ISM MPSI/PCSI, Informatique
Année 2023/2024

# Corrigé du TP Informatique 17

## Exercice 1

1. On saisit :

```
def fibo1(n):
    a=(1+np.sqrt(5))/2
    b=a**n
    if n%2==0:
        s=1
    else:
        s=-1
    return 1/np.sqrt(5)*(b-s/b)
```

2. On obtient :

```
>>> [fibo1(n) for n in range(10)]
[0.0, 1.0, 1.0, 2.0, 3.0000000000000004, 5.0000000000000009,
8.0000000000000018, 13.000000000000002, 21.000000000000004, 34.000000000000007]
```

Il s'agit d'un calcul flottant avec les approximations inhérentes à ce format. Si l'argument $n$ est trop grand, le résultat est inutilisable du fait des limitations du format flottant :

```
>>> fibo1(2000)

Warning (from warnings module):
  File "D:\Drive\ITC\INT\EX004.py", line 18
    b=a**n
RuntimeWarning: overflow encountered in double_scalars
inf
```

3. On saisit :

```
def expo(x,n):
    res,a,e=1,x,n
    while e>0:
        if e%2==1:
            res*=a
        e//=2
        a*=a
    return res
```

4. On saisit :

```
def fibo2(n):
    a=(1+np.sqrt(5))/2
    if n%2==0:
        s=1
    else:
        s=-1
    b=expo(a,n)
    return 1/np.sqrt(5)*(b-s/b)
```

5. On saisit :

```
def fibo3(n):
    if n==0:
        return 0
    u,v=0,1
    for k in range(2,n+1):
        u,v=v,u+v
    return v
```

6. On saisit :

```
while abs(fibo1(n)-fibo3(n))<1:
    n+=1
print("Seuil=",n)
```

On trouve un seuil égal à 72.

7. On observe :

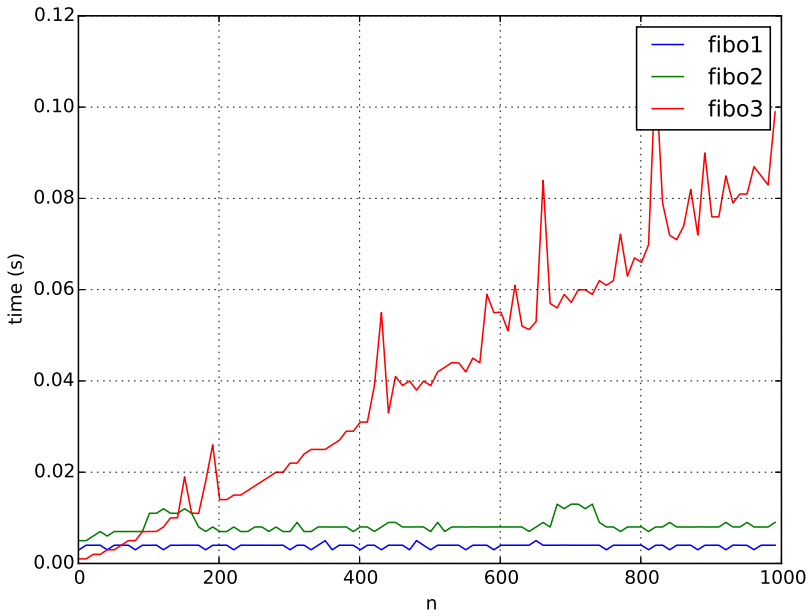


FIGURE 1 – Tracé des temps d'exécution du calcul de $(u_n)_n$

Le calcul avec l'exponentiation native ** est le plus rapide. On peut imaginer qu'il bénéficie d'une implémentation optimisée. Le calcul par exponentiation rapide est très performant également. Le calcul itératif exact avec des valeurs entières suit une tendance linéaire ce qui est cohérent avec sa complexité temporelle en $O(n)$. Ces écarts de performance ne doivent pas faire oublier que le calcul flottant n'est pas du calcul exact donc les fonctions ne sont pas réellement comparables ...

## Exercice 2

1. On saisit :

```
def expo(X,n):
    res=np.array([[1,0],[0,1]],dtype=object)
    a,e=X,n
    while e>0:
        if e%2==1:
            res=np.dot(res,a)
        e//=2
        a=np.dot(a,a)
    return res
```

2. On saisit :

```
def fibo4(n):
    A=np.array([[0,1],[1,1]],dtype=object)
    res=expo(A,n)
    return res[0,1]
```

3. On observe :

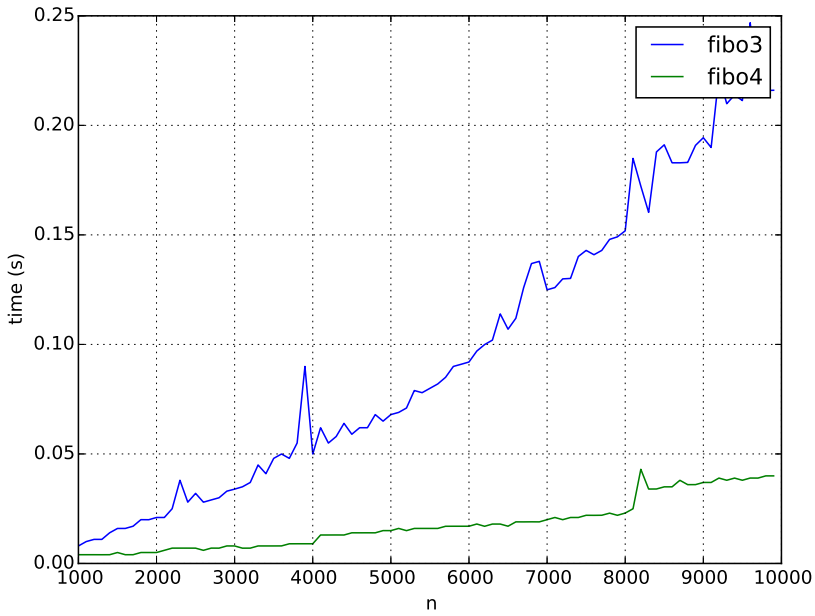


FIGURE 2 – Tracé des temps d'exécution du calcul de $(u_n)_n$

Le choix de l'exponentiation rapide pour le calcul de $A^n$ est sans conteste bénéfique. Le calcul de complexité de `fibo4` s'avère délicat car faire une hypothèse de coût constant des opérations arithmétiques pour de grandes valeurs de $n$ est véritablement trop simpliste ici. On constate d'ailleurs que l'annonce d'une complexité en $O(n)$ pour `fibo3` trouve déjà ses limites avec une tendance graphique qui ne semble plus réellement linéaire ...