1. RECURSIVITATE

1.1. Conceptul de recursivitate

În informatică şi în matematică, recursivitatea este un concept fundamental.

Spunem că o noţiune este definită **recursiv** dacă în cadrul definiţiei intervine însăşi noţiunea care se defineşte.

De exemplu – *un **descendent** al unei persoane este un copil al său sau un **descendent** al unui copil al său.* În acest caz, pentru a defini noţiunea de „*descendent*" am folosit noţiunea însăşi.

În matematică apar frecvent relaţii recursive, denumite şi „*relaţii de recurenţă*". De exemplu, să considerăm şirul *Fibonacci*: `0`, `1`, `1`, `2`, `3`, `5`, `8`, `13`, `21`, `...`

Observăm că în acest şir primii doi termeni sunt `0` şi `1`, iar în rest, fiecare termen se obţine însumând cei doi termeni care îl preced. Prin urmare, o definiţie recursivă pentru şirul *Fibonacci* este:

$$fib : N \to N$$

$$fib(n) = \begin{cases} n, & \text{dacă } n \le 1 \\ fib(n-1) + fib(n-2), & \text{dacă } n > 1 \end{cases}$$

Apare întrebarea: orice descriere de acest tip este corectă? Să considerăm următorul exemplu:

$$f : N \to N$$

$$f(n) = \begin{cases} 1, & \text{dacă } n = 0 \\ 1 + f(n+1), & \text{dacă } n > 0 \end{cases}$$

Observaţi că `f(0)=1`, iar dacă `n>1`, pentru a calcula `f(n)` trebuie să evaluăm mai întâi `f(n+1)`. Dar pentru a evalua `f(n+1)`, va fi nevoie de `f(n+2)`, deoarece `n>0` implică `n+1>0`, ş.a.m.d. Prin urmare, funcţia `f` nu poate fi evaluată decât pentru `0`.

Regula fundamentală pentru ca recursia să fie definită corect este:
1. trebuie să existe cazuri elementare, care se pot rezolva direct;
2. pentru cazurile care nu se rezolvă direct, recursia trebuie să progreseze către un caz elementar.

Exemplul precedent încalcă a doua parte a acestei reguli.

O greşeală care apare frecvent este însă şi încălcarea primei reguli. De exemplu, am solicitat unui elev să descrie o funcţie recursivă care să calculeze suma cifrelor unui număr natural. A abordat problema excelent: suma cifrelor unui număr natural

n se obține adunând ultima cifră a numărului n (n%10) cu suma cifrelor numărului natural care se obține eliminând ultima cifră din n (sum(n / 10)) şi a scris relația următoare:

```
sum: N → N
sum(n) = n%10+sum(n/10);
```

Evident, a uitat cazurile elementare. Ca urmare, recursia nu se opreşte niciodată: pentru a calcula suma cifrelor numărului natural, se elimină succesiv cifrele sale, până când argumentul n devine 0. Din acest moment, se evaluează la infinit sum(0), deoarece 0/10 este permanent 0.

Deci, corect ar fi fost:

$$sum(n) = \begin{cases} 0, & \text{dacă } n = 0 \\ n\%10 + sum(n/10), & \text{dacă } n > 0 \end{cases}$$

1.2. Mecanismul de realizare a recursivității

Recursivitatea se realizează prin intermediul funcțiilor.

O funcție se numeşte ***recursivă*** dacă se autoapelează.

Autoapelarea se poate realiza în două moduri: *direct* (în acest caz în corpul funcției apare explicit un apel recursiv) sau *indirect* (în corpul funcției apare apelul unei alte funcții care la rândul său apelează direct sau indirect funcția respectivă).

Mecanismul care face posibilă recursivitatea derivă din modul de funcționare a funcțiilor. Ca în cazul oricărui apel de funcție şi în cazul funcțiilor recursive se alocă zonă de memorie pe stivă pentru valorile parametrilor, precum şi pentru valorile variabilelor locale. Această zonă de memorie rămâne alocată pe tot parcursul execuției apelului funcției, fiind eliberată la momentul revenirii în funcția apelantă. Stiva nu este gestionată explicit de către programator, ci de către sistem. Pentru a înțelege mai exact să analizăm următoarea problemă.

Inversarea unui cuvânt

Să se scrie o funcție recursivă care citeşte un cuvânt, caracter cu caracter, şi afişează cuvântul în ordine, apoi cuvântul inversat. Marcajul de sfârşit de cuvânt este caracterul spațiu.

Calculul celui mai mare divizor comun al două numere naturale

Date a şi b ∈ **N**, scrieți o funcție recursivă de calcul al celui mai mare divizor comun al numerelor a şi b folosind algoritmul lui *Euclid*.

Funcţia Ackermann

Evaluați funcția *Ackerman* pentru m, n numere naturale date.

Funcția *Ackermann* este: ac : **N** x **N** → **N**,

Numărare

Fie a_1, a_2, ..., a_n un şir de n (0<n<20) numere întregi şi x un număr întreg. Scrieți o funcție recursivă care să determine numărul de apariții ale lui x în şir.

Generare

Fie n ∈ **N***. Să se genereze toate succesiunile de n (n<20) caractere '.' şi '_' (asemănător codurilor *Morse*).

De exemplu, pentru n=2 veți afişa:

```
. .
. _
_ .
_ _
```

Generarea produsului cartezian

Fie n∈ **N*** şi A_1, A_2, ..., A_n n mulţimi cu L_1, L_2, ..., respectiv L_n elemente. Să se genereze elementele produsului cartezian $A_1xA_2x...xA_n$.

Prin definiție produsul cartezian $A_1xA_2x...xA_n$ este mulțimea n-uplelor (e_1, e_2, ..., e_n) cu proprietatea că $e_i \in A_i$, pentru ∀i∈{1, 2, ..., n}.

De exemplu pentru n=3 şi L=(2, 3, 2), elementele produsului cartezian sunt: (1,1,1), (1,1,2), (1,2,1), (1,2,2), (1,3,1), (1,3,2), (2,1,1), (2,1,2), (2,2,1), (2,2,2), (2,3,1), (2,3,2). Observați că produsul cartezian are $L_1*L_2*...*L_n$ elemente.

Generarea permutărilor

Fie n∈ **N***. Scrieți un program recursiv de generare a permutărilor de ordin n.

De exemplu, pentru n=3 programul va genera:

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
```

```
3 2 1
```

Generarea aranjamentelor

Fie $n \in \mathbf{N}^*$ şi $m \in \mathbf{N}$, $m \leq n$. Scrieți un program recursiv care să genereze aranjamentele de n elemente luate câte m.

De exemplu, pentru $n=3$ şi $m=2$, programul va genera:

```
1 2
1 3
2 1
2 3
3 1
3 2
```

Generarea combinărilor

Fie $n \in \mathbf{N}^*$ şi $m \in \mathbf{N}$, $m \leq n$. Scrieți un program recursiv care să genereze combinările de n elemente luate câte m.

De exemplu, pentru $n=4$ şi $m=2$, programul va genera:

```
1 2
1 3
1 4
2 3
2 4
3 4
```