Algorithme de Karatsuba

I. Introduction

En informatique, l'algorithme de Karatsuba est un algorithme pour multiplier rapidement deux nombres de n chiffres avec une complexité temporelle en $O\left(n^{\log_2(3)}\right) \approx O\left(n^{1,585}\right)$ au lieu de $O\left(n^2\right)$ pour la méthode naïve. Il a été développé par Anatolii Alexevich Karatsuba en 1960 et publié en 1962.

II. Fonctionnement

Pour multiplier deux nombres de n chiffres, la méthode naïve multiplie chaque chiffre du multiplicateur par chaque chiffre du multiplicande. Cela exige donc n^2 produits de deux chiffres. Le temps de calcul est en $O\left(n^2\right)$.

En 1960, Karatsuba remarque que pour tout k, le calcul naïf d'un produit :

$$(a \times 10^k + b)(c \times 10^k + d) = ac \times 10^{2k} + (ad + bc) \times 10^k + bd$$

qui semble nécessiter les quatre produits ac, ad, bc et bd, peut en fait être effectué seulement avec les trois produits ac, bd et $(a-b)(c-d)$ en regroupant les calculs sous la forme suivante :

$$(a \times 10^k + b)(c \times 10^k + d) = ac \times 10^{2k} + (ac + bd - (a-b)(c-d)) \times 10^k + bd$$

Pour de grands nombres et en prenant $k = \frac{n}{2}$, la méthode peut être appliquée de manière récursive pour les calculs de ac, bd et $(a-b)(c-d)$ en scindant à nouveau a, b, c et d en deux et ainsi de suite. C'est un algorithme de type diviser pour régner.

La multiplication par la base de numération (10 dans l'exemple précédent, mais 2 pour les machines) correspond à un décalage de chiffre, et les additions sont peu coûteuses en temps ; ainsi, le fait d'être capable de calculer les grandeurs nécessaires en 3 produits au lieu de 4 mène à une amélioration de complexité.

III. Exemple

Exécutons l'algorithme pour calculer le produit 1237 × 2587.

- Pour calculer, 1237 × 2587, on décompose 1237 et 2587 en a_0 = 12 × 25 et a_2 = 37 × 87 :
  - On a alors 1237 × 2587 = a_0 × 10^4 + (a_0 + a_2 – a_1) × 10^2 + a_2
    avec a_1 = (12 – 37) × (25 – 87) = 25 × 62
    - Pour calculer 12 × 25, on décompose de même avec a0' = 1 × 2, a1' = (1 – 2) × (2 – 5) = -1 × -3 et a2' = 2 × 5
      - On a alors 12 × 25 = a0' 102 + (a0' + a2' - a1') 10 + a2'
        - Les calculs 1 × 2 = 2, 2 × 5 = 10 et -1 × -3 = 3 se réalisent en temps constant.
        - On obtient 12 × 25 = 2 × 100 + (2 + 10 – 3) × 10 + 10 = 300.
      - De la même façon, on obtient a1 = 25 × 62 = 1550.
      - De la même façon, on obtient a2 = 37 × 87 = 3219.
      - d'où 1237 × 2587 = 300 × 1002 + (300 + 3219 – 1550) × 100 + 3219 = 3000000 + 196900 + 3219 = 3200119.

Le calcul complet ne demande que 3×3 = 9 multiplications de chiffres au lieu de 4×4 = 16 multiplications par la méthode usuelle. Bien entendu, cette méthode, fastidieuse à la main, révèle toute sa puissance pour une machine devant effectuer le produit de grands nombres.

IV. Pseudocode

Nous allons implémenter l'algorithme de Katastuba en utilisant les représentations binaires des nombres. Au lieu de considérer les nombres en base 10 comme dans l'exemple ci-dessus, nous allons les considérer en base 2 et appliquer le même algorithme.

Voici la documentation officiel Python sur les opérateurs binaires :
https://wiki.python.org/moin/BitwiseOperators

et la fonction bit_length() de la classe in qui nous sera utile :
https://docs.python.org/3/library/stdtypes.html#int.bit_length

```
fonction karatsuba(nombre_1 : entier, nombre_1 : entier) -> entier :

    if nombre_1 est codé sur 1 bit ou moins
        ou nombre_2 est codé sur 1 bit ou moins
    alors
        retourner nombre_1 * nombre_2

    # L'objectif ici est de couper les nombres en 2 parties
    # On utilise les opérations sur les bits pour cela

    k = le maximum entre la moitié du nombre de bits de nombre_1
        et la moitié du nombre de bits de nombre_2
    a = on décale les bits du nombre_1 de k bits vers la droite
    b = on applique un XOR bit à bit entre nombre_1
        et a avec ses bits décalés de k bits vers la gauche
    c = on décale les bits du nombre_2 de k bits vers la droite
    d = on applique un XOR bit à bit entre nombre_2
        et c avec ses bits décalés de k bits vers la gauche

    ac = appel récursif pour calculer a*c
    bd = appel récursif pour calculer b*d
    p = appel récursif pour calculer (a-b)*(c-d)

    retourner la somme de
        ac avec ses bits decalés vers la gauche 2*k fois avec
        ac+bd-p avec ses bits decalés vers la gauche k fois avec
        bd.
```