Rapport Model

Auteur :

- ZHOU Runlin 28717281
- XU Mengqian 21306077
- ZHANG Zhile 21201131

Introduction:

Ce projet implique la mise en œuvre des fonctions de ***transformation de Fourier rapide*** (FFT) et de ***FFT inverse*** pour des vecteurs de taille 2^k en virgule flottante double. Le projet couvre également l'implémentation et la comparaison d'algorithmes naïfs et basés sur la FFT pour la multiplication de polynômes.

Le but du projet est pour évaluer et contraster leur efficacité dans des polynôme en grand dégrée.

Pour implémenter ces fonctions, nous avons créé un total de trois parties de fichiers de code:

1. utils.c / .h , fastFourierTrans.c / .h, multPoly.c / .h : placées dans archive /src.
2. test.c : Tester la validité des fonctions.
3. comparison.c / traceGraph.py : pour produire les résultats graphique.

ainsi qu'un fichier **makefile** pour Automatiser le processus de construction, un **fichier de résultat** en format de csv.

Nous expliquons progressivement la mise en œuvre de chaque partie du code et analysons les résultats obtenus.

Présentation des fichiers de code:

Utils.c / .h

Définit la représentation des nombres complexes et des polynômes, ainsi que certaines opérations de base sur les éléments. Le fichier comprend également des fonctions de calcul modulaires.

Les objects :

- ***struct numComplex* :** Il contient deux champs
  - *real* : partie réelle
  - *imaginary* : partie imaginaire
- ***struct polynomial* :** Il contient un tableau de numComplex avec deux champs *size* et *cpt.*
  - *cpt* : le nombre de l'élément,
  - *size* : la taille du tableau.

Les fonctions :

Gestion pour numComplex:

- `randomConsNumber(); creatComplexNum(real, imaginary); zeroComplexNum();`

  utilisées pour la création et l'initialisation d'objets. Les nombres aléatoires sont compris les décimales entre 0 et 20.
- Arithmétique élémentaire pour les nombres complexes:
  - `addComplexNumber(numA, numB);`

    Implémenter l'addition de deux nombres complexes en additionnant respectivement les parties réelles et les parties imaginaires.
  - `multComplexNumber(numA, numB);`

    Implémenter la multiplication de deux nombres complexes en utilisant la formule de multiplication $(a + bi) * (c - di) = (ac + bd) + (bc - ad)i$ et renvoyer le résultat.
  - `divComplexNum(n, num);`

    Implémenter la division d'un nombre complexe par un entier, en divisant respectivement la partie réelle et la partie imaginaire par n

Gestion pour polynomial:

- `createRandomPoly(int size);`

  ajouter les éléments itératifs dans un boucle forte
- `addElement(el, *poly); subElement(index, *poly);`

  Passage des paramètres sur le pointeur. En cas de succès, retourner 1, et 0 sinon.
  Les éléments sont ajoutés à la fin du tableau par défaut, et supprimés par l'index.

- `isEqPoly(polyA, polyB);`

fastFourierTrans.c / .h

L'idée de base de l'algorithme FFT est de décomposer un polynôme de longueur N en deux sous-signaux de longueur $\frac{N}{2}$, puis de combiner les résultats de ces sous-polynôme dans le domaine des fréquences.

Le nombre total d'opérations est en $O(N * log(N))$ car il y a $O(N)$ opérations à chaque niveau et il y a $O(log(N))$ niveaux au total.

- `extensionVec(poly, k);`

  Remplir le tableau de *original* avec $0 + 0i$, jusqu'à ce qu'il atteigne 2^k.

- `coreFFT(*num, size);`

  Cette fonction décompose récursivement le polynôme en parties de degrés impairs et pairs, utilisant les racines primitives pour effectuer les calculs. Les résultats sont stockées dans *res*. Les étapes principales sont les suivantes :

  1. Vérifier si le polynôme d'entrée ne comporte qu'un terme (degré 0). Sinon, divisez le polynôme d'entrée en parties de degrés impairs *p_odd* et pairs *p_even*.
  2. Effectuer des appels récursifs de la fonction `coreFFT` sur *p_odd et p_even.*
  3. Calculer le polynôme résultant *res*. Pour chaque élément, on applique l'équation suivant :

$$\textbf{For } j \textbf{ from } 0 \textbf{ to } \frac{n}{2} - 1 \textbf{ do}$$
$$\quad P(\omega^j) = P_e(\omega^{2j}) + \omega^j P_o(\omega^{2j}) = P_e(\tau^j) + \omega^j P_o(\tau^j).$$
$$\quad P(\omega^{\frac{n}{2}+j}) = P_e(\omega^{2j}) - \omega^j P_o(\omega^{2j}) = P_e(\tau^j) - \omega^j P_o(\tau^j).$$

- `fft(*num, size);`

  Avant qu'on appel la fonction `coreFFT`, on d'abord d'appel la fonction `extenstionVec`.

- `fftInverse(*num, size);`

  D'après qu'on appel la fonction `fft`, on fait la changement du tableau (*data[i] =data[size-i]*) et divide par n.

multPoly.c / .h

- `NaiveMultPoly(polyA, polyB);`
  1. calcule la taille du polynôme résultant ($polyA.size + polyB.size - 1$)et créer un tableau vide de la longueur correspondante.
  2. à l'aide de deux boucles fortes, multiplier chaque paire de coefficients des polynômes d'entrée. Puis on additionne le résultat au coefficient correspondant

     $$res.data[i+j] += polyA.data[i] * polyB.data[j]$$

  Donc, la complexité de l'algorithme est en $O(n^2)$

- `fftMultPoly(polyA, polyB);`
  1. définie le *maxDegree* du polynôme comme $polyA.size + polyB.size - 1$
  2. effectuer `fft` sur les deux polynômes d'entrée pour obtenir deux résultats intermédiaires, *resP* et *resQ*.
  3. créer un polynôme temporaire *stocker*, et multiplier les coefficients correspondants de *resP* et *resQ* avec: $res[i] = resP[i] * resQ[i]$
  4. retourne le résultat final en effectuant `fftInverse` avec res

  La fonction appel deux fois de `fft` et un fois de `fftInverse`. En total, le nombre d'operation est $3 * N * log(N)$, donc la complexité est en $O(N * log(N))$

---

comparison.c

Les fonctions:

- `measureTime((*function), polyA, polyB);`

  Mesurer le temps d'exécution de la fonction du paramètre en utilisant `time.h`

- `runBenchmarks(n);`

  Calculer le nombre de tests à effectuer pour des polynômes de degrés croissants, avec un intervalle de 100.

  Pour chaque itération, générer deux polynômes aléatoires (*polyA* et *polyB*), mesurer le temps d'exécution de `NaiveMultPoly` et `fftMultPoly`, et enregistrer les résultats dans le fichier benchmark_results.csv

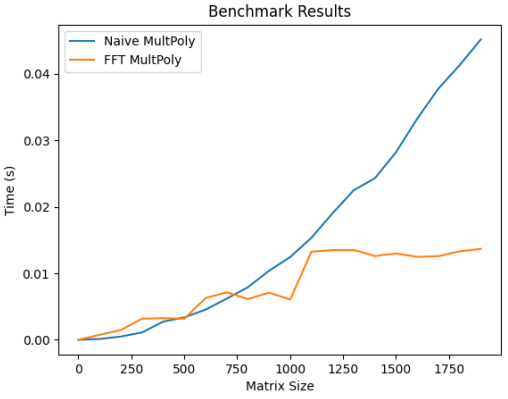
  l'ên-tête du ficher benchmark_results.csv se compose de trois parties:

- la taille du polynôme (Size),
- le temps d'exécution de la méthode Naïve
- le temps d'exécution de la méthode FFT

Analyse des résultats:

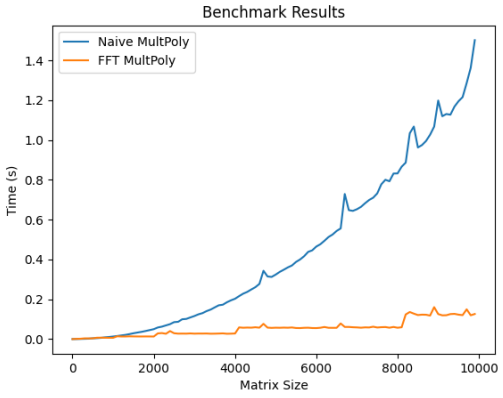
Nous observons que le temps d'exécution de l'algorithme naïf de multiplication de polynômes augmente continuellement, avec une croissance plus rapide que l'algorithme FFT, ce qui est conforme au comportement attendu

1. **Multiplication naïve de polynômes** :
   - À mesure que le degré du polynôme augmente, le nombre d'opération dans l'algorithme naïf augmente quadratiquement, entraînant une augmentation rapide du temps d'exécution.



graph 1.1, with n = 2000

2. **Multiplication de polynômes par FFT** :
   - Nous pouvons observer que le temps d'exécution de l'algorithme FFT tend à se stabiliser avec les polynômes de grand degré.
   - Selon la courbe, il y a des points d'inflexion dans le graphique. Cela est dû au fait que l'algorithme fft ne fonctionne que pour les polynômes avec $n = 2^k$. Donc les polynômes doivent être exécuté `extensionVec` lorsque



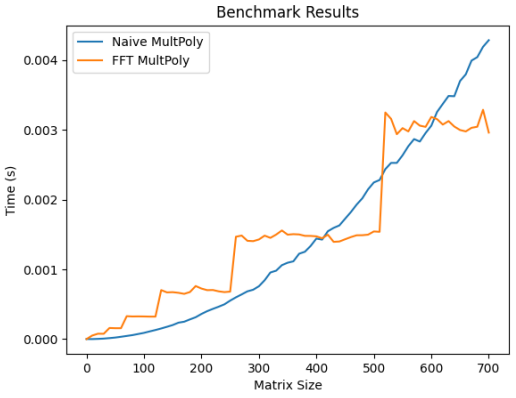
graph 1.2, with n = 10000

$n > 2^k$, Cela revient à calculer le polynôme de 2^{k+1}

Comparaison des deux algorithmes :

Il est évident que l'algorithme fft aura un avantage inégalé lorsque n est grand, mais pour des n plus petits, la situation est différente.

- Lorsque n est inférieur à 400, la méthode naïve est plus performante que fft
- Lorsque n est dans l'intervalle [400, 600], la méthode naïve est presque aussi efficace que fft
- Lorsque n est supérieur à 600, la méthode fft est significativement meilleure que la méthode naïve.



graph 1.3, with n = 700

Par conséquent, si on considère la complexité spatiale et la difficulté de la programmation, nous devrions utiliser la méthode naïve lorsque n<600

En conclusion, l'analyse détaillée montre que l'algorithme naïf est plus efficace pour de petits polynômes, mais à mesure que la taille du polynôme augmente, l'algorithme FFT devient progressivement plus performant

---

Difficultés dans la réalisation:

- Gestion du projet : Lors des tests ultérieurs, il est difficile de gérer le grand nombre de fichiers, de fonctions et d'objets : il y a de nombreuses fuites de mémoire, les fichiers ne sont pas bien classés, etc. En conséquent,
  - ajouter un certain nombre de méthodes à l'objet pour nous permettre de le manipuler.
  - créer les archives src / obj / bin pour placer les codes sources, les ficher objet, les exécutables.

  Avec cette modification, nous pensons que la lisibilité du code s'en trouve considérablement améliorée

- Précision des calculs :
  - Analyse de la stabilité numérique : en particulier pour les calculs impliquant des valeurs importantes ou similaires. Par exemple, si on a besoin de calculer la valeur de $x - y$, on peut le remplacer par $\frac{x^2-y^2}{x-y}$
  - Normalisation : Les données d'entrée sont normalisées et mises à l'échelle. Cela permet de réduire les erreurs de virgule flottante dans les calculs numériques
  - Ajout d'informations de sortie : l'ajout d'informations de sortie au projet, telles que les résultats intermédiaires, les variables intermédiaires, etc. Cela m'aidera à localiser la cause du problème.
  - Gestion des erreurs et des exceptions : incorporer des mécanismes stricts de gestion des erreurs et des exceptions dans le code afin de pouvoir détecter les fluctuations ou les tendances déraisonnables des calculs.

Conclusion:

La FFT a une complexité temporelle de O(n*log(n)), plus efficace que la version naïve. Cependant, en termes de complexité spatiale, la FFT nécessite plus de mémoire en raison des calculs séparés pour P et Q. Pour de petits degrés de polynômes, la version naïve n'a pas de différence notable, mais pour des degrés plus élevés, la FFT réduit significativement le temps.