelul ***suma*(*n*)**, observăm că mai întâi sunt calculaţi şi depuşi pe stivă termenii a_n, a_{n-1}, ... a_0, în acestă ordine, care apoi sunt adunaţi în ordine inversă şi rezultatul este returnat din aproape în aproape pe măsură ce se închid cele n apeluri deschise pe stivă.

In concluzie, implementarea recursivităţii prin funcţii definite recursiv este simplă pentru programator dar cel mai adesea este epuizantă pentru maşina de calcul, limitând drastic volumul datelor de intrare ce pot fi prelucrate. Limbajul C/C++ este un limbaj imperativ prevăzut cu instrucţiuni de ciclare foarte eficiente şi rapide, iar în acest limbaj fiecare program are la dispoziţie o singură stivă pentru apelul funcţiilor, stivă de dimensiune relativ mică. Din acest motiv în C/C++ sunt de preferat rezolvările iterative. Totuşi, există situaţii în care, din motive de simplitate, eleganţă şi uneori chiar eficienţă, sunt preferate funcţiile recursive. Iată un astfel de exemplu, la metoda înjumătăţirii intervalului pentru aflarea zerourilor unei funcţii continue:

```
#include<iostream>
#include<math.h>
using namespace std;
const double eps=0.1e-12;

double f(double x){
	return x*x-2;
}

double cauta(double a, double b){
	double med=(a+b)/2;
	if(abs(f(med))<eps || abs(a-b)<eps)return med;
	return f(a)*f(med)<=0 ? cauta(a,med) : cauta(med,b);
}
int main(void){
	cout.precision(12);
	cout<<cauta(0,10)<<endl;
	cout<<sqrt(2.0)<<endl;
	return 0;
}
/*REZULTAT
1.41421356237
1.41421356237
Press any key to continue . . .*/
```

In final un exerciţiu: încercaţi să calculaţi suma produselor termenilor unui şir (a_n)

$s_n = a_1 + a_1 a_2 + \ldots + a_1 a_2 \ldots a_n$

pe baza recurenţelor $s_n = s_{n-1} + p_n$ şi $p_n = p_{n-1} a_n$, cu $s_0 = 0$ şi $p_0 = 1$, abordând mai întâi o metodă recursivă şi apoi una nerecursivă, şi comparaţi eficienţa şi aria de aplicare a celor două rezolvări.