”Nimic nu costă mai scump ca neştiinţa”

Gr. C. Moisil

CURS 11 -RECURSIVITATE

Recursivitatea este una din noţiunile fundamentale ale informaticii. Utilizarea frecventă a recursivităţii s-a făcut după anii '80. Recursivitatea, folosită cu multă eficienţă în matematică, s-a impus în programare odată cu apariţia unor limbaje de nivel înalt, ce permit scrierea de module ce se autoapelează. Astfel PASCAL, LISP, ADA, ALGOL, C sunt limbaje recursive, spre deosebire de FORTRAN, BASIC, COBOL care sunt nerecursive.

Recursivitatea este un mecanism general de elaborare a programelor. Ea a apărut din necesităţi practice (transcrierea directă a formulelor matematice recursive) şi reprezintă acel mecanism prin care un subprogram (procedură, funcţie) se autoapelează. Un algoritm recursiv poate avea un echivalent nerecursiv şi invers. Timpul mare de execuţie şi spaţiul ridicat de memorie utilizat de un algoritm recursiv recomandă, atunci când este posibil, utilizarea unui algoritm nerecursiv. Totuşi, în cazul transformării algoritmului recursiv într-unul iterativ, acesta din urmă poate deveni mai complicat şi mai greu de înţeles. De multe ori, soluţia unei probleme poate fi elaborată mult mai uşor, mai clar şi mai simplu de verificat, printr-un algoritm recursiv.

Implementarea recursivităţii are la bază structura de date denumită *stivă*. Stiva este acea formă de organizare a datelor (structură de date) cu proprietatea că operaţiile de introducere şi scoatere a datelor se fac în vârful ei. O stivă funcţionează după principiul LIFO („Last in First Out”, în traducere „Ultimul intrat, primul ieşit”). Atunci când o procedură sau o funcţie se autoapelează se depun în stivă:

- *valorile parametrilor transmişi prin valoare*. Pentru toţi parametrii-valoare se vor crea copii locale apelului curent (în stivă), acestea fiind referite şi asupra lor făcându-se modificările în timpul execuţiei curente a procedurii (funcţiei). Când execuţia procedurii (funcţiei) se termină, copiile sunt extrase din stivă, astfel încât modificările operate asupra parametrilor-valoare nu afectează parametrii efectivi de apel, corespunzători. Zona de memorie aferente copiilor locale se dealocă în momentul revenirii la funcţia apelantă.
- *adresele parametrilor transmişi prin referinţă*. În acest caz nu se crează copii locale, ci operarea se face direct asupra spaţiului de memorie afectat parametrilor efectivi, de apel.
- *valorile tuturor variabilelor locale* (declarate la nivelul procedurii sau funcţiei). Pentru fiecare apel recursiv al unei proceduri (funcţii) se crează copii locale ale tutoror parametrilor transmişi prin valoare si variabilelor locale, ceea ce duce la risipă de memorie.

Din punct de vedere al modului în care se realizează autoapelul, există două tipuri de recursivitate: *directă* şi *indirectă*.

În cazul *recursivităţii directe* procedura (sau funcţia) se autoapelează (în corpul său). Exemplul clasic este definirea funcţiei factorial:

$n!=(n-1)!\cdot n$, $0!=1$, $n \in N$.

Calculul valorii 3! decurge astfel:

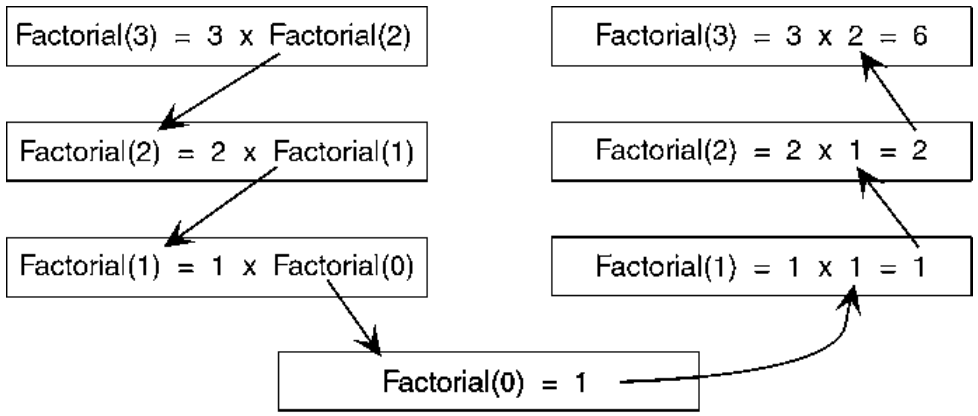

Fig. 8.1 Apelurile recursive pentru 3!

Recursivitatea indirectă are loc atunci când o procedură (funcţie) apelează o altă procedură (funcţie), care la rândul ei o apelează pe ea. Un astfel de exemplu ar fi următorul:

Se consideră două valori reale, pozitive a_0,b_0 şi n un număr natural. Definim şirul:

$$a_n=(a_{n-1}+b_{n-1})/2 \qquad b_n=a_{n-1}b_{n-1}$$

Orice algoritm recursiv are o condiţie de oprire pusă de programator, în caz contrar se va umple stiva (datorită propagării la nesfârşit a autoapelului) şi aplicaţia va genera eroare (Stack Overflow –Depăşire de stivă). De asemenea, în cazul unui număr mare de autoapelări, există posibilitatea ca să se umple stiva, caz în care programul se va termina cu aceeaşi eroare.

O funcţie recursivă trebuie astfel scrisă încât să respecte regulile:

- funcţia trebuie să poată fi executată cel puţin o dată fără a se autoapela
- funcţia recursivă se va autoapela într-un mod în care se tinde spre atingerea situaţiei de execuţie fără autoapel.

Exemplul 1. Calculul valorii n !.

```
#include<stdio.h>
#include<conio.h>
void main(void)
{ int n ;
```

```
int fact(int) ;
clrscr() ;
scanf("%d",&n) ;
printf("%d !=%d",n,fact(n));
getch();
}
int fact(int n)
{ if ( !n) return 1 ;
else return (n*fact(n-1));
}
```

Pentru cazul n=5, apelurile recursive se desfăşoară astfel:

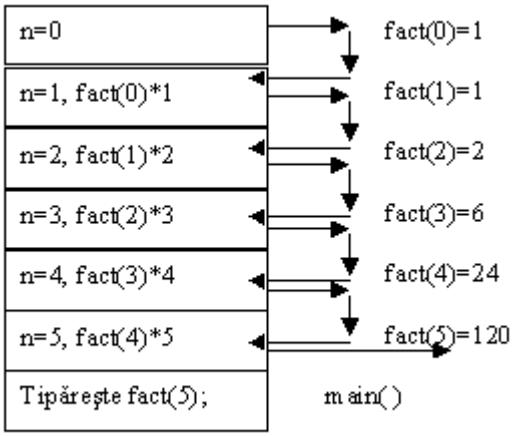

Fig. 8.2. Apelurile recursive pentru 5!

Exemplul 2. Algoritmul lui Euclid recursiv

```
#include<stdio.h>
#include<conio.h>
void main(void)
{ int m,n;
int cmmdc(int,int) ;
clrscr() ;
scanf("%d %d",&m,&n);
printf("cmmdc dintre %d si %d este %d",m,n,cmmdc(m,n));
getch();
}
int cmmdc(int m,int n)
{ if (n= =0) return (m) ;
else return cmmdc(n,m%n);
}
```

Exemplul 3. Se citesc coeficienţii unui şir de numere întregi. Să se calculeze suma elementelor acestui şir folosind o funcţie recursivă.

Definim suma celor n termeni din şir astfel:

$suma(n)=a_1+a_2+ ...+a_{n-1}+a_n$

$suma(n)=suma(n-1)+a_n$

$suma(1)=a_1$

```
#include<stdio.h>
#include<conio.h>
void main(void)
{ int i,n,a[30];
  int suma(int, int *);       //declarăm prototipul funcţiei recursive suma
  clrscr();
  printf("Introd dimensiunea sirului:");
  scanf("%d",&n);
  printf("Introd elementele sirului:\n");
  for(i=0;i<n;i++)
    { printf("a[%d]=");
      scanf("%d",a+i);
    }
  printf("Suma elem este %d\n",suma(n-1,a));
  getch();
}

int suma(int n, int *a)
{ if (n= =0) return(a[0]);
  else return(*(a+n)+suma(n-1,a));
}
```

Exemplul 4: Să se genereze primele n numere din şirul lui Fibonacci. Acesta se defineşte recurent astfel:

Fib(0)=Fib(1)=1;

Fib(n)=Fib(n-1)+Fib(n-2), pentru n>=2.

Fibonacci (născut în 1175) a fost unul din marii matematicieni ai Evului Mediu. Şirul 1, 1, 2, 3, 5, 8, 13, .. a fost introdus de către Fibonacci în anul 1202, atunci matematicianul fiind sub numele de *Leonardo Pisano* (Leonard din Pisa). Mai târziu matematicianul însuşi şi-a spus *Leonardus filius Bonacii Pisanus* (Leonard, fiul lui Bonaccio Pisanul). În secolul XIV şirul prezentat mai sus a fost denumit şirul lui Fibonacci prin contracţia cuvintelor *filius Bonacii*. Ideea acestui şir i-a venit în urma unei probleme propuse la un concurs de matematic, problemă care a fost formulată astfel:

“Plecând de la o singură pereche de iepuri şi ştiind că fiecare pereche de iepuri produce în fiecare lună o nouă pereche de iepuri, care devine “productivă” la vârsta de 1 lună, calculaţi câte perechi de iepuri vor fi după n luni. Se consideră că iepurii nu mor în decursul respectivei perioade de n luni”.

Prima pereche de iepuri o considerăm $F_1=1$ care după o lună dă naştere unei noi perechi de iepuraşi deci $F_2=2$. Notăm cu F_n numărul de perechi de iepuri după n luni. Numărul de perechi de iepuri după n+1 luni, notat F_{n+1}, va fi F_n (deoarece iepurii nu mor niciodată), la care se adaugă iepurii nou-născuţi. Iepurasii se nasc doar din perechi de iepuri care au cel puţin o lună, deci vor fi F_{n-1} perechi de iepuri nou-născuţi. Se obţine astfel relaţia de recurenţă:

$F_{n+1} = F_n + F_{n-1}$ iar aceasta este chiar relaţia de recurenţă care defineşte elementele şirului lui Fibonacci.

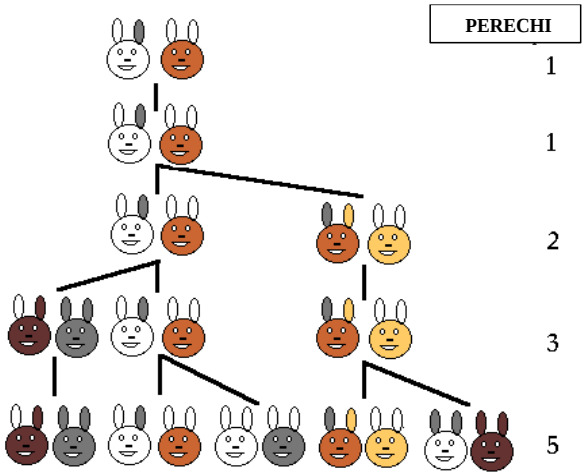

Fig. 8.3. Problema înmulţirii iepurilor

Această problemă a fost punctul de plecare pentru şirul lui Fibonacci, ulterior descoperindu-se alte aplicaţii, deosebit de importante şi interesante. Astfel, secţiunea de aur este legată de şirul lui Fibonacci iar alături de alte numere raţionale cum ar fi π şi ε, se regăseşte sistematic în Univers. Acest număr a fost cunoscut şi studiat încă din antichitate, sculptura şi arhitectura Greciei antice respectând cu rigurozitate secţiunea de aur, aceasta fiind considerată o măsură a armoniei şi echilibrului.

Exemplificăm apelurile recursive ale generării numerelor lui Fibonacci pentru cazul n=5:

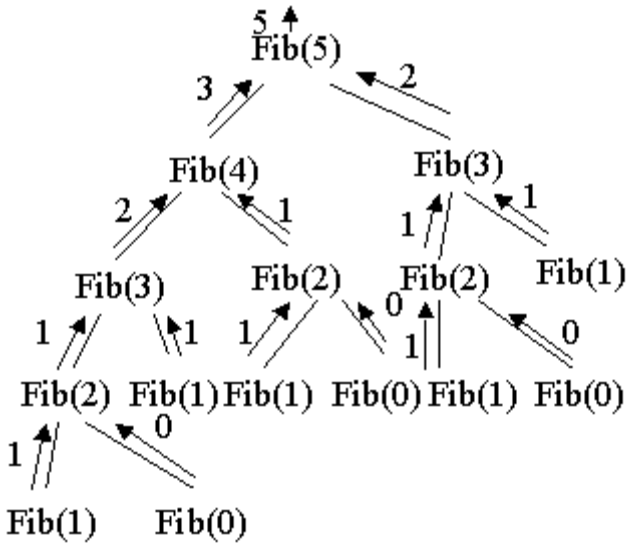

Fig. 8.4. Apelurile recursive pentru calculul Fibonacci(5)

Programul C:

```
#include<stdio.h>
#include<conio.h>
void main(void)
{ int i,n;
  int fib(int);          //declarăm prototipul funcţiei recursive ce calculează fiecare
                         //termen al şirului lui Fibonacci
  clrscr();
  scanf("%d",&n);
  for(i=0;i<n;i++) printf("fib[%d]=%d\n",i,fib(i));
  getch();
}
int fib(int n)
{ if((n= =0)||(n= =1)) return(1);
  else return(fib(n-1)+fib(n-2));
}
```

Observaţie: utilizarea unei astfel de funcţii recursive în acest caz este ineficientă. Un algoritm nerecursiv este mult mai eficient, putându-se genera elementele acestui şir pe baza relaţiei de recurenţă. Algoritmul recursiv prezentat, pentru fiecare element al şirului, face descompunerile pentru a ajunge până la primele două elemente, regenerând astfel repetat fiecare element al şirului.