Projet 1 :
la FFT et arithmétique rapide

UE : Fourier - Convolution

Étudiants :
Tahar AMAIRI
Hamza RAIS

Enseignant :
Fabrice BETHUEL

31 janvier 2022

Table des matières

1 Introduction

La multiplication des entiers est une opération très élémentaire qu'on apprend dès le plus jeune âge. Cependant, la question de la faire la plus efficacement possible constitue un domaine de recherche actif. On distingue deux types d'algorithmes :

- **la multiplication basée sur la notation décimale** : elle se base sur la décomposition décimale des nombres et nécessite de multiplier chaque chiffre du premier nombre par chaque chiffre du second. Par conséquent, il est nécessaire de connaître les tables de multiplications.
- **la multiplication rapide** : avec l'apparition des ordinateurs, de nombreux algorithmes plus rapides adaptés pour les grands nombres ont été mis au point.

Ce projet vise donc à présenter ces algorithmes dont celui de Schönhage-Strassen qui utilise la transformée rapide de Fourier (FFT). Nous verrons aussi la méthode de la multiplication "standard" ou bien l'algorithme de Karatsuba. Bien plus, nous comparerons leur efficacité.

Afin d'évaluer la performance de ces algorithmes, on utilisera le nombre d'opérations comme métrique, qui est exprimé en fonction du nombre n de chiffres des nombres à multiplier. Bien plus, cette unité rejoint la notation du big $\mathcal{O}$, très utilisée en informatique pour classifier les algorithmes selon leur temps d'exécution.

Pour des raisons de simplicité, nous nous placerons dans le cas où les nombres seront exprimés en base 10 et auront le même nombre de chiffre. Cependant, les algorithmes présentés peuvent être utilisés en dehors de ces cas spécifiques par exemple avec la base 2 utilisée par les ordinateurs. Par ailleurs, la plupart de ces algorithmes sont adaptés pour des entiers positifs. Ainsi, pour la multiplication d'entiers négatifs, il suffit de connaître au préalable le résultat du signe puis de passer à la valeur absolue : c'est donc un cas qui ne sera pas étudié ici.

Finalement, l'implémentation se fera sous Python avec un Jupyter Notebook en utilisant le module Numpy.

2 Algorithme de Schönhage-Strassen

2.1 Description

L'algorithme de Schönhage-Strassen est un algorithme de multiplication asymptotiquement rapide pour les grands nombres entiers. Il a été développé par Arnold Schönhage et Volker Strassen en 1971. Révolutionnaire, il consiste à faire un grand détour par la transformée de Fourier en utilisant la FFT.

2.2 Étapes

Avant d'expliciter le fonctionnement de cette algorithme, il est important de revenir sur certaines notions fondamentales :

1. Définition : La *transformée de Fourier discrète* transforme une suite de N nombres complexes $x_n = x_0, x_1, \ldots, x_{N-1}$ en une autre suite de nombres complexes, $X_k = X_0, X_1, \ldots, X_{N-1}$, qui est défini par :

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn}$$

De manière analogue, la *transformée inverse* (IFFT) est défini par :

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{\frac{i2\pi}{N}kn}$$

Cette discrétisation a une complexité de $\mathcal{O}(n^2)$ mais en utilisant une FFT avec l'algorithme récursive de Cooley-Tukey celle-ci devient $\mathcal{O}(nlog(n))$.

2. Théorème : Notons par $\mathcal{F}$ l'opérateur de la transformation de Fourier et $\mathcal{F}^{-1}$ son inverse. Soit f et g deux fonctions. Le *théorème de convolution* stipule donc :

$$\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g) \iff f * g = \mathcal{F}^{-1}(\mathcal{F}(f) \cdot \mathcal{F}(g))$$

En d'autres termes, pour obtenir le produit de convolution entre deux fonctions il suffit de calculer leur transformation de Fourier puis d'y appliquer l'inverse.

L'algorithme de Schönhage-Strassen repose essentiellement sur ces deux notions et c'est pour cela qu'il a une approche très innovante. En effet, lorsqu'on effectue une multiplication on manipule des entiers et non des fonctions. Ainsi, il faut trouver un moyen d'exprimer un entier comme une fonction et plus précisément comme un polynôme. Pour cela, il faut décomposer un entier dans sa base décimale et cette idée est très similaire à la décomposition binaire. Notons n le nombre de chiffres composant x, f la fonction polynomiale qui lui est associé et a_k ses coefficients, nous avons donc :

$$\forall x \in \mathbb{N},\ x = \sum_{k=0}^{2^n} a_k \cdot 10^k = f(10)$$

Après ce bref rappel, nous pouvons maintenant expliciter le fonctionnement de l'algorithme de Schönhage-Strassen :

1. Soit x et y deux entiers positifs de même taille n. La première étape consiste à décomposer ces deux entiers en deux polynômes f et g.
2. On calcule la transformée de Fourier de f et g avec la FFT. On notera respectivement le résultat F et G.
3. On effectue le produit point par point entre F et G. On calcule par la suite la transformée inverse de ce résultat. D'après le théorème de convolution, nous venons de calculer $f * g$.
4. Finalement, on évalue le produit de convolution obtenu dans la base 10 en recomposant le résultat. De cette manière, nous venons d'obtenir le résultat du produit $x \times y$.

Ici, nous avons appliqué l'algorithme avec deux entiers de même taille mais on peut ne pas se limiter à ce cas de figure en fixant n comme le maximum entre la taille des deux nombres à multiplier. Bien plus, on peut aussi étendre l'algorithme à d'autres bases autre que la décimale.

2.3 Exemple

Prenons $x = 38$ et $y = 19$ d'où $x \times y = 722$. Appliquons maintenant l'algorithme de Schönhage-Strassen :

1. On décompose x et y comme suit :

$$x = 8 \times 10^0 + 3 \times 10^1 + 0 \times 10^2 + 0 \times 10^3 \text{ d'où } a_k = (8, 3, 0, 0)$$
$$y = 1 \times 10^0 + 9 \times 10^1 + 0 \times 10^2 + 0 \times 10^3 \text{ d'où } a_k = (9, 1, 0, 0)$$

2. On calcule la transformée de Fourier discrète de x et y à l'aide de la FFT et des vecteurs a_k obtenus :

$$X = (11, 8 - 3i, 5, 8 + 3i)$$
$$Y = (10, 9 - i, 8, 9 + i)$$

3. On calcule maintenant le produit point par point entre X et Y :

$$X \cdot Y = (110, 69 - 35i, 40, 69 + 35i)$$

   Puis on détermine le produit de convolution à l'aide d'une transformation inverse appliquée à ce produit, d'où :

$$(72, 35, 3, 0)$$

4. Finalement, on redécompose ce produit de convolution dans la base décimale :

$$x \times y = 72 \times 10^0 + 35 \times 10^1 + 3 \times 10^2 + 0 \times 10^3 = 72 + 350 + 300 = 722$$

2.4 Complexité

La complexité de cet algorithme dépend énormément de celui utilisé pour effectuer la transformée de Fourier aux étapes 2 et 3. Comme nous utilisons deux fois la FFT et une seule fois l'IFFT, nous avons donc une complexité de $\mathcal{O}(3n \cdot log(n)) = \mathcal{O}(n \cdot log(n))$. On ajoute à cela le produit point par point s'effectue en temps linéaire, i.e, en $\mathcal{O}(n)$. Finalement, on retrouve une complexité de $\mathcal{O}(n \cdot log(n) \cdot loglogn)$ (cela dit, la démonstration ne se réduit pas à cela mais est un peu plus complexe).

2.5 Mise en oeuvre informatique

Présentons en premier lieu le pseudo-code de l'algorithme de Schönhage-Strassen :

Algorithm 1: Algorithme de Schönhage-Strassen

Input: $x, y \in \mathbb{N}$ de taille n
Output: résultat de l'opération $r = x \times y$

1 polX $\leftarrow$ `Decomposition`(x) `// Etape 1`
2 polY $\leftarrow$ `Decomposition`(y)
3 $X \leftarrow$ `FFT`(polX) `// Etape 2`
4 $Y \leftarrow$ `FFT`(polY)
5 $V \leftarrow$ `IFFT`$(X \cdot Y)$ `// Etape 3`
6 r $\leftarrow$ `Recomposition`(V) `// Etape 4`

Pour la première étape, nous utilisons la fonction `toPoly` pour décomposer un entier x en un polynôme :

```
1  #transforme un entier x en un polynôme
2  def toPoly(x):
3
4      digits = list(str(x))
5      digits.reverse()
6      poly = np.zeros(pow(2,len(digits)),dtype=int)
7
8      for i in range(0,len(digits)):
9          poly[i] = int(digits[i])
10
11     return poly
```

On transforme x à la ligne 4 en une liste de `string` puis on inverse son ordre à la ligne 5. Par la suite, nous convertissons chaque élément de cette liste en `int` à l'aide d'une boucle `for` tout en stockant le résultat dans `poly`.

Pour l'étape 2 et 3, afin de calculer le produit de convolution de deux polynômes, nous allons utiliser une unique fonction : `prodConv`. Celle-ci prend en paramètre deux polynômes obtenus avec la fonction `toPoly` :

```
#permet de calculer le produit de convolution entre les polynômes polX et polY
def prodConv(polX,polY):

    #Etape 2 :
        #on calcule tout d'abord la transformée de Fourier des deux
        #polynômes à l'aide de la FFT
    fftX = np.fft.fft(polX)
    fftY = np.fft.fft(polY)

    #Etape 3.1 :
        #on effectue le produit point par point des deux transformations
    prod = fftX * fftY

    #Etape 3.2 :
        #on calcule finalement la transformée inverse de Fourier :
        #le résultat correspond au produit de convolution souhaité
    ifft = np.fft.ifft(prod)

    return ifft
```

Pour calculer la FFT et l'IFFT, nous utilisons directement les fonctions pré-implémentées du module Numpy avec `np.fft.fft` et `np.fft.ifft`. Avec le résultat obtenu, on utilise la fonction `toNumerical` pour réévaluer le produit de convolution dans la base décimale :

```
#permet de transformer le produit de convolution en valeur numérique réelle
def toNumerical(prod):

    prod = (prod.real).astype(int)
    res = 0

    for i in range(0,len(prod)):
        res += prod[i] * pow(10,i)

    return res
```

Finalement, en assemblant toutes les étapes, nous avons la fonction `SchStr` implémentant l'algorithme dans son intégralité :

```
#permet de calculer le produit entre x et y en utilisant l'algorithme
#de Schönhage-Strassen
def SchStr(x,y):
    polX = toPoly(x)
    polY = toPoly(y)
    prod = prodConv(polX,polY)
    return toNumerical(prod)
```

3 Algorithme de Karatsuba

3.1 Description

Cet algorithme a été développé par Anatolli Alexevich Karatsuba en 1960. Il s'agit d'un algorithme de multiplication rapide consistant à multiplier x et y en passant par le calcul de trois sous-multiplications. Soient deux entiers $x, y \in \mathbb{N}$, qu'on peut écrire sous la forme suivante :

$$\forall k \in \mathbb{N}, k < n, \begin{cases} x = q_x \times b^k + r_x, & r_x < b^k \\ y = q_y \times b^k + r_y, & r_y < b^k \end{cases}$$

Par conséquent :

$$x \times y = (q_x \times b^k + r_x) \times (q_y \times b^k + r_y) = q_x q_y \times b^{2k} + (q_x r_y + r_x q_y) \times b^k + r_x r_y$$

Ici, $x \times y$ peut être obtenu à l'aide de quatre produits. On peut simplifier d'avantage le calcul en utilisant la forme ci-dessous :

$$x \times y = (q_x \times b^k + r_x) \times (q_y \times b^k + r_y) = q_x q_y \times b^{2k} + [(q_x + r_x)(q_x + r_y) - (q_x q_y + r_x r_y)] \times b^k + r_x r_y$$

De cette manière, on peut obtenir $x \times y$ à l'aide de seulement trois produits à la place de quatre initialement. Si x et y sont des nombres très grands, on peut appliquer la méthode de Karatsuba de manière récursive afin d'obtenir des calculs plus simples à chaque itération de l'algorithme. Ainsi, cet algorithme est de type **divide and conquer**.

3.2 Exemple

Calculons 1237×2587 à l'aide de la méthode de Karatsuba :

- On écrit 1237×2587 sous la forme : $x_0 \times 10^4 + (x_0 + x_2 - x_1) \times 10^2 + x_2$ où $x_0 = 12 \times 25, x_1 = (12 - 37) \times (25 - 87) = 25 \times 62$ et $x_2 = 37 \times 87$.
- Pour calculer 12×25, on écrit 12×25 sous la forme : $y_0 \times 10^2 + (y_0 + y_2 - y_1) \times 10 + y_2$ avec $y_0 = 1 \times 2, y_1 = (1 - 2) \times (2 - 5) = -1 \times -3$ et $y_2 = 2 \times 5$.
- Les calculs $1 \times 2 = 2$, $2 \times 5 = 10$ et $-1 \times -3 = 3$ se réalisent en temps constants.
- On obtient $x_0 = 12 \times 25 = 2 \times 100 + (2 + 10 - 3) \times 10 + 10 = 300$ et de la même façon : $x_1 = 25 \times 62 = 1550$ et $x_2 = 37 \times 87 = 3219$.
- D'où : $1237 \times 2587 = 300 \times 10^4 + (300 + 3219 - 1550) \times 10^2 + 3219 = 3200119$.

3.3 Complexité

La méthode de Karatsuba est plus efficace lorsque $n = 2^p$ avec $p \in \mathbb{N}$. Dans ce cas, la récursion s'arrête lorsque $n = 1$. On note $M(n)$ le nombre total d'opérations élémentaires que l'algorithme effectue pour multiplier deux nombres à n chiffres. Comme on peut négliger le coût des additions et soustractions lorsque n augmente (car coût constant), $M(n)$ correspond au nombre de multiplications. On obtient alors $M(n) = 3 \times M(n/2)$ et par récurrence on obtient le résultat asymptotique $M(n) = \mathcal{O}(n^{log_2(3)})$. Par conséquent, la complexité de l'algorithme de Karatsuba est $\mathcal{O}(n^{log_2(3)}) \approx \mathcal{O}(n^{1.585})$.

3.4 Mise en oeuvre informatique

L'algorithme de Karatsuba peut être implémenté avec n'importe quelle base b mais nous nous limiterons à la base décimale pour le pseudo-code :

Algorithm 2: Algorithme de Karatsuba

Input: $x, y \in \mathbb{N}$ de taille n
Output: résultat de l'opération $r = x \times y$

1 **if** $x < 10$ *or* $y < 10$ **then**
2 $\quad r \leftarrow x \times y$ // On effectue la multiplication standard
3 $k \leftarrow \texttt{floor}(\frac{n}{2})$ // On prend la partie entière de la division
4 $a \leftarrow \texttt{floor}(\frac{x}{10^k})$
5 $b \leftarrow x\%10^k$ // On prend le reste de la division
6 $c \leftarrow \texttt{floor}(\frac{y}{10^k})$
7 $d \leftarrow y\%10^k$
8 $z_0 \leftarrow \texttt{Karatsuba}(b, d)$ // Appels récursifs
9 $z_1 \leftarrow \texttt{Karatsuba}(a + b, c + d)$
10 $z_2 \leftarrow \texttt{Karatsuba}(a, c)$
11 $r \leftarrow (z_2 \times 10^{2k}) + ((z_1 - z_2 - z_0) \times 10^k) + z_0$

La fonction `floor` permet d'obtenir la partie entière d'une division et l'opérateur %, quant à lui, permet d'obtenir le reste. À partir de la ligne 8, nous effectuons les appels récursifs avec les constantes a, b, c et d. Présentons maintenant l'implémentation informatique effectuée sous Python :

```
1  #permet de calculer le produit entre x et y en utilisant l'algorithme de
2  #Karatsuba
3  def Kara(x,y):
4      if x < 10 or y < 10:
5          return x*y
6
7      k = max(len(str(x)),len(str(y)))
8      k = k // 2
9      a = x // 10**(k)
10     b = x % 10**(k)
11     c = y // 10**(k)
12     d = y % 10**(k)
13     z0 = Kara(b,d)
14     z1 = Kara((a+b),(c+d))
15     z2 = Kara(a,c)
16
17     return (z2 * 10**(2*k)) + ((z1 - z2 - z0) * 10**(k)) + (z0)
```

On remarque que cette implémentation suit de très près le pseudo-code. Pour obtenir la partie entière d'une division, on utilise l'opérateur // et pour le reste, l'opérateur %. Finalement, on transforme un entier x en `string` afin d'obtenir sa taille n via la fonction `len`.

4 Algorithme de multiplication "standard"

4.1 Description

L'algorithme standard est la méthode de multiplication qui est enseignée à l'école primaire. Cette méthode utilise la décomposition décimale des nombres. Elle consiste à multiplier chaque chiffre du premier nombre par chaque chiffre du second, puis de sommer les résultats en appliquant un décalage. Pour utiliser cette méthode, il est donc nécessaire de connaître les tables de multiplication.

4.2 Complexité

Cette méthode exige que les n chiffres de x soient multipliés avec les n chiffres de y. Il en résulte que le nombre d'opérations élémentaires est de l'ordre de $\mathcal{O}(n \times n)$, d'où une complexité quadratique.

4.3 Mise en oeuvre informatique

Avant de présenter le pseudo-code de cet algorithme, nous allons introduire une notation :

Notation : Soit un entier $x \in \mathbb{N}$. La notation x_i permet d'obtenir l'ième chiffre composant x à partir de la droite. Par exemple : si $x = 123$ alors $x_1 = 3, x_2 = 2$ et $x_3 = 1$.

Voici maintenant le pseudo-code et comme d'habitude, on se place dans le cas de l'introduction, i.e, base décimale et nombres de même taille n :

Algorithm 3: Algorithme de multiplication standard

Input: $x, y \in \mathbb{N}$ de taille n
Output: résultat de l'opération $r = x \times y$

$r \leftarrow 0_{2n}$ // une liste initialisée à 0 d'une taille $2n$
for $i \leftarrow 1$ *to* $n-1$ **do**
  for $j \leftarrow 1$ *to* $n-1$ **do**
    $k \leftarrow x_i \times y_j + r_{i+j+1}$
    $r_{i+j+1} \leftarrow k\%10$
    $r_{i+j} \leftarrow r_{i+j} + \texttt{floor}(\frac{k}{10})$

Tout comme pour l'algorithme de Karatsuba, l'implémentation sous Python est relativement directe :

```
#permet de calculer le produit entre x et y en utilisant l'algorithme de
#multiplication standard
def Stand(x,y):
    digX = list(str(x))
    digY = list(str(y))
    res = np.zeros(len(digX) + len(digY), dtype = int)
```

```
7
8       for i in range(len(digX) - 1, -1, -1):
9           for j in range(len(digY) - 1, -1, -1):
10              k = int(digX[i]) * int(digY[j]) + res[i+j+1]
11              res[i+j+1] = k % 10
12              res[i+j] += k//10
13
14      return  int("". join([str(digit) for digit in res]))
```

On utilise toujours la même astuce pour pouvoir parcourir les chiffres d'un entier en le transformant en liste de `string` (lignes 5 et 6). À l'aide du module Numpy, on crée un array `res` d'une taille $2n$ pour stocker le résultat (équivalent au r dans le pseudo-code). Il s'en suit après la double boucle `for` où on l'en retrouve les mêmes opérations que dans le pseudo code. Cependant, pour parcourir les listes `digX` et `digY` de droite à gauche (sens inverse, pour respecter notre notation) on décrémente i et j en les initialisant à $n-1$. Finalement, à la ligne 14, on utilise la fonction `join` pour transformer l'array `res` en `int`.

5 Autres algorithmes

Dans cette section, nous allons présenter succinctement quelques algorithmes qu'on n'a malheureusement pas pu implémenter soit à cause de leur complexité (e.g Fürer) ou bien de leur redondance avec les algorithmes précédents (e.g Toom-Cook).

5.1 Algorithme de Toom-Cook

5.1.1 Description

Toom-Cook, parfois connu sous le nom de Toom-3, est un algorithme de multiplication rapide pour les grands entiers. Il a été décrit par Andrei Toom pour la première fois en 1963 puis amélioré par Stephen Cook en 1966.

Son principe est très similaire à celui de Karatsuba : il se base sur l'idiome **divide and conquer**. En effet, c'est une généralisation de ce dernier : étant donné deux nombres entiers, a et b, Toom-Cook divise a et b en k parties plus petites, chacune de longueur l, et effectue des opérations sur ces parties. Au fur et à mesure que k augmente, on peut combiner plusieurs des sous-opérations de multiplication, en utilisant à nouveau de manière récursive celles-ci, et ainsi de suite. Bien que les termes Toom-3 et Toom-Cook soient parfois utilisés de manière incorrecte et interchangeable, Toom-3 n'est qu'une seule instance de l'algorithme de Toom-Cook, où $k = 3$.

5.1.2 Complexité

La complexité de cet algorithme dépend énormément de la valeur du k. En effet, pour $k = 1$, on revient à effectuer une multiplication standard d'où une complexité quadratique $\mathcal{O}(n^2)$ et pour $k = 3$ nous avons une complexité de $\mathcal{O}(n^{\frac{log(5)}{log(3)}}) \approx \mathcal{O}(n^{1.46})$. En général, pour un k fixé, Toom-k s'exécute en $\mathcal{O}(c(k) \cdot n^e)$, où $e = \frac{log(2k-1)}{log(k)}$, avec n^e le temps passé

sur les sous-multiplications, et c est le temps passé sur les additions et la multiplication des petites constantes. Finalement, pour $k = 2$, on retrouve la même complexité que l'algorithme de Karatsuba.

5.2 Algorithme de Fürer

5.2.1 Description

L'algorithme de Fürer est un algorithme de multiplication d'entiers avec une complexité asymptotique très faible. Il a été publié en 2007 par le mathématicien suisse Martin Fürer comme un algorithme asymptotiquement plus rapide que son prédécesseur, l'algorithme de Schönhage-Strassen. En effet, celui-ci repose aussi sur la FFT.

5.2.2 Complexité

L'algorithme de Fürer peut être utilisé pour multiplier des entiers de longueur n en $\mathcal{O}(n \cdot logn \cdot 2^{\mathcal{O}(log^* n)})$ où $log^* n$ est le logarithme itéré . Par rapport à l'algorithme de Schönhage-Strassen, la différence entre le $loglogn$ et le $2^{\mathcal{O}(log^* n)}$, d'un point de vue de la complexité, est asymptotiquement à l'avantage de l'algorithme de Fürer.

5.3 Algorithme de Harvey et van der Hoeven

5.3.1 Description

En 2019, les chercheurs David Harvey et Joris van der Hoeven ont annoncé la mise en point d'un algorithme pour multiplier efficacement les entiers avec la plus faible complexité connu à ce jour.

5.3.2 Complexité

Avec l'algorithme de Harvey et van der Hoeven, le nombre d'opérations élémentaires à effectuer est de l'ordre de $\mathcal{O}(n \cdot log(n))$. Ce qui est remarquable, c'est que c'est (conjecturalement) le meilleur ordre de grandeur possible. En effet, Schönhage et Strassen ont prédit que $\mathcal{O}(n \cdot log(n))$ est le "meilleur résultat possible".

6 Comparaison

Premièrement, la multiplication standard a une complexité quadratique. En 1962, apparaît, l'algorithme de Karatsuba avec une complexité de $\mathcal{O}(n^{\frac{log(3)}{log(2)}}) \approx \mathcal{O}(n^{1.58})$, ce qui en fait le premier algorithme sub-quadratique. Trois ans après, Andrei Toom et Stephen Cook mettent au point une généralisation de l'algorithme de Karatsuba ayant une complexité de $\mathcal{O}(n^{\frac{log(5)}{log(3)}}) \approx \mathcal{O}(n^{1.46})$ pour $k = 3$. En 1971, l'algorithme de Schönhage-Strassen, en utilisant la FFT, arrive à une complexité de $\mathcal{O}(n \cdot log(n) \cdot loglogn)$. En 2007, se basant aussi sur la FFT, Fürer développe un algorithme ayant une meilleure complexité asymptotique que celui de Schönhage-Strassen. Finalement, en 2019, l'algorithme de Harvey et van der Hoeven permet d'obtenir une complexité linéarithmique.

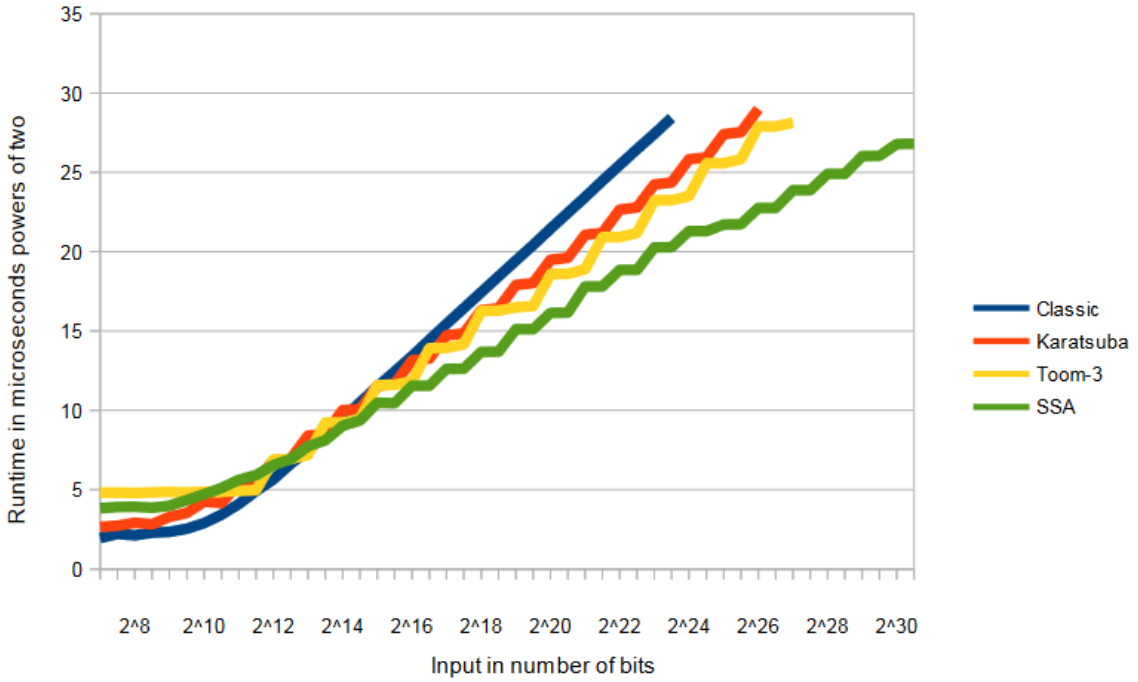

FIGURE 1 – Temps d'exécution pour l'opération x^2

Afin de comparer les différents algorithmes étudiés et implémentés, nous allons nous référer à ce graphique (voir [5]). En effet, l'implémentation du test pour effectuer la mesure du temps d'exécution n'est pas très compliquée (on peut utiliser le module `time`), cependant, pour obtenir des résultats cohérents il faut faire des calculs avec des entiers très grands or nous n'avons pas le matériel nécessaire (au niveau de la RAM notamment pour allouer assez d'espace pour les `numpy.array`).

On remarque qu'au fil du temps (i.e par année) la complexité s'améliore, cependant, ce n'est pas la tendance que l'on remarque sur le graphique. En effet, il ne faut pas oublier qu'on parle de complexité asymptotique et donc, certains algorithmes atteignent ce régime sous certaines conditions car ils sont adaptés pour les très grands entiers. Ainsi, il est préférable d'utiliser la multiplication standard pour les petits entiers, l'algorithme de Toom-Cook/Karatsuba pour les entiers de taille moyenne et l'algorithme de Schönhage-Strassen pour les grands entiers car, en pratique, celui-ci commence à être plus performant que les anciennes méthodes pour les nombres entre $2^{2^{15}}$ et $2^{2^{17}}$ (10 000 à 40 000 chiffres décimaux).

Finalement, concernant l'algorithme de Harvey et van der Hoeven, celui-ci est très peu pratique car il atteint le régime asymptotique pour des nombres d'une taille supérieure à $2^{1729^{12}}$ (ce qui est irréaliste pour une représentation numérique car, par exemple, l'univers compte près de $10^{23} \approx 2^{77}$).

7 Conclusion

En conclusion, le domaine d'application de la transformation de Fourier discrète ou continue ne se limite pas à celui du traitement du signal uniquement. En effet, comme nous l'avons pu le voir à travers ce projet, le détour utilisé par Schönhage et Strassen avec la FFT a permis d'obtenir un algorithme permettant la multiplication de très grands entiers de manière très rapide. Ainsi, la FFT a contribué énormément au domaine de l'arithmétique rapide qu'aujourd'hui l'algorithme de Schönhage-Strassen est utilisé pour approximer π ou pour la recherche de nombre de Mersenne premier.

8 Bibliographie

[1] *Algorithme de Fürer*. URL : https://fr.wikipedia.org/wiki/Algorithme_de_F%C3%BCrer.

[2] *Convolution theorem*. URL : https://en.wikipedia.org/wiki/Convolution_theorem.

[3] *Fast Fourier transform*. URL : https://en.wikipedia.org/wiki/Fast_Fourier_transform.

[4] *Karatsuba algorithm*. URL : https://en.wikipedia.org/wiki/Karatsuba_algorithm.

[5] Theo Kortekaas. *Multiplying large numbers and the Schonhage-Strassen algorithm*. URL : https://tonjanee.home.xs4all.nl/SSAdescription.pdf.

[6] *Long multiplication*. URL : https://rosettacode.org/wiki/Long_multiplication.

[7] *Multiplication algorithm*. URL : https://en.wikipedia.org/wiki/Multiplication_algorithm.

[8] *Multiplying 41*37 with Fast Fourier Transform by hand*. URL : https://www.youtube.com/watch?v=YDhsLhTK3Bs.

[9] *Schönhage–Strassen algorithm*. URL : https://en.wikipedia.org/wiki/Sch%C3%B6nhage%E2%80%93Strassen_algorithm.

[10] *Toom–Cook multiplication*. URL : https://en.wikipedia.org/wiki/Toom%E2%80%93Cook_multiplication.

List of Algorithms