Critère de divisibilité (binaire)

Denis Bouhineau (denis.bouhineau@imag.fr)

August 21, 2024

1 Critère de divisibilité, quotient et reste

Reconnaître qu'un nombre est multiple de 2, 4, 5, 8, 10,… est (quasi) immédiat.
Pour reconnaître qu'un nombre est multiple de 3, 6, 9, 12,… c'est un peu plus compliqué, mais, depuis longtemps, on sait faire sans grands calculs (réduction par addition des chiffres).
Pour les multiples de 7, 11, 13,… le calcul de la division euclidienne est toujours possible mais plus coûteux (sous réserve que l'on sache encore faire une division euclidienne). Pourtant il existe des méthodes (chacune spécifique du nombre en question) permettant de se ramener à des calculs similaires à ceux de la preuve par 9 (réduction par addition ou soustraction). Par exemple pour 7, il *suffit* de soustraire aux dizaines 2 fois les unités et de recommencer jusqu'à obtenir un petit nombre. Exemple numérique dans l'exemple, 2023 => 202 - 2*3* = *196* => *19* - *2*6 = 7 montre que 2023 est multiple de 7. Cependant ces méthodes sont moins connues car le choix du facteur 2 et de la soustraction (ou de la multiplication) dépendent de 7. Pour connaître les facteurs pour 11, 13,… voir https://fr.wikipedia.org/wiki/Liste_de_crit%C3%A8res_de_divisibilit%C3%A9. Il y a autant de méthodes que de nombres, chacun a ses coefficient et son choix addition/soustraction spécifiques, ce n'est pas pratique, il faudrait connaître tous ces coefficients par cœur (ou savoir les retrouver) et savoir s'il faut additionner ou soustraire. On peut se rappeler de 2/Soustaction pour 7, 1/Soustraction pour 11, 4/Addition pour 13,…

C'est rarement explicité (ni expliqué) mais ces méthodes sont liées à la base 10 et conviennent bien avec le calcul mental, mais moins avec les calculs sur machine (qui s'effectuent en binaire).

Pourtant, si l'on pense binaire, on peut trouver une méthode similaire (par réduction via des additions) sans paramètres et beaucoup plus efficace en machine.
Supposons donc que l'on veuille trouver si N (un nombre quelconque) est multiple de M :

- Si N et M sont tous les deux **pairs**, on peut les diviser tous les deux par 2 (ce qui est assez facile en décimal et immédiat sur machine). Le nouveau N sera multiple de M ssi le précédent N était multiple de M.
- Si N est **impair** et M **pair**, alors N ne sera pas multiple de M.
- Si N est **pair** et M **impair**, alors on peut diviser N par 2. Le nouveau N sera multiple de M ssi le précédent N était multiple de M.
- Si N et M sont **impairs**, alors on peut soustraire M de N (ou l'additionner), cela ne change pas le critère de divisibilité, mais le nouveau N est pair, on peut alors se ramener au cas précédent, et diviser N par 2.

Dans tous les cas, à chaque étape, on fait une division par 2 (ou on conclue), la méthode s'arrête assez vite.

Exemple pour N=2023 et M=7 : 2023 => (2023 - 7) / 2 = 1008 => 1008 / 2 = 504 => 504 / 2

= 252 => 252 /2 = 126 => 126 / 2 = 63 => (63 - 7) / 2 = 28 => 28 / 2 = 14 => 14 / 2 = 7.

Remarques :

- la méthode semble plus longue, c'est exacte, car elle repose sur des divisions/décomposition par 2 au lieu de divisions/décomposition par 10, elle est donc environ 3 fois plus longue, mais reste en temps linéaire selon le nombre de chiffres de N
- en utilisant la méthode avec soustraction, l'alternance des pairs/impairs permet de trouver le quotient poids faible d'abord quand le nombre est multiple (quand le nombre n'est pas multiple c'est encore possible mais plus compliqué et n'est plus poids faible d'abords).

2 Code - Vrac - Annexes

2.1 Critère de divisibilité/multiplicité

[1]:
```
def pair(V):
    return (V&1)==0

def impair(V):
    return (V&1)==1
```

[2]:
```
def multipleDe(V,M):
    match V,M :
        case 0,M : return True
        case V,M if V==M : return True
        case V,M if 0<V<M : return False
        case V,M if pair(V) and pair(M) : return multipleDe(V >> 1,M >> 1)
        case V,M if pair(V) and impair(M) : return multipleDe(V >> 1,M)
        case V,M if impair(V) and pair(M) : return False
        case V,M if impair(V) and impair(M) : return multipleDe((V-M) >> 1,M)
        case _ : return "not defined"
```

[3]:
```
multipleDe(2023,7)
```

[3]:
```
True
```

[4]:
```
for i in range(50):
    print(i,multipleDe(i,7))
```

```
0 True
1 False
2 False
3 False
4 False
5 False
6 False
7 True
8 False
```

```
9 False
10 False
11 False
12 False
13 False
14 True
15 False
16 False
17 False
18 False
19 False
20 False
21 True
22 False
23 False
24 False
25 False
26 False
27 False
28 True
29 False
30 False
31 False
32 False
33 False
34 False
35 True
36 False
37 False
38 False
39 False
40 False
41 False
42 True
43 False
44 False
45 False
46 False
47 False
48 False
49 True
```

[5]:
```
for V in range(100000):
    if multipleDe(V,7) != (V%7==0):
        print(V,multipleDe(V,7),V%7)
print("fin")
```

```
fin
```

[6]:
```
for V in range(10000):
    for M in range(1,100):
        if multipleDe(V,M) != (V%M==0):
            print(V,M,multipleDe(V,M),V%M)
print("fin")
```

```
fin
```

2.2 Quotient (si multiple)

[7]:
```
def QuotientSiMultiple(V,M):
    match V,M :
        case 0,M : return 0
        case V,M if V==M : return 1
        case V,M if 0<V<M : return False
        case V,M if pair(V) and pair(M) : return QuotientSiMultiple(V >> 1,M >>␣
 ↪1)
        case V,M if pair(V) and impair(M) : return QuotientSiMultiple(V >>␣
 ↪1,M)<<1
        case V,M if impair(V) and pair(M) : return False
        case V,M if impair(V) and impair(M) : return (QuotientSiMultiple((V-M)␣
 ↪>> 1,M)<<1)|1
        case _ : return "not defined"
```

[8]:
```
multipleDe(2030,7),QuotientSiMultiple(2030,7)
```

[8]:
```
(True, 290)
```

[9]:
```
multipleDe(2024,7),QuotientSiMultiple(2024,7)
```

[9]:
```
(False, 216)
```

[10]:
```
for V in range(10000):
    for M in range(1,100):
        if (multipleDe(V,M)!=(V%M==0)) or (multipleDe(V,M) and␣
 ↪(QuotientSiMultiple(V,M)!=(V//M))):
            print(V,M,multipleDe(V,M),QuotientSiMultiple(V,M),V%M,V/M)
print("fin")
```

```
fin
```

Remarques :

- le calcul peut se faire poids faible d'abord (sous réserve que l'on sache (à l'avance (?)) que le nombre est multiple)
- cela donne un autre calcul, récursif terminal, mais avec des accumulateurs (à initialiser)

[11]:
```
def QuotientSiMultipleRecTerminalAvecAcc(V,M,Q,P):
    match V,M :
        case 0,M : return Q
        case V,M if V==M : return Q|P
        case V,M if 0<V<M : return False
        case V,M if pair(V) and pair(M) : return␣
 ↪QuotientSiMultipleRecTerminalAvecAcc(V >> 1,M >> 1, Q, P)
        case V,M if pair(V) and impair(M) : return␣
 ↪QuotientSiMultipleRecTerminalAvecAcc(V >> 1, M, Q, P<<1)
        case V,M if impair(V) and pair(M) : return False
        case V,M if impair(V) and impair(M) : return␣
 ↪QuotientSiMultipleRecTerminalAvecAcc((V-M) >> 1, M, Q|P, P<<1)
        case _ : return "not defined"
```

[12]:
```
multipleDe(2030,7),QuotientSiMultipleRecTerminalAvecAcc(2030,7,0,1)
```

[12]:
```
(True, 290)
```

[13]:
```
for V in range(10000):
    for M in range(1,100):
        if (multipleDe(V,M)!=(V%M==0)) or (multipleDe(V,M) and␣
 ↪(QuotientSiMultipleRecTerminalAvecAcc(V,M,0,1)!=(V//M))):
            ␣
 ↪print(V,M,multipleDe(V,M),QuotientSiMultipleRecTerminalAvecAcc(V,M,0,1),V%M,V/
 ↪M)
print("fin")
```

```
fin
```

Version sans paramètres supplémentaires apparents (les paramètres supplémentaires (les accumulateurs) continuent d'exister dans les calculs, leur ajout et leur initialisation est prise en charge lors d'une étape préliminaire) :

[14]:
```
def QuotientSiMultipleRecTerminal(V,M):
    return QuotientSiMultipleRecTerminalAvecAcc(V,M,0,1)
```

[15]:
```
multipleDe(2030,7),QuotientSiMultipleRecTerminal(2030,7)
```

[15]:
```
(True, 290)
```

2.3 Quotient et Reste (dans tous les cas)

Pour calculer le quotient dans les cas où le reste n'est pas nul, on peut se ramener au cas où le reste est nul sous réserve de connaître ce reste et de le retrancher au nombre (V) initial.
Pour connaître ce reste (dans le cas où le nombre V n'est pas multiple de M) :
* dans le cas où le reste calculé (faux) est entre 0 et M (strict), il faut observer d'abord que ce reste est faux car à chaque étape où l'on a divisé M (pair) par 2, on a modifié le reste observé (par ex. s'il valait 1, en divisant par 2, il vaudra (1+M)/2) , on peut remonter au reste initial, en défaisant

chaque étape de division par 2 (au niveau du reste), par une opération de multiplication par 2 du reste (observé) modulo M * dans le cas où le V est impair et M pair, il faut chercher à se ramener au cas V pair (en retranchant à V la partie de M non pair (*) et faire l'opération inverse pour les restes observés dans le sens inverse)

Remarques :

- c'est moins immédiat que ce qui précède mais possible.

- on perd la propriété que le calcul se fasse poids faible d'abord (le calcul demande l'ensemble des chiffres pour être effectué)