Compte rendu TP2 API

Paul Blanchet Eloi Charra

21/10/2021

1 Introduction

Lors de ce TP, nous avons écrit 5 algorithmes différents qui calculent les N premiers nombres de la suite de fibonacci. La suite est définie comme : $F_0 = 0, F_1 = 1$ et $F_n = F_n - 1 + F_n - 2$ pour $n \geq 2$. Pour chaque algorithme nous analyserons les résultats obtenus ainsi que la durée des calculs pour 5 valeurs de N différentes.

2 Fibonacci

2.1 Récursion

Pour la première version de l'algorithme de fibonacci, on remarque que la complexité en temps croît rapidement en fonction de N entré. On obtient :

N	Résultat	Temps
10	55	0
20	6765	0
40	102334155	1s
75	?	trop long
100	?	trop long

Cette version récursive de l'algorithme possède une complexité de $\mathcal{O}(2^n)$ ce qui fait que des valeurs supérieures à 40 mettront plus de 5 secondes à se calculer et cette durée ne cessera d'augmenter. Cette complexité est due à l'addition des deux sous calculs de `n - 1` et `n - 2`.

2.2 Iteration

La version itérative de l'algorithme possède une complexité de $\mathcal{O}(n)$ (car l'algorithme ne comporte qu'une boucle `for`), ce qui nous permet d'obtenir les résultats plus rapidement que la version récursive.

N	Résultat	Temps
10	55	0
20	6765	0
40	102334155	0
75	2111485077978050	0
100	3736710778780434371	0

Nous remarquons que les valeurs sont calculées instantanément, il faudra attendre d'avoir N supérieur à 10000000000000 (dix billions) pour que le résultat apparaisse en plus de 1 seconde. De plus, nous avons changé la taille sur laquelle les entiers sont stockés en choisissant le type `long` (ou `long long`) pour pouvoir acceuilir des valeurs plus grandes dans nos variables.

2.3 Récursion v2

Cette version se repose sur la version itérative en calculant `(F(n),F(n-1))` à partir de `(F(n-2),F(n-1))`. Nous obtenons les mêmes résultats que la version itérative :

N	Résultat	Temps
10	55	0
20	6765	0
40	102334155	0
75	2111485077978050	0
100	3736710778780434371	0

Cet algorithme possède une complexité de $\mathcal{O}(n)$.

2.4 Nombre d'or

Nous pouvons remarquer qu'en utilisant cet aglorithme, les résultats sont globalement identiques aux algorithmes précédents. Cependant, nous pouvons voir une légère différence qui s'accentue en même tant que N grandit dû au fait que des nombres réels sont utilisés en plus de l'arrondi de $\sqrt{5}$. De plus nous pouvons voir que pour `N = 100`, nous obtenons un nombre négatif puisque nous dépassons la zone mémoire associée à cette variable.

N	Résultat	Temps
10	55	0
20	6765	0
40	102334155	0
75	2111485077978055	0
100	-9223372036854775808	0

La complexité de cet algorithme est de $\mathcal{O}(2log_2 n)$ puisque nous appelons deux fois la fonction `exponentiation_rapide` et qu'à chaque tour de récursion de celle-ci, nous divisons n par 2.

2.5 Matrices

Pour l'algorithme utilisant des matrices, nous remarquons également un temps de réponse instantané. De plus, tous les résultats sont justes. Cet algorithme utilise la fonction `exponentiation_rapide_mat` pour effectuer une puissance de matrice. Cet algortihme possède une complexité de $\mathcal{O}(log_2 n)$ puisque nous divisons par 2 notre paramètre n d'entrée à chaque récursion.

N	Résultat	Temps
10	55	0
20	6765	0
40	102334155	0
75	2111485077978050	0
100	3736710778780434371	0

3 Conclusion

Pour conclure, nous pouvons constater que la version la plus simple à écrire (récursion) est la moins performante de toutes n'arrivant pas à calculer des valeurs supérieures à $F(42)$ en temps raisonnable. La version utilisant le nombre d'or est plus performante mais possède néanmoins des approximations de calculs qui faussent les résultats. Enfin, les 3 algorithmes restants (matrices, récursion v2 et itération) sont des algorithmes performants permettant d'obtenir des résultats justes et rapidement pour de grandes valeurs de N.