Sfaturi pentru viitori olimpici (1)

TESTAREA și DEPANAREA programelor

Mihai Stroe

Datorită apropierii fazelor avansate ale olimpiadelor, inițiem o serie de două articole orientate pe testarea, depanarea și optimizarea programelor. Aceste aspecte sunt deosebit de importante. În concursurile de informatică, elaborarea unui algoritm eficient de rezolvare nu este suficientă pentru obținerea punctajului maxim pentru o anumită problemă; trebuie scris un program, care trebuie testat, depanat și eventual optimizat.

Articolul curent tratează problema testării și depanării programelor; următorul număr al *GInfo* va conține un articol care se ocupă de optimizarea acestora...

Cadrul general

Mulți concurenți "economisesc" timp de implementare, scriind programe greu de urmărit. De multe ori, mult mai mult timp se pierde depanând astfel de programe; acest timp ar fi putut fi economisit programând atent și îngrijit.

În continuare, vom presupune că vă aflați în următoarea situație:

- aveți o idee de rezolvare pentru o anumită problemă;
- ați verificat că algoritmul găsit funcționează pe toate cazurile și sunt șanse mari ca o implementare eficientă să se încadreze în timpul de execuție;
- trebuie să scrieți un program care să rezolve problema.

Scrierea unui program ușor de urmărit și de depanat

Claritatea și structurarea codului

Procesul implementării constă în a scrie și a depana simultan programul. Cu cât greșelile sunt depistate mai rapid, cu atât este mai bine.

Pentru a fi ușor de urmărit și ușor de depanat, un program trebuie să fie structurat. Astfel, diferitele etape din cadrul rezolvării trebuie să fie conținute în funcții și proceduri diferite sau, dacă sunt conținute în același subprogram din motive de eficiență, vor fi separate prin spații libere și comentarii.

Numele subprogramelor trebuie să fie explicative (de exemplu "init" pentru o procedură de inițializare, "pantă" pentru un subprogram care determină panta unei drepte etc.), astel sugerând și "funcția" pe care o îndeplinesc.

Încapsularea etapelor în subprograme diferite ușurează și testarea programului.

Imediat după ce a fost scris, fiecare subprogram sau program poate fi testat (și depanat, dacă este cazul). Este mult mai ușor să detectăm o eventuală eroare dacă examinăm o secvență de cod relativ scurtă, decât dacă examinăm întreg programul. Fiecare subprogram conține rezolvarea unei subprobleme care la rândul ei are date de intrare și date de ieșire (rezultate) care astfel pot fi verificate imediat.

Alternativa, uneori mai eficientă din punct de vedere al timpului de implementare (și recomandată pentru avansați), constă în a presupune că toate subprogramele vor fi scrise corect; chiar și în acest caz, delimitarea subprogramelor simplifică testarea și depanarea. Dacă se detectează o eroare, subprogramele pot fi testate unul câte unul.

Un exemplu de testare incrementală constă în testarea imediată a procedurii de citire a datelor și a celor care decurg imediat din ea, cum ar fi inițializarea unor structuri. Aceasta necesită scrierea unor teste (le veți scrie oricum, pentru a testa programul; de ce să nu le scrieți de la început?) și executarea procedurii de citire imediat după scrierea ei. Această testare se recomandă atunci când formatul datelor de intrare este complex. De asemenea, erorile de intro-

ducere a datelor de test se detectează mult mai ușor în această fază.

Ați scris un program ușor de urmărit dacă, uitându-vă din nou la el peste șase luni, veți înțelege ce și cum face și îl veți putea urmări pas cu pas. Pentru aceasta, următoarele indicații pot fi folositoare:

- organizați programul astfel încât să reflecte ideea de rezolvare;
- folosiți facilitățile de indentare oferite de mediul de dezvoltare;
- folosiți spații libere între blocuri de cod care realizează acțiuni diferite;
- folosiți comentarii; cele câteva secunde "pierdute" pentru a scrie un comentariu explicativ pot reduce cu câteva minute timpul de depanare;
- folosiți nume de variabile sugestive (de exemplu, un program care conține variabile **a**, **aa** și **aaa** poate deveni greu de urmărit) și încercați, pe cât posibil, să nu folosiți aceeași variabilă în scopuri diferite (excepțiile se admit dacă refolosirea este necesară datorită limitărilor de memorie);
- expresiile aritmetice care apar trebuie să fie ușor de urmărit; astfel, se recomandă folosirea parantezelor pentru a clarifica ordinea operațiilor și folosirea spațiilor libere în expresii etc.;
- încercați să folosiți cât mai puțin metoda "copy-paste și modificări minore" pentru a rezolva subprobleme similare, deoarece puteți greși la acele modificări; pe deasupra, lungimea programului rezultat va fi mai mare, deci depanarea va fi mai dificilă; dacă eficiența nu este afectată, se recomandă scrierea unor funcții/proceduri care vor fi apelate cu parametri diferiți.

Comentariile trebuie să fie scurte (nu scrieți "povești"). Ele se folosesc pentru a delimita și a explica rezolvarea anumitor subprobleme (mai ales în cazul în care, din motive de eficiență, acestea sunt tratate în același subprogram) și pentru a explica aspecte de implementare. De exemplu, o expresie aritmetică poate arăta "ciudat" după câteva optimizări; în acest caz, ea va fi precedată, într-un comentariu, de expresia "ușor de citit" și, eventual, de semnificația ei.

Se folosesc comentarii și pentru a explica semnificația și modul de calcul al elementelor unor anumite matrice. De exemplu, fie o problemă de *programare dinamică*, în care N persoane se împart în K grupe, în fiecare grupă având persoane consecutivi din șir, astfel încât să se obțină o penalizare minimă, după anumite criterii. Valoarea $A[i, j]$ poate reprezenta "penalizarea minimă care se obține împărțind primii j oameni în i grupe". Ar fi neplăcut să implementați rezolvarea, apoi să recitiți enunțul, să vă mutați atenția asupra unor alte aspecte și... să afișați $A[N, K]$ în loc de $A[K, N]$, deoarece ați uitat ce semnificau indicii. Un comentariu bine plasat micșorează posibilitatea apariției unei astfel de erori.

De asemenea, pe parcursul depanării, se scrie câte un comentariu pentru fiecare secvență al cărei conținut nu este clar la prima recitire.

Detectarea existenței erorilor

Evident, dacă programul se termină printr-un *crash*, sau rezultatul afișat de program pe un anumit test nu corespunde așteptărilor, sunt mari șanse ca în program să existe o eroare. Verificați că ați înțeles bine problema și că ați tastat corect valorile din fișierul de intrare!

O mare parte din erori sunt detectate și localizate folosind opțiunile de compilare. În *Pascal*, acest lucru se face folosind *directivele de compilare*; cele mai importante sunt **$R** - *range checking*, **$S** - *stack checking* și **$I** - *input/output checking*. Prezența unei anumite directive de compilare în starea "**+**" instruiește compilatorul să genereze cod care să facă anumite verificări; de exemplu, la **$R+** se verifică domeniul variabilelor și al indicilor, la **$S+** depășirea stivei, iar la **$I+** succesul operațiilor de intrare/ieșire.

Prezența fragmentelor de cod suplimentare duce la detectarea mai ușoară a unor erori "clasice". Pe de altă parte, executarea lor poate încetini considerabil programul. De aceea, se recomandă ca directivele de compilare să fie setate pe "**+**" la testarea pentru detectarea erorilor și pe "**-**" la testarea pentru eficiență, folosind teste mari. În programul final, directivele vor fi setate pe "**-**".

Multe erori pot fi detectate și localizate mai ușor dacă programul face verificări pe parcurs și afișează informații importante în timpul execuției.

Pentru anumite părți ale programului este util să scrieți subprograme care să ușureze depanarea, verificând starea anumitor variabile și structuri de date. De exemplu, dacă sunteți începător în lucrul cu *heap*-uri, puteți scrie un subprogram care să testeze dacă *heap*-ul pe care îl folosiți este valid. În etapa de depanare veți introduce câteva apeluri ale acestui subprogram; dacă la un moment dat *heap*-ul nu este valid, ați detectat o eroare.

O altă metodă de a ușura depanarea este să afișați pe ecran valorile anumitor variabile, în locurile esențiale. Dacă, la execuția pe un test simplu, o parte din variabile nu au valorile așteptate, au fost detectate erori. Un efect similar se poate obține folosind *watches* și *breakpoint*-uri, dar în prima fază poate fi mai eficient să afișăm pe ecran și să examinăm valorile la sfârșit. Dacă programul este corect, nu mai este nevoie de *watches* și *breakpoint*-uri, iar procesul de testare durează mult mai puțin.

Metoda de mai sus nu pare aplicabilă dacă ați dori să afișați mai mult de 25 de valori pe linii diferite. Pentru un astfel de caz, puteți face afișarea într-un fișier; astfel, programul va avea un fișier de ieșire și un fișier "**log** al execuției". Puteți examina ambele fișiere după rulare.

Liniile și subprogramele care ajută la detectarea erorilor nu vor fi șterse după testarea și depanarea programului.

Alternativa, mult mai utilă, constă în a le transforma în comentarii. Astfel, ele pot fi refolosite; dacă programul este corect, dar nu și suficient de rapid, rescrierea și optimizarea unor subprograme poate duce la apariția unor noi erori.

O întrebare frecventă este: *Dar oare toate aceste precauții, scrierea comentariilor etc. nu necesită prea mult timp?* Răspunsul este că timpul pierdut cu scrierea unui program ușor de depanat este câștigat în faza de depanare. În plus, o parte din etapele prezentate, cum ar fi cea în care scriem subprograme pentru a verifica starea unor structuri de date, pot fi amânate pentru începutul etapei de depanare, și folosite în cazul apariției unor erori. Este mult mai ușor să scriem o astfel de funcție, decât să examinăm de fiecare dată structura "ochiometric". Aveți grijă să nu greșiți tocmai subprogramul de verificare!

Localizarea și corectarea erorilor

Ați scris un program, dar acesta nu dă rezultatul dorit. Mai mult, știți că de la un anumit moment încolo programul nu se comportă corect. Poate a afișat pe ecran o valoare necorespunzătoare, sau poate s-a oprit cu o eroare de depășire a stivei. Ce faceți în continuare?

Erori frecvente

În această subsecțiune vom enumera cele mai frecvente erori. Unele dintre ele pot fi evitate în faza de scriere a programului. În orice caz, dacă programul nu se comportă cum doriți, există mari șanse ca motivul să fie unul din cele de mai jos. Acestea sunt deci principalele erori pe care trebuie să le aveți în vedere pe parcursul depanării.

Folosirea pointerilor poate introduce erori greu de detectat. De asemenea, multe din structurile alocate dinamic (de exemplu, listele) sunt dificil de depanat (nu putem vedea o listă în fereastra de *watches*). De aceea, se recomandă evitarea alocării dinamice; dacă totuși nu este suficientă memorie pentru a aloca totul static, puteți aloca dinamic vectori, matrice etc.

În cazul în care aveți o matrice prea mare (de exemplu o matrice de numere întregi *A* cu 300 de linii și 300 de coloane, care nu încape în 64K) veți aloca liniile acesteia ca vectori. Veți avea alocat static un vector de pointeri către liniile matricei; în *Pascal*, un element al matricei va fi accesat cu `A[i]^[j]`.

Dacă folosiți o structură de pointeri, este bine să scrieți subprograme care vă afișează pe ecran conținutul structurii într-o formă inteligibilă, pentru a facilita depanarea. De exemplu, se poate vizualiza valoarea atașată elementului curent în prelucrare `p^.info`.

Un alt tip frecvent de erori este cel al depășirii limitelor tipurilor de date. De exemplu, în *Pascal*, adunând două valori care depășesc *MAXLONGINT*, putem obține un rezultat negativ. Este datoria voastră să vă protejați față de astfel de depășiri.

Neinițializarea variabilelor poate conduce la erori dificil de detectat. De exemplu, la alocarea dinamică a unui vector, acestuia i se rezervă o zonă de memorie, dar ea nu este inițializată cu `0`. Un program care presupune că zona este inițializată cu `0` se va comporta într-o manieră imprevizibilă. Dar nici variabilele locale ale unor subprograme nu se inițializează cu zero.

Calculele cu numere reale pot introduce erori. De exemplu, testarea egalității a două numere reale nu se face cu "`x = y`", ci cu "`abs(x - y) < eps`", unde `eps` este o constantă cu o valoare foarte mică, de exemplu `0.0000001`.

Trecerea bruscă a unui programator de la *Pascal* la *C* poate fi cauza unor erori clasice. De exemplu, programatorul *Pascal* este obișnuit cu **`array`**-uri care se indexează începând de la 1; în *C*, un **`array`** cu *N* elemente are indicii elementelor de la `0` la *N* - 1. De asemenea, ciclurile **`for`** se scriu "`for(i = 0; i < N; i++);`", iar fostul programator *Pascal* este tentat să scrie "`i <= N`". Același programator poate folosi "`=`" pentru comparare în loc de "`==`". Recomand ca trecerea de la *Pascal* la *C* să se facă în timp util, astfel încât programatorul să se poată obișnui cu noul limbaj (deci cu cel puțin 2-3 luni înainte de concursurile importante).

O altă eroare constă în nedeclararea unor variabile locale acolo unde este nevoie. De exemplu, o procedură recursivă care folosește o variabilă contor **`i`** trebuie să o declare local; folosirea variabilei globale nu asigură restaurarea ei automată după fiecare întoarcere din apelurile efectuate. Pe de altă parte, această variabilă ocupă spațiu în stivă; din motive de eficiență, se poate prefera folosirea unei variabile globale și scrierea unor instrucțiuni în scopul restaurării ei.

Ultimul tip de erori din listă, dar poate cel mai des întâlnit, este cel al erorilor "de neatenție". Acestea sunt, probabil, erorile care au costat participanții la olimpiade mai multe puncte (ușor de obținut) decât orice altceva în ultimii 10 ani. Sunt motivele pentru care unul dintre cei mai valoroși concurenți poate pierde o medalie sau o calificare la un concurs internațional.

Astfel de erori sunt: scrierea unui indice **`i`** în loc de un **`j`** într-un ciclu **`for`**, afișarea valorii $A[n, k]$ în loc de $A[k , n]$, scrierea unui subprogram (de exemplu, subprogramul de inițializare) și neapelarea lui când este nevoie, intershimbarea numărului de linii *n* cu numărul de coloane *m* etc. Dacă obișnuiți să faceți astfel de erori, este foarte important să citiți cu atenție secțiunile programului pe care vă concentrați în timpul depanării. O măsură preventivă constă în a reciti fiecare subprogram imediat după ce l-ați scris; șansele de a observa dacă aveți o astfel de eroare sunt mult mai mari dacă examinați o secțiune mică de cod.

Tot în categoria erorilor de neatenție încadrăm netratarea anumitor cazuri particulare care pot apărea (de exemplu, graful cu `0` noduri, dacă nu se specifică explicit în enunț că nu putem întâlni un astfel de caz), nerespectarea formatului de afișare, afișarea în alt fișier decât cel specifi-

cat în enunț și nerezolvarea unui subpunct al problemei pentru că nu ați citit cu atenție enunțul. Fiți foarte atenți cu aceste tipuri de erori, și citiți enunțul și formatul de afișare de cel putin 3-4 ori! Mulți programatori experimentați, printre care și autorul acestui articol, au pierdut sute, poate mii de puncte din cauza unor astfel de erori.

Este recomandat să cunoașteți care sunt erorile pe care le faceți cele mai frecvent. Gândiți-vă la modalități pentru a le evita (de exemplu, dacă aveți erori datorate lucrului cu pointerii, încercați să alocați totul static!) și pentru a le descoperi rapid în cazul în care totuși apar. Dacă ajungeți la observații utile și cu aplicabilitate generală, le puteți trimite redacției, și poate le veți regăsi în următorul articol pe această temă!

Depanarea propriu-zisă

Fără a insista prea mult asupra celor prezentate, reamintim că anumite operațiuni care nu au fost făcute anterior (cum ar fi afișarea valorilor unor variabile pe ecran) pot fi introduse în această fază.

În continuare ne vom concentra asupra depanării folosind *watches* și *breakpoint*-uri.

Elevii începători (și nu numai) ar trebui să învețe să stăpânească foarte bine instrumentele de depanare puse la dispoziție de mediul de programare. De asemenea, se recomandă învățarea *shortcut*-urilor prin care acestea pot fi folosite. Timpul de depanare poate fi redus semnificativ dacă, pe lângă scrierea unui program structurat, *breakpoint*-urile sunt plasate în punctele esențiale, se urmărește execuția în interiorul subprogramelor numai dacă este cazul etc.

Există un punct până în care programul funcționează corect și un punct în care valorile unor variabile nu sunt cele așteptate. Inițial, primul punct este începutul programului, iar ultimul este sfârșitul sau între aceste două puncte se află cel puțin o eroare. O parte din strategiile descrise mai sus pot restrânge acest interval.

Vrem să restrângem intervalul până când localizăm eroarea (sau una dintre ele) într-o secvență mică de cod, după care o putem observa și elimina.

Primul pas constă în punerea în *watches* a variabilelor care vor fi urmărite. Puneți variabilele care pot fi afectate și cele care vă ajută să înțelegeți ce se întâmplă. De exemplu, dacă aveți o problemă cu N puncte în plan, coordonatele punctelor vor fi în *watches* în majoritatea cazurilor, chiar dacă ați stabilit că sunt citite corect și nu sunt modificate ulterior.

Găsiți subprogramul în care este localizată eroarea, folosind *breakpoint*-uri și opțiunea *Step Over* din meniul *Debug* (*shortcut*-ul în *Borland Pascal* și *Borland C* este *F8*). Puneți *breakpoint*-uri între punctul de început (până la care totul este O.K.) și punctul de sfârșit, apoi actualizați-le în timp ce executați bucăți din program, într-o manieră similară căutării binare. Reluați execuția și restrângeți cât mai mult intervalul.

Dacă pe parcursul acestei operații vă gândiți că o anumită secvență de cod poate fi "suspectă", examinați-o pe aceea înainte de a continua procesul descris anterior. Puneți un *breakpoint* înainte și unul după, apoi rulați programul pentru a vă confirma sau infirma suspiciunea.

Un instrument de depanare foarte util, dar neglijat de unii, sunt *breakpoint*-urile condiționale. Concret, uneori se poate ca eroarea să nu apară la prima trecere printr-un anumit punct, ci după mai multe treceri. Setarea unui *breakpoint* și execuția (cu *Ctrl+F9* în *Borland*) până la detectarea erorii poate necesita timp. Soluția constă în folosirea unui *breakpoint* condițional. Un astfel de *breakpoint* se activează (duce la întreruperea execuției) când o anumită condiție este îndeplinită.

O metodă eficientă de a folosi *breakpoint*-urile condiționale este bazată pe *căutarea binară*. Contorizați numărul de treceri printr-o anumită zonă (declarați o variabilă contor pe care o incrementați la fiecare trecere) și verificați, în condiția *breakpoint*-ului, dacă aceasta a ajuns la o anumită valoare. Dacă eroarea încă nu a apărut, creșteți valoarea din condiție; dacă a apărut, o scădeți etc.

În cazurile în care eroarea este semnalată printr-o anumită combinație (pe care o cunoașteți - de exemplu, o distanță negativă) de valori ale variabilelor din *watches*, iar combinația este ușor de detectat, folosiți un *breakpoint* condițional și opriți execuția exact în locul unde ați detectat eroarea. Dacă este mai dificil, dar nu imposibil să detectați combinația care conduce la apariția erorii, scrieți o funcție care verifică, la fiecare trecere, dacă eroarea a apărut. În momentul în care apare eroarea, întrerupeți execuția (folosind un *breakpoint*) și examinați contextul.

Când aveți o eroare într-un program, este recomandat să nu considerați că o anumită secțiune este 100% corectă. Autorul, după ani de experiență, a reușit "performanța" de a face greșeli de neatenție în cadrul unor subprograme clasice, cum ar fi unul care implementa *algoritmul Roy-Floyd* pentru determinarea distanțelor minime între oricare două noduri dintr-un graf, și de a depana inutil, timp de mai multe minute, alte părți mult mai "suspecte" ale programului până la localizarea erorii. În acest caz, principiul "nevinovației până la proba contrarie" nu se aplică; toate subprogramele pot conține erori.

Rescrierea

V-ați aflat vreodată în situația de a depana pentru un timp îndelungat o secțiune dintr-un program, în care sunteți sigur că există o eroare, fără a reuși să o detectați?

O posibilă soluție constă în rescrierea secțiunii. Rescrieți secțiunea în alt mod, cât mai clar posibil, aplicând observațiile din acest articol. Nu optimizați decât după ce ați eliminat eroarea. Cu puțin noroc, s-ar putea ca rescrierea să nu mai conțină erori, sau acestea să fie mai ușor de depanat.

În orice caz, păstrați și versiunea anterioară (în comentarii sau în alt fișier). S-ar putea să refolosiți anumite bucăți de cod (să sperăm că nu și cea cu eroare...).

Generarea datelor de test

Depanarea nu se poate face în absența datelor de test. Acest pas este de fapt unul dintre primii pași ai rezolvării problemei; se recomandă ca algoritmul să fie verificat pe testul din exemplu și pe alte câteva teste, înainte de a trece la implementare.

Se știe că, pentru a crește probabilitatea de detectare a erorilor, datele de test trebuie să fie acoperitoare (cazuri particulare, cazuri de teste mari etc.) și variate. Totuși, în multe cazuri, elevii folosesc testul din exemplul problemei (care poate fi total neconcludent), mai scriu maxim două teste mici "de mână" și apoi se declară satisfăcuți și consideră că au rezolvat problema. Mulți dintre acești elevi obțin punctaje mici, după care unii dintre ei depun contestații, susținând că programul lor "mergea pe exemplu". Intenția noastră nu este de a-i ridiculiza pe acești elevi, ci de a-i ajuta să obțină punctele pe care le merită și de care, probabil, îi desparte o eroare pe care ar putea-o detecta și corecta în câteva minute.

În general, testele pe care un elev și le dă în timpul concursului se împart în două categorii:

- *teste pentru depanare rapidă* - aceste teste sunt în general de dimensiuni mici. Elevul poate urmări ușor comportarea programului, poate sesiza dacă valorile unor variabile nu sunt cele așteptate etc.; în această categorie intră testul din exemplu și încă 2-3 teste mici generate manual;
- *teste pentru eficiență* - acestea sunt în general testele mari. Dacă intrarea constă într-un număr, testele mari se generează manual. Pe de altă parte, dacă datele de intrare au o structură mai complexă (de exemplu, un graf) este util să se scrie un mic program care generează date de test. Se pot genera date de test aleatoare și cazuri limită (de exemplu, o matrice 100×100 plină cu 0). Cu ajutorul testelor din această categorie se verifică dacă programul are un comportament normal pentru seturi mari de date (de exemplu, nu se blochează), dacă se încadrează în timpul de execuție și dacă rezultatul afișat "pare" corect. Generatorul poate fi apoi folosit pentru a construi teste aleatoare, de dimensiuni mici și medii, pentru care verificarea corectitudinii rezultatului afișat este mai simplă; acestea pot fi folosite pentru a detecta erori care au "scăpat" de faza testelor mici generate manual.

În unele cazuri se recomandă scrierea unui program pentru verificarea unor caracteristici ale datelor de ieșire afișate. De exemplu, dacă soluția constă într-un drum de lungime minimă într-un graf, se poate scrie un program care verifică dacă muchiile afișate aparțin într-adevar grafului, dacă lungimea drumului este cea scrisă în fișier etc. Acest program folosește o parte din codul sursă al rezolvării problemei, cum ar fi subprogramul de citire a datelor.

Dacă scrieți astfel de programe pentru generare de teste și verificare, aveți grijă să nu greșiți la conceperea sau la scrierea lor! De exemplu, dacă intrarea problemei constă într-un graf conex, dar generatorul construiește grafuri nu neaparat conexe, s-ar putea să căutați o greșeală inexistentă în rezolvarea problemei!

Pe parcursul testării, citiți datele din fișierul de intrare cu numele din enunț și respectați formatul datelor de intrare! Ar fi neplăcut să pierdeți punctele acordate pentru o problemă pentru că faceți citirea din alt fișier sau scrieți rezultatele pe ecran.

În afară de testarea întregului program, poate fi util să testați și să depanați, independent, anumite subprograme. De exemplu, fie o procedură care calculează înfășurătoarea unei mulțimi de puncte. Poate că folosiți această procedură într-un context care nu se deduce direct din datele de intrare ale problemei. Pentru a o testa, puteți scrie o procedură auxiliară care citește dintr-un fișier coordonatele a N puncte, apoi scrieți/generați un astfel de fișier, apelați doar procedurile de citire și calcul al înfășurătorii convexe și examinați rezultatele.

După ce ați terminat testarea, depanarea și optimizarea, asigurați-vă că programul respectă formatul de afișare din enunț. Transformați în comentarii instrucțiunile care produceau *output* suplimentar pentru depanare și instrucțiunile care așteptau ca utilizatorul să apese o tastă, dacă aveți așa ceva (de exemplu, dacă ați afișat ceva pe ecran). La evaluare, nimeni nu va apăsa o tastă după terminarea programului (în afară de cazul în care acest lucru este specificat explicit în enunț), ceea ce va duce la depășirea timpului de execuție!

Concluzii

Testarea și depanarea programelor sunt operațiile care vă asigură că algoritmul de rezolvare de care sunteți atât de încântat va fi concretizat într-un program care va obține punctajul dorit.

Pentru începători, s-ar putea ca unele dintre indicațiile din acest articol să nu fie suficient de clare, deoarece se pot referi la situații pe care aceștia nu le-au întâlnit încă. Recitiți acest articol din când în când; privindu-l prin prisma experienței câștigate, s-ar putea să obțineți noi informații utile.

Este foarte important să tratați cu seriozitate etapele de testare și depanare. Pe măsură ce le veți stăpâni din ce în ce mai bine și anumite activități vă vor intra în reflex, timpul suplimentar alocat lor va scădea, ca și numărul de erori cu care vă veți confrunta.

Mențiuni

Țin să precizez faptul că acest articol este inspirat din articolele având aceeași temă de pe site-ul web al *USA Computing* (http://ace.delos.com/usacogate), site care este dedicat pregătirii elevilor și studenților din toată lumea pentru adevăratele concursuri de informatică.

Mihai Stroe este masterand la Universitatea Politehnica din București și poate fi contactat prin e-mail la adresa mihai_stroe@yahoo.com.