

## CURS 8

### Complexitatea algoritmilor

Un *algorithm* este o succesiune de prelucrări care se aplică mecanic asupra unor date de intrare în scopul obținerii unor date de ieșire.

Principalele caracteristici ale unui algoritm sunt:

- *corectitudine* – proprietatea unui algoritm de a furniza date de ieșire corecte pentru orice date de intrare;
- *determinism* – pentru un anumit set de date de intrare un algoritm trebuie să furnizeze întotdeauna același date de ieșire;
- *generalitate* – un algoritm nu trebuie să rezolve doar o problemă particulară, ci o întreagă clasă de probleme;
- *claritate* – prelucrările efectuate de un algoritm trebuie să nu fie ambigue;
- *finitudine* – un algoritm trebuie să se termine într-un timp finit (i.e., după un număr finit de pași).

În practică, finitudinea unui algoritm nu este suficientă pentru a-i garanta utilitatea, deoarece s-ar putea ca timpul după care el se va termina să fie prea mare în raport cu cerințele noastre. Din acest motiv, un algoritm trebuie analizat și din punctul de vedere al *eficienței* sale, respectiv să verificăm dacă algoritmul se termină într-un anumit timp maxim, convenabil în practică din punctul nostru de vedere. Dacă, în plus, numărul de pași după care algoritmul se termină este minim, atunci algoritmul respectiv este *optim*.

De exemplu, dacă am calcula suma numerelor naturale mai mici sau egale decât un  $n$  dat adunând, pe rând, fiecare număr de la 1 la  $n$  am obține un algoritm care s-ar termina într-un timp finit și, în plus, ar fi eficient pentru valori "rezonabile" ale lui  $n$ . Totuși, algoritmul optim constă în calculul sumei  $1 + 2 + \dots + n$  folosind formula  $\frac{n(n+1)}{2}$ .

În practică, eficiența unui algoritm se studiază prin prisma complexității sale computaționale, respectiv a resurselor necesare pentru executarea sa. Cele mai importante resurse luate în considerare în evaluarea complexității computaționale a unui algoritm sunt *timpul de executare* și *spațiul de memorie* necesar. Deoarece exprimarea timpului de executare în unități de timp nu este relevantă, aceasta depinzând foarte mult de configurația hardware și software a sistemului de calcul, se preferă exprimarea acestuia printr-o expresie care estimează numărul maxim de operații elementare efectuate de algoritmul respectiv în raport de dimensiunile datelor de intrare, folosind notația asimptotică  $\mathcal{O}(\text{expresie})$ . De exemplu, dacă un algoritm are complexitatea computațională  $\mathcal{O}(n^2)$  înseamnă că dimensiunea datelor sale de intrare este egală cu  $n$  (variabila din expresie) și algoritmul efectuează aproximativ  $n^2$  operații elementare (expresia) pentru a rezolva problema respectivă. Pentru a exprima complexitatea unui algoritm din punct de vedere al spațiului de memorie necesar se utilizează aceeași notație, însă precizând explicit acest lucru (în mod implicit, complexitatea unui algoritm se referă la timpul său de executare).

Utilizarea notației asimptotice permite o comparare simplă a performanțelor a doi sau mai mulți algoritmi care rezolvă o aceeași problemă, fiind evident faptul că un algoritm este cu atât mai eficient cu cât numărul de operații elementare efectuate și, eventual, spațiul de memorie necesar este mai mic.

*Operațiile elementare* pe care le efectuează un algoritm sunt:

- operația de atribuire și operațiile aritmetice;
- operația de decizie și **operația de salt**;
- operațiile de citire/scriere.

În programarea actuală, de tip structurat, nu se mai permite utilizarea **operației de salt** (instrucțiunea `goto` din limbajele C/C++). Totuși, instrucțiunile repetitive sunt descompuse în astfel de instrucțiuni înainte de a fi executate de procesor! De exemplu, o instrucțiune repetitivă poate fi simulată în limbajul C++ astfel:

```
#include <stdio.h>

int main()
{
    int x = 1;
    Eticheta:    cout << x++ << endl;
                if(x <= 10) goto Eticheta;

    return 0;
}
```

Astfel, pentru a estima complexitatea unei instrucțiuni repetitive care execută de  $n$  ori un anumit bloc de instrucțiuni, vom considera faptul că instrucțiunea repetitivă este atomică (i.e., nu vom lua în considerare numărul de operații elementare efectuate pentru executarea sa), deci complexitatea sa totală va fi  $O(n \cdot \text{complexitate\_bloc\_instrucțiuni})$ :

Instrucțiune repetitivă	Complexitatea totală	Exemplu
<pre>for i in range(n):     instrucțiune cu complexitatea <math>O(1)</math></pre>	$O(n)$	<pre>n = int(input("n = ")) L = [] for i in range(n):     L.append(i+1)</pre>
<pre>for i in range(n):     instrucțiune cu complexitatea <math>O(m)</math></pre>	$O(n \cdot m)$	<pre>#presupunem că len(s) = n #și len(t) = m s = input("s: ") t = input("t: ") for x in s:     print(x, t.count(x))</pre>
<pre>for i in range(n):     instrucțiune cu complexitatea <math>O(1)</math> for i in range(m):     instrucțiune cu complexitatea <math>O(1)</math></pre>	$O(n + m)$	<pre>s = input("s: ") t = input("t: ") #presupunem că len(s) = n #și len(t) = m L = [] for x in s:     L.append(x) for x in t:     L.append(x)</pre>

Instrucțiune repetitivă	Complexitatea totală	Exemplu
<pre>for i in range(n):     for j in range(m):         instrucțiune cu complexitatea <math>O(1)</math></pre>	$O(n \cdot m)$	<pre>n = int(input("n = ")) m = int(input("m = ")) L = [] for i in range(n):     for j in range(m):         L.append((i, j))</pre>
<pre>for i in range(n):     for j in range(m):         instrucțiune cu complexitatea <math>O(p)</math></pre>	$O(n \cdot m \cdot p)$	<pre>#presupunem că A este o #matrice cu n linii și m #coloane, iar L o listă #cu p elemente for x in L:     print(f"Valoarea {x}: ")     for i in range(n):         for j in range(m):             if x == A[i][j]:                 print(i, j)</pre>
<pre>for i in range(n): {     instrucțiune cu complexitatea <math>O(m)</math>     instrucțiune cu complexitatea <math>O(p)</math> }</pre>	$O(n(m + p))$	<pre>#presupunem că A este o #listă cu n elemente, B o #listă cu m elemente, iar #C o listă cu p elemente for x in A:     p = B.count(x)     q = C.count(x)     if p &gt;= 1 and q &gt;= 1:         print(x)</pre>

Evident, exemplele de mai sus rămân valabile pentru orice alt tip de instrucțiune repetitivă (e.g., instrucțiunea while), dar în exemplele prezentate am preferat utilizarea instrucțiunii repetitive for pentru a evidenția într-un mod simplu faptul că instrucțiunea respectivă se execută de  $n$  ori. De asemenea, în locul unei instrucțiuni cu o anumită complexitate putem considera și un bloc de instrucțiuni cu aceeași complexitate. Atenție, în limbajul Python unele dintre exemplele de mai sus pot fi scrise mult mai concis, folosind, de exemplu, secvențe de inițializare. De exemplu, primul exemplu de mai sus poate fi scris condensat astfel: `L = [i+1 for i in range(int(input("n = ")))]`. Evident, complexitatea rămâne aceeași, respectiv  $O(n)$ .

Practic, complexitatea computațională a unui algoritm se determină estimând numărul maxim de operații elementare efectuate în raport de dimensiunile datelor de intrare. De exemplu, complexitatea maximă a algoritmului clasic pentru calculul valorii maxime dintr-o listă formată din  $n$  valori se poate determina astfel:

Instrucțiune	Operații elementare	
n = int(input("Numar elemente: "))	1 afișare + 1 citire	
lista = []	1 atribuire	
for i in range(n):	de n ori:	3n operații elementare
elem = int(input("Element:"))	1 afișare + 1 citire	
lista.append(elem)	1 atribuire	
maxim = lista[0]	1 atribuire	
for i in range(1, n):	de n-1 ori:	2(n-1) operații elementare
if lista[i] > maxim:	1 operație de decizie	
maxim = lista[i]	1 atribuire	
print("Maximul:", maxim)	1 afișare	
TOTAL:	5n + 3 operații elementare	

În concluzie, complexitatea algoritmului din punct de vedere al timpului de executare este  $\mathcal{O}(5n + 3) \approx \mathcal{O}(n)$ , iar complexitatea din punct de vedere al memoriei utilizate este tot  $\mathcal{O}(n)$ , deoarece se memorează cele  $n$  valori într-o listă.

Expresiile utilizate în notația asimptotică a complexității computaționale se simplifică folosind următoarele două reguli (se presupune faptul că  $n \rightarrow \infty$ ):

- constantele (multiplicative sau aditive) sunt ignorate:  $\mathcal{O}(5n + 3) \approx \mathcal{O}(5n) \approx \mathcal{O}(n)$
- dintr-o expresie se păstrează doar termenul dominant:  $\mathcal{O}(3n^2 + 5n + 7) \approx \mathcal{O}(3n^2) \approx \mathcal{O}(n^2)$  sau  $\mathcal{O}(2^n + 3n^2) \approx \mathcal{O}(2^n)$

În continuare, vom analiza din punct de vedere al complexității computaționale mai mulți algoritmi:

a) *algoritmul de sortare prin selecția minimului*

Instrucțiune	Operații elementare	
n = int(input("Numar elemente: "))	1 afișare + 1 citire	
lst = []	1 atribuire	
for i in range(n):	de n ori:	3n operații elementare
elem = int(input("Element: "))	1 afișare + 1 citire	
lst.append(elem)	1 atribuire	
for i in range(n-1):	de $(n-1)+\dots+1 = \frac{n(n-1)}{2}$ ori:	
for j in range(i+1, n):		
if lst[i] > lst[j]:	1 operație de decizie	maxim $\frac{3n(n-1)}{2}$ operații elementare
lst[i], lst[j] = lst[j], lst[i]	2 atribuiri (maxim!)	
print("Lista sortata:")	1 afișare	
for i in range(n):	de n ori:	n operații elementare
print(lst[i], end=" ")	1 afișare	
TOTAL:	$\frac{3n^2+5n+8}{2}$ operații elementare	

În concluzie, complexitatea acestui algoritm din punct de vedere al timpului de executare este  $\mathcal{O}\left(\frac{3n^2+5n+8}{2}\right) \approx \mathcal{O}(n^2)$ , iar complexitatea din punct de vedere al memoriei utilizate este tot  $\mathcal{O}(n)$ , deoarece se memorează cele  $n$  valori într-o listă.

b) *determinarea valorilor distincte (fără duplicate) dintr-o listă L formată din n numere întregi*

O variantă de rezolvare a acestei probleme constă în utilizarea unei liste auxiliare *distincte* care să memoreze valorile distincte găsite până la un moment dat. Practic, parcurgem lista *L* element cu element și dacă elementul curent nu se găsește în lista *distincte*, atunci îl adăugăm la sfârșitul său:

```
L = [int(x) for x in input("Lista: ").split()]

distincte = []
for x in L:
    if x not in distincte:
        distincte.append(x)

print("Elementele distincte:")
print(*distincte, end=" ")
```

Se observă faptul că instrucțiunile marcate cu roșu induc complexitatea algoritmului, respectiv  $\mathcal{O}(nd)$ , unde  $d$  reprezintă numărul valorilor distincte din lista inițială. Deoarece complexitatea unui algoritm trebuie exprimată doar în funcție de dimensiunile datelor de intrare (iar valoarea  $d$  nu reprezintă o dimensiune a datelor de intrare!), vom aproxima complexitatea sa maximă prin  $\mathcal{O}(n^2)$ , care se obține când numărul valorilor distincte  $d$  este aproximativ egal cu numărul  $n$  al valorilor din lista  $L$ . De asemenea, putem observa faptul că algoritmul are complexitatea  $\mathcal{O}(n)$  când numărul valorilor distincte  $d$  este mult mai mic decât  $n$  (de exemplu,  $d$  va fi egal cu 10 dacă vom considera elementele listei  $L$  ca fiind doar cifre). Complexitatea din punct de vedere al memoriei utilizate este  $\mathcal{O}(n)$ . Evident, un algoritm cu o complexitate medie mai bună, respectiv  $\mathcal{O}(n)$ , se poate obține folosind colecții de date de tip mulțime sau de tip dicționar!

*c) suma cifrelor unui număr natural  $n$*

Varianta clasică de rezolvare a acestei probleme constă în adăugarea fiecărei cifre a numărului  $n$  într-o sumă și apoi eliminarea sa:

```
n = int(input("n = "))

s = 0
while n != 0:
    s = s + n % 10
    n = n // 10

print(f"Suma cifrelor: {s}")
```

Se observă faptul că numărul operațiilor elementare efectuate este proporțional cu numărul  $c$  al cifrelor numărului  $n$ , deci este  $\mathcal{O}(c)$ . Complexitatea unui algoritm trebuie exprimată doar în funcție de dimensiunile datelor de intrare, trebuie să calculăm numărul  $c$  în funcție de numărul  $n$ . Presupunând faptul că numărul  $n$  are  $c$  cifre, rezultă că  $10^{c-1} \leq n < 10^c$ . Logarithmăm inegalitatea în baza 10 și obținem că  $\log_{10} 10^{c-1} \leq \log_{10} n < \log_{10} 10^c$ , deci  $c-1 \leq \log_{10} n < c$ . Din ultima inegalitate rezultă că  $\lfloor \log_{10} n \rfloor = c-1$ , deci  $c = \lfloor \log_{10} n \rfloor + 1$ . În concluzie, complexitatea acestui algoritm este  $\mathcal{O}(\lfloor \log_{10} n \rfloor)$ . Atenție, aceeași complexitate are și următoarea variantă, mult mai concisă, specifică limbajului Python:

```
print(f"Suma cifrelor: {sum([int(x) for x in input('n = ')])}")
```

## Clase uzuale de complexitate computațională

În continuare, vom prezenta clasele de complexitate computațională cele mai des întâlnite, în ordine crescătoare:

a) **clasa  $\mathcal{O}(1)$  – complexitate constantă**

**Exemple:** suma a două numere sau alte formule simple (e.g., rezolvarea ecuației de gradul I sau II, calculul minimului sau maximului dintre două numere etc.), adăugarea unui element la sfârșitul unei liste, determinarea numărului de elemente dintr-o colecție, accesarea unui element al unei liste prin indexul său

b) **clasa  $\mathcal{O}(\log_b n)$  – complexitate logaritmică**

**Exemple:** suma cifrelor unui număr natural -  $\mathcal{O}(\log_{10} n)$ , operația de căutare binară a unei valori într-o listă sortată cu  $n$  elemente -  $\mathcal{O}(\log_2 n)$

c) **clasa  $\mathcal{O}(n)$  – complexitate liniară**

**Exemple:** citirea/scrierea/o singură parcurgere a unui liste cu  $n$  elemente, testarea apartenenței unei valori la o listă cu  $n$  elemente sau un șir format din  $n$  caractere, numărarea aparițiilor unei valori într-o listă cu  $n$  elemente sau într-un șir format din  $n$  caractere

d) **clasa  $\mathcal{O}(n \log_2 n)$**

**Exemple:** metodele de sortare *Quicksort* (sortarea rapidă), *Mergesort* (sortarea prin interclasare), *Timsort* (implementată în funcția `sort` din Python) și *Heapsort* (sortarea cu ansamble)

e) **clasa  $\mathcal{O}(n^2)$  – complexitate pătratică**

**Exemple:** metoda de sortare prin interschimbare, metoda de sortare Bubblesort, compararea fiecărui element al unui liste cu  $n$  elemente cu toate celelalte elemente din listă, citirea/scrierea/o singură parcurgere a unui matrice pătratică de dimensiune  $n$

f) **clasa  $\mathcal{O}(n^k)$ ,  $k \geq 3$  – complexitate polinomială**

**Exemple:** sortarea fiecărei linii dintr-o matrice pătratică de dimensiune  $n$  folosind sortarea prin interschimbare sau Bubblesort -  $\mathcal{O}(n^3)$ , algoritmul Roy-Floyd-Warshall pentru determinarea drumurilor minime într-un graf orientat ponderat -  $\mathcal{O}(n^3)$

g) **clasa  $\mathcal{O}(a^n)$ ,  $a \geq 2$  – complexitate exponențială**

**Exemple:** generarea tuturor submulțimilor unei mulțimi cu  $n$  elemente -  $\mathcal{O}(2^n)$ , partiționarea unei mulțimi cu  $n$  elemente în două submulțimi cu aceeași sumă a elementelor -  $\mathcal{O}(2^n)$

**Observații:**

1. Complexitatea unui algoritm NU poate fi mai mică decât complexitatea citirii datelor de intrare și scrierii datelor de ieșire! De exemplu, complexitatea minimă a oricărui algoritm de generare a tuturor submulțimilor unei mulțimi cu  $n$  elemente este  $\mathcal{O}(2^n)$ , deoarece, indiferent de metoda de generare a submulțimilor, trebuie afișate cele  $2^n$  submulțimi!
2. Complexitatea unei operații poate fi mai mică decât complexitatea unui algoritm care o implementează, deoarece un algoritm presupune citirea unor date de intrare și scrierea unor date de ieșire! De exemplu, operația de căutare binară a unei valori într-o listă sortată cu  $n$  elemente are complexitatea  $\mathcal{O}(\log_2 n)$ , dar un algoritm care utilizează această operație utilizează presupune și citirea listei sortate crescător, deci complexitatea algoritmului va fi  $\mathcal{O}(n + \log_2 n) \approx \mathcal{O}(n)$ .
3. Evident, există și alte clase de complexitate relativ des întâlnite, în afara celor menționate anterior. De exemplu, există clasa  $\mathcal{O}(m + n)$  – interclasarea a două liste sortate crescător având  $m$ , respectiv  $n$  elemente; clasa  $\mathcal{O}(m \cdot n)$  – parcurgerea unui tablou bidimensional cu  $m$  linii și  $n$  coloane; clasa  $\mathcal{O}(m \cdot n \cdot p)$  – înmulțirea unei matrice cu  $m$  linii și  $n$  coloane cu o matrice cu  $n$  linii și  $p$  coloane etc.
4. Există complexități mai mari decât cea exponențială (e.g., generarea tuturor permutărilor unei mulțimi cu  $n$  elemente are complexitatea  $\mathcal{O}(n!)$ ), dar algoritmi cu o astfel de complexitate sunt, în realitate, inutilizabili sau, mai precis, sunt utilizabili doar pentru valori foarte mici ale lui  $n$ . De exemplu, considerând un PC pe care interpretorul limbajului Python execută aproximativ 500000 de operații elementare pe secundă, adică o operație elementară durează aproximativ  $2 \cdot 10^{-6}$  secunde, generarea tuturor submulțimilor unei mulțimi cu  $n = 30$  de elemente va dura aproximativ  $2^{30} \cdot 2 \cdot 10^{-6}$  secunde  $\approx 2140$  de secunde  $\approx 35$  de minute, pentru  $n = 50$  va dura aproximativ  $2^{50} \cdot 2 \cdot 10^{-6}$  secunde  $\approx 625500$  ore  $\approx 26062$  de zile  $\approx 72$  de ani, iar pentru  $n = 100$  de elemente va dura aproximativ  $2^{100} \cdot 2 \cdot 10^{-6}$  secunde  $\approx 8 \cdot 10^{16}$  ani  $\approx 80$  de milioane de miliarde de ani, adică o valoare de 6 milioane de ori mai mare decât vârsta estimată a Universului!

În continuare, vom prezenta câțiva algoritmi a căror complexitate se estimează mai greu:

a)

```
n = int(input("n = "))
i = 0
p = 1
while i <= n:
    j = 1
    while j <= p:
        print(j, end= " ")
        j = j + 1
    print()
    i = i + 1
    p = p * 2
```

După o analiză superficială, acest algoritmul pare să aibă complexitatea  $\mathcal{O}(n \cdot p)$ , datorită celor două instrucțiuni `while` imbricate. Analizând cu atenție algoritmul, vom observa faptul că, pentru fiecare valoare  $i$  cuprinsă între 0 și  $n$ , el afișează numerele de la 1 la  $2^i$ , ceea ce înseamnă că, în total, va afișa  $2^0 + 2^1 + \dots + 2^n = 2^{n+1} - 1$  numere, deci complexitatea sa este  $\mathcal{O}(2^n)$ , adică o complexitate exponențială! Această complexitate exponențială este indusă de faptul că variabila  $p$  își dublează valoarea la fiecare pas, deci are o creștere exponențială. Dacă variabila  $p$  ar avea o creștere liniară (e.g.,  $p = p + 1$ ), atunci, pentru fiecare valoare  $i$  cuprinsă între 0 și  $n$ , se vor afișa numerele de la 1 la  $i + 1$ , ceea ce înseamnă că, în total, va afișa  $1 + 2 + \dots + (n + 1) = \frac{(n+1)(n+2)}{2}$  numere, deci complexitatea sa ar fi  $\mathcal{O}(n^2)$ , adică o complexitate pătratică!

b)

```
a = int(input("a = "))
b = int(input("b = "))

p = 1
while p < a:
    p = p * 2
while p <= b:
    print(p, end=" ")
    p = p * 2
```

După o analiză superficială, acest algoritmul pare să aibă complexitatea  $\mathcal{O}(b)$ , datorită celor două instrucțiuni `while` secvențiale. Analizând cu atenție algoritmul, vom observa faptul că la fiecare pas valoarea variabilei  $p$  se dublează, deci, numărul total de iterații  $k$  pe care le va efectua algoritmul se obține rezolvând inecuația  $2^k \leq b$ , de unde obținem  $k \leq \log_2 b$ . În concluzie, acest algoritm, care afișează puterile lui 2 cuprinse între  $a$  și  $b$ , are complexitatea  $\mathcal{O}(\log_2 b)$ , deci o complexitate logaritmică!

c)

```
v = [int(x) for x in input("Valorile: ").split()]
n = len(v)
i = 0
j = n - 1
while i < j:
    while i < n and v[i] < 0:
        i = i + 1
    while j >= 0 and v[j] >= 0:
        j = j - 1
    if i < j:
        v[i], v[j] = v[j], v[i]
print("\nValorile:\n", v, sep="")
```

După o analiză superficială, acest algoritmul pare să aibă complexitatea  $\mathcal{O}(n^2)$ , unde  $n$  este numărul de elemente din lista  $v$ , datorită celor două instrucțiuni `while` secvențiale imbricate în altă instrucțiune `while`. Analizând cu atenție algoritmul, vom observa faptul că, în realitate, cei 2 indici  $i$  și  $j$  nu se suprapun, ci parcurg, fiecare, o porțiune din lista  $v$  (i.e., indicele  $i$  parcurge lista de la stânga spre dreapta, iar indicele  $j$  de la dreapta spre



stânga), iar în momentul în care cei 2 indici se "întâlnesc", algoritmul se termină. Astfel, per total, cei 2 indici vor parcurge o singură dată întreaga listă  $v$ , deci complexitatea algoritmului este  $\mathcal{O}(n)$ , respectiv o complexitate liniară. Algoritmul realizează un fel de sortare a elementelor listei, respectiv mută valorile negative înaintea celor pozitive, fără ca valorile negative sau cele pozitive să fie sortate conform unui criteriu. Metoda de parcurgere utilizată se numește *metoda arderii lumânării la două capete* (*two pointers*) și, într-o formă puțin modificată, este folosită și în metoda de sortare Quicksort.

## Funcții recursive

În general, prin *recursivitate* se înțelege proprietatea unor noțiuni de a se defini prin ele însele. De exemplu, numerele naturale poate fi definite recursiv folosind următoarele două axiome ale lui Peano ([https://en.wikipedia.org/wiki/Peano\\_axioms](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](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.,  $n! = 1 \cdot 2 \cdot \dots \cdot n$ ), folosind următoarea relație de recurență:

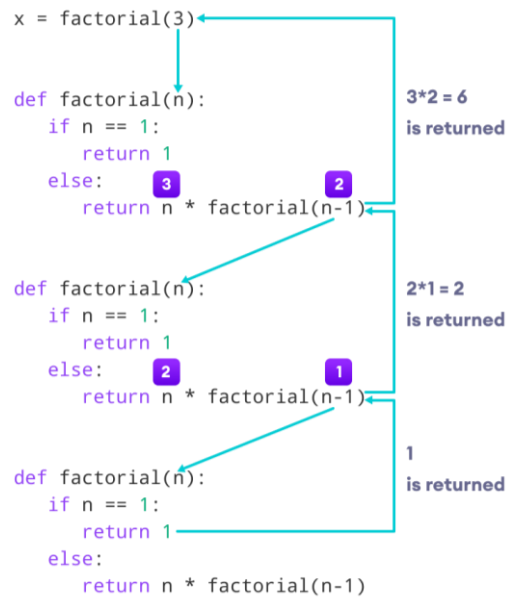
$$n! = \begin{cases} 1, & \text{dacă } n = 1 \\ n \cdot (n - 1)!, & \text{dacă } n \geq 2 \end{cases}$$

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

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)
```

În momentul apelării unei funcții, *contextul de apel* asociat (numele, variabilele locale, parametrii, adresa de revenire etc.) se salvează în stiva alocată programului, iar în momentul terminării executării sale, contextul de apel este eliminat din stivă.

De exemplu, pentru apelul  $f = \text{factorial}(3)$ , stiva programului (reprezentată invers) va avea următoarea evoluție (sursa imaginii: <https://www.programiz.com/python-programming/recursion>):



**Observație:** Orice funcție recursivă trebuie să conțină, pe lângă componenta recurentă (care conține autoapelurile), și o condiție de oprire (care nu conține niciun autoapel) realizabilă. În caz contrar, în momentul apelării sale, se va produce o recursivitate "infinită", care va conduce la depășirea numărului maxim de apeluri recursive permis (implicit, acesta este egal cu 1000) și la apariția unei erori. De exemplu, în cazul apelului `f = factorial(1000)`, se va afișa următoarea eroare:

```

def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

print(factorial(1000))
  
```

```

Traceback (most recent call last):
  File "C:/Users/Ours/OneDrive - unibuc.ro/Desktop/Test_Python/Test_curs.py", line 16, in <module>
    print(factorial(1000))
  File "C:/Users/Ours/OneDrive - unibuc.ro/Desktop/Test_Python/Test_curs.py", line 13, in factorial
    return n * factorial(n - 1)
  File "C:/Users/Ours/OneDrive - unibuc.ro/Desktop/Test_Python/Test_curs.py", line 13, in factorial
    return n * factorial(n - 1)
  File "C:/Users/Ours/OneDrive - unibuc.ro/Desktop/Test_Python/Test_curs.py", line 13, in factorial
    return n * factorial(n - 1)
  [Previous line repeated 995 more times]
  File "C:/Users/Ours/OneDrive - unibuc.ro/Desktop/Test_Python/Test_curs.py", line 11, in factorial
    if n == 0:
RecursionError: maximum recursion depth exceeded in comparison

Process finished with exit code 1
  
```

Numărul maxim de apeluri recursive care pot fi salvate pe stivă poate fi aflat folosind metoda `getrecursionlimit` și modificat folosind metoda `setrecursionlimit`, ambele din modulul `sys`:

```

1 import sys
2
3 def factorial(n):
4     if n == 0:
5         return 1
6     return n * factorial(n - 1)
7
8 rmax = sys.getrecursionlimit()
9 print("Numarul maxim de apeluri recursive initial:", rmax)
10
11 sys.setrecursionlimit(5000)
12 rmax = sys.getrecursionlimit()
13 print("Numarul maxim de apeluri recursive modificat:", rmax)
14
15 n = 1000
16 print(str(n) + "! = " + str(factorial(n)))
17

```

Run: "C:\Program Files (x86)\Python38-32\python.exe" "C:/Users/Ours/OneDrive - unibuc.ro/Desktop/Test\_Python/Test\_curs.py"

Numarul maxim de apeluri recursive initial: 1000  
 Numarul maxim de apeluri recursive modificat: 5000  
 1000! = 4023872600770937735437024339230039857193748642107146325437999104299385123986298028592044208486969484800479988610197196058631666872994888558901323

Process finished with exit code 0

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

a) *Șirul lui Fibonacci* este definit prin următoarea relație de recurență:

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

O implementare directă a acestei relații, care va furniza valoarea termenului de rang  $n$  a șirului lui Fibonacci, este următoarea funcție recursivă:

```

def fibo(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 și crește în raport cu valoarea lui  $n$ . În capitolul următor, dedicat complexității computaționale, se va demonstra faptul că numărul de apeluri recursive efectuate este exponențial în raport cu  $n$ , aceasta fiind cauza timpului mare de executare. O implementare eficientă se poate obține iterativ:

```

def fibo(n):
    if n <= 1:
        return n

    a, b = 0, 1
    for i in range(n):
        a, b = b, a+b
    return a

```

- b) *Algoritmul lui Euclid* permite determinarea celui mai mare divizor comun a două numere întregi nenule  $a$  și  $b$  prin împărțiri repetate, respectiv: cât timp restul împărțirii lui  $a$  la  $b$  este nenul înlocuim  $a$  cu  $b$  și  $b$  cu restul împărțirii lui  $a$  la  $b$ , iar ultimul rest nenul obținut va fi  $\text{cmmdc}(a, b)$ . De exemplu, pentru  $a = 120$  și  $b = 18$ , 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 obține  $\text{cmmdc}(120, 18) = 6$ . Se observă faptul că ultimul rest nenul este egal cu ultimul împărțitor, deci o variantă de implementare iterativă a acestui algoritm este următoarea:

```
def cmmdc(a, b):
    r = a % b
    while r != 0:
        a, b = b, r
        r = a % b
    return b
```

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

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

Implementarea acestei relații de recurență printr-o funcție recursivă este foarte simplă:

```
def cmmdc(a, b):
    if b == 0:
        return a
    return cmmdc(b, a % b)
```

Observăm faptul că funcțiile au complexități egale, respectiv numărul de iterații este aproximativ egal cu numărul de autoapeluri.

- c) *Calculul sumei cifrelor unui număr natural* se poate realiza folosind următoarea relație de recurență:

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

Implementarea acestei relații de recurență printr-o funcție recursivă este banală:

```
def sc(n):
    if n < 10:
        return n
    return n%10 + sc(n//10)
```

Ce relație există între numărul de autoapeluri din varianta recursivă și numărul de iterații din varianta iterativă?

- d) *Suma elementelor dintr-o listă* se poate defini recursiv ca fiind suma dintre primul element al listei și suma elementelor din restul listei dacă lista este nevidă, respectiv 0 dacă lista este vidă:

```
def suma_lista(L):
    if len(L) == 0:
        return 0
    return L[0] + suma_lista(L[1:])
```

Într-un mod similar se poate calcula și suma elementelor strict pozitive dintr-o listă:

```
def sumapoz_lista(L):
    if len(L) == 0:
        return 0
    if L[0] > 0:
        return L[0] + sumapoz_lista(L[1:])
    else:
        return sumapoz_lista(L[1:])
```

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

```
def frecventa(litera, sir):
    if len(sir) == 0:
        return 0
    return int(sir[0] == litera) + frecventa(litera, sir[1:])
```

- f) 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 obține prin șirul de operații  $4 \rightarrow 40 \rightarrow 404 \rightarrow 202 \rightarrow 101$ , iar numărul 133 prin șirul de operații  $4 \rightarrow 2 \rightarrow 24 \rightarrow 12 \rightarrow 6 \rightarrow 64 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 84 \rightarrow 42 \rightarrow 424 \rightarrow 212 \rightarrow 106 \rightarrow 1064 \rightarrow 532 \rightarrow 266 \rightarrow 133$ .

Pentru a afișa șirul de operații prin care se poate obține un număr natural nenul din numărul 4, vom aplica asupra sa operațiile inverse operațiilor date:

- 1'. înmulțirea numărului cu 2;
- 2'. eliminarea ultimei cifre, dacă aceasta este 4;
- 3'. eliminarea ultimei cifre, dacă aceasta este 0.

De exemplu, pentru numărul 101 vom obține următorul șir de operații  $101 \rightarrow 202 \rightarrow 404 \rightarrow 40 \rightarrow 4$ .

```
def numar4(n):
    if n != 4:
        if n % 10 == 0 or n%10 == 4:
            numar4(n//10)
        else:
            numar4(2*n)
    print(" ->", n, end="")
else:
    print(4, end="")
```

Complexitatea funcțiilor recursive se exprimă în raport de numărul apelurilor efectuate pentru o anumită dimensiune a datelor de intrare. O modalitate pentru determinarea efectivă a complexității unei funcții recursive o vom prezenta în capitolul următor, dedicat tehnicii de programare *Divide et Impera*.

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