Recursivitate

1. Introducere
2. Exemple și implementări. Analiza complexității.

1. Introducere

Recursivitatea este una dintre noţiunile fundamentale ale informaticii şi constă în posibilitatea unui subprogram de a se autoapela o dată sau de mai multe ori.

Recursivitatea a apărut din necesitatea de a transcrie direct formule matematice recursive. În timp acest mecanism a fost extins şi pentru alţi algoritmi.

În cazul autoapelării unui algoritm (unei funcţii), în ceea ce priveşte transmiterea parametrilor se procedează ca la orice apel de funcţie (subprogram). Pentru memorarea parametrilor se foloseţte o zonă de memorie numită stivă iar memorarea parametrilor se realizează în ordinea in care aceştia apar în antet. Parametri pot fi transmişi prin valoare sau prin referinţă (caz în care de fapt se transmite o adresă). În cadrul subprogramului, parametrii transmişi şi memoraţi în stivă sunt variabile.

Funcţiile recursive tipice corespund unei relaţii de recurenţă de tipul f(n)=rec(f(n-1)), n fiind parametrul după care se face recursivitatea) funcţia poate avea mai mulţi parametri). La fiecare nou apel valoarea parametrului n se decrementează, până când n ajunge 1 sau 0, iar valoarea f(1) sau f(0) se poate calcula direct.

Orice subprogram recursiv poate fi rescris si nerecursiv, iterativ, prin repetarea explicitã a operatiilor executate la fiecare apel recursiv. O funcţie recursivã realizează repetarea unor operatii fãrã a folosi instructiuni de ciclare.

In unele cazuri, ca de exemplu operaţiile cu arborilor binari, utilizarea funcţiilor recursive este mai naturală şi oferă un cod sursă mai simplu şi mai elegant. In schimb, pentru implementarea operaţiilor specifice listelor, pentru anumite calcule sau pentru operaţii de căutare este mai simplu şi mai eficient să se utilizeze variante iterative.

Observaţii

- Recursivitatea nu este de neînlocuit! Orice funcţie recursivă se poate transforma într-o structură ciclică.
- În cazul unui număr foarte mare de autoapelări, există posibilitatea ca stiva implicită (folosită de compilator) să se ocupe total, caz în care programul se va termina cu eroare.
- Recursivitatea presupune mai multă memorie.
- Un algoritm recursiv poate fi privit ca fiind ierarhizat pe niveluri (ce corespund nivelurilor din stivă), astfel încât:
  - Ce se execută pe un nivel se execută pe orice nivel.
  - Subprogramul care se autoapelează trebuie să conţină instrucţiunile corespunzătoare unui nivel.

- Funcţiile recursive au cel puţin un argument, a cărui valoare se modifică de la un apel la altul şi care este verificat pentru oprirea procesului recursiv.
- Orice subprogram recursiv trebuie să conţină o instrucţiune "if" (de obicei la început), care să verifice condiţia de oprire a procesului recursiv. In caz contrar, se ajunge la un proces recursiv ce tinde la infinit şi se opreşte numai prin umplerea stivei.

Executia unei functii care implementeaza un algoritm recursiv se realizează astfel:

- se creeaza in stiva de recursie o "inregistrare de activare" in care sunt memorati:
  - parametrii de apel;
  - adresa instructiunii de retur (cu care va continua programul dupa terminarea executiei functiei);
- se rezerva spatiu pentru variabile locale.
- se executa instructiunile functiei care folosesc pentru parametri si variabile locale parametrii memorati in "inregistrarea de activare";
- se scoate din stiva "înregistrarea de activare" (decrementarea vârfului stivei), stiva fiind ordonată; se continuă cu instrucţiunea dată de adresa de retur memorată în "inregistrarea de activare".

Asadar, *variabilele globale (statice)* sunt memorate intr-o zona de memorie fixă, mai exact în segmentele de date. *Variabilele automate (locale)* se memorează in stivă, iar *variabilele dinamice* in "heap"-uri (cu malloc in C şi cu new in C++).

Consumul de memorie al unui algoritm recursiv este proporţional cu numărul de apeluri recursive ce se fac. Variabilele recursive consumă mai multa memorie decât cele iterative. La prelucrarea unei liste, dacă primul element nu este vid, se prelucrează acesta, urmând apoi ca restul listei sa fie considerat ca o noua lista mai mica, etc.

2. Exemple și implementări. Analiza complexității.

Pentru exemplele de mai jos, urmăriți pas cu pas execuția programul si reprezentati grafic succesiunea apelurilor (pentru 4 -5 pasi), modificarea valorilor parametrilor și conținutul stivei.

E1. Să se implementeze și testeze funcțiile recursive pentru

a) crearea unei liste liniare simplu înlănțuite (elementele trebuie să apară în listă în ordinea introducerii acestora)
b) afișarea elementelor listei de la ultimul element la primul.

E2. Folosind o funcţie recursivă, să se determine cel mai mare divizor comun al două numere introduse de la tastatură.

```
int cmmdc(int m, int n)
{
	if(!n) return m;
	return cmmdc(n,m%n);
}
```

E3. Să se calculeze suma $S_n = \sum_{i=0}^{n} \frac{x^i}{i!}$

Se utilizează trei funcţii: factorial, putere şi S.

```
int fact(int n)
{
	if (n==0)
		return 1;
	return n*fact(n-1);
}

float putere(float x, int n)
{
	if (n==0)
		return 1;
	return x*putere(x,n-1);
}

float S(float x, int n)
{
	if (n==0)
		return 1;
	return S(x,n-1)+putere(x,n)/fact(n);
}
```

Varianta 1

```
S=1;
for(i=1;i<=n;i++)
	s+=putere(x,i)/fact(i);
```

Varianta 2. Relaţia de recurenţă a termenilor se poate scrie $T_i = \frac{x}{i} T_{i-1}$ iar varianta recursivă va fi

```
float S(float x, int n)
{
	static float T=1;
	if (n==0)
	{
		T=1;
		return 1;
	}
	return S(x,n-1)+(T*=x/n);
}
```

Obs.: variabilele statice îşi păstrează valoarea la ieşirea din funcţie, valoare ce se regăseşte la următorul apel. Variabila statică T păstrează rezultatul (calculele efectuate) anterior.

E4. Căutarea binară (varianta recursivă)

```
Int binSearch(int a[],int b,int
st,int dr)
{
  int m;
  assert(st<=dr);
  if(st==dr)
     return (b==a[dr]) ? dr: -1 //*1
  else {                          //*2
      m=(st+dr)/2;
      if(b<=a[m])
            return
            binSearch(a,b,st,m);
      else
            return
            binSearch(a,b,m+1,dr);
      }
 }
```

a- timpul pentru secvenţa *1
b- timpul pentru secvenţa *2

T(n) satisface relaţia de recurenţă

$$T(n) = \begin{cases} T(n/2) + a & \text{daca n} > 1 \\ b & \text{daca n} = 1 \end{cases}$$

se demonstrează prin inducţie că dacă T(n) satisface relaţia de recurenţă de mai sus atunci:

$$T(n) \le a * \log(n) + b \ \Rightarrow \text{O(log n)}$$

E5. Şirul lui Fibonacci

Folosind o metodă recursivă, să se determine primii *n* termeni ai şirului lui Fibonacci, F(n) pe baza relatiei de recurentã: F(n) = F(n-2)+F(n-1) si primele 2 numere din şir F(0)=F(1)=1.

```
int Fibonacci(int n)
{
        if(n<2)
        return n;
        else
        return Fibonacci(n-1)+Fibonacci(n-2);
}
```

Realizati o implementare optimizata a algoritmului recursiv in care odata calculata valoarea F(k) ea este stocata intr-un tablou si accesata in urmatorul apel recursiv. Comparati timpul de executie intre cele doua variante recursive pentru diferite valori ale lui n > 30.
Rezolvaţi problema şi nerecursiv.

E6. Numere prime

Folosind o metodă recursivă, să se stabilească dacă un număr întreg citit de la tastatură este număr prim sau nu.

```
bool isPrime(int p, int i) // p este variabila citită de la tastatură
{
        if(i= =p) return 1;
        if(p%i= =0) return 0;
        return isPrime(p,i+1);
}
```

Care este semnificaţia argumentului “i” şi ce valoare va avea la apelul funcţiei din main?

E7. Permutări

O categorie de probleme cu soluţii recursive sunt cele care conţin un apel recursiv într-un ciclu, deci un numãr variabil (de obicei mare) de apeluri recursive. Aici se încadrează algoritmii de tip “backtracking”.
Să se scrie o funcţie care generează şi afişează toate permutările posibile ale primelor n numere naturale.

E8. Problema turnurilor din Hanoi

Se dau ***n*** discuri: a_1, a_2, ... , a_n de dimensiuni diferite, cu $d_1 < d_2 < ... < d_n$, d_i - fiind diametrul discului a_i. Discurile respective sunt stivuite pe o tija.
Se cere sa se deplaseze aceasta stiva pe o alta tija, folosind ca manevra o tija auxiliara, respectandu-se conditia: Un disc nu poate fi plasat decat peste un disc cu diametrul mai mare decat al acestuia.
Problema P(n) a deplasarii a ***n*** discuri, se rezolva prin deplasari succesive ale discurilor de pe o tija pe alta. Deplasarea de pe o tija pe alta este echivalenta cu deplasarea a ***n-1*** discuri de pe tija intiala (***t****i*) pe tija de manevra (***t****m*), apoi plasarea celui mai mare disc pe tija finala (***t****f*), pentru ca la sfarsit sa se aduca de pe tija de manevra pe tija finala cele ***n-1*** discuri.

PseudoCod:

```
Hanoi(n, ti, tf, tm)
{
     if(n=1) then muta (ti, tf) //deplaseaza discul superior
                                //de pe ti pe tf
     else  Hanoi(n-1, ti, tm, tf)
             muta(ti, tf)
             Hanoi(n-1, tm, tf, ti)
}
```

Pentru o problema P(1), timpul T(1) = 1 , pentru o mutare.

Pentru P(n) , timpul:

$$T(n)=2*T(n-1)+1 \qquad (1)$$

Dorim sa aflam ordinul de complexitate a lui T(n). Asociem relaţiei (1) ecuaţia caracteristică:

$$x=2x+1 \qquad (2)$$

Rezulta $x_0=-1$ şi $T(n)- x_0=2*T(n-1)- x_0$

Dacă se noteaza $f(n)=T(n)- x_0$ se obţine

$$f(n)=2*f(n-1= \qquad (3)$$

Aplicând (2) rezultă $f(n)=2*f(n-1)=2*2*f(n-2)=...=2^{n-1} * f(1)$.

Inlocuind f(n) cu T(n)+1 şi f(1) cu 2 se obţine $T(n)=2^n-1$.

Prin urmare, ordinul de complexitate este $\boldsymbol{O(2^n)}$, adică o complexitate exponenţială.

Soluţia nerecursivă se bazează pe analiza stărilor (mutărilor) discurilor: se repetă de $(2^n–1)$ ori funcţia “muta” recalculând la fiecare pas numărul tijei sursă şi al tijei destinaţie (numărul discului nu contează deoarece nu se poate muta decât discul din vârful stivei de pe tija sursă):

```
int n=4, i; // n = numar de discuri
for (i=1; i < (1 << n); i++) // numar de mutari= 1<<n = 2n
printf("Muta de pe %d pe %d \n", (i&i-1)%3,((i|i-1)+1)%3);
```

E9. Problema labirintului

Se dă o matrice cu elemente 0 şi 1 reprezentând "harta" unui labirint (0-spatiu liber; ......1-zid). Se cere să se determine un drum între o poziţie iniţială şi o poziţie finală date. Un drum este format din poziţii libere învecinate pe verticală sau orizontală (pe aceeaşi linie sau aceeaşi coloană).

```
  123456789
1 ±±±±±±±±±     ± - zid
2 ±     ± ±
3 ±±±I±±± ±     I - pozitia initiala
4 ±     ±F±     F- pozitia finala
5 ±±± ±   ±
6 ±±±±±±±±±
```

Soluţia acestei probleme este un vector de poziţii în matrice. În cazul nostru soluţia este:

((3,4), (4,4), (4,5), (4,6), (5,6), (5,7), (5,8), (4,8))

Fiecare poziţie din drum este rezultatul unei alegeri între mai multe variante de inaintare. De exemplu din pozitia iniţială (3,4) se alege între variantele (2,4) şi (4,4).
Metoda de rezolvare se bazează pe încercări. Dintr-o anumită poziţie se inaintează atât cât este posibil, pe rând în toate direcţiile libere. Atunci când o pistă se "inchide" fără să se ajungă în poziţia finală, se revine urmărind drumul parcurs pentru a încerca alte piste.

Matricea labirint va fi o matrice de caractere:

L[i,j]='±' =>zid
L[i,j]=' ' =>pozitie libera.

Pentru a putea indexa variantele de înaintare dintr-o poziţie se foloseste vectorul "***dir***" cu incremenţii care trebuie aplicaţi indecşilor de linie şi coloană pentru înaintare într-o direcţie.

Laborator de Structuri de Date si Algoritmi – Lucrarea nr. 8

Direcţia:	**1**	**2**	**3**	**4**
Increment pt. *x*	0	0	-1	1
Increment pt. *y*	-1	1	0	0

Se poate aplica următoarea strategie recursivă: căutarea drumului pâna la poziţia finală, pornind dintr-o poziţie dată, va însemna căutarea drumului pornind pe rând din toate poziţiile învecinate din care acestă căutare nu a fost făcută anterior.
Să se scrie o variantă de rezolvare a problemei labirintului folosind o funcţie recursivă. Pentru iniţializarea matricii L (labirintul) veţi folosi o funcţie citeste_labirint care citeşte matricea labirint dintr-un fişier text, al cărui nume îl primeşte drept parametru:

Conţinutul fişierului:

```
6 9
±±±±±±±±±
±     ± ±
±±±I±±± ±
±     ±F±
±±± ±   ±
±±±±±±±±±
```

E10. Problema Jeep-urilor

Un jeep are un rezervor care poate contine ***r*** litri de benzina, consuma ***c*** litri de benzina la 100 km. Initial jeep-ul este parcat intr-o oaza din Sahara si are rezervorul gol. In oaza exista ***n*** canistre, fiecare continind ***r*** litri de benzina. Presupunem ca jeep-ul poate transporta la un moment dat o singura canistra plus benzina din rezervor. Rezervorul poate fi umplut numai atunci cand este golit complet.

Sa se determine cea mai mare distanta fata de oaza (pozitia initiala) care poate fi parcursa de jeep utilizand toate cele ***n*** canistre.

NOTARE

E1: 2p
E2: 0.5p
E3: 0.5p
E4: 0.5p
E5: 2p
E6: 0.5p
E7: 2p
E8: 0.5p
E9: 3p
E10: 3p