# TEOREME de programare

*Clara Ionescu*

**Dacă nu v-ați convins până acum că modelele de rezolvare ale anumitor subprobleme constituie adevărate teoreme, pentru care atât corectitudinea cât și optimalitatea se pot demonstra, vă propunem să urmăriți următoarele tipuri de algoritmi și să analizați felul în care ați procedat până acum.**

## Intersecția

În numărul precedent am rezolvat câteva probleme în care *se dădea un șir* și *se cerea crearea mai multora* pe baza unei proprietăți care permitea descompunerea șirului dat în mai multe subșiruri. Evident, se poate pune întrebarea, cum procedăm când *se dau mai multe șiruri* și trebuie să creăm *un singur șir* care va conține ***intersecția*** lor. Aici prin intersecție înțelegem șirul care conține toate elementele comune, existente în toate șirurile date. Deci este o problemă care cere *selecționarea* anumitor elemente pe baza proprietății de a fi prezente în fiecare șir dat.

Evident, în diverse limbaje de programare, această problemă se rezolvă în funcție de instrumentele pe care acestea le pun la dispoziția programatorilor. De exemplu, în *Pascal* există tipul **set**, care poate simplifica mult efectuarea acestei operații, dacă tipul de bază și domeniul valorilor elementelor permit declararea șirurilor ca fiind de acest tip. Acum, ne-am propus să tratăm problema enunțată în condiții generale, independent de limbajul de programare ales, doar din punct de vedere algoritmic. De asemenea, presupunem că fiecare șir conține elemente distincte și că aceste șiruri nu sunt ordonate. Veți spune că acest lucru nu trebuie să constituie un impediment, deoarece am putea să le ordonăm și să proiectăm un algoritm simplu care determină intersecția șirurilor. Dar ordonarea, în acest caz, ar însemna o risipă de timp pe care nu întotdeauna ne-o putem permite. În concluzie, vom aplica teorema numită *selecționare* care conține în interior o *decizie*.

**P24.** Se consideră toți divizorii a două numere naturale. Să se determine toți divizorii comuni!

**P25.** Să ne amintim de o problemă rezolvată la începutul acestui serial: Cercetând lumea păsărilor din Delta Dunării, *k* specialiști au notat, fiecare, tipurile de păsări observate. Acum îi "trimitem" acolo pe ornitologii noștri atât vara, cât și iarna și le cerem să ne spună, pe baza observării speciilor prezente vara, respectiv iarna, care sunt speciile care nu migrează.

**P26.** Cunoscând serile libere a patru persoane, să se stabilească serile în care ar fi posibil să meargă împreună la cinema!

Prezentăm un subalgoritm general care stabilește șirul acelor elemente din șirul *X* care se găsesc și în șirul *Y*.

```
n, m: întreg                {numărul elementelor șirurilor date}
X, Y: tablou(1..n): tip element   { elementele șirurilor date}
buc: întreg          { numărul elementelor șirului intersecție}
Z: tablou(1..n): întreg  { elementele comune în șirurile date}

subalgoritm Intersecție(n,X,m,Y,bucZ,Z):
  buc ← 0
  pentru i ← 1, n execută:
  j ← 1
    cât timp (j<=m) și (X[i]<>Y[j]) execută:
      j ← j + 1
    sfârșit cât timp
    dacă j <= m atunci
      buc ← buc + 1
      Z[buc] ← X[i]
    sfârșit dacă
  sfârșit pentru
sfârșit subalgoritm
```

Implementarea algoritmului prezentat nu ridică nici o problemă care să îl împiedice pe cititor să rezolve cu ajutorul lui problemele enunțate, deci nu le vom detalia.

În continuare vom exemplifica modul în care acest model se poate aplica în rezolvarea unor probleme care la prima vedere nu au nimic comun cu problema intersecției.

**1.** Vom putea aplica algoritmul într-o *decizie* unde proprietatea care trebuie verificată se definește astfel: *fiind date două mulțimi, să se stabilească dacă au sau nu cel puțin un element comun!*

**2.** Dacă se cere stabilirea *unui singur element comun*, avem de realizat o *selecție*.
**3.** Dacă se cere *determinarea unui element comun dacă există un astfel de element*, avem o *căutare secvențială*.
**4.** În schimb, dacă trebuie *să numărăm câte elemente comune există în două șiruri date*, avem o *numărare*.

Vom exemplifica modul de aplicare a algoritmului `Intersecție` pentru a răspunde la întrebarea **3**. În rezolvare căutăm în șirul *X* un element care există și în *Y*. Deci vom avea o căutare secvențială care conține o decizie. Evident, algoritmul are calitatea că nu continuă căutarea după ce s-a găsit primul element comun.

```
subalgoritm Element(n,X,m,Y,există,e):
  i ← 1
  există ← fals
  cât timp (i<=n) și not există execută:
  j ← 1
    cât timp (j<=m) și (X[i]<>Y[j]) execută:
      j ← j + 1
    sfârșit cât timp
    dacă j <= m atunci
      există ← adevărat
      e ← X[i]
    altfel i ← i + 1
    sfârșit dacă
  sfârșit cât timp
sfârșit subalgoritm
```

## Reuniunea

Dacă vrem să rezolvăm probleme care prelucrează mulțimi implementate cu ajutorul șirurilor, trebuie să tratăm și problema ***reuniunii***.

**P24.** Cunoscând divizorii primi a două numere, să se determine divizorii primi ai celui mai mic multiplu comun a celor două numere!
**P25.** Se consideră orarul a doi profesori dintr-o școală. Să se determine orele în care cel puțin unul dintre ei poate să suplinească un eventual profesor care lipsește!

Vom prezenta și de data aceasta subalgoritmul general în care avem următoarele date:

*n, m*: *întreg* {*numărul elementelor șirurilor date*}
*X, Y*: *tablou(1..n): tip element* { *elementele șirurilor date*}
*buc*: *întreg* { *numărul elementelor șirului reuniune*}
*Z: tablou(1..n): întreg* { *elementele șirului reuniune*}

În rezolvare, mai întâi vom copia toate elementele din șirul *X* în șirul *Z*, inițializând lungimea *buc* cu lungimea acestui șir. Apoi, vom selecționa din șirul *Y* acele elemente care nu se află în *X*. În consecință, algoritmul conține o *copiere* și o *selecționare* care conține o *decizie*.

```
subalgoritm Reuniune(n,X,m,Y,bucZ,Z):
  Z ← X
  buc ← n
  pentru j ← 1, m execută:
  i ← 1
    cât timp (i<=n) și (X[i]<>Y[j]) execută:
      i ← i + 1
    sfârșit cât timp
    dacă i > n atunci
      buc ← buc + 1
      Z[buc] ← Y[j]
    sfârșit dacă
  sfârșit pentru
sfârșit subalgoritm
```

În locul prezentării rezolvărilor problemelor, vom vedea modul în care acest subalgoritm se poate aplica în rezolvarea unor probleme clasice. De exemplu, este posibil ca într-o problemă să fie specificată cerința de *a transforma un șir de numere reale într-o mulțime*. Această cerință "tradusă" în termenii modelelor noastre poate fi formulată astfel: *să se copieze elementele unui șir într-un șir nou astfel încât acesta să conțină numai elemente distincte*. În concluzie, în șirul rezultat vom reține acele elemente din șirul dat care încă nu apar în șirul rezultat.

```
subalgoritm CreareMulțime(n,X,bucZ,Z):
  buc ← 0
  pentru i ← 1, n execută:
  j ← 1
    cât timp (j<=buc) și (X[i]<>Z[j]) execută:
      j ← j + 1
    sfârșit cât timp
    dacă j > buc atunci
      buc ← buc + 1
      Z[buc] ← X[i]
    sfârșit dacă
  sfârșit pentru
sfârșit subalgoritm
```

## Concluzii

Să observăm că pentru rezolvarea unor probleme care necesită subalgoritmi pentru rezolvarea problemelor enunțate, majoritatea începătorilor ar implementa, și ei, un algoritm pătratic, dar cu două structuri repetitive de tipul **pentru**. Veți spune că amândouă abordările conduc la algoritmi pătratici... și veți avea dreptate. Totuși, există o diferență mare între o structură **pentru** și una **cât timp**. Știm că nucleul din structura **pentru** se execută de *n* ori, iar corpul structurii **cât timp** doar în cazul cel mai defavorabil are acest număr de executări. Deci, este recomandabil să nu fim neglijenți în alegerea tipului structurilor repetitive.

Problema intersecției și reuniunii se prezintă altfel în cazul în care șirurile sunt odornate. Dar acest subiect îl vom trata în numărul viitor.