TEOREME de programare

Clara Ionescu

În numărul precedent al GInfo am trecut în revistă clasa de probleme în care prelucrarea unui șir de date se concretizează în determinarea unui singur rezultat (o sumă, un produs, o reuniune etc.). Dacă într-o astfel de problemă rezultatul cerut trebuie determinat în prezența unei anumite proprietăți, este posibil ca algoritmul prezentat să prevadă multe operații inutile. Teorema următoare elimină acest neajuns.

Decizia

Să căutăm trăsătura comună a următoarelor probleme!

P5. Să se stabilească dacă un număr dat este prim sau nu!
P6. Având la dispoziție șirul cuvintelor care desemnează lunile anului, să se stabilească despre un cuvânt dat dacă reprezintă o lună din an sau nu!
P7. Măsurând pe durata lunii iunie temperatura apei *Mării Negre* în dreptul stațiunii *Mamaia*, să se stabilească dacă apa mării s-a încălzit încontinuu sau nu!
P8. Cunoscând mediile generale ale elevilor dintr-o clasă, să se stabilească dacă un anumit elev este premiant sau nu (se află sau nu printre elevii care au primele trei medii din clasă)!

Trăsătura comună a acestor probleme constă în felul în care algoritmul va furniza răspunsul. Soluția va fi dată sub forma unui mesaj prin care se confirmă sau se infirmă proprietatea cerută. Acest mesaj de multe ori poate fi simplu: `'DA'` sau `'NU'` și se afișează în funcție de valoarea unei variabile logice, stabilită în algoritm.

Totuși, printre enunțurile de mai sus există și diferențe. De exemplu, problema **P5** cere verificarea unui singur număr. Evident, va trebui să studiem acest singur număr, astfel încât să putem decide dacă el este prim sau nu. Problema **P6** cere verificarea existenței unei valori printre mai multe valori date. Deci studiem mai multe date, dar acest "studiu" se reduce la o simplă comparare a elementului curent din șirul dat cu valoarea căutată. În problema **P7** verificăm o proprietate a unui întreg șir de date, deci trebuie să analizăm relația dintre valorile elementelor (ele trebuie să formeze un șir crescător). Problema **P8** cere verificarea apartenenței unei valori date la o parte a unui șir dat.

Să rezolvăm problema **P6** cu algoritmul prezentat în numărul precedent al revistei!

Începem cu faza preliminară în care stabilim variabilele și semnificațiile lor. Deocamdată ne "străduim" să respectăm descrierea algoritmului cunoscut:

n: *întreg* {*numărul elementelor șirului de prelucrat*}
x: *tablou(1..n): tip element* {*elementele șirului dat*}
S: *logic* {*rezultatul*}

În plus, vom folosi o funcție logică `P` în care vom verifica proprietatea cerută. Parametrul funcției `P` va fi un element al șirului, iar subalgoritmul de rezolvare a problemei are ca parametri șirul dat, dimensiunea acestuia și valoarea logică stabilită în funcție de existența sau absența proprietății căutate. În subalgoritmul următor am înlocuit identificatorul `S` cu `găsit`, pentru a-l face mai sugestiv.

```
subalgoritm Decizie(n,x,găsit):
  găsit ← fals
  pentru i ← 1, n execută:
    găsit ← găsit sau P(x[i])
  sfârșit pentru
sfârșit subalgoritm
```

Să observăm o proprietate importantă! Dacă la un moment dat valoarea variabilei `găsit` devine `adevărat`, valoarea ei nu se va mai schimba până la sfârșitul executării algoritmului. Respectiv, în cazul problemei **P5**, dacă am găsit un număr diferit de 1 și de numărul dat care divide numărul considerat, adică am aflat că numărul dat nu este prim, nu putem ajunge în final la concluzia că acesta de fapt este prim! Rezultă că toate operațiile efectuate după descoperirea unui divizor sunt inutile. Să modificăm, pe baza acestei observații, subalgoritmul `Decizie`!

În primul rând vom schimba structura repetitivă de tipul **pentru** într-o structură cu număr *necunoscut* de pași,

pentru a putea întrerupe executarea în momentul în care am obținut răspunsul la întrebare. Evident, în cazul problemei **P6**, dacă nici un element al șirului analizat nu coincide cu elementul căutat, se va parcurge întreg șirul.

```
subalgoritm Decizie1(n,x,găsit):
  i ← 1
  cât timp (i ≤ n) și nu (P(x[i])) execută:
    i ← i + 1
  sfârșit cât timp
  găsit ← i ≤ n
sfârșit subalgoritm
```

Să analizăm acum problemele din cealaltă categorie, mai exact, cele care seamănă cu problema **P7**. În această problemă fiecare element al șirului dat trebuie să aibă o aceeași proprietate, și anume, trebuie să aibă valoarea mai mare sau egală decât elementul precedent. Această caracteristică a șirului se poate formula și în felul următor: *în șir **nu** există nici un element care **nu** are proprietatea cerută*. Transformând subalgoritmul `Decizie1` prin aplicarea negației în două locuri, obținem algoritmul de rezolvare pentru această clasă de probleme. Deoarece semnificația deciziei se referă la toate elementele șirului, înlocuim variabila `găsit` cu `toate`.

```
subalgoritm Decizie2(n,x,toate):
  i ← 1
  cât timp (i ≤ n) și (P(x[i])) execută:
                        {am negat P(x[i])}
    i ← i + 1
  sfârșit cât timp
  toate ← i > n                  {am negat i ≤ n}
sfârșit subalgoritm
```

Probleme legate de implementare

Din nou vom intra în... laborator.

În momentul codificării subalgoritmilor `Decizie` pot apărea diverse probleme. În anumite limbaje de programare o expresie logică formată din subexpresii între care apar operatorii logici **și**, respectiv **sau**, se evaluează în întregime, dar există și limbaje în care, în funcție de operator și valoarea primei subexpresii, evaluarea se întrerupe după stabilirea valorii primei subexpresii. De exemplu, dacă valoarea de adevăr a primei subexpresii este `adevărat` și urmează operatorul **sau**, cea de-a doua subexpresie nu se mai evaluează, deoarece indiferent de valoarea acestuia, rezultatul final va fi `adevărat` (`adevărat` **sau** `adevărat` este `adevărat`, respectiv `adevărat` **sau** `fals` este tot `adevărat`). De asemenea, dacă valoarea de adevăr a primei subexpresii este `fals` și urmează operatorul **și**, cea de-a doua subexpresie nu se mai evaluează, deoarece indiferent de valoarea acesteia, rezultatul final va fi `fals` (`fals` **și** `adevărat` este `fals`, respectiv `fals` **și** `fals` este tot `fals`).

În cazul acelor limbaje de programare care, în schimb, nu întrerup evaluarea acestor expresii logice, sau dacă programatorul nu alege atent ordinea subexpresiilor în cazul primei categorii de limbaje de programare, codificarea deciziei, conform algoritmilor prezentați, pot genera erori neplăcute. Despre ce este vorba? Fie expresia logică din structura repetitivă **cât timp** din algoritmul `Decizie2`:

`(i ≤ n)` **și** `(P(x[i]))`.

Dacă `i > n`, în urma evaluării subexpresiei `i ≤ n` obținem `fals`, dar se evaluează și subexpresia `P(x[i])` în condițiile în care știm deja că `i > n`, deci se va verifica un element `x[i]` care nu există în șirul dat!

În concluzie, în implementarea subalgoritmilor de tipul celor prezentați trebuie să căutăm soluții prin care să evităm producerea unor erori în timpul executării sau obținerea unor rezultate incorecte. În cele ce urmează propunem câteva soluții pentru evitarea acestei "capcane".

a. putem crea un al (n + 1)-lea element fictiv;

b. putem opri reluarea executării nucleului structurii repetitive cu "un pas mai repede", urmând să analizăm motivele ieșirii din ciclu;

c. putem despărți subexpresiile, urmând ca pe cea de-a doua s-o evaluăm doar atunci când acest lucru este necesar;

d. să evităm folosirea, în cea de-a doua subexpresie, a valorii `x[i]`; acest lucru este posibil dacă introducem o variabilă logică pentru a reține valoarea de adevăr a subexpresiei `P(x[i])` de la pasul precedent, astfel permițând ieșirea din structura repetitivă în momentul în care avem confirmarea proprietății sau după parcurgerea întregului șir în cazul în care acest lucru nu a avut loc.

Să rezolvăm problema **P5** (numărul dat este prim sau nu?). Implementăm subalgoritmul `Decizie1` sub forma unei funcții, deoarece rezultatul este o singură valoare. Într-o primă abordare, pornim de la definiția numărului prim (un număr prim are exact doi divizori: 1 și el însuși). În concluzie, vom verifica dacă numărul dat are vreun divizor printre numerele 2, 3, ..., n-1.

```
function prim1(nr:Word):Boolean;        { P5 }
var diviz:Word;
begin
  diviz:=2;
  while (diviz <= nr - 1) and (nr mod diviz <> 0) do
    Inc(diviz);
  prim:=diviz > nr - 1
end;
```

Algoritmul poate fi îmbunătățit considerabil dacă tratăm separat cazul acelor numere care nu necesită nici o împărțire pentru stabilirea proprietății. Acestea sunt numărul 1 și numerele pare. În rest, proprietatea de divizibilitate o vom verifica doar cu numere impare, știind că un număr impar nu poate avea nici un divizor par. În plus, nu vom genera divizori mai mari decât rădăcina pătrată a numărului, deoarece divizorii posibili sunt mai mici decât această valoare.

```
function prim2(nr:Word):Boolean;        {P5}
var diviz:Word;
    p:Boolean;
    rad:Word;
begin
  if nr = 1 then p:=false
  else
    if not Odd(nr) then        {numere pare}
      p := nr = 2
           {singurul număr par prim este 2}
    else begin                 {numere impare}
      diviz := 3;
      rad := Trunc(Sqrt(nr));
      p := true;
      while (diviz <= rad) and p do
        if nr mod diviz = 0 then
          p := false         {diviz divide nr}
        else
          Inc(diviz,2)
    end;
  prim := p
end;
```

În această variantă a implementării am preferat să folosim variabila locală `p` în locul scrierii explicite a expresiei de verificare a divizibilității, pentru a mări lizibilitatea programului. Nu am putut folosi identificatorul `prim`, deoarece astfel în instrucțiunea **while** am fi avut un apel recursiv al funcției `prim`.

În continuare vom rezolva problema **P6.** Această problemă cere să stabilim dacă un cuvânt dat (`cuv`) reprezintă o lună din an sau nu. Presupunem că în vectorul de *string*-uri, denumit `luni`, având tipul `Tluni` avem păstrate cuvintele care reprezintă lunile anului. În funcția `Exista` variabila locală `gasit` primește valoarea `true` în urma depistării unui cuvânt care desemnează o lună care coincide cu șirul dat.

```
function Exista(luni:Tluni;cuv:string):Boolean;
var gasit:Boolean;
    i:Byte;
begin
  gasit := false;
  i := 1;
  while (i <= 12) and not gasit do
    if cuvant = luni[i] then
      gasit := true
    else
      Inc(i);
  Exista := gasit
end;
```

Problema **P7** o vom rezolva implementând algoritmul `Decizie2`. Presupunem că tipul `tablou` este cunoscut. În această implementare în limbajul *Pascal* condiția din instrucțiunea **while** este "sigură", deoarece avansând cu `i` până inclusiv la valoarea `n - 1`, compararea lui `x[i]` cu `x[i+1]` la ultima iterație înseamnă compararea ultimelor două elemente din șir. Dacă am "reușit" să ajungem cu `i` la valoarea `n`, înseamnă că proprietatea este satisfăcută.

```
function Monoton(n:Byte;x:tablou):Boolean;
var i:Byte;
begin
  i := 1;
  while (i <= n - 1) and (x[i] <= x[i+1]) do
    Inc(i);
  Monoton:= i > n - 1
end;
```

În rezolvarea problemei **P8** presupunem că în program s-au declarat tipurile și variabilele:

```
type elev = record
              nume:string;
              medie:Real
            end;
     elevi = array [1..30] of elev;
var clasa:elevi;
    nume:string;     {numele elevului cautat}
    n:Byte;
```

De asemenea, pentru a putea rezolva problema în mod eficient, presupunem că tabloul `clasa` este ordonat descrescător după `medie`. În continuare prezentăm funcția `premiant` cu care rezolvăm problema.

```
function premiant(nume:string;clasa:elevi):
                                        Boolean;
var i:Byte;
begin
  i := 1;
  while (i <= 3) and not (nume = clasa[i].nume) do
    Inc(i);
  premiant := i <= 3
end;
```

Evident, nu trebuie să "ne păzim" de eventuala depășire a dimensiunii șirului, deoarece într-o clasă de elevi cu siguranță vom avea mai mulți decât trei copii.

Concluzii

Dacă într-o problemă se cere doar verificarea unei anumite proprietăți, este recomandabil ca algoritmul să fie prevăzut cu posibilitatea opririi analizei imediat după ce proprietatea s-a stabilit. Dacă într-un șir dat se caută un element având o anumită proprietate, de asemenea este la limita minimei optimizări să nu căutăm elementul respectiv după ce l-am găsit! Evident, cu totul altfel se pune problema, dacă trebuie să găsim toate elementele având acea proprietate. În aceste probleme va trebui să parcurgem toate datele.