ABORDĂRI METODICE PRIVIND IMPLEMENTAREA UNOR TEHNICI DE PROGRAMARE PRIN PRISMA COMPLEXITĂŢII ALGORITMILOR

***Angela GLOBA*, Universitatea de Stat Tiraspol**
angelagloba@gmail.com

Rezumat. În articolul respectiv sunt expuse abordări didactice privind predarea unor subiecte din cursul universitar "Tehnici de programare". Sunt evidenţiate avantajele rezolvării unei probleme prin mai multe metode, aplicând diferite tehnici de programare şi structuri de date.
Cuvinte-cheie: tehnici de programare.
Abstract. The current article presents didactic approaches concerning subjects at "Programing Techniques" course. There are highlighted some advantages in solving a problem by various methods, applying different programing techniques and data structures.
Keywords: programing techniques.

1. Introducere

Anatole France[1] afirma că „dibăcia învăţătorului nu este decât de a trezi curiozitatea minţilor tinere, ca să le potolească apoi această curiozitate, pe care numai fiinţele umane fericite o au vie şi sănătoasă. Cunoştinţele vârâte cu de-a sila în minte o astupă şi o înăbuşă. Ca să mistui ştiinţa trebuie s-o fi înghiţit cu poftă."

Misiunea centrală a unui profesor universitar este de a contribui la formarea specialiştilor calificaţi [1-4]. În contextul respectiv vom examina unele metode şi procedee didactice privind pregătirea viitorilor profesori şcolari care, credem noi, ulterior vor educa şi dezvolta noi generaţii de tineri capabili şi competitivi atât pe piaţa autohtonă, cât şi cea internaţională.

Fiind într-o veşnică competiţie cu timpul, cu noile tehnologii şi cu noile oportunităţi, profesorul şcolar este obligat să ţină pasul cu aceste noi provocări. Contribuind în mod activ la formarea intelectuală a noii generaţii, profesorul şcolar trebuie să ştie cum să motiveze copii talentaţi, cum să selecteze copii după aptitudini şi cum să lucreze cu fiecare copil dotat, astfel ca posibilităţile lui intelectuale să fie dezvoltate la maxim. Mai jos vom examina aplicarea unor tehnici de programare soluţionând probleme cu diverse grade de dificultate.

II. Despre complexitatea algoritmilor şi aplicarea tehnicilor de programare

Desfăşurarea concursurilor naţionale şi internaţionale la Informatică au drept obiectiv principal descoperirea noilor talente în domeniul informaticii. Pregătirea pentru competiţii şi participarea la ele joacă un rol important în însuşirea temeinică şi profundă a

[1] Poet, journalist, nuvelist francez; Jacques Anatole François Thibault, laureat al Premiului Nobel pentru Literatură în 1921.

cunoştinţelor, ele contribuie la dezvoltarea şi desăvârşirea intereselor cognitive şi a activităţii creatoare şi contribuie, de asemenea, la lărgirea orizontului candidatului. Un elev ori student, care participă la un concurs naţional sau internaţional de informatică, trebuie să cunoască bine sau chiar foarte bine tehnicile de programare şi să posede cunoştinţe vaste în domeniul matematicii [1]. În dependenţă de tehnica de programare aleasă, el poate soluţiona eficient problema examinată.

Astfel, sunt tehnici de programare care oferă:

a) posibilitatea scrierii unui program mic, dar de cele mai dese ori greu de parcurs (pas cu pas) şi care încarcă considerabil memoria (recursia);
b) toate soluţiile unei probleme necesită un timp de calcul mare (backtracking);
c) poate fi aplicată o soluţie rapidă, dar nu de fiecare dată a celei mai bune, optime (greedy);
d) există o soluţie optimă, în timp de calcul util, dar care cere de la programator cunoştinţe matematice vaste (programarea dinamică).

Aceste afirmaţii pot fi demonstrate şi prin rezolvarea pas cu pas a mai multor exemple. Să examinăm mai jos unele modele de soluţionare.

Exemplul 1. *Problema rucsacului. Se consideră un vector de dimensiune n* $A=\{a_1,a_2,...,a_n\}$ *cu elemente numere întregi pozitive distincte şi un număr întreg pozitiv S. Se cere de găsit acele elemente* a_i *din A a căror sumă este S.*

Rezolvare. De exemplu, pentru $A=\{2,5,8,12,34,15\}$ şi $S=25$ în calitate de soluţii pot fi (5,8,12) sau (2,8,15) ş.a.m.d.

În principiu, o soluţie a problemei poate fi găsită parcurgând sistematic toate submulţimile lui *A* şi verificând dacă suma elementelor lor este *S*. Evident, se va aplica tehnica backtracking, recursiv. Pentru $n=6$ vor fi verificate $2^6-1=63$ submulţimi (fără mulţimea vidă), valoare acceptabilă ca timp de calcul.

Ce se va întâmpla dacă *A* are câteva sute sau chiar mii de componente? Se ştie că problema rucsacului este NP – completă şi pentru *n* destul de mare nu poate fi rezolvată în timp de calcul real.

Sunt clase de probleme ale rucsacului uşor de rezolvat, una din ele o formează vectorii cu creştere mare. Această problemă stă la baza sistemului de criptare Merkle - Hellman: primul sistem definit cu cheie publică.

Definiţie. Vectorul rucsac $A=\{a_1,a_2,...,a_n\}$ se numeşte vector cu creştere mare dacă $a_m \geq \sum_{t=1}^{m-1} a_t, m \geq 2$.

Exemplul 2. *Se consideră un vector de dimensiune n (*$n \leq 30$*)* $A=\{a_1,a_2,...,a_n\}$ *cu elemente numere întregi pozitive distincte şi un număr întreg pozitiv S. Se ştie că elementele acestui vector, fiind aranjate în ordine ascendentă, formează un vector cu creştere mare. Se cere de găsit acele elemente* a_i *din A a căror sumă este maxim posibilă şi nu întrece pe S. Datele de intrare se vor citi din fişierul text date.in, iar rezultatele se vor scrie în fişierul text date.out.*

Rezolvare.

În acest caz, pentru a rezolva problema rucsacului este suficient să fie realizaţi următorii paşi:

1. se sortează crescător vectorul *A*; se va aplica metoda de sortare QuikSort – una din cele mai rapide metode de sortare;
2. se parcurge vectorul *A* de la dreapta spre stânga;
3. cunoscând valoarea *S*, se cercetează mai întâi valoarea de adevăr a relaţiei $a_n \le S$;
4. dacă valoarea relaţiei este *false*, atunci a_n nu poate aparţine sumei pe care o căutăm şi se trece la examinarea elementului a_{n-1};
5. dacă valoarea relaţiei este *true*, atunci a_n trebuie să fie în suma pe care o căutăm, deoarece toate elementele a_i rămase nu pot depăşi în sumă pe *S*; $S \leftarrow S - a_n$;
6. ş.a.m.d., algoritmul se va opri la valoarea a_1.

Pentru realizarea acestui algoritm se va defini $S = \begin{cases} S, a_i > S \\ S - a_i, a_i \le S \end{cases}, \quad i = n,...,1$

Algoritmul (Greedy) implementat în limbajul de programare Pascal este:

```
Program p1;
var x:array[1..30] of longint;
   i,n:byte;
   s:longint;
Procedure quick(s,d:integer);
var a,b,t,tm:integer;
begin
 a:=s;
 b:=d;
  repeat
   while x[a]<x[b] do b:=b-1;
    t:=x[a];
    x[a]:=x[b];
    x[b]:=t;
    a:=a+1; tm:=1;
     if a<b then
      begin
       while x[a]<x[b] do a:=a+1;
       t:=x[b]; x[b]:=x[a]; x[a]:=t;
       b:=b-1; tm:=0;
      end;
   until b<=a;
   if s<a-tm then quick(s,a-tm);
   if a-tm+1<d then quick(a-tm+1,d);
end;
```

```
Begin
  assign(f,'date.in');
  reset(f);readln(f,n,s);
  assign(g,'date.out');
  rewrite(g);
  for i:=1 to n do read(x[i]);
  close(f);
  quick(1,n);
  for i:=n downto 1 do
   if x[i]<=s then
    begin
     s:=s-x[i];
     writeln(g,x[i],' ');
    end;
End.
```

Exemplul 3. *Se consideră un vector de dimensiune n* ($n \le 30$) $A=\{a_1, a_2, ..., a_n\}$ *cu elemente numere întregi pozitive distincte şi un număr întreg pozitiv S (*$1 \le S$, $a_i \le 2 \cdot 10^8$*). Se cere de găsit acele elemente* a_i *din A care formează un vector cu creştere mare şi a căror sumă este maxim posibilă, şi nu întrece pe S. Datele de intrare se vor citi din fişierul text date.in care va conţine pe prima linie numerele n şi S, iar pe fiecare linie i, la i = 1, 2, 3, ..., n, – elementele* a_i*. Fişierul text de ieşire date.out va conţine pe prima linie suma maximal apropiată de S, iar pe liniile următoare câte un element* a_i *aranjate în ordine crescătoare. Setul de date iniţiale admite o singură soluţie a problemei. Timpul de execuţie a programului nu va depăşi 0,1 secunde. De exemplu,*

date.in	date.out
6 40 14 2 5 20 18 8	39 5 14 20

Rezolvare.

Aplicând metoda backtracking, se poate obţine soluţia problemei, însă timpul de calcul va fi încălcat considerabil, lucru inacceptabil în cadrul concursurilor de informatică (programare). La fel, nu poate fi aplicat algoritmul greedy (un algoritm rapid ca timp de calcul), descris mai sus, el fiind acceptat numai în cazul când vectorul iniţial este cu creştere mare. Este adevărată afirmaţia: orice subvector al unui vector cu creştere mare este un vector cu creştere mare. De exemplu, pentru *S*=40 şi $A=\{2,5,8,14,18,20\}$ se va calcula suma maximal apropiată de *S*, care este egală cu 38 (sumă formată de elementele 18 şi 20), ceea

ce nu este adevărat. De fapt, suma maximal apropiată de *S* este egală cu 39 (sumă formată de elementele 5,14 şi 20).

Deoarece este menţionat că setul de date iniţiale admite o singură soluţie a problemei, se poate aplica tehnica programării dinamice, metoda înainte, în caz contrar problema este NP -completă şi nu posedă un algoritm de calcul în timp util.

Conform restricţiilor avem $1 \le S \le 2{,}1 \cdot 10^8$, $1 \le a_i \le 2{,}1 \cdot 10^8$, deci vom declara *S* şi vectorul *A* de tip longint. Se va utiliza alocarea statică a memoriei.

```
type multime=set of byte;
var A,suma,val_max:array[1..30] of longint;
   elem:array[1..30] of multime;
   n,i,j,k,delta:byte;
   max,S,mm:longint;
   f,g:text;
```

Elementele vectorului iniţial *A* se vor aranja în ordine crescătoare. Soluţia se va forma analizând elementele vectorului *A* de la dreapta spre stânga.

În vectorul suma se vor reţine sumele S_i maxime, care pot fi formate cu elementul a_i. Elementele vectorului val_max conţin valoarea maximă care poate fi adăugată la S_i pentru a obţine un vector cu creştere mare. Vectorul elem va contine indicii acelor elemente din *A*, care formează suma S_i.

De exemplu: pentru *S*=40 şi $A = \{2,5,8,14,18,20\}$ se va obţine:

i	A	suma	val_max	Explicaţie
6	20	20	20	Elementul căutat a_i, care poate fi adăugat cu suma[6] trebuie să fie mai mic ca 20.
5	18	38	2	Elementul căutat a_i, care poate fi adăugat cu suma[5] trebuie să fie mai mic ca 2.
4	14	34	6	Elementul căutat a_i, care poate fi adăugat cu suma[4] trebuie să fie mai mic ca 6.
3	8	28	8	Elementul căutat a_i, care poate fi adăugat cu suma[3] trebuie să fie mai mic ca 8.
2	5	39	1	Elementul căutat a_i, care poate fi adăugat cu suma[2] trebuie să fie mai mic ca 1.
1	2	36	2	Nu mai sunt elemente ale vectorului *A*, care pot fi adăugate la suma[1].

Elementele ce formează sumele S_i:

```
elem[1]= 1 4 6
elem[2]= 2 4 6
elem[3]= 3 6
elem[4]= 4 6
```

elem[5]= 5 6
elem[6]= 6
Soluţia:
39
5
14
20

```
Program p2;
type multime=set of byte;
var A,suma,val_max:array[1..30] of longint;
    elem:array[1..30] of multime;
    maxim,S,temp:longint;
    n,i,j,k,d:byte;
    f,g:text;

Begin
  assign(f,'date.in');assign(g,'date.out');
  reset(f);rewrite(g);
  readln(f,n,s);
  for i:=1 to n do
  begin
    readln(f,A[i]);
  end;
  close(f);

  for i:=1 to n do elem[n]:=[];

  for i:=1 to n-1 do
    for j:=i+1 to n do
      if A[i]>A[j] then
        begin
          temp:=A[i];
          A[i]:=A[j];
          A[j]:=temp;
        end;

  if A[n]<=S then
    begin
      suma[n]:=A[n];
      elem[n]:=[n];
      val_max[n]:=A[n];
```

```
   end else val_max[n]:=S;

  for k:=n-1 downto 1 do
   begin
    maxim:=A[k];d:=k;
    if A[k]<=S then
    begin
     for i:=k+1 to n do
      if A[k]<val_max[i] then
       begin
        temp:=A[k]+suma[i];
        if (temp<=S) and (temp>maxim) then
        begin
         maxim:=temp;d:=i;
        end;
       end;
      suma[k]:=maxim;
      elem[k]:=elem[d]+[k];
      val_max[k]:=abs(val_max[d]-A[k]);
      if val_max[k]>A[k] then val_max[k]:=A[k];
    end;
   end;

  maxim:=suma[1];j:=1;
  for i:=2 to n do
   if suma[i]>maxim then begin maxim:=suma[i];j:=i;end;

  writeln(g,maxim);

  for i:=1 to n do
   if i in elem[j] then writeln(g,A[i]);

  close(g);
End.
```

Complexitatea temporală a programului este de ordinul $O(n^2)$. Deoarece n=30, numărul de operaţii efectuate în program este destul de mic în comparaţie cu capacitatea de prelucrare a calculatoarelor personale din laboratorul de Informatică. Prin urmare, timpul de calcul cerut de program va fi cu mult mai mic decât 0,1 secunde.

Este suficient să modificăm în exemplul 3 limitele datelor de intrare $1 \le n \le 255$; $1 \le S \le 10^{255}$; $1 \le a_i \le 10^{255}$ şi lucrurile vor lua cu totul altă întorsătură. Tehnica de

programare utilizată rămâne aceeaşi: programarea dinamică, metoda înainte. Evident, nu vom opta pentru tehnica backtracking, deoarece pentru $n = 255$ sunt necesare de a verifica $2^{255} - 1 \approx 6 \cdot 10^{76}$ submulţimi, lucru inacceptabil din punctul de vedere al timpului de calcul.

Conform restricţiilor problemei $1 \le S \le 10^{255}$; $1 \le a_i \le 10^{255}$, putem deduce că astfel de numere nu se încadrează în tipul de date integer, deci se vor declara de tip string. Deoarece $1 \le n \le 255$; $1 \le S \le 10^{255}$; $1 \le a_i \le 10^{255}$, volumul de memorie necesar pentru a declara un vector cu 255 elemente de tip string este egal cu 255*255=65025 octeţi, volum inaccesibil pentru alocarea statică a memoriei. Nu se poate aloca memorie nici dinamic pentru acest vector. În acest caz se vor utiliza structuri dinamice de date şi anume - listele dublu înlănţuite. Poate fi utilizată următoarea structură:

```
type referinta=^Lista;
    Lista=record
     indice:byte;
     A,suma,val_max:string;
     anti,next:referinta;
    end;
```

Este cunoscut că nu se pot aduna două şiruri de caractere. De aceea pentru efectuarea operaţii aritmetice cu numere mari se vor defini de către programator operaţiile de adunare şi scădere utilizând string-uri. Compararea a două şiruri de caractere este foarte simplă: se compară codurile (numărul de ordine al simbolului dat în tabelul ASCII) caracterelor de pe aceeaşi poziţie; este mai mare acel şir de caractere al cărui cod respectiv este mai mare. Dacă se consideră două numere 1234 şi 643 definite cu ajutorul a două variabile de tip string $a = '1234'$ şi $b = '643'$, atunci rezultatul comparării $a < b$ va fi *true*. Rezultatul obţinut este incorect. În acest caz se vor defini două funcţii ce permit compararea numerelor mari definite cu ajutorul tipului de date string:

```
Function Comparare(x,y:string):boolean;
var temp:boolean;
begin
  temp:=false;
  if length(x)<length(y) then temp:=true;
  if (length(x)=length(y)) and (x<y) then temp:=true;
  Comparare:=temp;
end;

Function Comparare1(x,y:string):boolean;
var temp:boolean;
begin
  temp:=false;
  if length(x)<length(y) then temp:=true;
  if (length(x)=length(y)) and (x<=y) then temp:=true;
```

```
  Comparare1:=temp;
end;
```

Complexitatea algoritmului este de ordinul $O(n^2)$. Pentru $n = 255$, timpul de execuţie a programului va fi mai mic decât 2 secunde.

III. Unele concluzii şi recomandări

Abilităţile şi competenţele obţinute în cadrul cursului „Tehnici de programare" ţin de:

a) capacitatea de analiză a tehnicilor de sortare şi căutare în scopul implementării celei mai raţionale tehnici pentru o problemă concretă;
b) implementarea tehnicilor de programare studiate într-un limbaj de programare;
c) elaborarea algoritmilor care utilizează diverse structuri de date statice şi dinamice;
d) capacitatea de evaluare a algoritmilor după criteriul complexităţii lor şi alegerea celui mai bun algoritm corespunzător acestui criteriu.

Aceste finalităţi de studii îi va permite viitorului specialist să-şi construiască o carieră de succes şi să obţină, în viitor, rezultate de performanţă cu elevii săi. Aşadar un viitor profesor de informatică trebuie să fie pregătit pentru a lucra cu elevii dotaţi, trebuie să ştie cum să-i identifice şi cum să le dezvolte capacităţile în continuare.

BIBLIOGRAFIE:

1. L. Chiriac, A. Globa, N. Bobeică. *Procedee metodice privind pregătirea şi desfăşurarea olimpiadelor de informatică.* Mahematics & information technologies: research and education. Dedicated to the 65th anniversary of the Moldova State University, MITRE 2011, Chişinău, august 22-25, p.168-169.

2. A.Globa, *Aspecte metodice privind elaborarea algoritmilor de operare cu numere mari*. The 20 Conferance on Applied, and Industrial Matematics dedicated to academician Mitrofan M. Ciobanu, Chisinau, August 22-25, 2012.

3. I.Bolun, A.Globa, ş.a., *Olimpiada Republicană la Informatică. Ediţia 2014*. Chişinău, ASEM, 2014.

4. S. Focşa-Semionov, Învăţarea autoreglată. Teorie şi aplicaţii, Chişinău, Epigraf, 2010.