CURS 4
RECURSIVITATE

În general, prin *recursivitate* se înțelege proprietatea unor noțiuni de a se defini prin ele însele. De exemplu, numerele naturale pot fi definite recursiv folosind următoarele două axiome ale lui Peano (https://en.wikipedia.org/wiki/Peano_axioms): "*Zero este un număr natural.*" și "*Succesorul oricărui număr natural este tot un număr natural.*".

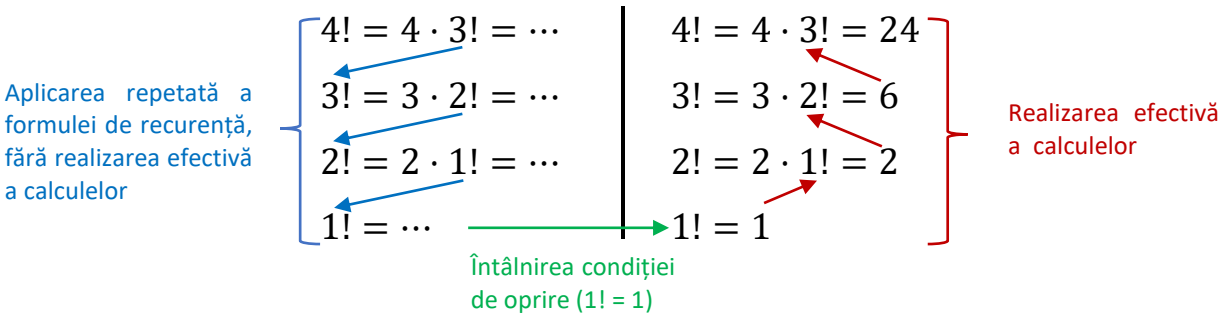
În programare, o *funcție recursivă* este o funcție care se autoapelează, direct sau indirect.

Deoarece recursivitatea indirectă (i.e., o funcție `f` apelează o funcție `g`, iar funcția `g` apelează, la rândul său, funcția `f`) este foarte rar utilizată în programare (de exemplu, pentru a calcula media aritmetico-geometrică: https://en.wikipedia.org/wiki/Arithmetic-geometric_mean), în continuare vom prezenta doar recursivitatea directă.

Un exemplu clasic de funcție recursivă îl reprezintă calculul factorialului unui număr natural n (i.e., $n! = 1 \cdot 2 \cdot \ldots \cdot n$), folosind următoarea relație de recurență:

$$n! = \begin{cases} 1, & \text{dacă } n = 1 \\ n \cdot (n-1)!, & \text{dacă } n \geq 1 \end{cases}$$

Folosind această formulă de recurență, putem calcula valoarea 4! astfel:



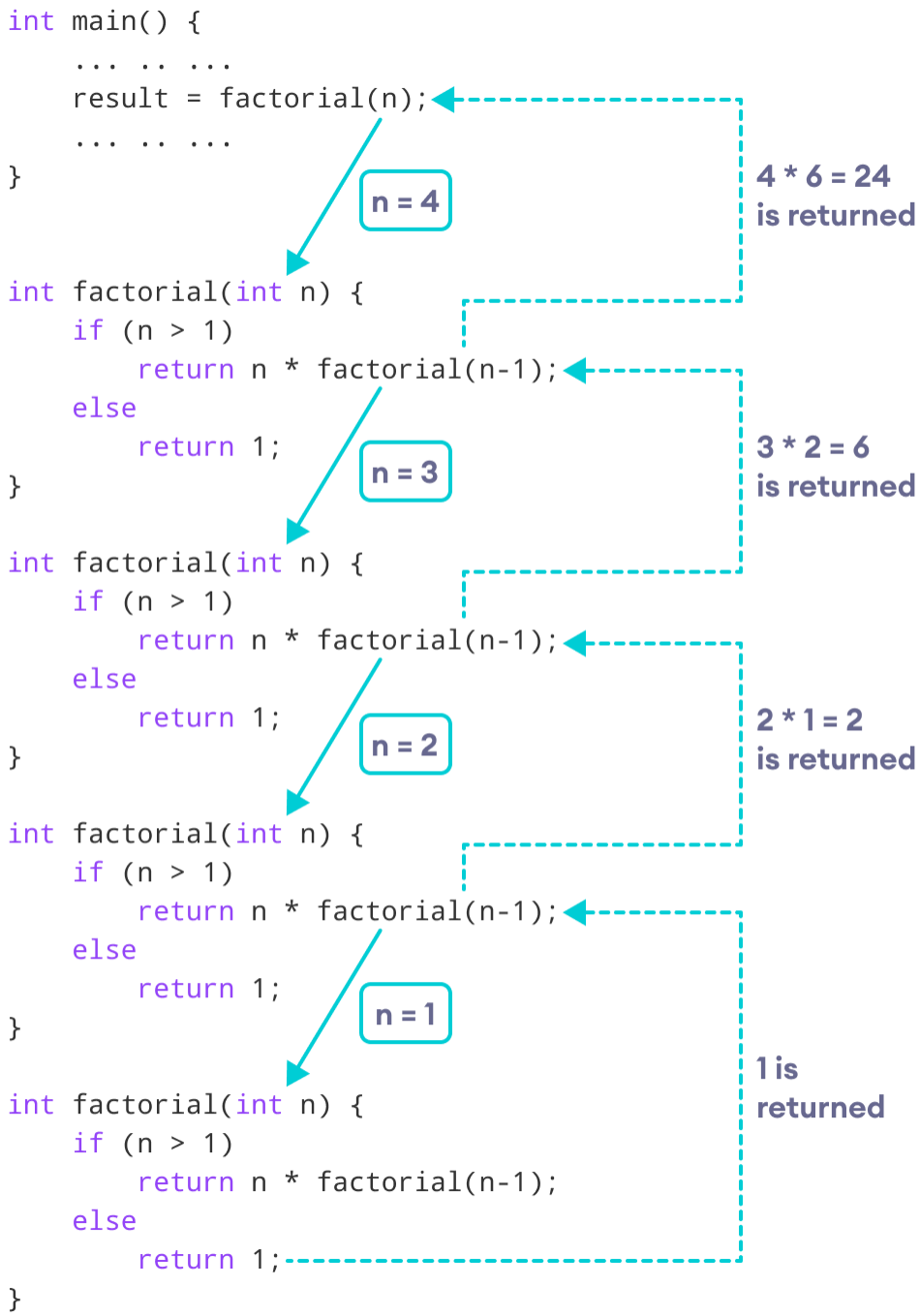
O funcție care implementează în limbajul C relația de recurență de mai sus este următoarea:

```
unsigned long long int factorial(unsigned int n)
{
    if(n == 1) return 1;
    return n * factorial(n-1);
}
```

Implementarea mecanismului de recursivitate într-un limbaj de programare se realizează folosind o structură de date de tip *stivă* (*stack* – http://elf.cs.pub.ro/sda-ab/wiki/laboratoare/laborator-04) pe care se salvează contextele de apel asociate apelurilor funcției respective. Astfel, în momentul apelării unei funcții, *contextul de apel* curent, format din numele funcției, adresa de revenire din apel, copii ale valorilor

parametrilor transmişi prin valoare, variabilele locale şi valoarea returnată de funcţie, se salvează pe stivă, folosind o operaţie de tip `push`, iar în momentul terminării executării apelului respectiv contextul de apel asociat este eliminat din stivă, folosind o operaţie de tip pop.

De exemplu, pentru apelul `f = factorial(4)`, stiva programului (reprezentată invers pentru o implementare alternativă) va avea următoarea evoluţie (sursa imaginii: https://www.programiz.com/python-programming/recursion):



Observaţie: Orice funcţie recursivă trebuie să conţină, pe lângă componenta recurentă (care conţine autoapelurile), şi o condiţie de oprire (care nu conţine niciun autoapel) realizabilă. În caz contrar, în momentul apelării sale, se va produce o recursivitate "infinită", care va conduce la depăşirea dimensiunii maxime permise pentru stiva programului şi apariţia unei erori.

În continuare, vom prezenta câteva exemple clasice de funcţii recursive:

a) *Şirul lui Fibonacci* este definit prin următoarea relaţie de recurenţă:

$$f_n = \begin{cases} 0, & \text{dacă } n = 0 \\ 1, & \text{dacă } n = 1 \\ f_{n-2} + f_{n-1}, & \text{dacă } n \geq 2 \end{cases}$$

O implementare directă a acestei relaţii, care va furniza valoarea termenului de rang n a şirului lui Fibonacci, este următoarea funcţie recursivă:

```
unsigned long long int fibo(unsigned int n)
{
    if (n <= 1)
        return n;
    return fibo(n-2) + fibo(n-1);
}
```

Pentru $n \geq 40$ se observă faptul că timpul de executare este destul de mare şi creşte în raport cu valoarea lui n. În capitolul următor, dedicat complexităţii computaţionale, se va demonstra faptul că numărul de apeluri recursive efectuate este exponenţial în raport cu n, aceasta fiind cauza timpului mare de executare. O implementare eficientă se poate obţine iterativ, astfel:

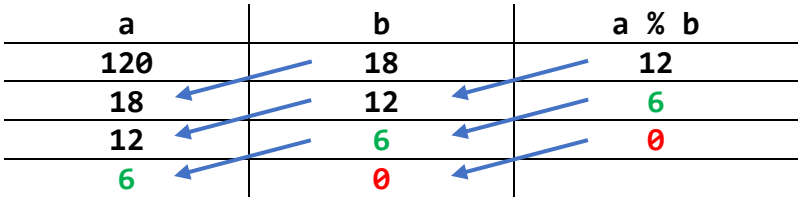
```
unsigned long long int fibo(unsigned int n)
{
    unsigned long long int a, b, c;
    unsigned int i;

    if (n <= 1)
        return n;

    a = 0;
    b = 1;
    for(i = 0; i < n; i++)
    {
        c = a+b;
        a = b;
        b = c;
    }

    return a;
}
```

b) *Algoritmul lui Euclid* permite determinarea celui mai mare divizor comun a două numere întregi nenule a şi b prin împărţiri repetate, respectiv: cât timp restul împărţirii lui a la b este nenul înlocuim a cu b şi b cu restul împărţirii lui a la b, iar ultimul rest nenul obţinut va fi $cmmdc(a, b)$. De exemplu, pentru $a = 120$ şi $b = 18$, cel mai mare divizor comun se va calcula astfel:



a	b	a % b
120	18	12
18	12	6
12	6	0
6	0	

Ultimul rest nenul este egal cu 6, deci vom obţine $cmmdc(120, 18) = 6$. Se observă faptul că ultimul rest nenul este egal cu ultimul împărţitor, deci o variantă de implementare iterativă a acestui algoritm este următoarea:

```
int cmmdc(int a, int b)
{
    int r;

    if(b == 0) return a;

    r = a % b;
    while(r != 0)
    {
        a = b;
        b = r;
        r = a % b;
    }

    return b;
}
```

Analizând algoritmul, putem deduce foarte uşor următoarea formulă de recurenţa pentru calculul celui mai mare divizor comun a două numere întregi:

$$cmmdc(a, b) = \begin{cases} a, & \text{dacă } b = 0 \\ cmmdc(b, a\%b), & \text{dacă } b \neq 0 \end{cases}$$

Implementarea acestei relaţii de recurenţă printr-o funcţie recursivă este directă:

```
int cmmdc(int a, int b)
{
    if(b == 0) return a;
    return cmmdc(b, a%b);
}
```

Observăm faptul că funcţiile au complexităţi egale, respectiv numărul de iteraţii este aproximativ egal cu numărul de autoapeluri.

c) *Calculul sumei cifrelor unui număr natural* se poate realiza folosind următoarea relaţie de recurenţă:

$$sc(n) = \begin{cases} n, & \text{dacă } n < 10 \\ n\%10 + sc(n/10), & \text{dacă } n \geq 10 \end{cases}$$

Implementarea acestei relaţii de recurenţă printr-o funcţie recursivă este banală:

```
unsigned int sc(unsigned int n)
{
    if (n < 10)
        return n;
    return n%10 + sc(n/10);
}
```

Ce relaţie există între numărul de autoapeluri din varianta recursivă şi numărul de iteraţii din varianta iterativă?

d) *Simularea unei instrucţiuni repetitive de tipul* `for(i = a; i <= b; i++)` se poate realiza folosind o funcţie recursivă astfel:

```
void for_rec(int a, int b)
{
    if(a <= b)
    {
        printf("%d ", a);
        for_rec(a+1, b);
    }
}
```

O consecinţă a utilizării unei structuri de date de tip stivă pentru implementarea mecanismului recursivităţii o constituie salvarea contextelor de apel ale unei funcţii în ordinea lor inversă, deci mutând instrucţiunea de afişare a valorii curente `printf("%d ", a)` după autoapelare funcţiei vom obţine o simulare a unei instrucţiuni repetitive de tipul `for(i = b; i >= a; i--)`:

```
void for_rec_invers(int a, int b)
{
    if(a <= b)
    {
        for_rec_invers(a+1, b);
        printf("%d ", a);
    }
}
```

Atenţie, apelarea acestei funcţii se va realiza tot pentru valori `a` şi `b` cu proprietatea că `a <= b`, de exemplu `for_rec_invers(1, 10)`!

e) Folosind ideea din exemplul anterior, putem să citim de la tastatură şi să afişăm invers un şir de numere întregi terminat cu valoarea 0 (care se consideră că nu face parte din şir) fără a folosi nicio structură de date de tip tablou, astfel:

```
void inversare()
{
    int x;

    scanf("%d", &x);
    if(x != 0)
    {
        inversare();
        printf("%d ", x);
    }
}
```

Deşi standardul limbajului C nu încurajează utilizarea recursivă a funcţiei `main()`, totuşi, în majoritatea compilatoarelor de C, acest lucru este posibil:

```
#include <stdio.h>

int main()
{
    int x;

    scanf("%d", &x);
    if(x != 0)
    {
        main();
        printf("%d ", x);
    }
}
```

f) *Testarea recursivă a primalităţii unui număr natural* se poate realiza simulând o instrucţiune de tipul `for` pentru valorile posibilului divizor `d` al unui număr natural `n`, cuprinse între `2` şi `n/2`:

```
int prim_rec(int n, int d)
{
    if(d <= n/2)
        if(n % d == 0)
            return 0;
        else
            return prim_rec(n, d+1);
    return 1;
}
```

Evident, funcţia se va apela prin `prim_rec(n, 2)`.

g) *Suma elementelor dintr-un tablou unidimensional* se poate defini recursiv ca fiind suma dintre primul element al tabloului și suma celorlalte elemente din tablou dacă tabloul este nevid, respectiv 0 dacă tabloul este vid:

```
int suma(int v[], int i)
{
    if(i >= 0)
        return v[i] + suma(v, i-1);
    return 0;
}
```

Deoarece parametrul `i` reprezintă indexul elementului curent din tabloul `v`, funcția se va apela prin `suma(v, n-1)`, unde `n` reprezintă numărul de elemente din tabloul `v`.

Într-un mod similar se poate calcula și suma elementelor strict pozitive dintr-un tablou unidimensional:

```
int suma_pozitive(int v[], int i)
{
    if(i >= 0)
        if(v[i] > 0)
            return v[i] + suma_pozitive(v, i-1);
        else
            return suma_pozitive(v, i-1);
    return 0;
}
```

h) *Frecvența unei litere într-un șir de caractere* se poate calcula recursiv astfel:

```
unsigned int frecventa(char* sir, char litera)
{
    if(strlen(sir) > 0)
        if(sir[0] == litera)
            return 1 + frecventa(sir+1, litera);
        else
            return frecventa(sir+1, litera);
    return 0;
}
```

Reamintim faptul că un șir de caractere este o secvență de caractere cuprinsă între o adresă de tip `char*` și caracterul `'\0'`, iar numele unui șir de caractere este chiar adresa primului său element, deci `sir+1` reprezintă șirul obținut prin ștergerea primului caracter din șirul respectiv.

i) Din numărul 4 se poate obține orice număr natural nenul aplicând în mod repetat următoarele operații:
1. împărțirea numărului la 2;
2. adăugarea cifrei 4 la sfârșitul numărului;
3. adăugarea cifrei 0 la sfârșitul numărului.

De exemplu, numărul 101 se poate obţine prin şirul de operaţii $4 \to 40 \to 404 \to 202 \to 101$, iar numărul 133 prin şirul de operaţii $4 \to 2 \to 24 \to 12 \to 6 \to 64 \to 32 \to 16 \to 8 \to 84 \to 42 \to 424 \to 212 \to 106 \to 1064 \to 532 \to 266 \to 133$.

Pentru a afişa şirul de operaţii prin care se poate obţine un număr natural nenul din numărul 4, vom aplica asupra sa operaţiile inverse operaţiilor date:
1'. înmulţirea numărului cu 2;
2'. eliminarea ultimei cifre, dacă aceasta este 4;
3'. eliminarea ultimei cifre, dacă aceasta este 0.

Astfel, pentru numărul 101 vom obţine următorul şir de operaţii $101 \to 202 \to 404 \to 40 \to 4$.

```
void numar4(unsigned int n)
{
    if(n != 4)
    {
        if(n % 10 == 0 || n%10 == 4)
            numar4(n/10);
        else
            numar4(2*n);
        printf(" -> %u", n);
    }
    else
        printf("4");

}
```

Încheiem prezentarea funcţiilor recursive menţionând faptul că, în general, acestea consumă multă memorie (pentru salvarea contextelor de apel), sunt mai greu de depanat decât funcţiile iterative şi, de multe ori, necesită un timp de executare mai mare. Din aceste motive, se recomandă utilizarea unei funcţii recursive doar în cazul în care complexitatea sa computaţională este echivalentă cu cea a variantei iterative, dar implementarea sa este mai simplă.