Analyse d'un algorithme

Position du Problème

Trois questions vont attirer notre attention.

- Lorsqu'on utilise une boucle itérative conditionnelle (tant que), il faut s'assurer que l'algorithme ne va pas nous entrainer dans une suite d'instructions qui ne s'arrête pas : c'est le problème de la **terminaison**.
- Il faut s'assurer que l'algorithme va faire exactement ce qu'on attend de lui : c'est le problème de la **correction**.
- Il faut maitriser la quantité de calcul que va nous demander un algorithme, notamment en vue d'améliorer la solution au problème posé : c'est le problème de la **complexité**.

1

Terminaison

Terminaison

Que fait l'algorithme suivant ?

```
k=0
while  k>=0:
            k+=1
            print(k)
```

Par nature l'exécution d'un algorithme doit conduire à la réalisation d'un nombre fini d'instructions.
Il est clair que les algorithmes que vous pouvez écrire ne contiennent qu'un nombre fini de lignes d'instructions mais l'écriture d'une instruction itérative peut conduire à lancer l'exécution d'une séquence d'instructions qui ne se termine pas. Il faut veiller à ce que cela ne se produise pas : c'est l'étude de la **terminaison** d'un algorithme.

Position du problème

Une boucle **for** se finit lorsque l'indice a atteint la valeur finale ; si celle-ci est convenablement spécifiée, il n'y a pas de risque de boucle infinie.

Lorsque l'itération est rédigée avec une boucle **while**, le passage successif dans les instructions ne s'arrête pas si la condition qui suit le **while** ne se réalise jamais.
C'est là que se situe donc le travail pour justifier la terminaison.

Variant de boucle

Pour vérifier la terminaison d'un algorithme on utilise souvent la propriété mathématique suivante :

> Une suite de nombres entiers strictement décroissante admet, à partir d'un certain rang, des valeurs négatives ou nulles.

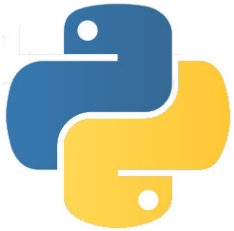
> Pour vérifier la terminaison d'une structure itérative **while**, on identifie un **variant de boucle** qui :
>
> - prend des valeurs entières positives,
> - diminue strictement à chaque passage dans la boucle.

Exemple 1 : Compte à rebours

```
k=10
while k>0:
        k=k-1
        if k>0:
                print(k)
        else:
                print('GO!')
```

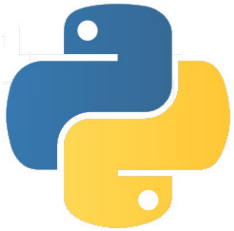
Exemple 2 : Compter sur ses doigts

```
k=0
while k<10:
        k=k+1
        print(k)
```



Exemple 3 : Fonction maxiwhile

```
def maxiwhile(uneliste):
    """
    Renvoie la plus grande valeur
    presente dans une liste
    Entree: uneliste (type list)
    Sortie: m (type int ou float)
    """

    m=uneliste[0]
    i=0
    while i<len(uneliste):
        if uneliste[i]>m:
            m=uneliste[i]
    return m
```



Attention, cette technique n'est pas généralisable à tous les algorithmes. La terminaison d'un algorithme est un problème indécidable : il n'existe pas de procédure permettant de déterminer si n'importe quel algorithme termine ou non.

```
def syracuse(n):
    syr=n
    compteur=0
    while syr!=1:
        if syr%2==0:
            syr=syr//2
        else:
            syr=3*syr+1
        compteur+=1
    return compteur
```

Sorties de boucle

L'instruction **break** permet de forcer une sortie de boucle.

```
k=0
while k>=0:
        k+=1
        print(k)
        if k>100:
                print('STOP!')
                break
```

Attention :

- A utiliser avec modération
- Ne pas confondre avec l'arrêt complet d'une fonction avec **return**

2

Correction

Invariant de boucle

Dans cette partie on étudie une méthode pour justifier qu'un algorithme est correct : c'est à dire qu'on cherche à justifier que l'algorithme fait exactement ce qu'on attend de lui.
Précisément, on se propose de justifier que nos structures itératives (boucles **for** et **while**) retournent à la fin de leur exécution les valeurs souhaitées.

On cherche pour cela un **invariant de boucle** :

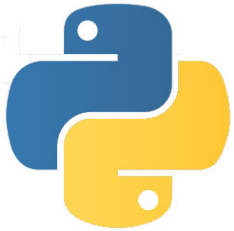
Un invariant de boucle est une propriété qui, si elle vraie juste avant un tour de boucle, reste vraie après son exécution.

En particulier : si un invariant de boucle est vrai avant le premier passage dans la structure itérative, il est encore vrai lorsque celle-ci prend fin.

Le choix d'un invariant de boucle convenable permet donc souvent de connaître l'état des variables après le passage dans une structure itérative et de justifier que l'algorithme retourne des valeurs correctes.
Étant donné une propriété P, il s'agit donc de vérifier :

1. **Initialisation :** La propriété P est vraie avant la première itération.
2. **Hérédité :** Si P est vraie avant une itération alors P est vraie après cette itération.
3. **Terminaison :** P est vraie si la boucle se termine.

Si P est bien choisit : elle fournit une propriété qui aide à conclure sur la correction de l'algorithme.



Exemples


```
def somme(valeurs):
        """Renvoie la somme d'une liste
        de valeurs
        Entree: valeurs (type list)
        Sortie: s (type float) somme des
        valeurs
        """

        s=0
        for i in range(len(valeurs)):
                s+=valeurs[i]
        return s
```

Montrer que la propriété $s = \sum_{k=0}^{i} \text{valeur[k]}$ est un invariant de boucle et en déduire que l'algorithme est correct.


```
def maxi(uneliste):
    """
    Renvoie la plus grande valeur
    presente dans une liste
    Entree: uneliste (type list)
    Sortie: m (type int ou float)
    """

    m=uneliste[0]
    for i in range(len(uneliste)):
        if uneliste[i]>m:
            m=uneliste[i]
    return m
```

Montrer que l'algorithme est correct.

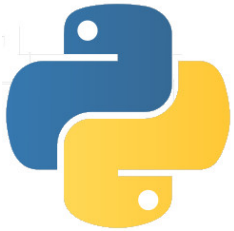
3

Complexité

Position du problème

Dans de nombreux contextes de la vie courante les programmes informatiques peuvent être amenés à opérer sur un grand nombre d'informations. Même si les algorithmes conçus se terminent et sont corrects, ils doivent donner les réponses dans un temps raisonnable (parfois très court pour être satisfaisants). De même, deux solutions différentes peuvent résoudre un même problème : il s'agit de trouver celle qui sera la plus efficace.

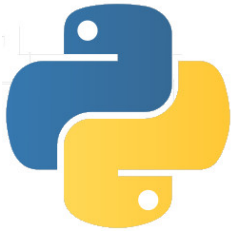
Dans cette, nous abordons quelques méthodes d'analyse qui nous permettront de comparer et d'estimer la performance de nos algorithmes : on parle d'étude de **complexité**.



Complexité en espace

On considère un litre de gaz à dans les conditions normales de température et de pression ($p = 1\,\text{bar}$ et $T = 298\,\text{K}$). On souhaite modéliser le mouvement des particules qui constituent ce système gazeux. Pour cela on doit disposer de la position de chaque particule à t=0 (repérées par x_i, y_i, z_i), de la vitesse de chaque particule à t=0 (notées par v_{x_i}, v_{y_i}, v_{z_i}) et d'un modèle (par exemple le modèle des gaz parfaits). Chaque donnée numérique est entrée comme un flottant codé sur 64 bits

Quel est l'espace mémoire nécessaire (exprimé en octets) pour stocker les valeurs numériques associées aux positions et aux vitesses des particules ?



Complexité en temps

Lancer le script harmonique_complexite.py avec une durée d'observation de 10 s. Ce script réalise une résolution numérique de l'équation d'un oscillateur harmonique par la méthode d'Euler.

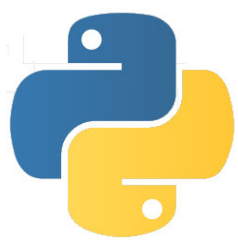
Commenter le résultat obtenu

Comment estimer la complexité d'un algorithme ?

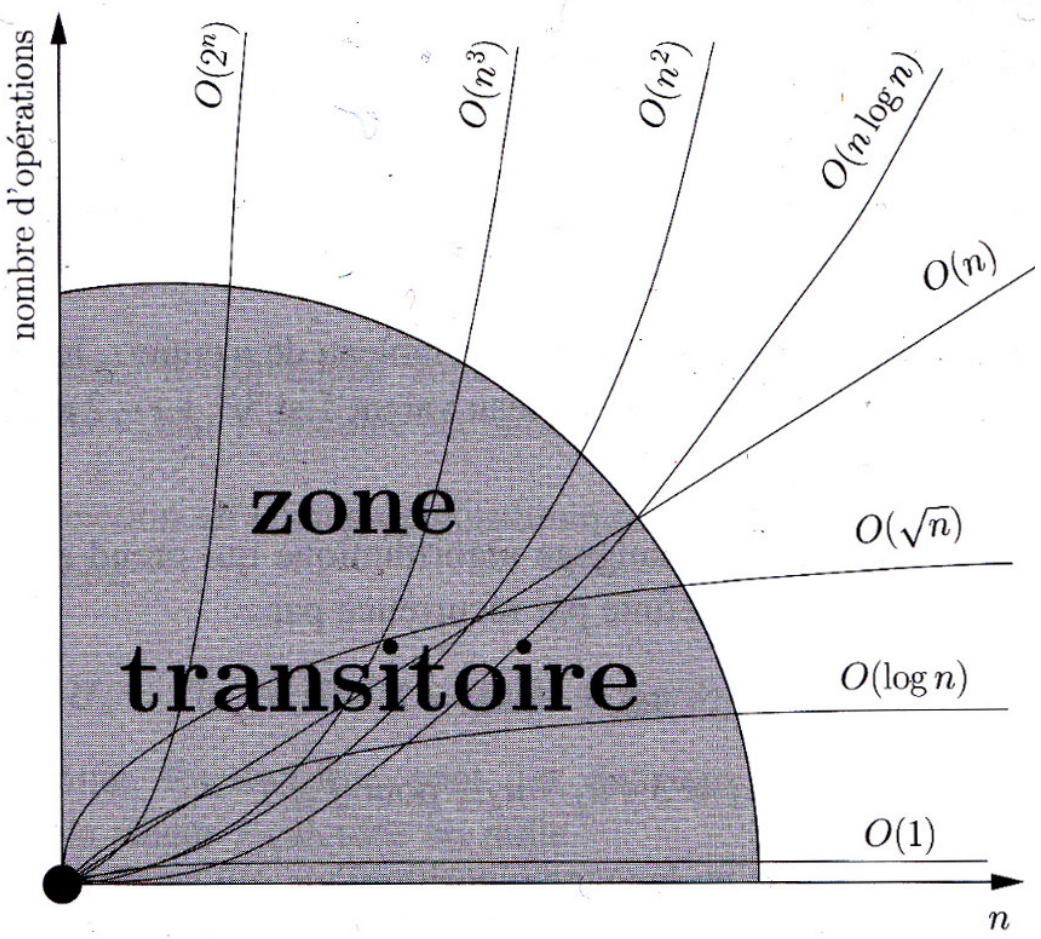
- Solution 1 : compter le nombre d'opérations élémentaires à effectuer. (on suppose alors souvent que toutes les opérations élémentaires ont le même temps d'exécution)
- Solution 2 : Identifier comment le nombre N de données apparait dans les boucles itératives de l'algorithme.

Indiquer le résultat affiché par les trois algorithmes suivants pour n=3 puis estimer leur complexité.


```
n=eval(input("Donner un entier: "))

#algo1
for i in range(11):
        print(i*n)

#algo2
for j in range(n):
        print(j*j)

#algo3
for k in range(n):
        for p in range(n):
                print(k*p, end=" ")
        print()
```



Les principales complexités temporelles



Recherche d'un élément dans une liste


```
def recherche(item, uneliste):
        """documentation"""
        resultat=False
        for i in range(len(uneliste)):
                if uneliste[i]==item:
                        resultat=i
        return resultat
```

- Rédiger la documentation pour cette fonction.
- Montrer que l'algorithme est correct.
- Quelle est la complexité de cet algorithme ?
- Comment améliorer l'algorithme `recherche` pour le rendre plus performant ?