Recursivitate

Un algoritm (functie) recursiv(a) este un algoritm (functie) care ***se autoapeleaza***

Ex.: Calculul factorialului

n! = n * (n-1) * (n-2) * … * 2 * 1 <=> n! = n * (n-1)!

- Recursivitatea corespunde unei relatii de recurenta

Recursivitate

```
int FactorialIterativ(int n)
{
    int sum = 1;
    if (n <= 1)
        return sum;
    while (n > 1)
    {
        sum *= n;
        n--;
    }
    return sum;
}
```

```
1: int FactorialRec(int n)
2: {
3:     if (n <= 1)
4:         return 1;
5:     return n * FactorialRec(n - 1);
6: }
```

Recursivitate

3 REGULI:

1. Un algoritm recursiv trebuie sa contina un caz de baza
   - Conditia de iesire din recursie (de oprire a apelului recursiv)
   - Tipic: o problema suficient de mica pt. a fi rezolvata direct

2. Un algoritm recursiv trebuie sa isi modifice starea si sa se deplaseze spre cazul de baza
   - Relatia de recurenta
   - Tipic: cu fiecare apel se modifica datele cu care este apelat algoritmul

3. Un algoritm recursiv se auto-apeleaza

Recursivitate

```
int FactorialIterativ(int n)
{
    int sum = 1;
    if (n <= 1)
        return sum;
    while (n > 1)
    {
        sum *= n;
        n--;
    }
    return sum;
}
```

```
1: int FactorialRec(int n)
2: {
3:     if (n <= 1)
4:         return 1;
5:     return n * FactorialRec(n - 1);
6: }
```

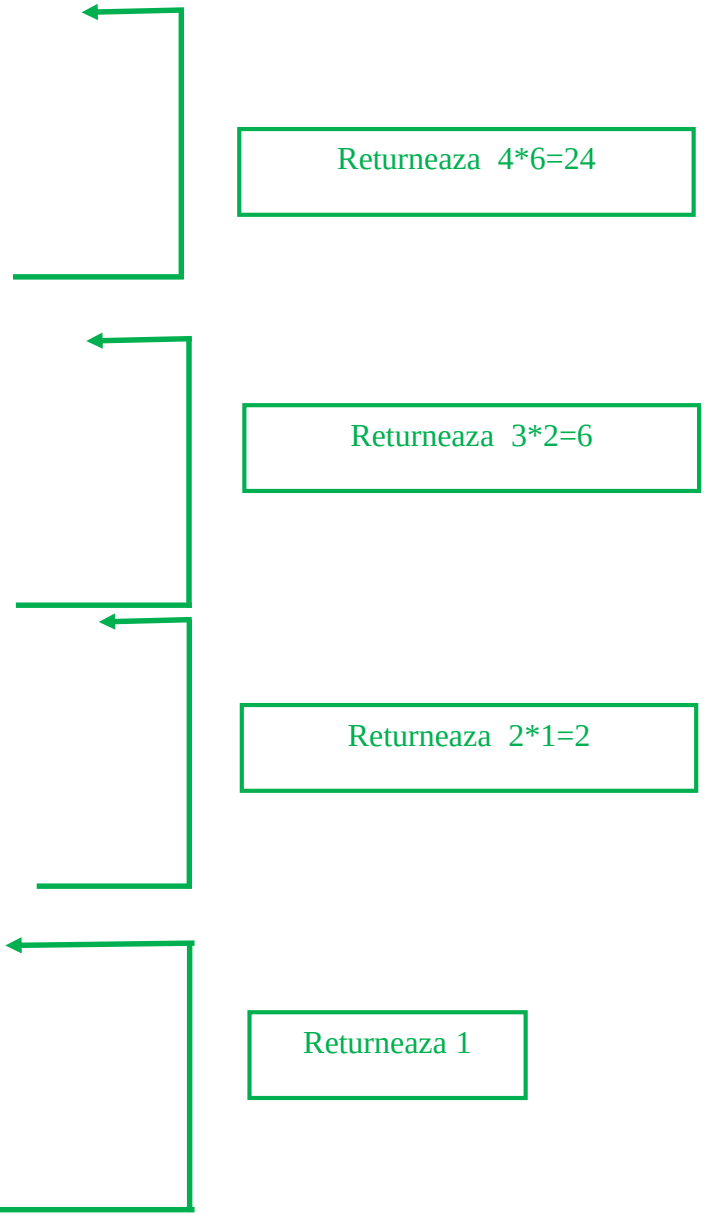
Conditia de oprire a recursiei

Apelul recursiv

Parametrii de apel evolueaza

Recursivitate

```
….. in main…

cout<< FactorialRec(4);

……..

int FactorialRec(int n)
{                    4
        if (n <= 1) return 1;
        return n * FactorialRec(n - 1);
}

                4                3

int FactorialRec(int n)
{                    3
        if (n <= 1) return 1;
        return n * FactorialRec(n - 1);
}

                3                2

int FactorialRec(int n)
{                    2
        if (n <= 1) return 1;
        return n * FactorialRec(n - 1);
}

                2                1

int FactorialRec(
{                    1
        if (n <= 1) return 1;
        return n * FactorialRec(n - 1);
}
```



Recursivitate

Observatii legate de stocarea datelor unui program in memorie

Variabilele globale sau statice

- sunt memorate intr-o zona de memorie fixă, mai exact în segmentul de date

Variabilele automate (locale)

- se memorează in stivă

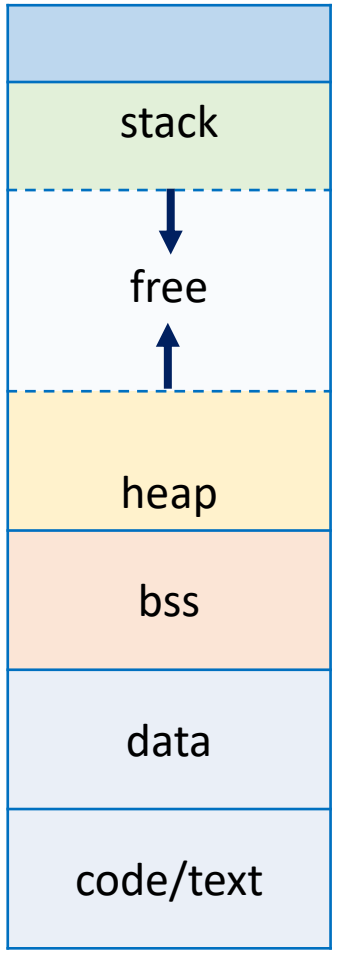
Variabilele dinamice (*cu malloc in C şi cu new in C++*)

- in memoria "heap"

Recursivitate

Observatii legate de stocarea datelor unui program in memorie

Virtual memory space



Command-line arguments, environment variables

Automatic variables, arguments, return addresses

Dynamically allocated data

Unitialized data (static, global)

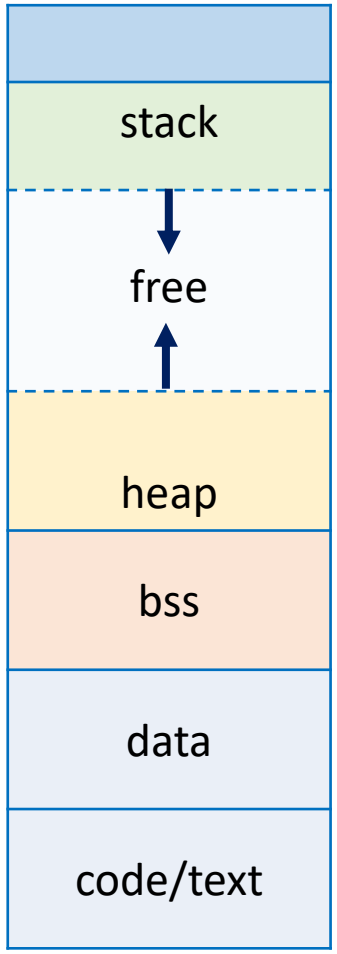
Initialized data (static, global, constant)

Program code

Recursivitate

Observatii legate de stocarea datelor unui program in memorie

Virtual memory space



Command-line arguments, environment variables

Automatic variables, arguments, return addresses

Dynamically allocated data

Unitialized data (static, global)

Initialized data (static, global, constant)

Program code

```
char* s = "hello";

int i;

char* foo(void)
{
  char* p;

  static int x = 0;

  x += 1;

  i = 1;

  p = malloc( sizeof(char) * strlen(s) + i * x);

  for(int j=0; j < x; ++j)
    strcat(p,s);
  return p;
}
```

.data: assigned "hello" at start
Available until termination

.bss: assigned 0 at start
Available until termination

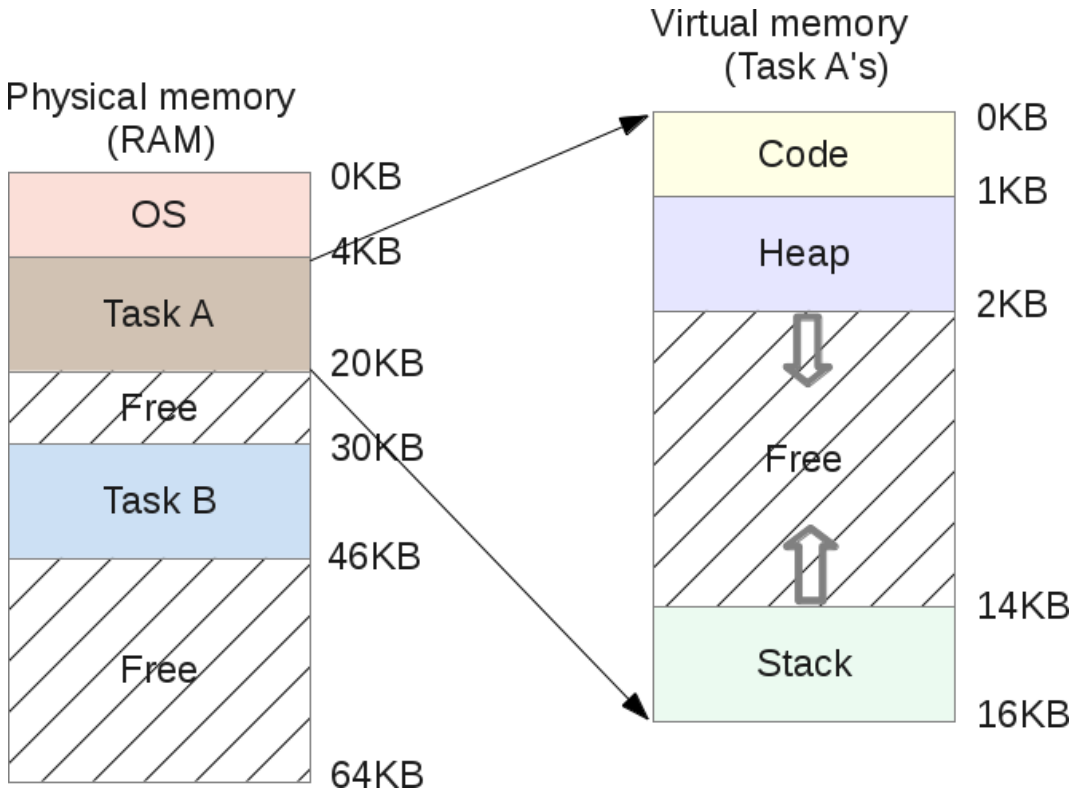
Stack: allocated at function call
Deallocated on return from foo( )

.data: assigned 0 at start
Available until termination

Heap: allocates 6 bytes at malloc()
Deallocated on free( )

Recursivitate

Observatii legate de stocarea datelor unui program in memorie



Memory limits on Windows: https://msdn.microsoft.com/en-us/library/aa366778.aspx

Read more: https://answers.microsoft.com/en-us/windows/forum/windows_10-performance/physical-and-virtual-memory-in-windows-10/e36fb5bc-9ac8-49af-951c-e7d39b979938

Recursivitate

Executia unei functii recursive:

- se creeaza in segmentul de stiva 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);
- se continuă cu instrucţiunea dată de adresa de retur memorată în "inregistrarea de activare".

Recursivitate

```
1: int FactorialRec(int n)
2: {
3:     if (n <= 1)
4:         return 1;
5:     return n * FactorialRec(n - 1);
6: }
```

```
FactorialRec(4);
```

Trace table (stack)

Apel	n	Linie retur	Valoare retur
1	4	5	
..	..	..	

Recursivitate

Atentie!

- 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**.

- Consumul de memorie al unui algoritm recursiv este proporţional cu numărul de apeluri recursive ce se fac. Functiile recursive consumă mai multa memorie decât cele iterative.

Recursivitate

Sirul lui Fibonacci: 1 1 2 3 5 8 13 21 34 55 89 144 …

Relatia de recurenta: $F_n = F_{n-1} + F_{n-2}$

$F_1 = 1, F_0 = 0$

```
int FibonacciIterative(int n)
{
    if (n <= 1) return n;
    int prevPrev = 0;
    int prev = 1;
    int result = 0;
    for (int i = 2; i <= n; i++)
    {
        result = prev + prevPrev;
        prevPrev = prev;
        prev = result;
    }
    return result;
}
```

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

Recursivitate

Sirul lui Fibonacci: 1 1 2 3 5 8 13 21 34 55 89 144 …

Varianta optimizata:

```
#define DIM_MAX 41
int resultHistory[DIM_MAX];

int FibonacciRecursiveOpt(int n)
{
        if (n == 0) return 0;
        if (n == 1) return 1;
        if (resultHistory[n] != -1)
                return resultHistory[n];

        int result = FibonacciRecursiveOpt(n - 1) +
                     FibonacciRecursiveOpt(n - 2);
        resultHistory[n] = result;

        return result;
}
```

```
void main() {

int n = 40;

for (int i = 1; i <= n; i++)
        resultHistory[i] = -1;

cout << FibonacciRecursiveOpt(n);
}
```

Recursivitate

Recursivitatea este mai inceata si limitativa (stack overflow)

N	Recursiv (ms)	Recursiv opt. (ms)	Iterativ (ms)
30	54	0.0023	0.0019
40	5663	0.0027	0.0019
50	709972	0.0075	0.0047
100	too long	0.0205	0.0067
1000	too long	0.3034	0.0075
5000	stack overflow	stack overflow	0.0213
100000	stack overflow	stack overflow	0.0426

Ex: Fibonacci
Intel Core i7-6700HQ @2.6GHz

- datorita numarului mare de operatii push/pop ale registrilor in stiva de memorie

Dimens. implicita a stivei de memorie in Visual Studio: 1MB (change it: *Properties - Configuration Properties - Linker - System - Stack Reserve Size*)

Recursivitate

Exemple:

- Crearea unei liste liniare simplu inlantuite prin inserari repetate la sfarsitul listei

- Parcurgerea dus-intors a unei liste liniare simplu inlantuite