TEOREME de programare

Clara Ionescu

Pe parcursul a opt luni (aproape a trecut un an școlar) v-am prezentat modele de proiectare și implementare a unor subalgoritmi care apar frecvent în rezolvarea problemelor. Aceste modele sugerează că programarea ar trebui să se desfășoare planificat, disciplinat, respectând anumite cerințe, pentru că altfel se diminuează beneficiul ei și vor fi înțelese numai de autorii lor.

Interclasarea a două șiruri ordonate

În numărul precedent am rezolvat problema reuniunii a două mulțimi, presupunând că mulțimile sunt implementate sub forma unor șiruri. Se poate observa ușor că, în cazul unor șiruri de date ordonate, această reuniune se va realiza altfel, mai ales dacă se cere ca și rezultatul să constituie un șir ordonat. De exemplu, fie problema:

P27. Se consideră mai multe tabele nominale ale elevilor din diferite școli, participanți la olimpiada județeană. Să se realizeze un singur tabel nominal în care să fie cuprinse numele tuturor elevilor din tabelele date!

În această problemă se dau mai multe șiruri ordonate și se cere unul singur, ordonat și acesta. Rezolvarea revine la crearea unui șir ordonat din *două* șiruri, apoi din șirul rezultat și următorul dat se va obține unul nou etc. În cele ce urmează vom prezenta un subalgoritm general care creează un șir ordonat din alte două șiruri ordonate. Deocamdată facem abstracție de enunțul problemei **P27** în care, elementele șirurilor sunt cu siguranță distincte.

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

În rezolvare vom avansa în paralel în cele două șiruri date. Primul element în șirul nou va fi fie `X[1]`, fie `Y[1]`. Pentru a ști care dintre cele două elemente va fi ales, trebuie să le comparăm. Dacă `X[1]` este mai mic decât `Y[1]`, atunci pe acesta îl "consumăm" din șirul *X* și pregătim indicele elementului următor din acesta. În caz alternativ, procedăm la fel cu indicele *j*, după ce așezăm elementul `Y[1]` în șirul *Z*. Dacă cele două elemente sunt egale, vom copia unul dintre aceștia în *Z*, și vom avansa în ambele șiruri. Vom efectua acești pași până când careva din șiruri se sfârșește, moment în care copiem elementele rămase din celălalt șir în șirul nou. Din moment ce nu putem ști care șir s-a "consumat" integral, vom scrie secvența respectivă pentru ambele șiruri.

```
subalgoritm Interclasare1(n,X,m,Y,bucZ,Z):
  buc ← 0
  i ← 1
  j ← 1
  cât timp (i <= n) și (j <= m) execută:
    buc ← buc + 1
    dacă X[i] < Y[j] atunci
      Z[buc] ← X[i]
      i ← i + 1
    altfel
      dacă X[i] = Y[j] atunci
        Z[buc] ← X[i]
        i ← i + 1
        j ← j + 1
      altfel
        Z[buc] ← y[j]
        j ← j + 1
      sfârșit dacă
    sfârșit dacă
  sfârșit cât timp
  cât timp i <= n execută:
    buc ← buc + 1
    Z[buc] ← X[i]
    i ← i + 1
  sfârșit cât timp
  cât timp j <= m execută:
    buc ← buc + 1
    Z[buc] ← Y[j]
    j ← j + 1
  sfârșit cât timp
sfârșit subalgoritm
```

Analizând acest subalgoritm, observăm că dacă am fi avut "norocul" ca `X[n]` să fie egal cu `Y[m]`, atunci ultimele două structuri **cât timp** nu se executau niciodată. Vom exploata această observație, așezând două elemente fictive (numite frecvent *santinele*) după ultimele elemente din cele două șiruri. Vom fi atenți să fie egale și să aibă valorile mai mari decât ultimul element (cel mai mare) din fiecare șir. În subalgoritmul `Interclasare2` vom nota această valoare cu `infinit`.

```
subalgoritm Interclasare2(n,X,m,Y,bucZ,Z):
  buc ← 0
  i ← 1
  j ← 1
  X[n+1] ← infinit
  Y[m+1] ← infinit
  cât timp (i < n+1) sau (j < m+1) execută:
    buc ← buc + 1
    dacă X[i] < Y[j] atunci
      Z[buc] ← X[i]
      i ← i + 1
    altfel
      dacă X[i] = Y[j] atunci
        Z[buc] ← X[i]
        i ← i + 1
        j ← j + 1
      altfel
        Z[buc] ← y[j]
        j ← j + 1
      sfârșit dacă
    sfârșit dacă
  sfârșit cât timp
sfârșit subalgoritm
```

Valoarea elementului fictiv (`infinit`) nu va face parte din șirul rezultat. Dacă am avea nevoie de acest element, pentru că șirul rezultat se va interclasa cu un altul, evident, putem să-l adăugăm.

Subalgoritmul `Interclasare2` este valabil în caz general. Să vedem cum s-ar aplica și ce îmbunătățiri s-ar putea realiza în scopul rezolvării unei probleme cum este **P27**, unde se știe că elementele sunt *distincte*. Evident, nu vom pune întrebarea privind posibila egalitate între `X[i]` și `Y[j]`. Dar mai știm ceva: șirul nou va conține fiecare element existent în șirul *X* și *Y*, deci se cunoaște numărul elementelor sale care este *n* + *m*, lucru care în caz general nu este cunoscut (se admit și elemente identice în cele două șiruri date). În acest caz, structura repetitivă utilizată va fi de tip **pentru**.

Rezultă că algoritmul de interclasare nu este "unul singur". În primul rând, dacă este posibil (și în majoritatea cazurilor este) vom adăuga elemente *santinele* la sfârșitul șirurilor care se vor interclasa, astfel eliberându-ne de grija elementelor rămase neprelucrate într-un șir, după ce celălalt s-a "epuizat". În al doilea rând, vom studia proprietatea elementelor din șirurile de interclasat: dacă ele sigur sunt distincte, vom putea implementa un algoritm care se prezintă sub următoarea formă:

```
subalgoritm Interclasare3(n,X,m,Y,bucZ,Z):
  i ← 1
  j ← 1
  X[n+1] ← infinit
  Y[m+1] ← infinit
  pentru buc = 1,n+m execută:
    dacă X[i] < Y[j] atunci
      Z[buc] ← X[i]
      i ← i + 1
    altfel
      Z[buc] ← y[j]
      j ← j + 1
    sfârșit dacă
  sfârșit pentru
sfârșit subalgoritm
```

Concluzii

Ce s-ar mai putea spune? Este nevoie totuși de concluzii... În primul rând sperăm că se observă o tendință de reglementare a modului în care se vor implementa operații relativ simple care apar frecvent în algoritmi. Cu toții știm că este important să nu le realizăm neglijent. Aceste teoreme (*calcule pe baza elementelor unui șir* (1), *decizia* (2), *selecția* (3), *căutarea secvențială* (4), *numărarea* (5), *stabilirea maximului/minimului* (6), *selectarea* elementelor pe baza unei proprietăți (7), *partiționarea* (8), *reuniunea* (9), *intersecția* (10) și *interclasarea* (11)) conțin modele de implementare a unor astfel de operații "elementare".

Alte modele veți cunoaște în momentul studierii *Șabloanelor de programare* (*Design patterns*), atunci când veți decide să alegeți pentru viitor meseria de programator.

Rezultă că atunci când verificăm comportamentul și performanțele unui algoritm, nu va fi suficient să aplicăm o metodă de verificare de tipul "cutiei negre" sau "cutiei transparente". Astfel vom afla doar că algoritmul furnizează un rezultat "aparent" corect pentru date de intrare în cazul cărora am putut calcula și compara rezultatul (rezultatele) în minte. Dacă am obținut același rezultat în minte ca și cu ajutorul programului, ne putem bucura, dar nu putem fi siguri că algoritmul respectiv este cel mai bun (este optim) în ceea ce privește performanțele sale.

În momentul evaluării performanțelor trebuie să avem în vedere cele două resurse la care apelăm: *timpul* și *spațiul*. Vom evalua ordinul de mărime a numărului de operații care se vor efectua asupra datelor de prelucrat, apoi vom căuta să reducem acest număr. Vom încerca să renunțăm la structuri de date auxiliare, apelând la soluții care rezolvă problema "pe loc", adică folosind pentru rezultat structura de date construită pentru datele de intrare.

Bibliografie

1. **Szalávi Péter, Zsakó László**, *Programozási tételek*, Neumann János Számítógép - tudományi Társaság, 2001