Rapport de MAO : Arithmétique Multiprécision

Martin FLEURIAL, Célia ROCHE

Résumé

Dans ce rapport, nous

Table des matières

1 Représentation des données en mémoire

Exercice 1

Soit m un entier de n chiffres en base 10. En stockant les chiffres de m en base 10 dans un tableau de byte, le nombre de bits utilisés est $q = 8n$, car chaque "case" du tableau utilise 8 bits. Calculons le nombre q' de bits utilisés si nous pouvions stocker l'écriture binaire de m dans la mémoire :

$$\begin{aligned} m \text{ utilises } q' \text{ bits} &\Longleftrightarrow 2^{q'-1} \leqslant m < 2^{q'} \\ &\Longleftrightarrow q' - 1 \leqslant \frac{\ln m}{\ln 2} < q' \\ &\Longleftrightarrow \frac{\ln m}{\ln 2} < q' \leqslant \frac{\ln m}{\ln 2} + 1 \end{aligned}$$

De plus

$$\begin{aligned} m \text{ a } n \text{ chiffres en base } 10 &\Longleftrightarrow 10^{n-1} \leqslant m < 10^n \\ &\Longleftrightarrow (n-1)\frac{\ln 10}{\ln 2} \leqslant \frac{\ln m}{\ln 2} < n\frac{\ln 10}{\ln 2} \end{aligned}$$

Donc

$$(n-1)\frac{\ln 10}{\ln 2} < q' < n\frac{\ln 10}{\ln 2} + 1$$

Posons $e = \left|\frac{q'-q}{q}\right|$. Alors

$$\begin{aligned} &\left|1 - \frac{n\frac{\ln 10}{\ln 2} - 1}{8n}\right| < e < \left|1 - \frac{(n-1)\frac{\ln 10}{\ln 2}}{8n}\right| \\ \Longleftrightarrow &\left|1 - \frac{\ln 10}{8\ln 2} + \frac{1}{8n}\right| < e < \left|1 - \frac{\ln 10}{8\ln 2} + \frac{\ln 10}{8n\ln 2}\right| \end{aligned}$$

Donc à la limite on a

$$e = 1 - \frac{\ln 10}{8\ln 2} \approx 0.5848$$

On voit alors que en stockant les chiffres de m en base 10 dans un tableau de byte, 58% de la mémoire est utilisée inutilement par rapport à l'écriture binaire classique.

1.1 Exercice 2

1. Procédons à la multiplication de 325 par 28 en base 10 :
   Supposons ces entiers écrits sous forme de 2 tableaux contenant respec-

tivement 3 et 2 chiffres, on a alors

$$
\begin{aligned}
325 \times 28 &= [5;2;3] \times [8;2] \\
\text{(a)} &= 10^0 \times 8 \times [5;2;3] + 10^1 \times 2 \times [5;2;3] \\
\text{(b)} &= [40;16;24] + [100;40;60] \\
\text{(c)} &= [0;0;6;2] + [0;0;5;6] \\
\text{(d)} &= [0;0;11;8] \\
\text{(e)} &= [0;0;1;9] \\
&= 9100
\end{aligned}
$$

(a) On distribue 28 dans 325 en multipliant aussi par la puissance de 10 associée.

(b) On effectue les multiplications terme à terme

(c) Dans chaque tableau, on procède de gauche à droite en ajoutant le quotient du nombre par 10 à la case suivante (la retenue), et en remplaçant le nombre par le reste obtenu. Lorsqu'on atteint la dernière case, on en crée une nouvelle si besoin

(d) On ajoute les tableaux terme à terme

(e) On répète (c) sur le tableau obtenu

2. On constate à l'étape (b) que les nombres contenus dans les cases des tableaux ne sont pas dans $\{0; \dots; 10-1\}$. Ce dépassement de plage est la raison pour laquelle on utilise la base 2^{32} et pas 2^{64}. En effet, si nous travaillions en base 2^{64}, en cas de dépassement nous obtiendrions directement le reste (car la machine calcule dans $\mathbb{Z}/2^{64}\mathbb{Z}$) mais nous ne pourrions par obtenir la retenue, ce qui fausserait le calcul. En travaillant en base 2^{32} mais en stockant les entiers sur une plage de longueur 2^{64}, nous pouvons obtenir la retenue et faire des calculs exacts.

2 Normalisation et comparaison

2.1 Exercice 3

1.
```
procedure normalize (var t : int_array);
begin
while (t <> nil) and (t[length (t) - 1] = 0) do
    SetLength (t, length (t) - 1);
end;
```

2. Il est nécessaire de passer t en entrée/sortie à la fonction pour que la procédure ne duplique pas l'espace mémoire occupé par t. En effet, en passant un paramètre en entrée, le langage copie la mémoire occupée par t à un autre endroit et fait les opérations sur cette mémoire ci. Cependant, ici nous manipulons des tableaux de tailles arbitraires, et donc qui occupent beaucoup d'espace mémoire. En passant t en entrée/sortie, le langage procède à un passage par adresse (ou par référence) de t, c'est à dire qu'elle manipule un pointeur, ce qui est beaucoup moins coûteux que de dupliquer la mémoire.

3.
```
function smaller (a, b: int_array) : Boolean;
var i: LongWord;
begin
    normalize (a); normalize (b);
    smaller := False;
    if (length (a) < length (b)) then
        smaller := True
    else if (length (a) = length (b)) then
    begin
        i := length (a) - 1;
        // On compare jusqu'à trouver
        // des chiffres différents
        while (a[i]=b[i]) and (i>0) do
            i := i - 1;
        // On compare les chiffres
        // différents et on conclut
        if (a[i]<=b[i]) then
            smaller := True
    end;
end;
```

3 Quotient d'un int_array par un longword et conversion décimale

3.1 Exercice 4

1. Montrons par récurrence la propriété P_N suivante

$$d\sum_{n=0}^{N} q_{N-n}b^n + r_{N+1} = \sum_{n=0}^{N} a_{N-n}b^n$$

On sait que

$$\begin{cases} r_0 & = 0 \\ r_n b + a_n & = dq_n + r_{n+1} \, (*) \end{cases}$$

Donc pour $n = 0$ on a $dq_0 + r_1 = a_0$, donc P_0 est vraie.
Supposons maintenant la propriété vraie jusqu'au rang $N \geqslant 0$ et montrons la au rang $N + 1$.
On a

$$\begin{aligned} \sum_{n=0}^{N+1} q_{N+1-n}b^n &= q_{N+1}b^0 + \sum_{n=1}^{N+1} q_{N+1-n}b^n \\ &= q_{N+1}b^0 + b\sum_{n=0}^{N} q_{N-n}b^n \end{aligned}$$

Alors

$$
\begin{aligned}
d\sum_{n=0}^{N+1} q_{N+1-n}b^n + r_{N+2} &= dq_{N+1}b^0 + db\sum_{n=0}^{N} q_{N-n}b^n + r_{N+2} \\
\text{par } (*) &= r_{N+1}b + a_{N+1} + db\sum_{n=0}^{N} q_{N-n}b^n \\
&= b(d\sum_{n=0}^{N} q_{N-n}b^n + r_{N+1}) + a_{N+1} \\
\text{par hypothèse de récurrence} &= b\sum_{n=0}^{N} a_{N-n}b^n + a_{N+1} \\
&= \sum_{n=0}^{N} a_{N-n}b^{n+1} + a_{N+1} \\
&= \sum_{k=1}^{N+1} a_{N+1-k}b^k + a_{N+1}b^0 \\
&= \sum_{n=0}^{N+1} a_{N+1-n}b^n
\end{aligned}
$$

Donc P_{N+1} est vraie, donc la propriété est héréditaire. Alors, pour tout $N \in \mathbb{N}$ on a

$$
d\sum_{n=0}^{N} q_{N-n}b^n + r_{N+1} = \sum_{n=0}^{N} a_{N-n}b^n
$$

2.
```
procedure div_by_digit (a : int_array; d : longword;
                        var q : int_array; var r : longword);
var i, N : LongWord;
begin
    Normalize (a);
    setLength (q, length (a));
    r := 0;
    N := length (a) - 1;
    for i := N downto 0 do
        begin
            q[i] := (r * BASE + a[i]) div d;
            r := (r * BASE + a[i]) mod d;
        end;
    Normalize (q);
end;
```

4 Somme et différence de deux int_array

4.1 Exercice 5

Pour réaliser la somme de deux int_array, il faut réaliser l'addition avec retenue. C'est à dire additionner terme à terme (chiffre à chiffre) les int_array, et lorsque la somme est plus grande que la base (ici 2^{32}), on ajoute la retenue au chiffre suivant.

Exemple :

$$\begin{aligned} 325 + 28 &= [5;2;3] + [8;2] \\ (a) &= [13;4;3] \\ (b) &= [3;5;3] \end{aligned}$$

On voit à l'étape que la valeur de la retenue est le quotient de la division euclidienne de la somme par la base, et que on remplace la somme par le reste. On peut généraliser cela :

Soient $m_1 = \sum_{n=0}^{N_1} c_n b^n$ et $m_2 = \sum_{n=0}^{N_2} c'_n b^n$ avec $m_1 \geqslant m_2$. Alors on pose $N_3 = \max(N_1; N_2) = N_1$ et on définit $\tilde{c}_n$ ainsi :

$$\begin{cases} c_0 + c'_0 = r_0 b + \tilde{c}_0 & \text{si } n = 0 \\ c_n + c'_n + r_{n-1} = r_n b + \tilde{c}_n & \text{si } 1 \leqslant n \leqslant N_2 \\ c_n + r_{n-1} = r_n b + \tilde{c}_n & \text{si } N_2 \leqslant n \leqslant N_1 \end{cases}$$

où les $r_n \geqslant 0$ et $\tilde{c}_n \in \{0, \ldots, b-1\}$. Alors $m_1 + m_2 = \sum_{n=0}^{N_3+1} \tilde{c}_n b^n$.

```
procedure sum (a, b: int_array; var s: int_array);
var i: LongWord;
    somme: QWord;
    retenue: LongWord;
begin
    Normalize (a); Normalize (b);
    if smaller (a, b) then
        sum (b, a, s)
    else
     begin
        setLength (s, length (a) + 1);
        retenue := 0;
        for i:=0 to high (s) do
        begin
            // On utilise un QWord dans
            // le cas où la somme est plus
            // grande que la base pour
            // pouvoir calculer la retenue
            if i <= High (b) then
                somme := a[i] + retenue + b[i]
            else
                somme := a[i] + retenue;
            retenue := 0;
            if somme >= BASE then
                 retenue := somme div BASE;
            s[i] := somme mod BASE;
        end;
        Normalize (s);
    end;
end;
```

De la même manière pour la différence, on peut généraliser l'algorithme . En reprenant les notations précédentes :

$$\begin{cases} c_0 - c'_0 = r_0 b + \tilde{c}_0 & \text{si } n = 0 \\ c_n - c'_n + r_{n-1} = r_n b + \tilde{c}_n & \text{si } 1 \leqslant n \leqslant N_2 \\ c_n + r_{n-1} = r_n b + \tilde{c}_n & \text{si } N_2 \leqslant n \leqslant N_1 \end{cases}$$

où les $r_n \leqslant 0$ et $\tilde{c}_n \in \{0, \ldots, b-1\}$ et alors $m_1 - m_2 = \sum_{n=0}^{N_3} \tilde{c}_n b^n$

```
procedure diff (a, b: int_array; var s: int_array);
var i: LongWord;
    difference, retenue: Int64;
begin
    Normalize (a); Normalize (b);
    setLength (s, length (a));
    retenue := 0;
    for i:=0 to High (s) do
    begin
        // On utilise un Int64 dans le cas
        // où la différence est négative pour
        // pouvoir calculer la retenue
        if i <= High (b) then
            difference := a[i] + retenue - b[i]
        else
            difference := a[i] + retenue;
        retenue := 0;
        // En Pascal, le reste peut être négatif.
        // Si on ne veut pas qu'il le soit,
        // il faut lui ajouter
        // la base et retirer 1 au quotient.
        if difference < 0 then
             retenue := (difference div BASE) - 1;
        s[i] := difference mod BASE;
    end;
    Normalize (s);
end;
```

5 Produit naïf de deux int_array

5.1 Exercice 6

Pour réaliser le produit de deux int_array, on applique l'algorithme décrit dans l'exercice 2, en utilisant la base 2^{32}.

```
procedure prod (a, b : int_array; var p : int_array);
var n, m: Longword;
    produit: QWord;
    retenue, retenue_ld, bs_atteinte: LongWord;
begin
    Normalize (a); Normalize (b);
    setLength (p, length (a) + length (b) + 1);
```

```
    // La somme n+m n'est pas croissante
    // dans la boucle ci-dessous, donc il
    // faut conserver la retenue quand on retourne sur une
    // valeur inférieure de n+m, et l'ajouter
    // quand atteint n+m+1. De plus, quand on
    // atteint high (a) + high (b) il faut déplacer
    // la retenue à high (a) + high(b) + 1
    retenue_ld := 0;
    bs_atteinte := 0;
    for n:=0 to high (a) do
    begin
        retenue := 0;
        for m:=0 to high (b) do
        begin
            produit := a[n]*b[m] + retenue;
            if (n + m = bs_atteinte + 1) then
                produit := produit + retenue_ld;
            retenue := produit div BASE;
            p[n+m] := (p[n+m] + produit) mod BASE;
            if m = high (b) then
            begin
                retenue_ld := retenue;
                bs_atteinte := n + m;
             end;
            if (n = high (a)) and (m = high (b)) then
                p[n+m+1] := retenue;
        end;
    end;
    Normalize (p);
end;
```

6 Tests (et un peu d'arithmétique)

6.1 Exercice 7

1. Soit un entier m en base $b = 2^{32}$, alors m s'écrit : $m = \sum_{k=0}^{r} c_k b^k$ avec $c_k \in \{0; \ldots; b-1\}$ et $c_r \neq 0$.

   Dans $\mathbb{Z}/17\mathbb{Z}$, $\overline{b} = \overline{2^{32}} = \overline{2}^{32} = \left(\left(\left(\underbrace{\left(\overline{2}^2\right)^2}_{=\overline{16}=\overline{-1}}\right)^2\right)^2\right)^2 = \overline{1}$.

Ainsi, dans $\mathbb{Z}/17\mathbb{Z}$,

$$\begin{aligned}
\overline{m} &= \overline{\sum_{k=0}^{r} c_k b^k} \\
&= \sum_{k=0}^{r} \overline{c_k b^k} \\
&= \sum_{k=0}^{r} \overline{c_k} \times \underbrace{\overline{b^k}}_{=\overline{1}} \\
&= \overline{\sum_{k=0}^{r} c_k}
\end{aligned}$$

Donc un entier m en base $b = 2^{32}$ est divisible par 17 si, et seulement si, la somme de ses chiffres est divisible par 17.