Mini-projet 1

October 7, 2022

1 Mini-projet 1: Calcul des nombres de Fibonacci : le point de vue expérimental

1.1 Fibonacci

1.1.1 Que sont les nombres de Fibonacci ?

Les nombres de Fibonacci sont une séquence de nombres entiers disposés en 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,... Chaque nombre est la somme des deux nombres précédents. Voici quelques faits intéressants sur les nombres de Fibonacci :

$$\begin{cases} u_0 = 0 \\ u_1 = 1 \\ u_n = u_{n-1} + u_{n-2} \end{cases}$$

Cette suite s'appelle la suite de Fibonacci et c'est une suite infinie. Chaque nombre de la série ou séquence de Fibonacci est représenté par Fn.

1.1.2 Règles pour les nombres de Fibonacci

Le premier nombre de la liste des nombres de Fibonacci est exprimé par F0 = 0 et le deuxième nombre de la liste des nombres de Fibonacci est exprimé par F1 = 1. Les nombres de Fibonacci suivent une règle selon laquelle, Fn = Fn-1 + Fn-2, où n >1. Le troisième nombre de Fibonacci est donné par F2 = F1 + F0. Comme nous le savons, F0 = 0 et F1 = 1, la valeur de F2 = 0 + 1 = 1. La séquence des nombres de Fibonacci va comme 0, 1, 1, 2, et ainsi de suite. La règle des nombres de Fibonacci, si elle est expliquée en termes simples, dit que chaque nombre de la séquence est la somme de deux nombres qui le précèdent dans la séquence.

1.1.3 Implémentation

Il est proposé ici d'implémenter les 3 versions du calcul des nombres de Fibonacci :

1.1.4 Version de base récursive

La suite des nombres un de Fibonacci est définie par la relation de récurrence suivante :

u0=1 , u1=1 , un=un1+un2 nN, n>1 La relation de récurrence exprime le nombre à l'ordre n en fonction des nombres et , respectivement à l'ordre et . Ainsi, on définit directement en fonction de et de . Cette écriture est très compacte et très expressive; en particulier, elle ne fait pas intervenir de boucle comme dans le code de la fonction déjà rencontré dans la définition d'une fonction.

Il serait donc intéressant de définir la fonction fibonacci de manière aussi simple dans le langage algorithmique qu'en mathématiques. Cela nécessite de pouvoir définir une fonction en l'appelant elle-même : on parle alors de récursivité ou de récursion.

1.1.5 1.1.1- Algorithme Récursive de la suite de Fibonacci

Fonction Fibonacci(n: entier): entier;

Var résultat: entier;

Debut

```
Si(n = 0) Alors
    résultat := 0;
Sinon Si(n = 1) Alors
    résultat := 1;
Sinon
    résultat := Fibonacci(n - 1) + Fibonacci(n - 2);
FinSi

Renvoyer résultat;
```

Fin

1.1.6 une analyse de complexité pour la version recursive

Pour analyser la complexité de cet algorithme, on remarque que chaque appel à Fibonacci() se fait en temps constant . Il suffit donc de compter le nombre d'appels, pour n en entrée, que l'on va noter An. La suite An vérifie :

Cette suite ressemble à la suite de Fibonacci. On peut de point de vue mathématique et on trouve que

Cette suite ressemble à la suite de Fibonacci. On peut l'étudier mathematiquement et on trouve que An $\in \Theta(O^n)$

où O= est le nombre d'or. La complexité de l'algorithme est donc exponentielle.

1.2- Version de base itérative

1.2.1- Algorithme itérative de la suite de Fibonacci

```
Fonction fibIter ( n : Entier ) : Entier
  Début

   | f0 <- 0
   | f1 <- 1
   Si ( n = 0 ) Alors
   | | fn <- f0
   | FinSi
   | Si ( n = 1 ) Alors
```

```
    | | fn <- f1
    | FinSi
    | Pour k <- 2 à n Faire
    | | fn <- f1 + f0
    | | f0 <- f1
    | | f1 <- fn
    | FinPour
    | Retourner fn
  Fin
```

1.2.2- une analyse de complexité pour la version itérative En observant la fonction Fibobacci iterative , on constate qu'elle met en œuvre deux types d'opérations, des affectations et des sommes. Plus précisément, le nombre d'affectations A(n) et le nombre de sommes S(n) effectuées par l'algorithme Sous l'hypothèse que chaque affectation et chaque somme prend un temps constant, on s'attend à ce que le temps de calcul de f(n) par cet algorithme soit à peu près linéaire comme fonction de n

1.3- Version basée sur l'exponentiation de matrice Il est basés sur une exploitation astucieuse de mise à la puissance d'une matrice 2 × 2. On donne les propriétés conduisant à cet algorithme :

1.3.1- Algorithme de la méthode d'exponetiation rapide algorithme: exporap

```
Entrées: a apartient a n ,n appartient a n

Résultat:a puissance n
    si n=0 alors
         renvoyer 1
    sinon

    Si n est impair alors
        renvoyer a x exporap(a puissance 2,(n-1)/2)
    Sinon
        renvoyer a x exporap(a puissance 2 ,n/2)
  Fin du si
Fin du si;
```

1.3.2- une analyse de complexité pour la version itérative En utilisant l'algorithme d'exponentiation rapide, cela demande $(2^3 log(n1)) = (logn)$

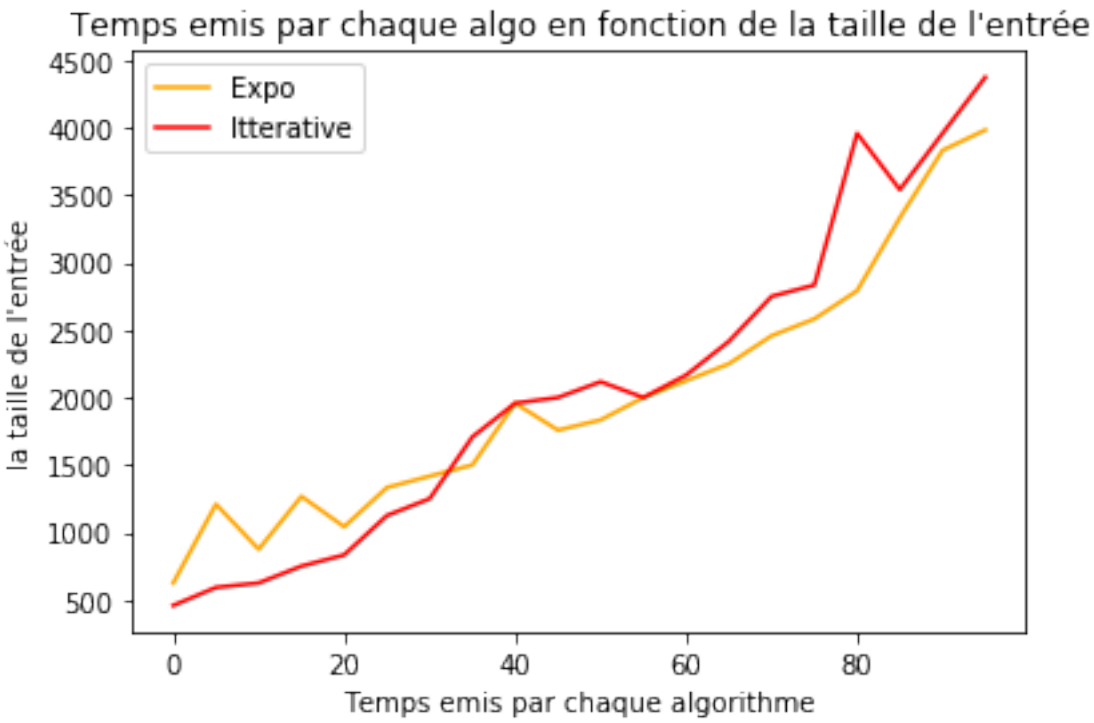
opérations. Le coût total de cette méthode est donc seulement de (log n). puisque ona en entrée: une matrice A de taille 2X2, n entier et on obtient on sortie : la matrice Aˆn

1.3.3 les résultats des tests en fonction de la taille de l'entrée Afin de calculer la durée d'éxecution de chaque algorithme on a utiliser une méthode prédefinie de Java qui est System.nanoTime() au debut et à la fin d'éxecution de chaque algorithme et la difference entre ces deux valeurs(Debut,Fin) est

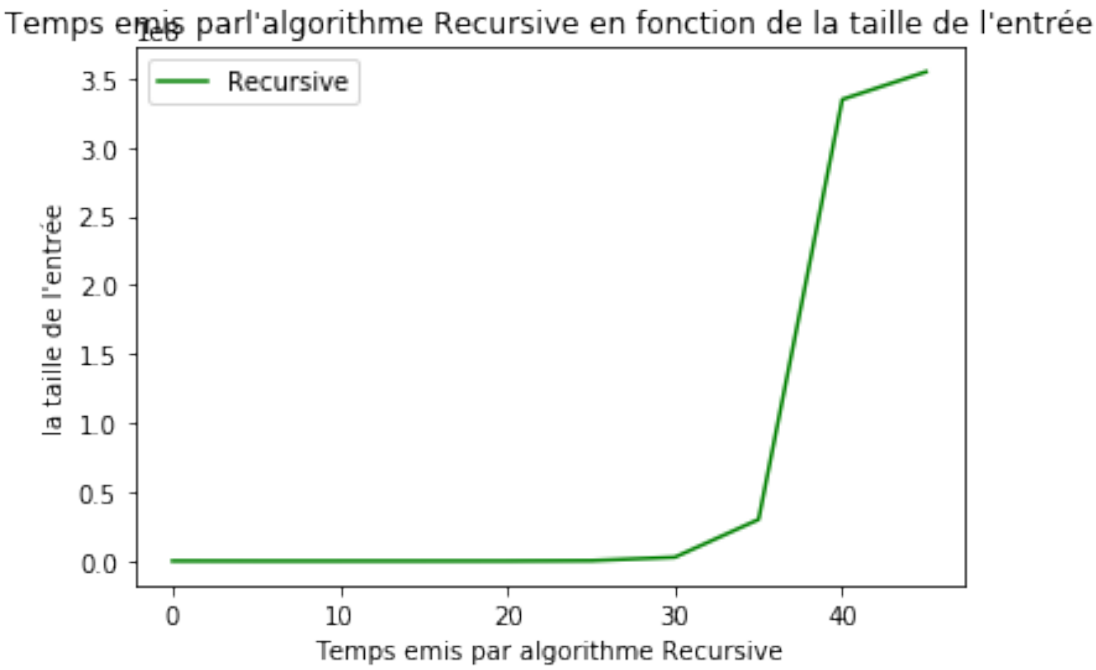
le temps d'execution de l'algorithme comme le montre le tableau ci-dessous.

n	10	20	30	40	50	60	70	80	90
le temps emis par fibo récur	333	22125	30135625	354751666					
temps mis par fibo itéra	592	750	1125	1708	2000	2117	2416	2834	3541
temps mis par fibonacci l'expo$_M$	1209	1267	1333	1500	1758	2000	2250	2583	3333

[15]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
import math


from matplotlib.pyplot import plot, show, legend

i= [3984, 3833, 3333, 2792, 2583, 2458, 2250, 2125, 2000, 1834, 1758, 1958,␣
 ↪1500, 1417, 1333, 1042, 1267, 875, 1209, 625]
x = [95, 90, 85, 80, 75, 70, 65, 60, 55, 50, 45, 40, 35, 30, 25, 20, 15, 10, 5,␣
 ↪0]
plot(x, i, color='orange', label='Expo')

i = [4375, 3959, 3541, 3958, 2834, 2750, 2416, 2166, 2000, 2117, 2000, 1958,␣
 ↪1708, 1250, 1125, 833, 750, 625, 592, 458]
plot(x, i, color='red', label='Itterative')
plt.title("Temps emis par chaque algo en fonction de la taille de l'entrée")
plt.xlabel("Temps emis par chaque algorithme")
plt.ylabel("la taille de l'entrée")
legend()
show()
```



[17]:
```
import matplotlib.pyplot as plt
from matplotlib.pyplot import plot, show, legend

y = [354751666, 334751666, 30135625, 2732125, 246709, 22125, 2167, 333, 209,␣
 ↪37417]
x = [45, 40, 35, 30, 25, 20, 15, 10, 5, 0]
plot(x, y, color='green', label='Recursive')

plt.xlabel("Temps emis par algorithme Recursive")
plt.ylabel("la taille de l'entrée")
plt.title("Temps emis parl'algorithme Recursive en fonction de la taille de␣
 ↪l'entrée")
legend()
show()
```


Temps emis parl'algorithme Recursive en fonction de la taille de l'entrée
1e8
Recursive
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
la taille de l'entrée
0
10
20
30
40
Temps emis par algorithme Recursive