Analyse d'algorithmes et validation de programmes

Projet
[ISTY]

- UVSQ, Paris Saclay -

Olivier Benaben

15 Février 2021

1 Introduction

1.1 Choix du problème

Pour ce projet j'ai choisis de m'intéresser aux `algorithmes de décomposition de nombres en produit de facteurs premiers`.

Ce choix vient du fait que ces algorithmes sont très importants ajourd'hui par la place stratégique qu'ils occupent dans le domaine de la cryptographie.
En effet le chiffrements asymétriques le plus connu et le plus utilisé - le *RSA* - consiste à générer ses clefs à partir de deux très grands nombres premiers, qui sont multipliés ensuite (on appelle cette partie de la clef N).
Une attaque du RSA consisterait donc à retrouver les deux premiers p et q tels que $N = p \times q$, et c'est là qu'interviennent les algorithmes de décomposition en produit de facteurs premiers.

Les meilleurs algorithmes actuels résolvant ce problème sur des machines "normales" (non-quantiques) sont *sous-exponentiels* mais aussi *super-polynomiaux*.
C'est clairement un problème appartenant à la classe NP : il est suffisement compliqué, mais en vérifier une solution est très simple (il suffit en effet de multiplier les solutions et d'en vérifier l'égalité avec le nombre en entrée.
En revanche on peut trouver l'algorithme de Shor qui, executé sur un ordinateur quantique, arrive à le résoudre en temps polynomial !
Le plus grand nombre factorisé par ce type de marchines s'élève cependant à ce jour à $35 = 7 \times 5$.

Dans ce devoir nous comparerons trois algorithmes de décompositions en facteurs premiers : une première approche par Division Successive,

Table des matières

2 Méthode par Divisions Successives

C'est l'algorithme le plus simple et le plus evident auquel on peut penser pour factoriser un nombre en facteurs premiers. C'est aussi un algorithme glouton, comme on peut voir a la lecture du code :

```
1  func trial_division(n int) []int {
2    var a []int
3    var f = 2
4
5    for n > 1 {
6      if n % f == 0 {
7        a = append(a, f)
8        n /= f
9      } else {
10       f += 1
11     }
12   }
13
14   return a
15 }
```

2.1 Principe

Cet algorithme a ete ennonce par Fibonacci de la maniere suivante :

On test pour tous les entiers premiers p_i entre 2 n si ce n peut etre divise par le p_i. En trouvant un premier p_p qui divise n, on trouve automatiquement le p_q associe tel que : $p_p \cdot p_q = n$. On se retrouve avec une liste de nombres qui divisent n. On selectionne alors les deux plus grands

2.2 Complexité

L'algorithme n'utilise que les variables A et b (on considerera les variables i, j et k commes negligeables).
La complexite en espace de l'algorithme est donc :

$$|A| + |b| = n \cdot n + n = n^2 + n$$

2.3 Améliorations

Wheel Factorisation

3 Méthode du Crible Rationnel

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4 #include <time.h>
5
6 double f (double x) {
7   return 4/(1 + x*x);
8 }
```

3.1 Complexité

L'algorithme n'utilise que les variables A et b (on considerera les variables i, j et k commes negligeables).
La complexite en espace de l'algorithme est donc :

$$|A| + |b| = n \cdot n + n = n^2 + n$$

3.2 Améliorations

Special number field sieve, General number field sieve

4 Méthode de Fermat

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4 #include <time.h>
5
6 double f (double x) {
7   return 4/(1 + x*x);
8 }
```

4.1 Complexité

L'algorithme n'utilise que les variables A et b (on considerera les variables i, j et k commes negligeables).
La complexite en espace de l'algorithme est donc :

$$|A| + |b| = n \cdot n + n = n^2 + n$$

4.2 Améliorations

Special number field sieve, General number field sieve

5 Autres algorithmes

5.1 Algorithme du Crible Algébrique

C'est aujourd'hui l'algorithme le plus efficace pour la factorisation en facteurs premier pour un grand nombre. Les derniers "records" de factorisation sonts tous faits à partir de cet algorithmes, et recemment il a été capable de factoriser un nombre RSA-240 (ie. un nombre de 240 bits).

Le Crible Algébrique étant la méthode de référence, c'est sur sa complexité que l'on fixe la longueur des clefs RSA pour garder un niveau de securité.

L'algorithme étant trop complexe et mathématiquement poussée, il ne m'était pas possible de l'impémenter pour ce projet. Voici cependant les étapes de la méthode :

1. Création de polynomes en rapport avec les entrées
2. Test et selection d'un des polynomes
3. Trouver des "relations" algebriques et relationelle avec des nombres friables en rapport avec la base factorielle du polynome.
4. Création et résolution d'une large matrice pour trouver les les sets de relation dont le produit des normes de ce set sont un carré parfait.
5. Trouver la racine carrée des polynomes sur un corps fini
6. Executer le Théorème des Restes Chinois
7. et enfin tester les resultats produits.

6 Ressources

Wikipedia
https://www.wikiwand.com/en/Integer_factorization
https://www.wikiwand.com/en/Dixon%27s_factorization_method
https://www.wikiwand.com/en/Quadratic_sieve
https://www.wikiwand.com/en/General_number_field_sieve
https://www.wikiwand.com/en/Fermat%27s_factorization_method
https://www.wikiwand.com/en/RSA_(cryptosystem)
https://www.wikiwand.com/en/Shor%27s_algorithm

RSA Cracking Challenge
https://medium.com/asecuritysite-when-bob-met-alice/cracking-rsa-a-challenge-generator-2b

Crible Algébrique
https://github.com/AdamWhiteHat/GNFS/blob/master/Integer%20Factorization%
20-%20Master%20Thesis%20-%20Per%20Leslie%20Jensen.pdf
https://github.com/AdamWhiteHat/GNFS/blob/master/The%20Multiple%20Polynomial%
20Quadradic%20Sieve%20-%20R.D.%20Silverman.pdf