Tehnici de elaborare a algoritmilor

Iterativitatea și recursivitatea

Furculiță Andreea, clasa a 11-a “C”

Profesor: Guțu Maria

2020

CUPRINS:

1. Introducere

Un limbaj de programare nu poate fi complet dacă prin intermediul său nu pot fi rezolvate probleme, în cadrul cărora nr de operaţii 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 şi în alte limbaje de programare, acest lucru este realizat prin două mecanisme de bază: iteraţia şi recursia. [11] Aceste două mecanisme stau la bază a două tipuri de algoritmi - iterativi şi recursivi.

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

2. Iterativitatea

2.2 Informaţii generale

Iteraţia reprezintă repetarea unei secvenţe de instrucţiuni de un număr specificat de ori sau până la îndeplinirea unei condiţii [6]. Mecanismul dat este utilizat atunci când este necesară realizarea unei anumite acţiuni de mai multe ori pe baza anumitor condiţii [4]. Mecanismul clasic de construcţie a iteraţiilor într-un cod de programare se face prin aşa-numitele cicluri. Algoritmii iterativi se opresc când condiţia ciclului devine falsă. Un ciclu poate fi infinit dacă condiţia de continuare a ciclului este mereu adevărată. [3]

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

Instrucţiunea *for*

Instrucţiunea **for** indică execuţia repetată a unei instrucţiuni, care poate fi şi compusă, în funcţie de valoarea unei variabile de control. Sintaxa instrucţiunii în studiu este:

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

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

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

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

Instrucţiunea situată după cuântul-cheie **do** se execută pentru fiecare valoare din domeniul determinat de valoarea iniţială şi de valoarea finală.

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

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

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

Instrucţiunea *while*

Instrucţiunea **while** conţine o expresie booleană care controlează execuţia repetată a altei instrucţiuni, care poate fi şi una compusă. Sintaxa instrucţiunii în studiu este:

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

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

Instrucţiunea **while** se consideră deosebit de utilă în situaţia în care numărul de execuţii repetate ale unei secvenţe de instrucţiuni este dificil de evaluat.

Instrucţiunea *repeat*

Instrucţiunea **repeat** indică repetarea unei secvenţe de instrucţiuni în funcţie de valoarea unei expresii booleene. Sintaxa instrucţiunii este:

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

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

În mod obişnuit, instrucţiunea **repeat** se utilizează în locul instrucţiunii **while** atunci când evaluarea expresiei care controlează repetiţia se face după executarea secvenţei de repetat. Aceasta este utilă în situaţia în care numărul de executări repetate ale unei secvenţe de instrucţiuni este dificil de evaluat. [2]

Avantaje şi dezavantaje

Avantaje:

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

Dezavantaje:

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

Exemple de programe

1. Scrieţi un program care ia un număr întreg pozitiv n şi afişează toate numerele de la 1 până la n.
Exemplu: 5 → 12345. [11]

```
Program iterativitate_1;
Var i, n : integer;
begin
   readln(n); {Se citeşte 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 afişate într-un rând, fără spaţii}
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 instrucţiunii repeat, utilizatorului i se va cere să introducă un caracter, iar la ecran se va afişa caracterul introdus, până când utilizatorul va introduce caracterul 'c', condiţia de la until va deveni adevărată şi se va ieşi din ciclu}
until c = 'c';
writeln('Stop'); {La introducerea caracterului 'c' are loc ieşirea din ciclu şi se trece la următoarea instrucţiune, astfel încât se afişează 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 înmulţirii cu x, astfel încât pentru i=1 r=x}
   for i:=1 to n do r:=r*x; {În dependenţă de puterea aleasă, variabila r, ce are la început valoarea 1, se va înmulţi cu x de n ori, obţinându-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 defineşte ca o situaţie în care un subprogram se autoapelează fie direct, fie prin intermediul altei funcţii sau proceduri. Subprogramul care se autoapelează se numeşte recursiv. [1]

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

Astfel, regula de consistenţă 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 algoritmii iterativi, algoritmii recursivi nu pot fi executaţi la infinit, deoarece mai devreme sau mai târziu programul nu va mai avea spaţiu de memorie disponibil pentru stiva unui nou apel. [3]

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

Avantaje şi dezavantaje

Avantaje:

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

Dezavantaje:

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

Algoritmii recursivi sunt mult mai avantajoşi decât cei iterativi, dacă cerinţele problemei sunt formulate recursiv. [10]

Exemple de programe

1. Scrieţi un program care ia un număr întreg pozitiv n apoi calculează şi afişează numărul cu poziţia n din şirul lui Fibonacci. Primele două numere ale şirului vor fi definite ca 1 şi fiecare număr ulterior reprezintă suma a două numere două anterioare, deci secvenţa este 1, 1, 2, 3, 5, 8, 13 ...
Exemplu: n=6 => nr din şir este 8. [11]

```
Program recursie_1;
var n : integer;

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

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

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

2. Scrieţi un program recursiv care calculează suma S(n)=1+3+5+...+(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); {Funcţia se autoapelează direct. Se
ajunge la suma S(2), ce poate fi calculată cu ajutorul cazului
elementar şi apoi se calculează, pe rând, valorile funcţiilor 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 funcţia
S}
end.
```

3. Scrieţi un program care calculează produsul P(n)=1x4x7x...x(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); {Funcţia se autoapelează direct. Se
ajunge la produsul P(2), ce poate fi calculat cu ajutorul cazului
elementar şi apoi se calculează, pe rând, valorile funcţiilor 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 afişează suma, apelându-se
funcţia S}

end.
```

4. Scrieţi un program recursiv care calculează produsul P(n)=2x4x6x2n. [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; {Funcţia se autoapelează direct. Se ajunge la
produsul P(2), ce poate fi calculat cu ajutorul cazului elementar şi
apoi se calculează, pe rând, valorile funcţiilor 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 afişează suma, apelându-se
funcţia S}
end.
```

5. Scrieţi un program recursiv care inversează un şir 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 algoritmii 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;
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 şi dezavantajele utilizării recursivităţii”, 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”.