Operatii cu numere mari

1.Utilitatea

De ce este necesara implementarea operatiilor cu numere mari? Sa consideram numerele naturale mari (cu maximum 500 de cifre). Nici unul dintre tipurile limbajului C/C++ nu permite lucrul cu astfel de numere. Prin urmare, trebuie sa implementam propriul nostru tip pentru numere mari si operatiile uzuale cu acest tip de numere (adunare, scadere, inmultire, impartire, comparare).

Solutie? Vom reprezenta un numar natural mare ca un vector in care retinem in ordine cifrele sale, incepand cu unitatile (astfel pe pozitia 0 se va afla cifra unitatilor, pe pozitia 1 cifra zecilor s.a.m.d). Insa pentru aceasta solutie, pentru fiecare numar vom avea nevoie de o variabila auxiliara care sa contorizeze numarul de cifre ale numarului nostru mare. Pentru rezolvarea acestei probleme pe pozitia 0 se va afla acest numar, iar pozitiile cifrelor vor incepe de pe pozitia 1. Exemplu, numărul 3781 îl vom reprezenta:

0	1	2	3	4
4	**1**	**8**	**7**	**3**

Declararea tipului corespunzator unui numar mare poate fi:

```
#define LG_MAX 500 + 1
//avem nevoie de acel 1 pentru a satisface si necesitatea lunigimii sirului de cifre
typedef int nrmare[LG_MAX];
```

2. Citirea unui numar mare

Pentru a citi un numar mare vom citi de la tastatura numarul intr-un sir de caractere, apoi vom transforma caracterele in numere si le vom memora in ordine inversa in vectorul cu cifrele numarului mare. Pentru a usura calculele,vom completa zona de memorie alocata numarului mare (celelalte componente ale vectorului de cifre) cu 0. Functia *citire()* va avea un singur parametru : vectorul in care retinem cifrele numarului mare citit (BigNumber x) si numarul cifre ale respectivului numar.

```
void citire(BigNumber x)
{
     int i,n; char s[LG_MAX];
     f>>s;                  //citim sirul de cifre
     n = strlen(s);         //calculam lungimea numarului
     for (i=n-1; i>0;i--) x[n-i-1]=(int) (s[i]-'0'); //retin cifrele invers
     x[0] = n; //retin numarul de cifre
}
```

3. Afisarea unui numar mare

Vom parcurge vectorul de cifre de la sfarsit catre inceput,afisand succesiv cifrele. Functia de afisare va avea un singur parametru (vectorul in care retinem cifrele numarului mare citit si numarul de cifre ale numarului mare).

```
void afisare(BigNumber x){
     int i;
```

```
      for(i=x[0];i>0;i--) cout<<x[i];
}
```

4. Compararea a două numere mari

Pentru a compara doua numere "uriase", incepem prin a afla numarul de cifre semnificative (deoarece, in urma anumitor operatii pot rezulta zerouri nesemnificative care "atarna" totusi la numarul de cifre). Aceasta se face decrementand numarul de cifre al fiecarui numar atata timp cat cifra cea mai semnificativa este *0*. Dupa ce ne-am asigurat asupra acestui punct, comparam numarul de cifre al celor doua numere. Numarul cu mai multe cifre este cel mai mare. Daca ambele numere au acelasi numar de cifre, pornim de la cifra cea mai semnificativa si comparam cifrele celor doua numere pana la prima diferenta intalnita. In acest moment, numarul a carui cifra este mai mare este el insusi mai mare. Daca toate cifrele numerelor sunt egale doua cate doua, atunci in mod evident numerele sunt egale.

Dupa cum se vede, algoritmul seamana foarte bine cu ceea ce s-a invatat la matematica prin clasa a II-a (doar ca atunci nu ni s-a spus ca este vorba de un "algoritm"). Functia *compara()* va avea ca parametri cele doua numere mari care se aduna si intoarce *-1*, *0* sau *1*, daca primul numar este mai mic, egal sau mai mare decat *al doilea*.

```
int compara(BigNumber a,BigNumber b) {
  // Elimina zero-urile semnificative, daca exista.
  while (a[0] && !a[a[0]]) a[0]--;
  while (b[0] && !b[b[0]]) b[0]--;

     //comparam numarul de cifre
  if (a[0] < b[0]) {
    return -1;
  } else if (a[0] > b[0]) {
    return +1;
  }
 //daca numerele au un numar egal de cifre
//parcurgem numerele incapand cu cifra cea mai semnificativa
// pana la prima diferenta intalnita
  for (int i = a[0]; i > 0; --i) {
    if (a[i] < b[i]) {
      return -1;
    } else if (a[i] > b[i]) {
      return +1;
    }
  }
  return 0;
}
```

5. Suma a doua numere mari

Pentru a calcula suma a doua numere mari vom parcurge simultan cele doua numere,incepand de la unitati.Vom aduna cele doua cifre de pe pozitii corespondente si vom lua in calcul si eventuala cifra de transport care s-a obtinut de la adunarea precedenta. Retinem in suma cifra obtinuta prin adunare si recalculam transportul. Suma ar putea avea o cifra in plus fata de cel mai mare dintre cele doua numere (daca la sfarsit cifra de transport este nenula). Functia *suma()* va avea ca parametri cele doua numere mari care se aduna si numarul mare rezultat.

```
//s = a+b;
void suma(BigNumber a,BigNumber b,BigNumber s)
{
    int i,cifra,t = 0,max;

    //completam numarul cel mai mic cu zeroouri nesemnificative
    if(a[0] < b[0]) { max = b[0]; for(i = a[0]+1; i <= b[0]; i++) a[i] = 0; }
    else            { max = a[0]; for(i = b[0]+1; i <= a[0]; i++) b[i] = 0; }

    for(i = 1; i <= max; i++)
    {
      cifra = a[i] + b[i] + t; //calculam noua cifra
      s[i] = cifra % 10;
      t = cifra/10;            //calculam cifra de transport
    }
    if(t) s[i] = t; else i--;
    s[0] = i;
}
```

6. Diferenta a doua numere mari

Vom presupune ca descazutul este mai mare sau egal cu scazatorul.Pentru a calcula diferenta a doua numere mari vom parcurge dezcazutul incepand de la unitati.Vom scadea din cifra curenta a descazutului cifra curenta a scazatorului si vom lua in calcul si eventuala cifra de transport,obtinuta la scaderea precedenta.Daca rezultatul este negativ,imprumutam 10 de pe pozitia urmatoare(este posibil doar daca descazutul este mai mare ca scazatorul),si in acest caz cifra de transport devine -1.Diferenta ar putea avea mai putine cifre decat descazutul ,astfel incat la sfarsit calculam numarul de cifre din diferenta,ignorand zerourile nesemnificative.

```
//d = a-b( a > b )
void diferenta(BigNumber a,BigNumber b,BigNumber d)
{
   int i,t = 0;
   if(a[0] < b[0])
   {
      diferenta(a,b,d);
   }
   else
   {
    for(i = 1; i <= a[0]; i++)
    {
```

```
        d[i] = a[i]-b[i]+t;            //diferenta
        if(d[i] < 0) d[i]+= 10,t = -1;  //calculam cifra de transport
        else t = 0;
      }
      i--;
      while(i && !d[i]) i--;
      d[0] = i;
    }
}
```

7. Inmultirea unui numar mare cu un scalar

Si aceasta operatie este o simpla implementare a modului manual de efectuare a calculului. La inmultirea "de mana" a unui numar mare cu unul de o singura cifra, noi parcurgem deinmultitul de la sfarsit la inceput, si pentru fiecare cifra efectuam urmatoarele operatii:

- Inmultim cifra respectiva cu inmultitorul;
- Adaugam "transportul" de la inmultirea precedenta;
- Separam ultima cifra a rezultatului si o trecem la produs;
- Celelalte cifre are rezultatului constituie transportul pentru urmatoarea inmultire;
- La sfarsitul inmultirii, daca exista transport, acesta are o singura cifra, care se scrie inaintea rezultatului.

Exact acelasi procedeu se poate aplica si daca inmultitorul are mai mult de o cifra. Singura deosebire este ca transportul poate avea mai multe cifre (poate fi mai mare ca *9*). Din aceasta cauza, la sfarsitul inmultirii, poate ramane un transport de mai multe cifre, care se vor scrie inaintea rezultatului. Iata de exemplu cum se calculeaza produsul *312* x *87*:

87 x 2 = 174 = 17 x 10 + 4
87 x 1 + 17 = 104 = 10 x 10 + 4
87 x 3 + 10 = 271 = 27 x 10 + 1
87 x 0 + 27 = 27 = 2 x 10 + 7
87 x 0 + 2 = 2 = 0 x 10 + 2

```
  312 x
   87
 ------
    4
   4
  1
 7
2
```

```
void Mult(BigNumber a, unsigned long X)
/* a <- a*X */
{ int i;
  unsigned long T=0;

  for (i=1;i<=a[0];i++)
    { a[i]=a[i]*X+T;
      T=a[i]/10;
      a[i]=a[i]%10;
    }
  while (T) /* Cat timp exista transport */
    { a[++a[0]]=T%10;
      T/=10;
    }
}
```

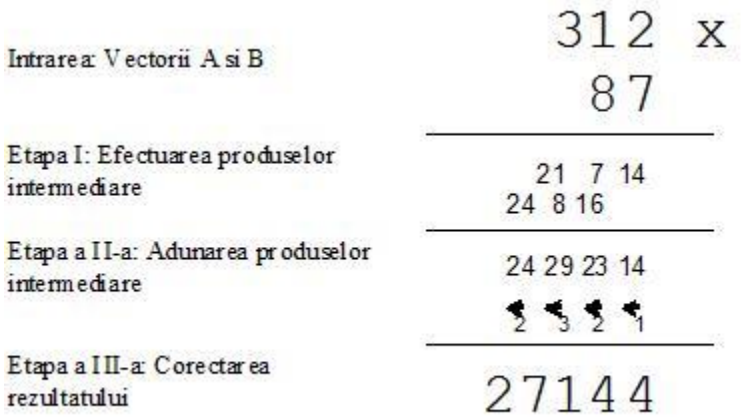
Inmultirea a doua numere mari

Daca ambele numere au dimensiuni mari si se reprezinta pe tipul de date Huge, produsul lor se calculeaza inmultind fiecare cifra a deinmultitului cu fiecare cifra a inmultitorului si trecand rezultatul la ordinul de marime (exponentul lui *10*) cuvenit. De fapt, acelasi lucru il facem si noi pe hartie. Considerand acelasi exemplu, in care ambele numere sunt "uriase", produsul lor se calculeaza de mana astfel

```
  312 x
   87
 2184
2496
27144
```

S-a luat deci fiecare cifra a inmultitorului si s-a efectuat produsul partial corespunzator, corectand la fiecare pas rezultatul prin calculul transportului. Rezultatul pentru fiecare produs partial s-a scris din ce in ce mai in stanga, pentru a se alinia corect ordinele de marime. Acest procedeu este oarecum incomod de implementat. Se pot face insa unele observatii care usureaza mult scrierea codului:

- Prin inmultirea cifrei cu ordinul de marime *10i* din primul numar cu cifra cu ordinul de marime *10j* din al doilea numar se obtine o cifra corespunzatoare ordinului de marime *10i+j* in rezultat (sau se obtine un numar cu mai mult de o singura cifra, caz in care transportul merge la cifra corespunzatoare ordinului de marime *10i + j + 1*).
- Daca numerele au *M* si respectiv *N* cifre, atunci produsul lor va avea fie *M + N* fie *M + N - 1* cifre. Intr-adevar, daca numarul *A* are *M* cifre, atunci $10^{M-1} \le A < 10^{M}$ si $10^{N-1} \le B < 10^{N}$, de unde rezulta $10^{M+N-2} \le A \times B < 10^{M+N}$.
- La calculul produselor partiale se poate omite calculul transportului, acesta urmand a se face la sfarsit. Cu alte cuvinte, intr-o prima faza putem pur si simplu sa inmultim cifra cu cifra si sa adunam toate produsele de aceeasi putere, obtinand un numar cu "cifre" mai mari ca *9*, pe care il aducem la forma normala printr-o singura parcurgere. Sa reluam acelasi exemplu:



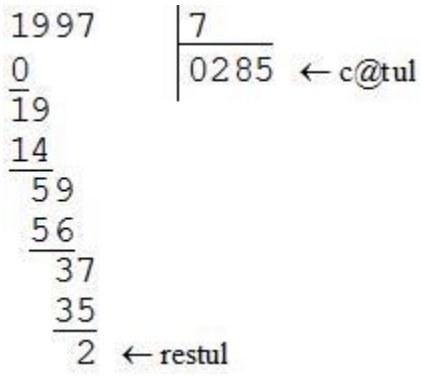
Aceasta operatie efectueaza *M*x*N* produse de cifre si *M*+*N* (sau *M*+*N-1*, dupa caz) "transporturi" pentru aflarea rezultatului, deci are complexitatea O(*M*x*N*). Iata si implementarea:

```
void MultHuge(BigNumber A, BigNumber B, BigNumber C)
/* C <- A x B */
{ int i,j,T=0;
```

```
  C[0]=A[0]+B[0]-1;
  for (i=1;i<=A[0]+B[0];) C[i++]=0;
  for (i=1;i<=A[0];i++)
    for (j=1;j<=B[0];j++)
      C[i+j-1]+=A[i]*B[j];
  for (i=1;i<=C[0];i++)
    { T=(C[i]+=T)/10;
      C[i]%=10;
    }
  if (T) C[++C[0]]=T;
}
```

Impartirea unui vector la un scalar

Ne propunem sa scriem o functie care sa imparta numarul *A* de tip Huge la scalarul *B*, sa retina valoarea catului tot in numarul *A* si sa intoarca restul (care este o variabila scalara). Sa pornim de la un exemplu particular si sa generalizam apoi procedeul: sa calculam catul si restul impartirii lui *1997* la *7*. Cu alte cuvinte, sa gasim acele numere *C* de tip Huge si $R \in \{0, 1, 2, 3, 4, 5, 6\}$ cu proprietatea ca *1997 = 7* x *C + R*.



La fiecare pas se coboara cate o cifra de la deimpartit alaturi de numarul deja existent (care initial este *0*), apoi rezultatul se imparte la impartitor (*7* in cazul nostru). Catul este intotdeauna o cifra si se va depune la sfarsitul catului impartirii, iar restul va fi folosit pentru urmatoarea impartire. Restul care ramane dupa ultima impartire este tocmai *R* pe care il cautam. Procedeul functioneaza si atunci cand deimpartitul are mai multe cifre. La sfarsit trebuie sa decrementam corespunzator numarul de cifre al catului, prin neglijarea zerourilor create la inceputul numarului. Numarul maxim de cifre al catului este egal cu cel al deimpartitului.

```
unsigned long Divide(Huge A, unsigned long X)
/* A <- A/X si intoarce A%X */
{ int i;
  unsigned long R=0;

  for (i=A[0];i;i--)
    { A[i]=(R=10*R+A[i])/X;
      R%=X;
    }
  while (!A[A[0]] && A[0]>1) A[0]--;
  return R;
}
```

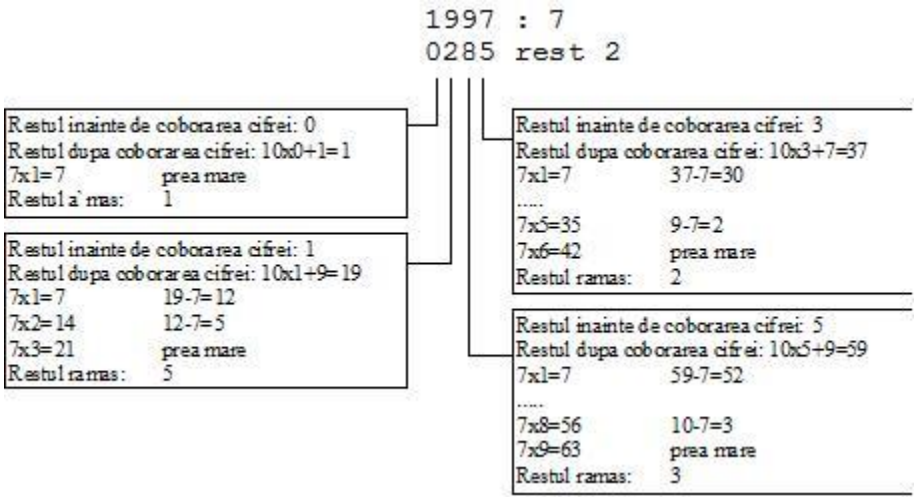
Daca dorim numai sa aflam restul impartirii, nu mai avem nevoie decat sa recalculam restul la fiecare pas, fara a mai modifica vectorul A:

```
unsigned long Mod(Huge A, unsigned long X)
/* Intoarce A%X */
{ int i;
  unsigned long R=0;
```

```
  for (i=A[0];i;i--)
    R=(10*R+A[i])%X;
  return R;
```

Impartirea a doi vectori

Daca se dau doi vectori *A* si *B* si se cere sa se afle catul *C* si restul *R*, etapele de parcurs sunt aceleasi ca la punctul precedent, numai ca operatorii "/" si "%" trebuie implementati de utilizator, ei nefiind definiti pentru vectori. Cu alte cuvinte, dupa ce "coboram" la fiecare pas urmatoarea cifra de la deimpartit, trebuie sa aflam cea mai mare cifra *X* astfel incat impartitorul sa se cuprinda de *X* ori in restul de la momentul respectiv. Acest lucru se face cel mai comod prin adunari repetate: pornim cu cifra *X=0* si o incrementam, micsorand concomitent restul, pana cand restul care ramane este prea mic. Sa efectuam aceeasi impartire, *1997*:*7*, considerand ca ambele numere sunt reprezentate pe tipul Huge.



Cazul cel mai defavorabil (cand *X*=*9*) presupune *9* scaderi si *10* comparatii, cazul cel mai favorabil (cand *X=0*) presupune numai o comparatie, deci cazul mediu presupune *4* scaderi si *5* comparatii. Cautarea lui *X* se poate face si binar, prin injumatatirea intervalului, ceea ce reduce timpul mediu de cautare la aproximativ *3* comparatii si *3* inmultiri, dar codul se complica nejustificat de mult (de cele mai multe ori).

```
void DivideHuge(Huge A, Huge B, Huge C, Huge R)
/* A/B = C rest R */
{ int i;

  R[0]=0;C[0]=A[0];
  for (i=A[0];i;i--)
    { Shl(R,1);R[1]=A[i];
      C[i]=0;
      while (Sgn(B,R)!=1)
        { C[i]++;
          Subtract(R,B);
        }
    }
  while (!C[C[0]] && C[0]>1) C[0]--;
}
```

campion.edu.ro/arhiva - set, muguri, sqr, doitrei, banda10