
RÉVISIONS Python 3 : Terminaison, correction, complexité

I] Terminaison et correction

Quand on écrit un algorithme on doit, en général, démontrer qu'il se termine en un nombre fini d'étapes. Il n'existe pas d'algorithme qui permet de déterminer si un algorithme se termine, d'ailleurs il existe des algorithmes où on ne sait pas (encore ?) le faire, par exemple la conjecture de Syracuse (soit u_0 un entier strictement positif, on construit la suite (u_n) par récurrence sur n , si u_n est pair alors on pose $u_{n+1} = \frac{u_n}{2}$, si u_n est impair alors on pose $u_{n+1} = 3u_n + 1$, la conjecture dit que u_n vaudra la valeur 1 en un nombre fini d'étapes).

S'il n'y a pas de boucle (et pas de récursivité) la terminaison est automatique, pour la correction, souvent une table des états suffit. S'il y a une boucle `while` ou si l'algorithme est récursif il y a plus de travail. Pour montrer la terminaison on détermine un variant de boucle. Tandis que pour montrer qu'un algorithme fait bien ce qu'on lui demande on utilise un invariant de boucle (et une preuve qui ressemble à une récurrence).

Comme exemple on va utiliser l'algorithme de division euclidienne :

```
def divE(a,b):
    """ a est un entier positif, b un entier strictement positif
    la fonction renvoie le couple (q,r) formé du quotient et du reste de
    la division euclidienne de a par b """
    r=a
    q=0
    while r>=b:
        r=r-b
        q=q+1
    return (q,r)
```

Terminaison : Ici on va utiliser $r - b$ comme variant de boucle, en effet la boucle s'arrête dès que cette quantité est négative ou nulle, notons $r_0 = a$ et r_j la valeur de r après le j -ième passage dans la boucle, comme $r_{j+1} = r_j - b$ et comme $b > 0$, le variant de boucle est une suite strictement décroissante d'entiers, elle est donc nécessairement finie (ie on quitte la boucle après un nombre fini d'étapes)

Correction : On utilise comme invariant de boucle l'expression $a = bq + r$. Notons q_0 et r_0 les valeurs de q et r avant passage dans la boucle, et q_n et r_n les valeurs après le n -ième passage.

Initialisation : on a $r_0 = a$ et $q_0 = 0$, ainsi, on a bien $bq_0 + r_0 = 0 + a = a$, l'invariant est bien initialisé.

Hérédité : On suppose $a = bq_n + r_n$ et qu'on a un $n+1$ -ième passage dans la boucle, on aura alors : $r_{n+1} = r_n - b$ et $q_{n+1} = q_n + 1$, ainsi $bq_{n+1} + r_{n+1} = b(q_n + 1) + r_n - b = bq_n + r_n = a$. Ainsi l'invariant est héréditaire. Ainsi en sortie de boucle on a $a = bq + r$, de plus, comme on est sorti de la boucle, on a $0 \leq r < b$, ainsi en sortie de boucle q et r contiennent bien le quotient et le reste de la division euclidienne de a par b .

Q1. Montrer la terminaison et la correction de l'algorithme suivant qui détermine la plus petite puissance de 2 supérieur ou égal à un entier n donné en paramètre.

```
def puis2(n):
    res=1
    while res<n:
        res=2*res
    return res
```

Q2. Exponentiation rapide . On considère la fonction suivante qui permet de calculer a^n , justifier sa terminaison et justifier qu'il fait bien ce qu'on lui demande.

```
def puisrap(a,n):
    A,N,R=a,n,1
    while N>0:
        if N%2==0:
            A=A**2
            N=N//2
        else:
            R=R*A
            N=N-1
    return R
```

II] Complexité

Si la donnée d'entrée d'un algorithme dépend d'un entier n (qui peut être le nombre d'éléments de la liste donnée en paramètre) on peut mesurer le nombre d'opération élémentaire C_n (ou autre) effectuées en fonction de n , on parle alors de complexité (ou de complexité temporelle), et on l'exprime souvent sous la forme d'un grand O, on parle de complexité constante si on est un $O(1)$ (ie indépendant de n), logarithmique si on est un $O(\ln(n))$, linéaire pour un $O(n)$, quasi linéaire pour un $O(n \ln(n))$, quadratique pour un $O(n^2)$, polynômiale s'il existe un β tel qu'on soit un $O(n^\beta)$ et exponentielle s'il existe un a tel qu'on soit un $O(a^n)$.

Remarque : Si on considère un temps de calcul de 1 ms pour $n = 1$, alors pour $n = 100$ on est à 10s pour un algorithme quadratique et à 4×10^{19} ans (sic) pour un algorithme en $O(2^n)$, ainsi un tel algorithme n'est pas praticable (un algorithme quadratique est à 1 journée pour $N = 10^4$).

Remarque : Dans certain cas le nombre d'opérations dépend des données, souvent on va donner la complexité dans le pire des cas.

Il existe aussi une notion de complexité spatiale (ou en espace) si on considère l'occupation mémoire.

Q3. Recherche d'un élément dans une liste

- (a) On considère le programme suivant de recherche d'un élément dans une liste :

```
def recherche(L, x):
    for i in range(len(L)):
        if x==L[i]:
            return True
    return False
```

Déterminer la complexité de cet algorithme dans le pire des cas.

- (b) On considère maintenant l'algorithme de recherche d'un élément dans une liste triée, par dichotomie :

```
def dichotomie(L, x):
    a=0
    b=len(L) -1
    while b-a >=0:
        c=(a+b)//2
        if x==L[c]:
            return True
        elif x<L[c]:
            b=c
        else:
            a=c
    return False
```

Déterminer la complexité de cet algorithme (dans le pire des cas).

Q4. Donner la complexité des algorithmes mis en jeux dans les exercices précédent.

Q5. Écrire une fonction qui détermine si deux listes de même taille sont identiques et donner la complexité dans le pire des cas.

Q6. Déterminer le nombre de multiplications (en fonction de n) de chacun de ces algorithmes.

```
def npow(x,n):
    l=[]
    for k in range(1,n+1):
        p=x**k
        l.append(p)
    return l
def fpow(x,n):
    l=[]
    p=1
    for k in range(1,n+1):
        p=x*p
        l.append(p)
    return l
```

Q7. Donner un exemple d'algorithme (juste le nom) dont la complexité est en temps constant, logarithmique, linéaire, quasi linéaire, quadratique, polynômiale et exponentielle.