



OPTIMIZAREA programelor

Mihai Stroe

Dacă în numărul anterior am tratat problema depanării și testării programelor, în acest articol vom trata problema optimizării acestora, necesară în situația în care programele depășesc cu puțin limita de timp precizată în enunțurile problemelor care trebuie rezolvate.

Acest articol tratează optimizarea programelor. Pentru a obține efectul maxim din acest articol, este recomandată parcurgerea articolului din numărul precedent, referitor la testarea și depanarea programelor.

Cele două articole sunt bazate pe experiența personală a autorului, pe observațiile mai multor elevi și studenți participanți la concursurile de informatică și pe materialele de pregătire puse la dispoziție de organizatorii *USA Computing Olympiad* (www.usaco.org).

Aceste articole se adresează în special participanților la concursurile de informatică, ținând cont de specificul acestora (timp de lucru destul de mic, probleme cu enunț clar etc.). O mare parte din observațiile conținute pot fi aplicate și în alte contexte, cum ar fi cel al dezvoltării de *software*; pe de altă parte, date fiind situațiile mai complexe din aceste contexte, există multe tehnici pe care nu le-am descris aici. Există posibilitatea de a aborda mai pe larg aceste subiecte într-un articol viitor.

Toate articolele din numerele anterioare ale revistei, care sunt menționate în această expunere, se găsesc în format *pdf* pe *site*-ul revistei, www.ginfo.ro.

Introducere

A optimiza un program înseamnă a-l modifica pentru a obține un program mai rapid.

Prima observație este ca un program mai rapid, dar care dă rezultate incorecte, este în cele mai multe cazuri inutil. De aceea, se recomandă ca, la rezolvarea unei probleme, să se scrie programe clare și ușor de depanat.

Aceste aspecte sunt, în prima fază, mai importante decât viteza de execuție. Este preferabil să avem un program care funcționează corect pe care să îl optimizăm, față

de un program greu de depanat și care produce rezultate eronate (chiar dacă le produce repede).

Pentru a avea un program rapid este esențial să avem un algoritm rapid. Astfel, un program care implementează un algoritm cu ordinul de complexitate $O(N^3)$ va rula, de obicei, mult mai încet decât unul cu ordinul de complexitate $O(N^2)$ pentru valori mari ale lui N , indiferent cât de mult îl vom optimiza. De aceea, găsirea unui algoritm bun de rezolvare este mai importantă decât optimizarea.

Nici acest pas nu trebuie dus la extrem. Dacă un algoritm mai lent, dar ușor de implementat, se va încadra în timp, acesta este preferabil algoritmului optim; în acest caz dispăre necesitatea optimizării.

În continuare vom presupune că se optimizează un program de dimensiune medie sau mare, care rezolvă o problema dată în concurs. Optimizarea unui program de 10-15 linii poate fi tratată mult mai simplu.

Pe parcursul etapei de optimizare se recomandă păstrarea tuturor versiunilor corecte ale programului, prin salvarea lor cu alte nume. Se elimină astfel necesitatea rescrierii unor mari părți de cod pentru a reveni la o variantă anterioară, dacă se dovedește că optimizările curente nu funcționează. Se recomandă ca numele unei versiuni să conțină și numărul acesteia, pentru a avea o imagine mai clară a etapelor din evoluția programului.

În același scop se poate folosi comentarea secțiunilor de cod care vor fi optimizate. Recomand această variantă pentru înlocuirea unor secțiuni reduse ca dimensiuni. Un program cu multe comentarii mari este mai greu de modi-



ficat, deoarece trebuie să folosim *PageUp* - *PageDown* de mai multe ori pentru a naviga între secțiuni. În plus, dacă totuși alegeți varianta comentării blocurilor mari, recomand inserarea în comentarii a unor informații care să permită revenirea rapidă la o versiune funcțională anterioară.

Programul se va optimiza până în momentul în care apreciem că se va încadra în timp pentru testele cele mai dificile pe care le putem construi. Un program care merge instantaneu nu va primi mai multe puncte decât un program care rulează foarte aproape de limita de timp, dar nu o depășește niciodată; optimizarea celui de-al doilea program este inutilă, iar timpul care ar fi alocat acesteia poate fi folosit în alte scopuri.

Pentru a demonstra importanța optimizării, amintesc o realizare a unei echipe a *Universității "Politehnica" din București*, din care am făcut parte, la etapa regională a concursului *ACM* din anul 2000.

Echipa a rezolvat o problemă dificilă cu un algoritm de complexitate $O(N^2)$, pentru care soluția oficială (pe care nu am reușit să o găsim) avea ordinul de complexitate $O(N \cdot \log N)$, folosind tehnicile de optimizare prezentate în acest articol.

Rezolvarea inițială, neoptimizată, nu se încadra în timp, motiv pentru care problema a fost abandonată... până în ultimele 7 minute ale probei, când nu mai era nimic altceva de făcut. Totuși, problema a fost rezolvată; au fost folosite structuri de date mai eficiente, o sortare ineficientă a fost înlocuită cu *QuickSort*, după sortare au fost eliminate duplicatele, s-au făcut câteva optimizări de cod etc.

Se pare că nici o altă echipă nu a rezolvat problema respectivă.

Ca fapt divers, am fi obținut locul 1 și fără a rezolva această problemă, dar în alte situații optimizarea unui program poate fi decisivă pentru obținerea unei performanțe. Există multe exemple în acest sens.

Optimizări în cadrul algoritmului

Există două reguli importante pentru optimizarea în cadrul unui algoritm:

- *Don't Do Anything Stupid!* – Nu face nimic aiurea!
- *Don't Do Anything Twice!* – Nu face un lucru de două ori!

Pentru a respecta aceste reguli este necesar să identificăm ce anume se face de mai multe ori și ce anume este inutil. Vom da exemple pentru ambele situații.

Pentru a nu calcula același lucru de două ori, folosim tehnici de preprocesare. Acestea sunt prezentate pe larg într-un articol din numărul din martie 2003 al *GInfo* și nu vor fi reluate în această expunere.

De asemenea, în multe cazuri este recomandată eliminarea valorilor duplicate dintr-o anumită structură, sau compactarea anumitor secvențe.

Felul în care se va evita lucrul inutil într-un program se poate exemplifica în multe feluri. Un exemplu foarte clar constă în generarea numerelor prime; pentru a determina

dacă un număr N este prim îl vom împărți nu la toate numerele mai mici ca N , ci la toate numerele mai mici sau egale cu \sqrt{N} . Dacă generăm și memorăm toate numerele prime începând de la 2, atunci la testarea numărului N îl vom împărți la toate numerele prime mai mici sau egale cu \sqrt{N} . De asemenea, nu vom testa numerele pare mai mari decât 2.

Faptul că, pentru o anumită problemă, aveți un algoritm de complexitate $O(N^2)$ nu înseamnă că trebuie ca toți pașii să fie $O(N^2)$. De exemplu, transformarea unei sortări din *BubbleSort* în *QuickSort* poate duce la un câștig important de viteză, chiar dacă un pas ulterior al algoritmului rămâne $O(N^2)$. Generalizând, dacă putem optimiza anumiți pași ai rezolvării, este bine să o facem, chiar dacă ordinul final de complexitate rămâne neschimbat.

În multe probleme, spațiul soluțiilor conține soluții simetrice. De exemplu, în cazul problemei găsirii tuturor variantelor de așezare a N dame pe o tablă de șah astfel încât să nu se atace reciproc, se pot obține foarte ușor soluții simetrice cu o anumită soluție. Încercați să folosiți simetriile pentru a reduce timpul de execuție, parcurgând o parte mai mică din spațiul soluțiilor și afișând soluțiile simetrice. Aveți grijă la soluțiile simetrice cu ele însele, deoarece acestea nu trebuie afișate de mai multe ori.

În unele probleme, este util să descompunem spațiul analizat. De exemplu, multe din problemele *NP* pe grafuri se pot rezolva independent pe componentele conexe ale grafului; luarea în considerare a acestui fapt duce la o creștere substanțială a vitezei.

Există probleme care se pot rezolva mai eficient dacă datele sunt procesate în altă ordine față de cea din fișierul de intrare. Gândiți-vă la modul în care este afectată rezolvarea dacă datele de intrare sunt sortate, parcurse în ordine inversă, sau parcurse de mai multe ori în ordine aleatoare.

Pentru metode eficiente de abordare a problemelor *NP*, studiați articolul din numărul din octombrie 2003 al *GInfo*. Sunt prezentate idei ca: simplificarea problemei, eliminarea explorării situațiilor neinteresante, explorarea "*Depth First Search with Iterative Deepening*" etc.

Există probleme pentru care anumiți pași se pot rezolva cu complexitate logaritmică, în loc de complexitate liniară. Pentru mai multe detalii, studiați articolul pe această temă apărut în *GInfo* în noiembrie 2003.

În problemele în care apar calcule cu numere mari (cu câteva zeci/sute de cifre), de multe ori se folosește un vector de cifre pentru memorarea numerelor mari. Elementele acestui vector sunt numere între 0 și 9 inclusiv.

O variantă de optimizare a operațiilor cu numere mari constă în a lucra în baza 10^K . Astfel, elementele vectorului



care reprezintă un număr mare vor fi întregi între 0 și 10^K-1 inclusiv. Evident, un număr cu C cifre va corespunde unui vector cu C/K elemente, deci se vor efectua mai puține operații; astfel, adunarea a două numere mari constă în procesarea a C/K poziții în loc de C poziții etc.

Se recomandă mare atenție la afișarea numerelor mari. Astfel, de exemplu, numărul 909 este reprezentat în baza 100 prin două elemente 9; trebuie avut grijă să nu afișăm 99.

Unii autori recomandă folosirea unor baze de numerație care să fie puteri ale lui 2, cum ar fi 256. Această abordare oferă o eficiență ridicată, dar programele obținute pot deveni greu de depanat. Personal, consider că varianta cu baza 10^K este satisfăcătoare pentru valori bine alese ale lui K .

O ultimă optimizare menționată aici constă în preferarea algoritmilor care lucrează cu numere întregi, față de cei pe numere reale. Motivul este simplu: operațiile cu numere întregi sunt mai rapide.

De exemplu, dacă datele de intrare sunt numere reale cu cel mult trei cifre înainte de virgulă și cel mult două cifre după, ele pot fi transformate în întregi de cinci cifre prin înmulțirea cu 100, după care toate operațiile sunt mult mai rapide.

Există și exemple de probleme în care transformarea nu este la fel de evidentă.

Optimizări de cod

Aceste optimizări se referă la modificări minore aduse codului sursă, cu rezultate importante.

Prima categorie de astfel de optimizări ține de preprocesare, tehnica amintită anterior. Astfel, dacă valoarea unei expresii este folosită de mai multe ori, ea poate fi memorată într-o variabilă, pentru a nu fi recalculată de fiecare dată. Mai multe detalii se găsesc în articolul dedicat acestui subiect.

De exemplu, în C/C++, în locul secvenței:

```
for (i=2; i <= sqrt(n); i++)
{
    //prelucrări
}
```

este indicat să se folosească secvența:

```
radical <= sqrt(n);
for (i=2; i <= radical; i++)
{
    //prelucrări
}
```

O categorie de optimizări de cod care nu ține de codul efectiv constă în alegerea opțiunilor de compilare. Astfel, pe parcursul testării și depanării programului se folosesc opțiuni recomandate în acest sens, în timp ce în etapa optimizării testarea se realizează folosind opțiunile "de vite-

ză". De exemplu, în *Pascal*, verificările de tipul *Range Checking* (Verificarea depășirii domeniului – §R) și *Stack Checking* (Verificarea depășirii stivei – §S) sunt setate pe minus.

Apelurile de funcții și proceduri introduc un timp suplimentar pentru alocarea pe stivă a spațiului pentru parametri, variabile locale etc. Dacă o funcție este apelată de puține ori, acest timp este neglijabil, dar pentru o funcție scurtă care se apelează de un milion de ori, timpul suplimentar începe să conteze. În aceste cazuri este recomandată scrierea *inline* a codului funcției. Dimensiunea programului poate crește, deoarece aceeași funcție poate fi apelată în mai multe locuri, cu parametri diferiți. De aceea, în multe cazuri, în etapa de implementare și depanare se va folosi funcția; trecerea la varianta *inline* se va face doar dacă programul nu este suficient de rapid.

Evident, dacă, în varianta neoptimizată a programului, o funcție este apelată de mai multe ori în același context (aceiași parametri etc.) se va folosi preprocesarea și se vor reține valorile deja calculate.

Dacă nu se dorește scrierea *inline* a unei funcții, o metodă de optimizare constă în reducerea numărului de parametri și variabile locale.

Folosirea tipurilor de date adecvate influențează viteza algoritmului. De exemplu, dacă avem un vector de întregi între 0 și 25 și îl inițializăm de multe ori cu anumite valori, îl vom declara vector cu elemente de tipul *byte* (în *Pascal*) și nu *longint*. Astfel, operațiile care implică întreaga structură se vor realiza mai rapid, deoarece zona de memorie accesată va fi mai mică.

Există și cazuri în care folosirea unui tip de date cu un domeniu mai mare poate duce la creșterea vitezei. Se recomandă testarea diferitelor variante.

Limbajul C/C++ oferă multe optimizări de cod care nu există în *Pascal*. Vom examina câteva dintre ele.

În aceste limbaje există funcții de genul *memcpy()*, care copiază eficient un bloc de memorie. Este recomandată folosirea acestor funcții, față de varianta în care copiem, pe rând, fiecare element. În *Pascal* există procedura similară *fillchar* pentru inițializarea unei zone de memorie, se permite copierea integrală a unui vector peste altul, dar nu există funcții și proceduri eficiente pentru copierea unei anumite zone.

De fapt, în general, dacă limbajul de programare pune la dispoziție o anumită funcție pentru a realiza o anumită operație, folosirea funcției respective este preferabilă scrierii unei funcții proprii.

În unele probleme de *programare dinamică* pe matrice putem memora doar ultimele două linii ale matricei, ele-

mentele din linia i calculându-se pe baza celor din linia $i-1$. Fie aceste linii reprezentate în doi vectori, A și B . De obicei, valorile din B sunt calculate pe baza celor din A , după care A este suprascris cu B și se trece la pasul următor.

Acest pas se poate face mult mai ușor dacă pentru A și B folosim *point*-eri; astfel, copierea constă într-o interschimbare de *point*-eri, deoarece valorile care se aflau în B vor fi oricum suprascrise. Pe de altă parte, în *Pascal* folosirea *point*-erilor pe parcursul rezolvării poate duce la creșterea timpului de execuție.

Se recomandă testarea ambelor variante și alegerea celei mai eficiente. De fapt, există o variantă care elimină ambele neajunsuri; lăsăm găsirea ei ca exercițiu pentru cititor.

Înmulțirile și împărțirile cu 2 (sau 2 la o anumită putere) se pot realiza prin deplasări pe biți la stânga, respectiv la dreapta. Pentru a ușura depanarea programului, se recomandă scrierea acestor operații în mod normal pentru etapa de testare și depanare și transformarea lor în deplasări numai dacă sunt necesare optimizări. Programatorii cu experiență pot ignora cu succes această recomandare.

Dacă o problemă se poate rezolva folosind ca structuri de date vectori, într-o primă variantă, și matrice, într-o a doua variantă, prima variantă este de preferat (presupunând că restul condițiilor, cum ar fi ordinul de complexitate, sunt identice). Motivul este următorul: pentru accesarea elementelor unei matrice se fac mai multe calcule cu indicii, decât pentru accesarea elementelor unui vector.

De asemenea, folosirea structurilor de tip înregistrare (**record** în *Pascal*, **struct** în *C/C++*) duce la încetinirea programului.

Astfel, din următoarele 3 variante de definire ale unui vector de puncte în plan (fiecare punct este definit prin două coordonate întregi, x și y):

- **type** punct = **record**
 x, y : integer;
 end;
 var puncte: **array**[1..1000] **of** punct;
- **var** puncte: **array**[1..1000, 1..2] **of** integer;
- **var** x, y : **array**[1..1000] **of** integer;

dacă eficiența contează, se va alege, în cele mai multe cazuri, cea de-a treia variantă.

În *C/C++*, unele variabile pot fi prefixate cu modificatorul **register**. Acesta îi spune compilatorului că variabila respectivă va fi folosită frecvent, deci accesul la ea ar trebui optimizat. Concret, de obicei variabila respectivă va fi păstrată într-un registru.

Opțiunea funcționează pentru variabile de tip **int** și **char**. Nu pot fi păstrate foarte multe variabile în regiștri

în același timp; dacă sunt prea multe variabile declarate, de tip **register**, compilatorul va ignora o parte din declarații.

Se recomandă folosirea acestui modificator în bucle, de exemplu, pentru variabila contor dintr-un ciclu **for**.

Tot în *C/C++*, folosirea aritmeticii *pointer*-ilor poate fi mai indicată decât folosirea indicilor într-un vector. Recomand studierea operațiilor cu *point*-eri pentru mai multe informații.

Ultima categorie de optimizări de cod amintită se referă la scrierea unor secvențe de cod în limbaj de asamblare, dacă acest lucru este permis în concurs. Din nefericire, autorul acestui articol nu este expert în acest domeniu, deci nu vă poate oferi mai multe detalii...

Concluzii

Există multe moduri de a optimiza un program. Unele din ele se referă la îmbunătățiri majore ale algoritmului, altele la îmbunătățiri minore, iar altele se referă la modificări de câteva caractere sau linii ale unor secvențe de cod.

Cunoașterea și punerea în practică a tehnicilor de optimizare prezentate în acest articol (și, eventual, a altora) pot aduce puncte prețioase, care nu s-ar fi putut obține altfel (deoarece, de exemplu, algoritmul optim de rezolvare este foarte greu de găsit, sau depășește experiența concurenților). Pe de altă parte, recomandăm insistent abordarea cu seriozitate a fazelor de găsire a algoritmului eficient, de implementare, de testare și de depanare. Fără aceasta, nu avem ce optimiza!

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

