Programarea Calculatoarelor

Cursul 8: Recursivitate

Razvan Itu
Universitatea Tehnică din Cluj-Napoca
Departamentul Calculatoare

Recursivitatea

- Tehnică de programare puternică și de uz general în care **o funcție se auto-apelează**
- Stă la baza inducției matematice
  - O tehnică puternică care demonstrează multe teoreme matematice
- Ajută la descompunerea unei probleme de dimensiuni mari în subprobleme – tehnica ***Divide et Impera***
  - Rezolvarea subproblemelor și combinarea rezultatelor duc la soluția problemei
- Ajută la determinarea tuturor soluțiilor posibile ale unei probleme – tehnica ***Backtracking***
- Permite scrierea programelor într-o manieră mult mai compactă și mai clară

Recursivitatea

- O funcție recursivă se poate **auto-apela**
  - **Direct** – funcția conține un apel al ei însăși
  - **Indirect** – funcția conține un apel al altei funcții, iar aceasta din urmă conține la rândul ei un apel către prima funcție
- La fiecare apel recursiv **se stochează la locații separate pe stivă**
  - Valorile variabilelor locale automate pentru apelul respectiv
  - Valorile parametrilor formali pentru apelul respectiv
  - Adresa de întoarcere din apelul respectiv
- De-a lungul apelurilor recursive **sunt stocate pe *heap*** și își păstrează locația
  - Variabilele statice
  - Variabilele globale

Terminarea recursivității

- O funcție recursivă trebuie să aibă în construcția ei o **condiție de terminare**
  - Altfel apelul recursiv poate să ducă la o buclă infinită (asemănătoare structurilor repetitive în care condiția de continuare este întotdeauna adevărată și care iterează la nesfârșit)!
- **Adâncimea recursivității** trebuie să nu fie una foarte mare
  - La fiecare apel recursiv se memorează pe stivă
    - Parametrii funcției
    - Variabilele locale automate
    - Adresa de revenire din apelul funcției
  - Aceasta variază în funcție de numărul de octeți conținuți în parametrii și variabilele locale automate
  - În cazul depășirii dimensiunii maxime a stivei, programul se termină subit în urma unei erori – ***stack overflow***

Recursivitate liniară

- Este cea mai simplă formă de recursivitate
- Apare atunci când o acțiune are o structură simplă repetitivă care constă într-un proces urmat de repetarea acțiunii respective
- Poate fi transformată în iterații nerecursive prin plasarea procesului într-o structură repetitivă (ex.: *for*, *while*)
  - Condiția de terminare a recursivității este utilizată pentru a termina iterarea în cadrul structurii repetitive
- Exemple
  - Calculul factorialului unei valori
  - Inversarea unui șir de valori
  - Calculul minimului unui șir de valori

Exemplu – Calculul lui *n*!

- Definiția recurentă a lui *n* factorial

$$fact(n) = \begin{cases} 1 & , \text{dacă} \quad n = 0 \\ n \times fact(n-1) & , \text{dacă} \quad n > 0 \end{cases}$$

- Se identifică cazul special (*n*=0) în care funcția *fact* nu se apelează recursiv
- În cazul apelului recursiv se observă că parametrul *n* scade cu 1 la fiecare apel => se îndreaptă către cazul special (condiția de terminare) dacă *n* este un număr natural

Exemplu – Calculul lui *n*!

```
#include <stdio.h>
#include <stdlib.h>

double factorial(int n){
    printf("Apel recursiv cu n=%d\n", n);
    double y;
    if (n<=0)
        y = 1;
    else
        y = factorial(n-1) * n; /* &2=adresa de intoarcere dupa apelul
                                   lui factorial(n-1) */
    printf("Iesirea din apelul recursiv cu n=%d\n", n);
    return y;
}


int main() {
    int n;
    printf("n=");
    scanf("%d", &n);
    printf("%d!=%g\n", n, factorial(n));
      /* &1=adresa de intoarcere dupa evaluarea lui factorial(n) */
    return 0;
}
```

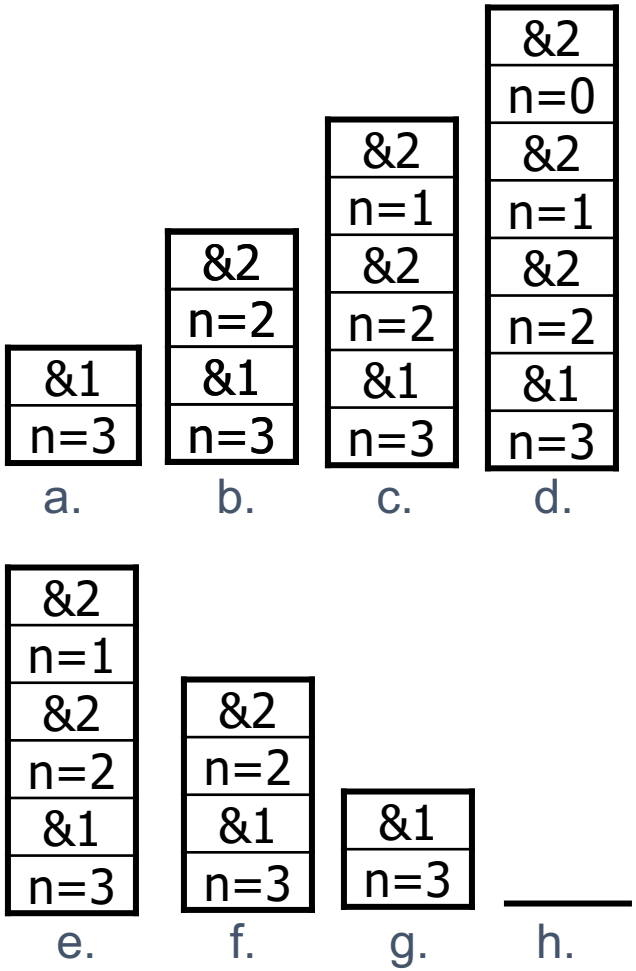
Exemplu – Calculul lui *n*!

Intrare și rezultate afișate de program:

```
n=3
Apel recursiv cu n=3
Apel recursiv cu n=2
Apel recursiv cu n=1
Apel recursiv cu n=0
Iesirea din apelul recursiv cu n=0
Iesirea din apelul recursiv cu n=1
Iesirea din apelul recursiv cu n=2
Iesirea din apelul recursiv cu n=3
3!=6
```

Exemplu – Calculul lui *n*!

- Evoluția stivei de apeluri:
  a. Imediat după intrarea în apelul lui factorial(3)
  b. Imediat după intrarea în apelul recursiv al lui factorial(2)
  c. Imediat după intrarea în apelul recursiv al lui factorial(1)
  d. Imediat după intrarea în apelul recursiv al lui factorial(0)
  e. Imediat după ieșirea din apelul recursiv (pentru n=0)
  f. Imediat după ieșirea din apelul recursiv (pentru n=1)
  g. Imediat după ieșirea din apelul recursiv (pentru n=2)
  h. Imediat după ieșirea din apelul recursiv (pentru n=3) în funcția main()



Exemplu – Inversarea unui șir de întregi

```
#include <stdio.h>
#include <stdlib.h>

void inverseaza(int *a, int stg, int dr)
{
    if (stg<dr)
    {
        int t=a[stg];
        a[stg]=a[dr];
        a[dr]=t;
        inverseaza(a,stg+1,dr-1);
    }
}


int main()
{
    int n=5;
    int a[]={3,6,4,1,2};
    inverseaza(a,0,n-1);
    for (int i=0; i<n; i++)
        printf("%d ",a[i]); // 2 1 4 6 3
    return 0;
}
```

Exemplu – Valoarea minimă dintr-un șir de întregi

```c
#include <stdio.h>
#include <stdlib.h>

int minim(int *a, int stg, int dr) {
    if (stg==dr)
        return a[stg];
    else
    {
        int m = minim(a,stg+1,dr);
        if (a[stg]<m)
            return a[stg];
        else
            return m;
    }
}


int main() {
    int n=5;
    int a[]={3,6,4,1,2};
    printf("%d",minim(a,0,n-1)); // 1
    return 0;
}
```

Recursivitate indirectă – Determinarea parității unui număr natural

```
#include <stdio.h>
#include <stdlib.h>

int este_impar(int);

int este_par(int n) {
    if (n==0)
        return 1;
    return(este_impar(n-1));
}

int este_impar(int n) {
    return (!este_par(n));
}

int main() {
    printf("%d %d", este_par(20), este_par(35)); //1 0
    return 0;
}
```

Recursivitate neliniară

- Funcția se auto-apelează de mai multe ori de-a lungul firului execuției ei
- Exemple
  - Șirul lui Fibonacci

$$fib(n) = \begin{cases} 0, \text{ dacă } n = 0 \\ 1, \text{ dacă } n = 1 \\ fib(n-1) + fib(n-2), \text{ dacă } n \geq 2 \end{cases}$$

  - Partițiile unui număr natural pozitiv
    - Exemplu pentru numărul 4:

      {1, 1, 1, 1}, {1, 1, 2}, {1, 2, 1},{1, 3}, {2, 1, 1}, {2, 2}, {3, 1}, {4}

Exemplu – Șirul lui Fibonacci

```c
#include <stdio.h>
#include <stdlib.h>

int fib(int n)
{
    if (n==0)
        return 0;
    if (n==1)
        return 1;
    return fib(n-1)+fib(n-2);
}

int main()
{
    printf("%d\n",fib(20)); //6765
    printf("%d\n",fib(42)); //Timp de executie ridicat!!!
    return 0;
}
```

Exemplu – Partițiile unui număr natural pozitiv

```
#include <stdio.h>
#include <stdlib.h>

void partitii(int i, int *x, int n) {
    for (int j=1;j<=n;j++) {
        x[i]=j;
        if (j<n)
            partitii(i+1,x,n-j);
        else {
            for (int k=0;k<=i;k++)
                printf("%d ",x[k]);
            printf("\n");
        }
    }
}
int main() {
    int n=4;
    int x[n];
    partitii(0,x,n);
    return 0;
}
```

Buclă infinită !

- Nu există condiție de terminare:

```
void afisare_cifre(int n) {
   afisare_cifre(n / 10);
   printf("%d\n", (n % 10));
}
```

- Există condiție de terminare, dar apelul recursiv nu modifică valoarea parametrului:

```
void afisare_cifre(int n) {
    if (n < 10)
       printf("%d\n", n);
    else
    {
       afisare_cifre(n);
       printf("%d\n", (n % 10));
    }
}
```

Depășirea stivei !

```c
#include <stdio.h>
#include <stdlib.h>

void f(int n) {
    double x[10000]; // Tablou alocat pe stiva la fiecare apel
    if (n==0)
        return;
    else
    {
        f(n-1);
        printf("%d ",n);
    }
}

int main() {
    f(10); // 1 2 3 4 5 6 7 8 9 10
    f(100); // Depasire de stiva !!!
    return 0;
}
```

Recursivitatea – important !

- Pentru a evita ciclul infinit scrieți un caz special în care funcția să nu se apeleze recursiv
- Atenție la funcțiile care se pot apela prin recursivitate indirectă
- Recursivitatea trebuie gândită bine – altfel se pot scrie programe total ineficiente
- Recursivitatea este o alternativă pentru structurile repetitive *for* si *while* utilizate în calcule ce necesită iterații multiple
- Funcțiile recursive sunt foarte utile când structurile de date (ex. arbori binari de căutare) sau paradigma de programare (ex. *divide et impera*) sunt recursive prin definiție

Recursivitatea – important !

- O funcție recursivă ar trebui să aibă următoarele trei proprietăți
  - Nu se auto-apelează la infinit
    - Se va identifica cazul special (sau condiția de terminare) în care funcția nu se auto-apelează
    - Se va asigura faptul că succesiunea de apeluri recursive va conduce către acest caz special
    - Pot să existe mai multe astfel de cazuri speciale
  - Fiecare caz special de oprire
    - Trebuie să returneze valoarea corectă pentru acel caz (în cazul funcțiilor care returnează o valoare)
    - Trebuie să execute acțiunile corecte pentru acel caz (în cazul funcțiilor care nu returnează nicio valoare)
  - Pentru cazurile care necesită apeluri recursive, dacă apelul recursiv se execută corect (sau returnează o valoare corectă) atunci acțiunea finală este corectă (sau calculul final este corect)