Informatique en CPGE (2016-2017)
Corrigé TP 8 : algorithmes, validité et complexité

Exercice 1

A priori, le calcul est plus simple pour 2^i que pour 1.01^i. Mais les entiers sont illimités dans Python donc, pour les calculs de 2^i, une grande partie du temps va être utilisée pour stocker en mémoire les entiers qui dépassent 64 bits et calculer avec, alors que pour le calcul de 1.01^i, Python utilisera des valeurs approchées et calculera très rapidement.

Quand on calcule cent millions de fois 2^{950} ou $1,9^{950}$, on constate également la différence de rapidité. Mais **attention**, ici la majeure partie du temps n'est pas prise par le calcul des puissances mais par la création avec "range" de la liste de 100 000 000 d'entiers. Il suffit pour s'en convaincre de tester le programme qui suit et on peut tenir compte de ce temps pour apprécier la différence réelle entre les temps de calcul des puissances.

```
from time import time

st=time()
for i in range(100000000):
    c=1
print(time()-st)
```

Exercice 2

Avec l'algorithme "naïf", on effectue n multiplications.

Le second algorithme utilise l'écriture binaire de n soit $n = a_k 2^k + a_{k-1}2^{k-1} + \ldots + a_1 2^1 + a_0 2^0$.

On obtient alors : $b^n = b^{a_k 2^k + a_{k-1}2^{k-1} + \ldots + a_1 2^1 + a_0 2^0} = b^{a_k 2^k} b^{a_{k-1}2^{k-1}} \ldots b^{a_1 2^1} b^{a_0 2^0}$
soit :

$$b^n = \left(b^{2^k}\right)^{a_k} \left(b^{2^{k-1}}\right)^{a_{k-1}} \ldots \left(b^{2^1}\right)^{a_1} \left(b^{2^0}\right)^{a_0}$$

Avec $b = 3$, et $n = 13$, soit 1101 en binaire, on obtient :

	initialisation	passage 1	passage 2	passage 3	passage 4
r	0	1	0	1	1
p	1	3	3	243	1594323
b	3	9	81	6561	43046721
n	13	6	3	1	0
$p * b^n$	1594323	1594323	1594323	1594323	1594323

Terminaison : la suite des valeurs de n est une suite d'entiers positifs strictement décroissante qui atteint 0 en un nombre fini k d'itérations (le nombre de chiffres dans l'écriture binaire de n).

Validité : l'invariant de boucle est pb^n. A l'initialisation, $pb^n = 1.b^n = b^n$. On montre que pb^n reste invariant lors d'un passage dans la boucle : $pb^n = pb^{2q+r} = pb^r b^{2q} = (pb^r)(b^2)^q = p'b'^{n'}$.

A la fin, $n = 0$ donc $pb^n = p$.

Complexité : à chaque passage dans la boucle, 5 opérations sont effectuées, et il y a k passages dans la boucle, avec k = int$(1 + \ln n / \ln 2)$, d'où un niveau de complexité en $\mathcal{O}(\ln(n))$.

Exercice 3

1. Avec l'algorithme "naïf", on effectue, pour chaque valeur de i : i multiplications dans la boucle "for j in range(1, i+1) ", puis 1 multiplication et 1 addition, soit i+2 opérations. Au total le nombre d'opérations est donc $\sum_{i=0}^{n}(i+2) = \sum_{i=0}^{n} i + \sum_{i=0}^{n} 2 = \frac{n(n+1)}{2} + 2(n+1) = \frac{n^2}{2} + \frac{5n}{2} + 2$.
   Le niveau de complexité est donc en $\mathcal{O}(n^2)$.

2. Le programme correspondant à l'*algorithme de Hörner*, est le suivant :

```
p=a[n]
for i in range(n-1,-1,-1):
    p=p*x+a[i]
print(p)
```

   A chaque passage dans la boucle on effectue une multiplication et une addition donc au total $2n$ opération. Le niveau de complexité est en $\mathcal{O}(n)$.
   L'algorithme le plus efficace est donc l'*algorithme de Hörner*.

   Pour 100000 valeurs et $n = 10$: 2,68s avec l'algo naïf et 0,49s avec Hörner.
   Pour 100000 valeurs et $n = 20$: 7,97s avec l'algo naïf et 1,00s avec Hörner.