PREPROCESAREA

Mihai Stroe

Articolul de față prezintă o categorie de optimizări importante în rezolvarea problemelor: soluțiile bazate pe preprocesare. Vom prezenta modul în care poate fi utilizată această tehnică pentru a îmbunătăți timpul de execuție al programelor.

Termenul de *preprocesare* se referă la calculul și stocarea în memorie a unor valori, în scopul folosirii lor ulterioare pe parcursul rezolvării. Există două niveluri la care se aplică aceste optimizări:

- calcule efectuate înaintea executării programului; în acest caz, programul care rezolvă problema conține tabele cu valori precalculate (constante);
- calcule realizate înaintea efectuării unor anumiți pași de rezolvare, pentru reducerea timpului de execuție a acestora.

Vom prezenta ideea de preprocesare care a apărut, în contextul olimpiadelor naționale, în 1996, la barajele pentru selectarea echipei lotului pentru *Balcaniada de Informatică*. Una dintre problemele propuse era următoarea (enunț ușor modificat):

"*Să se amplaseze pe muchiile unui cub numerele de la 1 la 12, astfel încât suma numerelor de pe muchiile care formează fiecare față să fie egală. Problema nu are date de intrare; se cer toate soluțiile.*"

Problema stabilea, prin enunț, o numerotare a muchiilor și amplasarea numărului 1 pe o anumită muchie. Timpul de execuție era de o secundă.

Evident, problema se rezolvă prin metoda *backtracking*. Din nefericire, pe calculatoarele existente la momentul respectiv, rezolvări nu foarte eficient implementate depășeau cu puțin timpul limită.

În aceste condiții, o parte dintre concurenți au aplicat următoarea tehnică:

- se rulează programul în timpul probei, obținându-se soluțiile într-un fișier;
- se observă că numărul de soluții este mic (80);
- se transformă fișierul rezultat într-un program care afișează cele 80 de soluții.

Cel mai simplu mod de a realiza transformarea constă în introducerea soluțiilor între ghilimele, prefixarea lor cu un `"writeln("` și finalizarea cu șirul `");"` (în *Pascal*). În acest mod, programul putea fi completat cu un **`"begin"`** și un **`"end."`** și ar fi rezultat un program *Pascal* corect care rezolvă problema (de fapt, afișarea se realiza într-un fișier; am ales această modalitate de prezentare pentru a simplifica înțelegerea soluției). Ideea este de a realiza afișarea, în programul care generează soluțiile, într-un mod care să ușureze cât mai mult scrierea programului care le va folosi. După cum se va vedea în continuare, pentru anumite probleme folosirea poate consta în mai mult decât tipărirea unor valori.

Bineînțeles, la corectare au apărut discuții legate de valoarea acestei rezolvări. Deoarece regulamentul nu era încălcat, s-a decis ca astfel de soluții să fie acceptate, regulă care a rămas valabilă pentru viitor. Altfel spus, programele bazate pe precalcularea soluțiilor sau a unor date ajutătoare nu pot fi depunctate; corectarea se realizează numai în funcție de rezultatele obținute la teste, fără a se ține cont de conținutul codului sursă.

Ca toate lucrurile bune care apar în contextul concursurilor (reamintim, printre altele, *backtracking*-ul "*omorât*"), această metodă a fost preluată și folosită din plin la concursurile care au urmat, inclusiv în cadrul concursurilor "*Bursele AGORA*". În prezent, cunoașterea acestei metode este considerată esențială, iar anumite probleme nu se pot rezolva eficient fără ajutorul ei.

Vom continua prin a prezenta câteva abordări din cea de-a doua categorie.

O problemă interesantă, datorită optimizărilor care pot fi introduse, este următoarea:

"*Se dau N puncte în plan de coordonate întregi ($3 \le N \le 100$). În aceste puncte locuiesc oameni. În două din aceste puncte se pot deschide restaurante. Fiecare om va mânca la restaurantul cel mai apropiat. Să se aleagă locațiile în care se vor deschide restaurante, astfel încât distanța totală parcursă de oameni de la casele lor până la restaurante să fie minimă. Oamenii care au restaurante acasă parcurg distanța 0.*"

Limitele problemei conduc la concluzia că rezolvarea prin metoda forței brute este acceptabilă. Să o examinăm:

distanța optimă devine `+infinit`
`pentru` *fiecare punct* `i` *între* `1` *și* `N` *(ales ca prim restaurant)* **`execută`**
  `pentru` *fiecare punct* `j` *între* `1` *și* N *(ales ca al doilea restaurant)* **`execută`**

distanța totală devine 0
pentru *fiecare punct* `K` *între* `1` *și* `N` **execută**
`d1` *devine distanța de la* `i` *la* `K` (*)
`d2` *devine distanța de la* `j` *la* `K` (*)
`min` *devine minimul dintre* `d1` *și* `d2`
`min` *se adună la distanța totală*
distanța totală este comparată cu distanța optimă și, dacă este cazul, o actualizăm pe aceasta.

Complexitatea acestei metode este $O(N^3)$, deci timpul de rulare ar trebui să fie rezonabil.

Cum implementăm pașii marcați cu (*)? Prima variantă constă în realizarea unei funcții `dist(i,j)` și apelarea ei:

```
function dist(i,j:integer):real;
begin
  dist:=sqrt(sqr(x[i] - x[j]) + sqr(y[i] - y[j]));
end;
```

Evident `d1` devine `dist(i,K)` și `d2` devine `dist(j,K)`.

Se efectuează $O(N^3)$ apeluri de funcție, ceea ce duce la performanțe slabe.

O variantă ușor îmbunătățită constă în calcularea distanțelor în cadrul programului, fără a scrie o funcție separată:

```
d1:=sqrt(sqr(x[i] - x[K]) + sqr(y[i] - y[K]));
d2:=sqrt(sqr(x[j] - x[K]) + sqr(y[j] - y[K]));
```

Nu se mai efectuează apeluri de funcții `dist`, dar, pe cazul cel mai defavorabil, se calculează aproximativ un milion de distanțe (ceea ce înseamnă un milion de extrageri de rădăcini pătrate). Soluția bazată pe preprocesare constă în definirea unei matrice `dist`:

```
var dist:array[1..100,1..100] of real;
```

Matricea va fi completată la începutul programului, imediat după citirea datelor:

```
for i:=1 to n do
  for j:=i + 1 to n do begin
    dist[i,j]:=sqrt(sqr(x[i] - x[j]) + sqr(y[i]
                                    - y[j]));
    dist[j,i]:=dist[i,j];
  end;
```

Din acest moment, distanțele între oricare două puncte sunt disponibile și pot fi folosite în orice moment prin citirea din tabelă. S-au efectuat aproximativ 5000 de operații de extragere a rădăcinii pătrate. În plus, s-au economisit înmulțiri și adresări în vectorii `x` și `y`. Complexitatea este aceeași; timpul de execuție este însă redus semnificativ.

După examinarea acestei soluții, sunt evidente alte tipuri de valori care pot fi precalculate: tabele de logaritmi, rădăcini pătrate, funcții trigonometrice, numere prime etc. De exemplu (revenind în contextul calculelor efectuate înaintea lansării în execuție), un program care va folosi intens primele 1000 de numere prime poate începe astfel:

```
const prime:array[1..1000] of longint
      = (2, 3, 5, 7, 11, 13, 17, ...);
```

Începutul programului poate fi obținut cu ajutorul unui alt program, care generează primele 1000 de numere prime și le afișează câte 20 pe linie (pentru a nu depăși numărul maxim de caractere pe linie permis), separate prin virgulă. Fișierul rezultat este apoi completat cu codul programului.

În acest caz, primele 1000 de numere prime ar fi putut fi calculate și la executarea programului; există însă contexte în care vectorii de constante sunt indispensabili.

O optimizare utilă în multe probleme constă în construirea vectorului / matricei sumelor parțiale. O vom ilustra cu ajutorul următoarelor două probleme:

- *Se consideră un vector D cu N ≤ 10000 elemente, numere întregi. Urmează K întrebări, K ≤ 10000. O întrebare constă în două numere întregi A și B, între 1 și N inclusiv, cu A ≤ B. Pentru fiecare întrebare afișați suma elementelor vectorului, situate între pozițiile A și B inclusiv.*
- *Se dă o matrice D cu M linii și N coloane (1 ≤ M, N ≤ 100). Urmează K întrebări, K ≤ 10000. O întrebare constă în 4 numere întregi L1, C1, L2, C2 cu 1 ≤ L1 ≤ L2 ≤ M și 1 ≤ C1 ≤ C2 ≤ N. Pentru fiecare întrebare, afișați suma elementelor din matrice, din dreptunghiul determinat de liniile L1 și L2 și coloanele C1 și C2.*

Metoda bazată pe parcurgerea vectorului / matricei este ineficientă și va obține foarte puține puncte. Există însă metode în care răspunsul la fiecare întrebare se calculează în timp constant, datorită unor preprocesări efectuate după citirea datelor.

Pentru prima problemă, se calculează vectorul `S` al sumelor parțiale, definit astfel:

```
S[0] = 0
S[1] = D[1]
S[2] = D[1]+D[2]
S[3] = D[1]+D[2]+D[3]
...
S[i] = suma primelor i elemente din vectorul D. Evident,
       S[i]=S[i-1]+D[i].
```

Calculul vectorului `S` se realizează în $O(N)$. După acest pas, la fiecare întrebare se răspunde scăzând `S[A - 1]` din `S[B]`. Astfel, `S[B]` este suma primelor `B` elemente, iar `S[A - 1]` este suma primelor `A - 1`; realizând scăderea prezentată anterior se obține suma elementelor situate între pozițiile `A` și `B`.

Pentru cazul matricei, calculele sunt puțin mai complicate. Definim `S[i, j]` ca fiind suma elementelor din dreptunghiul delimitat de liniile `1` și `i` și de coloanele `1` și `j`. Valorile din `S` se calculează unele din altele, astfel:

```
S[i, j]:=S[i-1, j]+S[i, j-1]-
S[i-1, j-1]+D[i, j].
```

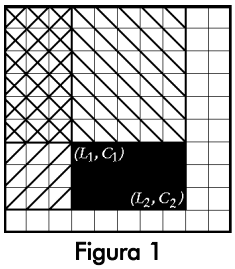

Figura 1

Cunoscând `S[i, j]`, suma elementelor din dreptunghiul determinat de liniile `L1` și `L2` și coloanele `C1` și `C2` se calculează astfel:

```
Suma:=S[L2, C2]-S[L1-1, C2]-S[L1, C2-1]
      +S[L1-1, C1-1].
```

Așadar răspunsul la o întrebare este calculat într-un timp de ordinul $O(1)$.

Calculul vectorului sumelor parțiale poate fi folosit și în unele probleme bazate pe matrice. Se pot calcula matricele sumelor parțiale pe linii, respectiv pe coloane. Mai exact, `LIN[i, j]` este suma primelor `j` elemente din linia `i`, iar `COL[i, j]` suma primelor `i` elemente din coloana `j`. Invităm cititorul să găsească o utilizare a acestor matrice.

Încheiem prezentarea cu câteva probleme rezolvabile cu ajutorul preprocesării, bazate pe noțiunile prezentate anterior.

Numere prime

"*Se dau două numere A și B, cu 0 < A < B < 5000000. Afișați suma numerelor prime aflate în intervalul* [*A, B*]. *Timp de execuție: 1 secundă/test.*"

Un program care determină numerele prime dintre A și B și calculează suma lor nu se va încadra în timp pe testele în care intervalul este mare. Există deci o soluție alternativă.

Calculăm un vector S, unde $S[i]$ conține suma primelor $i \cdot 1000$ numere prime, unde i este cuprins între 1 și 5000. Pe baza acestui vector și a cunoștințelor despre metoda vectorului sumelor parțiale, prezentate anterior, răspundem la cerința problemei astfel:

- calculăm pozițiile X și Y, multipli de 1000, situate între A și B, cu X cât mai apropiat de A și Y cât mai apropiat de B. De exemplu, pentru $A = 53478$ și $B = 674876$, avem $X = 54000$ și $Y = 674000$;
- suma numerelor prime situate între X și Y este:
$$S[Y/1000] - S[X/1000 - 1];$$
- suma numerelor prime dintre A și B se obține folosind suma determinată mai sus, la care adunăm numerele prime dintre A și X și pe cele dintre Y și B.

Vectorul S este calculat înainte de executarea programului, cu ajutorul unui program auxiliar (care, evident, se va termina după mai mult de o secundă, dar timpul său de rulare nu contează la evaluare). Programul auxiliar scrie vectorul S la începutul unui fișier; sumele sunt puse câte 10 - 20 pe linie și separate prin virgule. Fișierul este completat cu programul care realizează citirea datelor de intrare și calculează răspunsul, conform metodei de mai sus.

Pătratul și labirintul

"*Se dă o matrice cu elemente de 0 și 1, cu N linii și N coloane. Un pătrat pleacă din poziția (L1, C1) și trebuie să ajungă în poziția (L2, C2). Pătratul constă într-o matrice pătratică de cel puțin un element (deci poate fi un pătrat 2 × 2, 3 × 3 etc.), nu se poate deplasa peste elemente cu valoarea 1 și nu poate ieși din matrice. Să se determine cea mai mare latură a unui pătrat care poate să plece din (L1, C1) și să ajungă în (L2, C2), respectând restricțiile.*"

De exemplu, pentru matricea:

```
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 1 0 1
0 0 0 0 0 0
0 0 0 0 0 0
1 0 0 0 0 0
```

și $L1 = 1$, $C1 = 1$, $L2 = 4$, $C2 = 4$, pătratul maximal are latura 2 și poate fi deplasat prin trei mutări în jos și trei mutări în dreapta.

Rezolvarea clasică a acestei probleme se bazează pe algoritmul lui *Lee*. Acesta (considerat cunoscut în cadrul expunerii de față) este realizat pentru un pătrat de latura 1 și se bazează pe introducerea într-o coadă a pozițiilor în care se poate afla pătratul, în ordinea distanței lor față de poziția inițială.

Pentru a adapta algoritmul la pătratul de dimensiune variabilă, putem alege următoarea soluție:

pentru *fiecare dimensiune* `D` *a pătratului* **execută**
  se plasează pătratul în poziția inițială (dacă este posibil);
  se încearcă deplasarea pătratului, conformă algoritmului lui Lee;
  dacă deplasarea nu a reușit, se afișează soluția de la pasul precedent și se termină execuția.

În cadrul algoritmului lui *Lee*, verificarea dacă pătratul se poate deplasa într-o poziție vecină se realizează în D pași, examinând cele D elemente noi care urmează a fi ocupate prin deplasarea pătratului (pentru ca aceasta să fie posibilă, toate trebuie să fie nule). Complexitatea pe cazul cel mai defavorabil este $O(N^4)$.

Prima îmbunătățire a programului constă în calculul valorii optime a lui D cu ajutorul căutării binare. Se încearcă deplasarea unui pătrat de dimensiune $N/2$, dacă aceasta a reușit se încearcă un pătrat mai mare, în caz contrar un pătrat mai mic etc. Complexitatea a fost redusă la $O(N^3 \cdot \log N)$.

Se observă că verificarea dacă pătratul poate fi deplasat într-o poziție vecină se poate realiza cu ajutorul preprocesării. Se calculează matricele LIN și COL amintite anterior:

- $LIN[i, j]$ este suma primelor j elemente din linia i;
- $COL[i, j]$ este suma primelor i elemente din coloana j.

În acest context, condiția pentru ca, de exemplu, un pătrat de dimensiune D, aflat în poziția (L, C), să poată fi deplasat cu o poziție mai sus, este ca diferența $LIN[L\text{-}1, C\text{+}D\text{-}1] - LIN[L\text{-}1, C\text{-}1]$ să fie 0. Diferența respectivă dă suma elementelor situate pe linia L-1, între coloanele C și C+D-1. Pot fi găsite condiții similare pentru deplasarea în jos, spre stânga și spre dreapta. Complexitatea noii rezolvări este $O(N^2 \cdot \log N)$. Lăsăm ca exercițiu pentru cititor încercarea rezolvării problemei cu o complexitate $O(N^2)$.

Concluzie

Preprocesarea este o tehnică utilă. Stăpânirea ei poate conduce în multe cazuri la reducerea timpului de execuție, prin evitarea recalculării unor valori. În alte cazuri, se poate ajunge și la scăderi importante ale complexității.