Optimizarea programelor folosind OPERAȚII PE BIȚI

Cosmin-Silvestru Negruşeri

În cadrul acestui articol vă vom prezenta câteva metode prin care programele se pot optimiza dacă utilizăm eficient operațiile pe biți sau folosim operații pe biți în locuri în care, la prima vedere, nu s-ar părea că ar fi necesare.

De cele mai multe ori, atât în concursuri cât și în viața de zi cu zi a programatorului, atunci când implementăm o metodă care este folosită de o aplicație și vrem să facem această metodă eficientă ca timp de execuție, suntem învățați din școală (sau ar trebui să fim învățați, chiar dacă unii dintre profesori consideră că cel mai important algoritm învățat în liceu este *backtraking*-ul, soluția tuturor problemelor) să ne uităm la complexitatea algoritmului implementat și dacă observăm că în practică algoritmul este mai încet decât ne dorim noi să fie să încercăm să găsim un algoritm cu un ordin de complexitate mai mic.

Nu trebuie să uităm că ceea ce numim complexitatea unui algoritm este o aproximare a vitezei unui algoritm și nu o măsură absolută. Pentru dimensiuni mici ale datelor, câteodată nu se observă diferența dintre $O(n)$ și $O(n \cdot \log n)$ sau $O(n^{3/2})$. Mie personal mi s-a întâmplat ca la implementarea soluției unei probleme care în engleză se numește "*Bottleneck Minimal Spanning Tree*" să observ că rezolvarea în $O(m \cdot \log m)$ (folosind algoritmul de găsire a arborelui parțial de cost minim al lui *Kruskal*) să fie mai rapidă decât rezolvarea mai laborioasă în $O(m)$ a acestei probleme. Deci trebuie dată o atenție egală cu cea acordată complexității algoritmului și dimensiunii factorilor constanți care apar.

În continuare vom încerca să găsim soluții mai rapide decât cele naive pentru unele operații de bază (toate operațiile vor fi implementate pentru întregi pozitivi reprezentați pe 32 de biți).

Problema 1

Să se determine numărul de biți 1 din reprezentarea binară a numărului *n*.

Rezolvarea naivă a acestei probleme ar consta în parcurgerea secvențială a biților lui *n*.

În continuare vă prezint această rezolvare:

```
int count(long n){
  int num = 0;
  for (int i = 0; i < 32; i++)
    if (n & (1 << i)) num++;
  return num;
}
```

Dacă ne uităm cu atenție și analizăm rezultatul operației `n & (n - 1)` putem obține o soluție mai bună.

Să luăm un exemplu:

```
n          = (11011101010000)2
n - 1      = (11011101001111)2
n & (n - 1) = (11011101000000)2
```

Se vede clar de aici că efectul operației `n & (n - 1)` este anularea celui mai nesemnificativ bit cu valoarea 1.

De aici ideea algoritmului:

```
int count(long n){
  int num = 0;
  if (n)
    while (n &= n - 1) num++;
  return num;
}
```

Dar este acest algoritm mai rapid? Am testat pentru toate numerele de la 1 la 2^{24} și rezultatele au fost 2,654 secunde folosind metoda naivă și 0.821 folosind a doua metodă.

Rezultatul mult mai bun al celei de-a doua metode se bazează în principal pe faptul că ea execută un număr de pași egali cu numărul de biți cu valoarea 1 din număr, deci în medie jumătate din numărul de pași efectuați de prima metodă.

Problema 2

Să se determine paritatea numărului de biți de 1 din reprezentarea binară a unui număr *n*.

Din cele prezentate mai sus se pot determina două metode evidente:

```
int parity(long n){
  int num = 0;
  for (int i = 0; i < 32; i++)
    if (n & (1 << i)) num ^= 1;
  return num;
}

int parity(long n){
  int num = 0;
  if (n)
    while (n &= n - 1) num ^= 1;
  return num;
}
```

Putem obține o a treia rezolvare făcând câteva observații pe un exemplu:

Considerăm $n = (11011011)_2$. Rezultatul căutat este dat de valoarea `1 ^ 1 ^ 0 ^ 1 ^ 1 ^ 0 ^ 1 ^ 1`.

Împărțim pe *n* în partea lui superioară și partea lui inferioară:

```
1 1 0 1
  ^
1 0 1 1
  =
0 1 1 0
```

Aplicăm asupra rezultatului același procedeu:

```
0 1
 ^
1 0
 =
0 1
```

și

```
 1
 ^
 1
 =
 0
```

Să scriem algoritmul care reiese din acest exemplu, luând în considerare faptul că numărul *n* este reprezentat pe 32 de biți:

```
int parity(long n){
  n = ((0xFFFF0000 & n) >> 16) ^ (n & 0xFFFF);
  n = ((0xFF00 & n) >> 8) ^ (n & 0xFF);
  n = ((0xF0 & n) >> 4) ^ (n & 0xF);
  n = ((12 & n) >> 2) ^ (n & 3);
  n = ((2 & n) >> 1) ^ (n & 1);
  return n;
}
```

Deși constanta multiplicativă este mai mare, numărul de operații are ordin logaritmic față de numărul de biți ai unui cuvânt al calculatorului.

Problema 3

Să se determine cel mai puțin semnificativ bit de 1 din reprezentarea binară a lui *n*.

Rezolvarea naivă se comportă bine în medie, deoarece numărul de cicluri până la găsirea unui bit cu valoarea 1 este de obicei mic, dar din cele discutate mai sus putem găsi ceva mai bun.

Așa cum am arătat `n & (n - 1)` are ca rezultat numărul *n* din care s-a scăzut cel mai puțin semnificativ bit.

Folosind această idee obținem următoarea funcție:

```
int low1(long n){
  return n ^ (n & (n - 1));
}
```

Exemplu:

```
n               = (11011000)2
n - 1           = (11010111)2
n & (n - 1)     = (11010000)2
n               = (11011000)2
n ^ (n & (n-1)) = (00001000)2
```

Această funcție este foarte importantă pentru structura de date *Arbori Indexați Binar* prezentată într-un un articol mai vechi din *GInfo*.

Problema 4

Să se determine cel mai semnificativ bit cu valoarea 1 din reprezentarea binară a lui *n*.

Putem aplica și aici ideile prezentate mai sus: cea naivă și cea cu eliminarea biților, dar putem găsi și ceva mai bun.

O abordare ar fi cea a căutării binare (aplicabilă și în problema anterioară).

Verificăm dacă partea superioară a lui *n* este 0. Dacă nu este 0, atunci căutăm bitul cel mai semnificativ din ea, iar dacă este, ne ocupăm de partea inferioară, deci reducem la fiecare pas problema la jumătate.

```
int high1(long n) {
  long num = 0;
  if (!n) return -1;
  if (0xFFFF0000 & n) {
    n = (0xFFFF0000 & n)>>16;
    num += 16;
  }
  if (0xFF00 & n) {
    n = (0xFF00 & n) >> 8;
    num += 8;
  }
  if (0xF0 & n) {
    n = (0xF0 & n) >> 4;
    num += 4;
```

```
  }
  if (12 & n) {
    n = (12 & n) >> 2;
    num += 2;
  }
  if (2 & n) {
    n = (2 & n) >> 1;
    num += 1;
  }
  return 1 << num;
}
```

Problema 5

Să se determine indexul celui mai semnificativ bit de 1 din reprezentarea binară a lui *n*.

Metodele prezentate la rezolvarea problemei 4 pot fi folosite și aici, dar vom prezenta o nouă rezolvare.

Să luăm următorul șir de operații pentru un exemplu:

$n = (10000000)_2$
$n = n \mid (n >> 1)$
$n = (11000000)_2$
$n = n \mid (n >> 2)$
$n = (11110000)_2$
$n = n \mid (n >> 4)$
$n = (11111111)_2$

Se observă că aplicând o secvență asemănătoare de instrucțiuni cu cea de mai sus putem face ca un număr *n* să se transforme în alt număr care are un număr de biți de 1 egal cu 1 + indexul celui mai semnificativ bit cu valoarea 1 din *n*. De aici algoritmul este următorul:

```
int indexHigh1(long n){
  n = n | (n >> 1);
  n = n | (n >> 2);
  n = n | (n >> 4);
  n = n | (n >> 8);
  n = n | (n >> 16);
  return count(n) - 1;
}
```

Pentru ca această metodă să fie eficientă ne trebuie o metodă **count** eficientă.

Problema 6

Să se determine dacă *n* este o putere a lui 2 (întrebare interviu *Microsoft*).

```
int isTwoPower(long n){
  return (n & (n-1) == 0);
}
```

O altă abordare: preprocesarea

Operațiile prezentate mai sus sunt implementate în limbajele de asamblare, dar câteodată se folosesc algoritmi naivi și atunci o implementare inteligentă ajunge să fie mai rapidă decât instrucțiunea din limbajul de asamblare. Pentru unele supercalculatoare aceste instrucțiuni sunt instrucțiuni ale procesorului.

Se pot găsi alți algoritmi mai rapizi decât cei de mai sus, dar de obicei găsirea unui algoritm mai inteligent este mai grea decât folosirea unei abordări banale care aduce un câștig foarte mare: preprocesarea.

În continuare vom rezolva problemele prezentate anterior folosindu-ne de preprocesarea datelor.

Problema 1

Determinăm, pentru numerele de la 0 la 2^{16} - 1, un tablou `num`, în care `num[i]` este egal cu numărul de biți de 1 din reprezentarea binară a lui `i`.

De aici obținem soluția:

```
int count(long n){
  return num[n & 0xFFFF] + num[n >> 16];
}
```

Problema 2

Determinăm, pentru numerele de la 0 la 2^{16} - 1, un tablou `par`, în care `par[i]` este egal cu paritatea numărului de biți de 1 din reprezentarea binară a lui `i`.

De aici obținem soluția:

```
int parity(long n){
  return par[n & 0xFFFF] ^ par[n >> 16];
}
```

Problema 3

Determinăm, pentru numerele de la 0 la 2^{16} - 1, un tablou `l`, în care `l[i]` este egal cu cel mai nesemnificativ bit de 1 din reprezentarea binară a lui `i`.

De aici obținem soluția:

```
int low(long n){
  if (l[n & 0xFFFF]) return l[n & 0xFFFF];
  else return l[n >> 16] << 16;
}
```

Problema 4

Determinăm, pentru numerele de la 0 la 2^{16} - 1, un tablou `h`, în care `h[i]` este egal cu cel mai semnificativ bit de 1 din reprezentarea binară a lui `i`.

De aici obținem soluția:

```
int high(long n){
  if (h[n >> 16]) return h[n >> 16] << 16;
  else return h[n & 0xFFFF];
}
```

Problema 5

Determinăm, pentru numerele de la 0 la 2^{16} - 1, un tablou `iH`, în care `iH[i]` este egal cu indexul celui mai semnificativ bit de 1 din reprezentarea binară a lui `i` și este egal cu -1 pentru 0.

De aici obținem soluția:

```
int indexHigh(long n) {
  if (iH[n >> 16] != -1) return iH[n >> 16] + 16;
  else return iH[n & 0xFFFF];
}
```

Algoritmi mai serioși

Deși ceea ce am prezentat mai sus sunt mai degrabă trucuri de implementare decât algoritmi serioși și în general nu se acordă așa mare importanță detaliilor de genul acesta, câteodată asemenea trucuri se pot dovedi importante mai ales în timpul concursurilor.

Un exemplu ar fi problema rundei 12 de la concursul de programare *Bursele Agora*. Acolo se cerea găsirea numărului de dreptunghiuri conținute de o matrice binară *a* care au în colțuri 1.

Un algoritm de complexitate $O(n^2 \cdot m)$ ar fi fost următorul: se consideră fiecare pereche formată din două linii și se numără câte perechi $(a_{i,k}, a_{j,k})$ pentru care $a_{i,k} = 1$ și $a_{j,k} = 1$ există.

Fie acest număr *x*. Atunci numărul de dreptunghiuri care au colțurile pe aceste două linii va fi $x \cdot (x - 1) / 2$.

Un asemenea algoritm ar fi luat numai 64 de puncte deoarece restricțiile date erau mari ($n \le 200$, $m \le 2500$).

Putem rescrie condiția `a[i][k] == 1 && a[j][k] == 1` ca și `a[i][k] ^ a[j][k] == 1`).

Deci putem modifica rezolvarea anterioară în modul următor:

```
pentru fiecare i
  pentru fiecare j > i
    b = a[i] ^ a[j];
  sfârșit pentru
sfârșit pentru
  numaraBitiUnu(b)
```

Dacă în loc să păstrăm în `a[i][j]` un element al matricei binare, păstram 16 elemente, atunci în linia `b = a[i] ^ a[j]` se efectuează cel mult 2500 / 16 calcule în loc de 2500 de calcule. Acum ce a rămas de optimizat este metoda `numaraBitiUnu`.

Dacă folosim preprocesarea, atunci această metodă va fi compusă și ea 2500/16 calcule. Se observă că în acest fel am mărit viteza de execuție a algoritmului cu un factor de 16.

Alt exemplu ar fi problema rundei 17. În acea problemă se cere determinarea numărului de sume distincte care se pot obține folosind niște obiecte cu valori date folosind fiecare obiect pentru o sumă cel mult o dată.

Din nou o rezolvare directă, folosind marcarea pe un șir de valori logice n-ar fi obținut punctaj maxim. Pentru obținerea punctajului maxim următoarea idee ar fi putut fi folosită cu succes: șirul de valori logice se condensează într-un șir de întregi reprezentați pe 16 biți, și acum actualizarea se face asupra a 16 valori deodată. Deci și aici se câștigă un factor de viteză egal cu 8 (datorită detaliilor de implementare).

Un al treilea exemplu ar fi problema *Viteza* de la a 9-a rundă a concursului organizat de ***.campion*** anul acesta. Acolo se cerea determinarea celui mai mic divizor comun a două numere binare de cel mult 10000 de cifre.

După cum se știe, complexitatea algoritmului de determinare a celui mai mare divizor comun a două numere este logaritmică, acest fapt este adevărat atunci când operațiile efectuate sunt constante, dar în rezolvarea noastră apar numere mari, deci o estimare a numărului de calcule ar fi $maxn \cdot maxn \cdot \log maxn$ (împărțirea are în medie costul $maxn \cdot maxn$ dacă nu implementăm metode mai laborioase).

Acest număr este prea mare pentru timpul de rezolvare care ne este cerut. Pentru a evita împărțirea, care este operația cea mai costisitoare, putem folosi algoritmul de determinare a celui mai mare divizor comun binar care se folosește de următoarele proprietăți:

- $cmmdc(a, b) = 2 * cmmdc(a / 2, b / 2)$, dacă *a* și *b* sunt numere pare;
- $cmmdc(a, b) = cmmdc(a, b / 2)$, dacă *a* este impar;
- $cmmdc(a, b) = cmmdc(a - b, b)$, dacă *a* este impar, *b* este impar și $a > b$.

Aplicând acest algoritm numărul de calcule s-a redus la $maxn \cdot \log maxn$.

Dar nici acest număr nu este suficient de mic, și deci pentru accelerarea algoritmului folosim ideea utilizată în cele două probleme de mai sus.Împachetăm numărul în întregi în pachete de câte 30 de biți, și atunci deplasarea la stânga (adică împărțirea la 2) și scăderea vor fi de 30 de ori mai rapide.

Să încercăm acum rezolvarea unei probleme în care eficiența este cea mai importantă, mai importantă decât lizibilitatea codului obținut. Problema are următorul enunț:

Se consideră un număr n. Să se determine numerele prime mai mici sau egale cu n, unde $n \le 10.000.000$.

Practic această problemă ne va fi utilă în testarea rapidă a primalității numerelor mai mici sau egale cu *n* .

Putem încerca mai multe rezolvări, dar în practică cea mai rapidă se dovedește de obicei cea numită *ciurul lui Eratostene*. Ea se învață în școli încă din clasa a 6-a, deci nu o voi mai explica. Implementarea banală a algoritmului ar fi următoarea:

```
#define maxsize 10000000

char at[maxsize]; //vector de valori logice în care nume-
                  //rele prime vor fi marcate cu 0
int n;

int isPrime(int x){
  return (!at[x]) ;
}
```

```
void sieve(){
  int i, j;
  memset(at,0,sizeof(at)) ;
  for (i = 2; i <= maxsize; i++)
    if (!at[i])
      for (j = i + i; j <= maxsize; j += i)
        at[j]=1; //marcăm toți multipli lui i ca fiind
                 //numere prime
}
```

Observăm că jumătate din timpul folosit de noi la preprocesare este pierdut cu numerele pare. Marcându-le de la început vom putea ignora numerele pare în preprocesare:

```
#define maxsize 10000000

char at[maxsize];
int n;
int isPrime(int x){
  return (!at[x]) ;
}

void sieve(){
  int i, j;
  memset(at,0,sizeof(at));
  for (i = 4; i <= maxsize; i += 2)
    at[i] = 1;
  for (i = 3; i <= maxsize; i += 2)
    if (!at[i])
      for (j = i + i + i; j <= maxsize; j += i + i)
        at[j] = 1;
}
```

La marcarea multiplilor numărului prim *i* toate numerele până la $i \cdot i$ au fost deja marcate deoarece $i \cdot i$ este cel mai mic număr care nu este divizibil cu numere naturale mai mici sau egale cu $i - 1$.

Deci, avem o a treia versiune a programului:

```
#define maxsize 10000000
char at[maxsize];
int n;
int isPrime(int x){
  return (!at[x]) ;
}

void sieve(){
  int i, j;
  memset(at,0,sizeof(at));
  for (i = 4; i <= maxsize; i += 2)
    at[i] = 1;
  for (i = 3; i <= maxsize; i += 2)
    if (!at[i])
      for (j = i * i; j <= maxsize; j += i + i)
        at[j] = 1;
}
```

Ultima optimizare este optimizarea spațiului necesar. Într-un tip de date **char** putem împacheta opt valori logice și, punând un test suplimentar în metoda `isPrime` putem elimina și numerele pare, astfel vom avea nevoie de un spațiu de 16 ori mai mic.

```
#define maxsize 10000000

char at[maxsize/16];
int n;

int isPrime(int x) {
  if (!(x & 1))
    if (x==2) return 0 ;
    else return 1 ;
    else return (at[((x - 3) >> 1) >> 8] &
                 (1 << (((x - 3) >> 1) & 7))) ;
}
void sieve() {
  int i, j;
  memset(at, 0, sizeof(at)) ;
  for (i = 3; i <= maxsize; i += 2)
    if (!at[((i - 3) >> 1) >> 8] &
                  (1 << (((i - 3) >> 1) & 7)))
      for (j=i * i ; j <= maxsize ; j += i + i)
        at[((i - 3) >> 1) >> 8] |=
                 (1 << (((i - 3) >> 1) & 7))) ;
}
```

Concluzie

Asemenea optimizări ca și cele prezentate în cadrul acestui articol se pot dovedi foarte utile în unele cazuri, dar ele fac programul ilizibil.

Tocmai din acest motiv, asemenea trucuri trebuie aplicate numai în locurile critice ale codurilor sursă pentru a duce la o creștere semnificativă de viteză a aplicațiilor și trebuie documentate foarte bine, mai ales dacă se dorește sau este necesar ca alt programator să poată lucra cu codul respectiv mai târziu.

Bibliografie

1. James A. Storer, *An Introduction to Data Structures and Algorithms*, Springer Verlag, 2001
2. T. H. Cormen, C. E. Leiserson, R. R. Rivest, *Introducere în algoritmi*, Editura Computer Libris Agora, 2000
3. ***, *colecția GInfo*
4. http://www.liis.ro/~campion
5. http://www.aggregate.org/MAGIC/
6. http://www.topcoder.com – *RoundTables*

Cosmin-Silvestru Negrușeri este student în anul III la Universitatea Babeș-Bolyai din Cluj-Napoca. El poate fi contactat prin e-mail la adresa kosmin_2000@yahoo.com.