

# **Tehnici de elaborare a algoritmilor**

## **Iterativitatea și recursivitatea**

**Furculiță Andreea, clasa a 11-a "C"**

**Profesor: Guțu Maria**

**2020**

# CUPRINS:

1. Introducere.....	3
2. Iterativitatea .....	3
2.2 Informații generale .....	3
Instrucțiunea <i>for</i> .....	3
Instrucțiunea <i>while</i> .....	4
Instrucțiunea <i>repeat</i> .....	4
Avantaje și dezavantaje .....	5
Avantaje: .....	5
Dezavantaje: .....	5
Exemple de programe .....	5
3. Recursivitatea.....	8
Informații generale .....	8
Avantaje și dezavantaje .....	9
Avantaje: .....	9
Dezavantaje: .....	9
Exemple de programe .....	9
4. Concluzii .....	12
Bibliografie: .....	13

# 1. Introducere

Un limbaj de programare nu poate fi complet dacă prin intermediul său nu pot fi rezolvate probleme, în cadrul cărora nr de operații nu este static (de exemplu, în cazul găsirii sumei unui vector constând din **n** elemente, unde **n** nu este cunoscut la momentul scrierii programului). Atât în Pascal, cât și în alte limbaje de programare, acest lucru este realizat prin două mecanisme de bază: iterația și recursia. [11] Aceste două mecanisme stau la bază a două tipuri de algoritmi - iterativi și recursivi.

Teoretic, orice algoritm ce este iterativ, poate fi transformat în unul recursiv și invers. Practic însă, sunt anumite probleme ce au o natură fie iterativă fie recursivă. Algoritmii ce rezolvă aceste probleme pot fi exprimați mai ușor într-o reprezentare și mult mai complicat în cealaltă. [5]

## 2. Iterativitatea

### 2.2 Informații generale

Iterația reprezintă repetarea unei secvențe de instrucțiuni de un număr specificat de ori sau până la îndeplinirea unei condiții [6]. Mecanismul dat este utilizat atunci când este necesară realizarea unei anumite acțiuni de mai multe ori pe baza anumitor condiții [4]. Mecanismul clasic de construcție a iterațiilor într-un cod de programare se face prin așa-numitele cicluri. Algoritmii iterativi se opresc când condiția ciclului devine falsă. Un ciclu poate fi infinit dacă condiția de continuare a ciclului este mereu adevărată. [3]

Pascalul prezintă următoarele structuri iterative: ciclul **for**, ciclul **while... do** și ciclul **repeat... until**. [7]

### Instrucțiunea **for**

Instrucțiunea **for** indică execuția repetată a unei instrucțiuni, care poate fi și compusă, în funcție de valoarea unei variabile de control. Sintaxa instrucțiunii în studiu este:

*<Instrucțiune for> ::= **for** <Variabilă> := <Expresie> <Pas> <Expresie> **do** <Instrucțiune>;*

*<Pas> ::= **to** | **downto**.*

Variabila situată după cuvântul-cheie **for** se numește **variabilă de control** sau **contor**. Această variabilă trebuie să fie de tip ordinal.

Valorile expresiilor din componența instrucțiunii **for** trebuie să fie compatibile, în aspectul atribuirii, cu tipul variabilei de control.

Instrucțiunea situată după cuvântul-cheie **do** se execută pentru fiecare valoare din domeniul determinat de valoarea inițială și de valoarea finală.

Dacă instrucțiunea **for** utilizează pasul **to**, valorile variabilei de control sunt incrementate la fiecare repetiție, adică se trece la succesorul valorii curente. Dacă valoarea inițială este mai mare decât valoarea finală, instrucțiunea situată după cuvântul-cheie **do** nu se execută niciodată.

Dacă instrucțiunea **for** utilizează pasul **downto**, valorile variabilei de control sînt decrementate la fiecare repetiție, adică se trece la predecesorul valorii curente. Dacă valoarea inițială este mai mică decât valoarea finală, instrucțiunea situată după cuvântul-cheie **do** nu se execută niciodată.

Valorile variabilei de control nu pot fi modificate în interiorul ciclului.

### Instrucțiunea *while*

Instrucțiunea **while** conține o expresie booleană care controlează execuția repetată a altei instrucțiuni, care poate fi și una compusă. Sintaxa instrucțiunii în studiu este:

*<Instrucțiune while> ::= while <Expresie booleană> do  
<Instrucțiune>.*

Instrucțiunea situată după cuvântul-cheie **do** se execută repetat atâta timp, cât valoarea expresiei booleene este true. Dacă expresia booleană ia valoarea false, instrucțiunea de după **do** nu se mai execută.

Instrucțiunea **while** se consideră deosebit de utilă în situația în care numărul de execuții repetate ale unei secvențe de instrucțiuni este dificil de evaluat.

### Instrucțiunea *repeat*

Instrucțiunea **repeat** indică repetarea unei secvențe de instrucțiuni în funcție de valoarea unei expresii booleene. Sintaxa instrucțiunii este:

*<Instrucțiune repeat> ::= repeat <Instrucțiune>  
{; <Instrucțiune>} until <Expresie booleană>.*

Instrucțiunile situate între cuvintele-cheie **repeat** și **until** se execută repetat atât timp cât expresia booleană este falsă. Când această expresie devine adevărată, se trece la instrucțiunea următoare. Instrucțiunile aflate între **repeat** și **until** vor fi executate cel puțin o dată, deoarece evaluarea expresiei logice are loc după ce s-a executat această secvență.

În mod obișnuit, instrucțiunea **repeat** se utilizează în locul instrucțiunii **while** atunci când evaluarea expresiei care controlează repetiția se face după executarea secvenței de repetat. Aceasta este utilă în situația în care numărul de executări repetate ale unei secvențe de instrucțiuni este dificil de evaluat. [2]

## Avantaje și dezavantaje

### Avantaje:

- Necesarul de memorie este mai mic decât în cazul recursivității;
- Testarea și depanarea programelor este simplă; [1]
- Un algoritm iterativ poate fi mai rapid decât echivalentul său recursiv, din cauza eforturilor adiționale, precum apelarea funcțiilor și înregistrarea stivelor în mod repetat, prezente în cazul algoritmului recursiv. [3]

### Dezavantaje:

- Structura programului poate fi complicată;
- Volumul de muncă necesar pentru scrierea programului poate fi mare. [1]

## Exemple de programe

**1. Scrieți un program care ia un număr întreg pozitiv n și afișează toate numerele de la 1 până la n.**

**Exemplu: 5 → 12345.** [11]

```
Program iterativitate_1;
```

```
Var i, n : integer;
```

```
begin
```

```
  readln(n); {Se citește valoarea variabilei n de la tastatură}
```

```
  for i:= 1 to n do write(i); {Variabila i va lua, pe rând, câte o valoare de la 1 la n, cu pasul de 1, iar valorile variabilei vor fi afișate într-un rând, fără spații}
```

```
end.
```

**2. Scrieți un program care ia un număr întreg pozitiv n apoi calculează și afișează numărul cu poziția n din șirul lui Fibonacci. Primele două numere ale șirului vor fi definite ca 1 și fiecare număr ulterior reprezintă suma a două numere două anterioare, deci secvența este 1, 1, 2, 3, 5, 8, 13 ...**  
**Exemplu: n=6 => nr din șir este 8. [11]**

```
Program iterativitate_2;
var n : integer;
```

```
function fibonacci (n1:integer): integer;
var i, a, b, fib: integer;
begin
  a:=1; b:=1; fib:=1; i := 3; {a și b reprezintă primele 2 numere din șir,
  care sunt egale cu 1, iar variabilei fib i se atribuie valoarea 1 pentru
  cazurile în care poziția numărului este 1 sau 2. Variabila i arată
  poziția numărului și, întrucât pentru pozițiile 1 și 2 este specificată
  valoarea variabilei fib, i ia valori începând cu 3}
  while i <= n1 do
  begin
    fib := a + b;
    a := b;
    b := fib;
    inc(i);
  end; {În acest ciclu while are loc calcularea fiecărui număr din
  șir de la cel de pe poziția 3 până la cel de pe poziția necesară.
  Variabila b reprezintă predecesorul numărului de pe poziția i, iar
  variabila a-predecesorul predecesorului. Valoarea variabilei fib este
  reprezentată de suma variabilelor a și b, iar valorile acestor variabile
  sunt actualizate. La ieșirea din ciclu, variabila fib va conține rezultatul
  pentru i=n1}
  fibonacci := fib; {Funcției i se atribuie valoarea variabilei fib}
end;

begin
  readln(n); {Este citită poziția numărului}
  writeln('Al ',n, '-lea numar din sirul lui fibonacci este ', fibonacci(n));
  {Se afișează numărul de pe poziția data de la tastatură, fiind apelată
  funcția fibonacci}
end.
```

**3. Să se scrie un program care citește și afișează caracterul introdus de utilizator și se oprește când caracterul 'c' este introdus. [9]**

```
Program iterativitate_3;
Var c : char;
begin
repeat
    writeln('Introduceti un caracter. Introduceti 'c' pentru a opri
programul:');
    readln(c);
    write('Ati introdus: ');
    writeln(c); { Datorită instrucțiunii repeat, utilizatorului i se va cere
să introducă un caracter, iar la ecran se va afișa caracterul introdus,
până când utilizatorul va introduce caracterul 'c', condiția de la until
va deveni adevărată și se va ieși din ciclu}
until c = 'c';
writeln('Stop'); {La introducerea caracterului 'c' are loc ieșirea din
ciclu și se trece la următoarea instrucțiune, astfel încât se afișează
cuvântul "Stop"}
end.
```

**4. Scrieți un program care calculează și afișează rezultatul operației  $x$  la puterea  $n$ , unde  $x$  este un număr real, iar  $n$ -unul întreg.**

```
Program iterativitate_4;
Var x, r:real;
n, i: integer;
begin
    write('x='); readln(x); {Are loc citirea datelor de la tastatură}
    write('n='); readln(n);
    r:=1; {Variabilei r i se atribuie valoarea 1, care nu modifică
rezultatul înmulțirii cu x, astfel încât pentru i=1 r=x}
    for i:=1 to n do r:=r*x; {În dependență de puterea aleasă, variabila
r, ce are la început valoarea 1, se va înmulți cu x de n ori, obținându-
se  $x^n$  }
    writeln('x^n=', r); {Are loc afișarea rezultatului}
end.
```

**5. Scrieți un program care va calcula suma elementelor unui tablou cu  $n$  elemente.**

```
Program iterativitate_5;
```

```

type tablou=array[1..100] of integer;
var a:tablou;
i,n,s:integer;
begin
    writeln('Dati n'); readln(n); {Are loc citirea numărului de elemente
ale tabloului}
    s:=0; {Datorită faptului că s are la început valoarea 0, nu este
modificată suma elementelor, obținându-se un rezultat corect}
    for i:= 1 to n do begin { Datorită proprietăților instrucțiunii for,
operațiunile din corpul ciclului vor fi repetate exact n ori, ceea ce ne
permite să accesăm fiecare valoare din tablou}
        write('a['i,']='); readln(a[i]); {Are loc citirea elementelor tabloului
pentru fiecare poziție, de fiecare data când variabila i își schimbă
valoarea și se reia secvența de instrucțiuni}
        s:=s+a[i]; {Se adună la 0 elementul de pe fiecare poziție pe rând, în
dependență de valoarea lui i, variabila s schimbându-și valoarea, iar
ultima valoare fiind rezultatul final}
    end;
    writeln('Suma este ', s); {Are loc afișarea rezultatului}
end.

```

### 3. Recursivitatea

#### Informații generale

Recursia se definește ca o situație în care un subprogram se autoapelează fie direct, fie prin intermediul altei funcții sau proceduri. Subprogramul care se autoapelează se numește recursiv. [1]

Scrierea unui algoritm recursiv deseori începe prin specificarea **“cazului elementar”** - cazul în care nu se apelează recursiv funcția. Algoritmii recursivi se opresc când “cazul elementar” este întâmpinat. [3]

Astfel, regula de consistență în elaborarea unui program recursiv este următoarea:

Definirea corectă a unui algoritm recursiv presupune că în procesul derulării calculelor trebuie să existe:

- cazuri elementare, care se rezolvă direct;
- cazuri care nu se rezolvă direct, însă procesul de calcul în mod obligatoriu progresează spre un caz elementar. [1]



Spre deosebire de algoritmi iterativi, algoritmi recursivi nu pot fi executați la infinit, deoarece mai devreme sau mai târziu programul nu va mai avea spațiu de memorie disponibil pentru stiva unui nou apel. [3]

Recursivitatea se numește **directă** atunci când în corpul subprogramului apare apelul acelui subprogram și **indirectă** atunci când apelul apare în corpul altui subprogram care este apelat de către primul subprogram. [8]

## Avantaje și dezavantaje

Avantaje:

- Structura programelor este mai simplă decât a celor iterative;
- Volumul de muncă necesar pentru scrierea programului este mic.

Dezavantaje:

- Necesarul de memorie este mare;
- Testarea și depanarea programelor este complicată. [1]

Algoritmi recursivi sunt mult mai avantajoși decât cei iterativi, dacă cerințele problemei sunt formulate recursiv. [10]

## Exemple de programe

**1. Scrieți un program care ia un număr întreg pozitiv  $n$  apoi calculează și afișează numărul cu poziția  $n$  din șirul lui Fibonacci. Primele două numere ale șirului vor fi definite ca 1 și fiecare număr ulterior reprezintă suma a două numere două anterioare, deci secvența este 1, 1, 2, 3, 5, 8, 13 ...**  
**Exemplu:  $n=6 \Rightarrow$  nr din șir este 8. [11]**

```
Program recursie_1;  
var n : integer;
```

```
function fibonacci(n1: integer): integer;  
begin  
    if (n1=1) or (n1=2) then fibonacci :=1; {Funcției i se atribuie  
    valoarea de 1 pentru  $n_1=1$  și  $n_1=2$  (cazuri elementare)}  
    else begin fibonacci := fibonacci(n1-1) + fibonacci(n1-2); end;  
end; {Dacă  $n_1$  e diferit de 1 și de 2, atunci valoarea funcției se  
calculează sumând valoarea funcției date pentru predecesorul lui  $n_1$   
și cea a funcției date pentru predecesorul predecesorului, care, dacă  
nu reprezintă cazuri de bază se calculează, la rândul lor, în același
```

mod și tot așa, până se ajunge la suma funcțiilor pentru cazurile de bază, după care calculându-se și toate restul}

```
begin
readln(n); {Este citită poziția numărului}
writeln('Al ',n, '-lea numar din sirul lui fibonacci este ', fibonacci(n));
{Se afișează numărul de pe poziția data de la tastatură, fiind apelată
funcția fibonacci}
end.
```

## **2. Scrieți un program recursiv care calculează suma $S(n)=1+3+5+\dots+(2n-1)$ . [1]**

```
Program recursie_2;
type Natural_nenul=1..MaxInt;
var n:Natural_nenul;

Function S(n1:Natural_nenul):Natural_nenul;
begin
  if n1=1 then S:=1 {Are loc specificarea cazului elementar, care se
                    rezolvă direct}
  else S:=S(n1-1)+(2*n1-1); {Funcția se autoapelează direct. Se
ajunge la suma S(2), ce poate fi calculată cu ajutorul cazului
elementar și apoi se calculează, pe rând, valorile funcțiilor S(3),
S(4),..., până se ajunge la valoarea sumei S(n1)}
end;

begin
  writeln('Dati n');
  readln(n); {Are loc citirea lui n}
  writeln('Suma este ', S(n)); {Se afișează suma, apelându-se funcția
S}
end.
```

## **3. Scrieți un program care calculează produsul $P(n)=1 \times 4 \times 7 \times \dots \times (3n-2)$ . [1]**

```
Program recursie_3;
type Natural_nenul=1..MaxInt;
var n:Natural_nenul;

Function P(n1:Natural_nenul):Natural_nenul;
begin
```

```

    if n1=1 then P:=1 {Are loc specificarea cazului elementar, care se
                        rezolvă direct}
    else P:=P(n1-1)*(3*n1-2); {Funcția se autoapelează direct. Se
    ajunge la produsul P(2), ce poate fi calculat cu ajutorul cazului
    elementar și apoi se calculează, pe rând, valorile funcțiilor P(3),
    P(4),..., până se ajunge la valoarea produsului P(n1)}
end;

begin
    writeln('Dati n');
    readln(n); {Are loc citirea lui n}
    writeln('Produsul este ', P(n)); {Se afișează suma, apelându-se
    funcția S}

end.

```

#### **4. Scrieți un program recursiv care calculează produsul $P(n)=2 \times 4 \times 6 \times 2n$ . [1]**

```

Program recursie_4;
type Natural_nenul=1..MaxInt;
var n:Natural_nenul;

Function P(n1:Natural_nenul):Natural_nenul;
begin
    if n1=1 then P:=2 {Are loc specificarea cazului elementar, care se
                        rezolvă direct}
    else P:=P(n1-1)*2*n1; {Funcția se autoapelează direct. Se ajunge la
    produsul P(2), ce poate fi calculat cu ajutorul cazului elementar și
    apoi se calculează, pe rând, valorile funcțiilor P(3), P(4),..., până se
    ajunge la valoarea produsului P(n1)}
end;

begin
    writeln('Dati n');
    readln(n); {Are loc citirea lui n}
    Writeln('Produsul este ', P(n)); {Se afișează suma, apelându-se
    funcția S}

end.

```

#### **5. Scrieți un program recursiv care inversează un șir de caractere. [1]**

```

Program recursie_5;

```

```

type Natural_nenul=1..MaxInt;
var s:string;

Function Invers(s1:string):string;
begin
    if length(s1)=1 then Invers:=s1 {Are loc specificarea cazului
                                    elementar}
    else Invers:=s1[length(s1)]+Invers(copy(s1,1,length(s1)-1));
    {Funcția se autoapelează direct. Se ajunge la calcularea inversului
    șirului cu lungimea egală cu 2, care poate fi calculată cu ajutorul
    cazului elementar, după care se calculează, pe rând, inversele pentru
    șirurile cu încă un caracter, adăugându-se în față acel caracter, până
    se ajunge la inversul șirului introdus}
    end;

begin
    writeln('Dati un sir de caractere');
    readln(s); {Are loc citirea șirului de caractere}
    writeln('Sirul invers este ', Invers(s)); {Se afișează șirul inversat,
    apelându-se funcția Invers}
end. {Programul dat consumă multă memorie}

```

## 4. Concluzii

Iterativitatea asigură repetarea unor instrucțiuni de un anumit număr de ori în baza unor condiții, iar recursivitatea reprezintă situația în care un subprogram se autoapelează direct sau indirect, având loc, de asemenea, repetarea unor acțiuni.

Ambele clase de algoritmi au avantajele și dezavantajele sale. Nu putem afirma faptul că algoritmi iterativi sunt mai buni decât cei recursivi sau invers. Utilizarea lor depinde de fiecare problemă în parte.

Vom folosi algoritmi recursivi când natura problemelor este recursivă. Un algoritm recursiv poate fi transformat în unul iterativ și invers. Totuși, trebuie să alegem metoda cea mai optimală analizând necesarul de memorie al algoritmilor, timpul de execuție, structura programului, volumul de muncă necesar pentru scriere și gradul de complicitate al depanării algoritmilor.

## Bibliografie:

1. Anatol Gremalschi "Informatică. Manual pentru clasa a 11-a";
2. Anatol Gremalschi, Iurie Mocanu, Ion Spinei "Informatică. Manual pentru clasa a 9-a";
3. "Differences between iterative and recursive algorithms"  
<https://www.codeit-project.eu/differences-between-iterative-and-recursive-algorithms/>;
4. Haberman, B., H. Averbuch. "The Case of Base Cases: Why Are They So Difficult to Recognize? Student Difficulties with Recursion. – In: Proc. of 7th Annual Conference on Innovation and Technology in Computer Science Education, ACM, New York, 2002, pp. 84-88.";
5. "Iteration vs recursion"  
[https://www.ocf.berkeley.edu/~shidi/cs61a/wiki/Iteration\\_vs\\_recursion](https://www.ocf.berkeley.edu/~shidi/cs61a/wiki/Iteration_vs_recursion);
6. Meriam-Webster Dictionary. <http://www.merriam-webster.com>;
7. M. S. Schmalz "PASCAL Programming: § 4: Selection and Iteration Structures"  
<https://www.cise.ufl.edu/~mssz/Pascal-CGS2462/ifs-and-loops.html>.
8. "Recursivitatea"  
<https://informaticacnet.wordpress.com/2013/06/18/recursivitatea/>;
9. "Recursivitate"  
<http://staff.cs.upt.ro/~chirila/teaching/upt/id12-sda/lectures/ID-SDA-Cap5.pdf>;
10. "Tema 3. Avantajele și dezavantajele utilizării recursivității",  
<http://muhaz.org/nvmntul-profesional-si-tehnic-n-domeniul-tic.html?page=3>;
11. Vladimir Sulov "Iteration vs Recursion in Introduction to Programming Classes: An Empirical Study".