1 Boucles imbriquées

1.1 Principe

• Les programmes que nous avons rencontrés jusqu'à présent contenaient des boucles simples mais il est tout à fait possible de placer des boucles *while* ou *for* à l'intérieur d'une boucle *while* ou *for*. On parle alors de boucles imbriquées.

```
for i in range(3):
    for j in range(4):
        print(i * j)
```

• On fera attention à l'indentation et à donner un nom différent aux variables des boucles. Dans cet exemple, i prend d'abord la valeur 0 et j prend les valeurs de 0 à 3, puis i prend la valeur 1 et j prend les valeurs de 0 à 3, enfin i prend la valeur 2 et j prend les valeurs de 0 à 3. Ce qui nous donne comme résultat :

```
0
0
0
0
0
1
2
3
0
2
4
6
```

• Ce principe peut être utilisé au sein d'une liste en compréhension :

```
>>>L = [i + j for i in range(4) for j in range(3)]
>>>L
[0, 1, 2, 1, 2, 3, 2, 3, 4, 3, 4, 5]
>>>M = [[x, y] for x in range(5) for y in range(3) if x != y] # on peut ajouter une condition
>>>M
[[0, 1], [0, 2], [1, 0], [1, 2], [2, 0], [2, 1], [3, 0], [3, 1], [3, 2], [4, 0], [4, 1], [4, 2]]
```

1.2 Application au tri bulle

• On se donne une liste de nombres que l'on souhaite trier dans l'ordre croissant. C'est un problème algorithmique important et il existe de nombreuses façons de trier une liste, nous les verrons tout au long de l'année. Le tri bulle va utiliser des boucles imbriquées, c'est la raison pour laquelle nous l'étudions dans ce cours.

• Le principe du **tri bulle** est le suivant :

- ▶ On parcourt la liste en comparant deux éléments consécutifs, s'ils ne sont pas rangés dans le bon ordre, on les échange.
- ▶ Si au moins un échange a eu lieu lors de ce parcours, on recommence l'opération en repartant au début de la liste.
- ▶ Si aucun échange n'a eu lieu lors de ce parcours, c'est que la liste est triée dans l'ordre et on stoppe l'algorithme.

Exemple . On part de la liste $L = [4,\ 7,\ 3,\ 9,\ 5]$ que l'on souhaite trier dans l'ordre croissant.

- **Parcours 1**. On compare 4 et 7, ils sont bien rangés, on ne change rien : $L = [4,\ 7,\ 3,\ 9,\ 5]$.
  On compare 7 et 3 et on les échange : $L = [4,\ 3,\ 7,\ 9,\ 5]$.
  On compare 7 et 9 et on ne change rien : $L = [4,\ 3,\ 7,\ 9,\ 5]$.
  On compare 9 et 5 et on les échange : $L = [4,\ 3,\ 7,\ 5,\ 9]$.
- **Parcours 2**. On compare 4 et 3, on les échange : $L = [3,\ 4,\ 7,\ 3,\ 5,\ 9]$.
  On compare 4 et 7 et on ne change rien : $L = [3,\ 4,\ 7,\ 5,\ 9]$.
  On compare 7 et 5 et on les échange : $L = [3,\ 4,\ 5,\ 7,\ 9]$.
  On compare 7 et 9 et on ne change rien : $L = [3,\ 4,\ 5,\ 7,\ 9]$.
- **Parcours 3**. On compare 3 et 4 et on ne change rien : $L = [3,\ 4,\ 5,\ 7,\ 9]$.
  On compare 4 et 5 et on ne change rien : $L = [3,\ 4,\ 5,\ 7,\ 9]$.
  On compare 5 et 7 et on ne change rien : $L = [3,\ 4,\ 5,\ 7,\ 9]$.
  On compare 7 et 9 et on ne change rien : $L = [3,\ 4,\ 5,\ 7,\ 9]$.

Lors de ce troisième parcours, nous n'avons effectué aucun échange, ainsi la liste est triée et on stoppe l'algorithme.

Voici l'algorithme écrit en Python, il est à bien comprendre.

```
def tri_bulles (L):
    """ordonne la liste L dans l'ordre croissant en utilisant le tri bulle"""
    n =len(L) # la longueur de la liste
    trier = False # cette variable prend la valeur True lorsque l'on fait un parcours sans échange
    while trier == False:
        trier = True # pour l'instant, on n'a pas fait d'échange
        for i in range(n - 1): # on parcourt la liste
            if L[i] > L[i + 1]:
                L[i], L[i +1] = L[i + 1], L[i] # on échange les paires mal rangées
                trier = False # on indique que la liste n'est pas triée car on a fait un échange
    return(L) # quand on sort de la boucle, la liste est triée
```

- Ce principe de tri est classique et est à connaître, mais nous verrons que ce n'est pas la méthode la plus efficace.

2 Notion de complexité d'un algorithme.

- Pour améliorer l'efficacité d'un algorithme, il y a deux pistes possibles : réduire le temps de calcul et réduire les ressources mémoires utilisées. Dans ce paragraphe, nous allons nous intéresser uniquement à la complexité en temps.

- Donner le temps en secondes qu'un algorithme met pour s'exécuter n'a pas réellement de sens car cela dépend fortement de l'ordinateur utilisé et des ressources mémoires allouées au programme. Nous allons plutôt compter le nombre d'opérations qu'effectue l'algorithme pour estimer son efficacité.

2.1 Exemple de la recherche des diviseurs

- On cherche à afficher la liste des diviseurs d'un entier naturel n. On peut écrire l'algorithme naïf suivant :

```
def diviseurs(n):
""" affiche les diviseurs positifs de n"""
    for i in range(1, n + 1):
        if n % i == 0: #test de la divisibilité par i
            print(i)
```

• S'intéresser à la complexité de cet algorithme revient à compter le nombre d'opérations qu'il effectue. Cet algorithme calcule n restes de divisions euclidiennes, effectue n comparaisons et au plus n affichages. Empiriquement, c'est la division euclidienne qui est la plus coûteuse en temps, **il y a n divisions euclidiennes.**

• On peut améliorer cet algorithme si l'on se souvient que si $n = pq$ avec $p \geq \sqrt{n}$ alors $q \leq \sqrt{n}$. Il suffit donc de chercher chaque diviseur p inférieur ou égal à $\sqrt{n}$ et d'afficher également $q = n/p$. Voici une amélioration du programme précédent qui tient compte de cette remarque :

```
import math as m

def diviseurs2(n):
"""affiche les diviseurs positifs de n"""
    for i in range(1, int(m.sqrt(n)) + 1):
        if n % i == 0: #test de la divisibilité par i
            print(i)
            print(n // i)
```

• Lors de l'exécution de cette fonction, **il y a $\lfloor\sqrt{n}\rfloor$ (partie entière) divisions euclidiennes effectuées.**

Conclusion : si l'on néglige les opérations de comparaison et d'affichage et que l'on suppose que le temps de calcul pour effectuer une division euclidienne est t alors le temps d'exécution de l'algorithme 1 est tn et celui de l'algorithme 2 est $t\sqrt{n}$. Cela peut représenter un gain de temps conséquent si n est grand.

2.2 Définitions et notation

2.2.1 Taille du problème

• En général, le temps d'exécution d'un algorithme varie en fonction d'un paramètre d'entrée que l'on appelle la **taille du problème**. Dans l'exemple précédent, la taille du problème est l'entier n dont on souhaite trouver les diviseurs. En effet, on a remarqué que le temps d'exécution de l'algorithme est proportionnel à n ou à $\sqrt{n}$.

La taille du problème sera parfois précisée dans l'énoncé mais dans certains cas ce sera à vous de la choisir. Voici quelques exemples de choix possibles.

Exemples : i) Lorsque le problème dépend d'un entier naturel n, il y a deux choix logiques : l'entier lui-même ou le nombre de chiffres de n.

ii) Vous étudierez l'année prochaine des algorithmes permettant de trier des listes de nombres entiers. La taille de la donnée sera le nombre d'éléments du tableau.

iii) Si vous étudiez un programme travaillant sur un texte, il sera naturel de prendre pour taille du problème le nombre de caractères du texte.

iv) Si vous travaillez avec des matrices carrées, il sera naturel de prendre le nombre de lignes de la matrice comme taille du problème.

2.2.2 Que compter ?

• Dans l'exemple, nous avons choisi de nous concentrer uniquement sur le nombre de divisions euclidiennes effectuées et de négliger le coût des autres opérations.

• En général, nous comptons **le nombre d'opérations effectuées : additions, multiplications, comparaisons, affectations...** On peut être amené à considérer que certaines opérations sont négligeables par rapport à d'autres, la plupart du temps ce sera précisé dans l'énoncé.

Exemple : Si l'on souhaite comparer plusieurs algorithmes de tris de listes d'entiers, on pourra dénombrer les comparaisons effectuées par ces algorithmes.

2.2.3 Notation

• Dans l'exemple précédent, avec notre algorithme optimisé de recherche de diviseurs, nous avons vu que le temps de calcul est proportionnel à $t\sqrt{n}$. Le paramètre t est le temps mis par Python pour effectuer une division euclidienne, ce réel dépend bien évidemment de la machine utilisée. Ce paramètre ne change pas l'efficacité intrinsèque de notre algorithme.

Ce qui nous intéresse n'est pas un temps précis d'exécution mais un ordre de grandeur de ce temps d'exécution en fonction de la taille des données.

On dira que notre algorithme a une complexité en $O(\sqrt{n})$ (se lit grand o).

• Une dernière notion à considérer est celle du terme dominant dans le temps d'exécution d'un algorithme. Par exemple, si l'on a déterminé que le temps d'exécution est proportionnel à n^2+3n dès que la taille n devient grand, il est connu que le terme $3n$ est négligeable devant n^2. On dira que n^2+3n est un $O(n^2)$.

Remarques : i) *Toutes ces notations seront précisées dans le cours de mathématiques.*

ii) *On examine parfois la complexité dans le meilleur des cas, c'est dans dire dans le cas où les paramètres d'entrée sont les plus favorables. Cependant, c'est bien sûr la complexité dans le pire des cas qu'il faudra le plus souvent estimer.*

2.3 Récapitulatif des complexités usuelles

• Voici les ordres de grandeur des temps d'exécution que l'on rencontre en pratique pour un problème de taille $n = 10^6$ sur un ordinateur personnel effectuant 1 milliard d'opérations par seconde.

	Nom courant	Temps	Remarques
$O(1)$	temps constant	$1ns$	le temps d'exécution ne dépend pas de la taille des données : rare.
$O(log(n))$	logarithmique	$10ns$	la base du logarithme n'a pas d'importance
$O(n)$	linéaire	$1ms$	
$O(n^2)$	quadratique	$15min$	cette complexité n'est pas acceptable pour des données de taille $> 10^6$
$O(n^k)$	polynomiale	30 ans si $k = 3$	il n'est pas rare de voir des complexités en $O(n^3)$ ou $O(n^4)$
$O(2^n)$	exponentielle	10^{300000} ans	impraticable sauf pour des tailles de données très petites

2.3.1 Un exemple simple de complexité linéaire

• On considère la fonction suivante :

```
def factorielle (n):
    """renvoie la factorielle de l'entier naturel n"""
    P = 1 # on initialise le produit
    for i in range(2, n + 1):
        P = P * i # on multiplie entre eux tous les entiers de 2 à n
    return(P)
```

• On s'intéresse au nombre de multiplications en fonction de la donnée n. Si on note $C(n)$ ce nombre de multiplications, on a clairement $C(n) = n - 1$. Ainsi $C(n) = O(n)$, on dit que la complexité est linéaire.

2.3.2 Un exemple d'amélioration de la complexité

• On considère la suite définie par :

$$\begin{cases} u_0 = 2 \\ \forall n \in \mathbb{N}^*,\ u_n = \dfrac{1}{2}\left(u_{n-1} + \dfrac{3}{u_{n-1}}\right) \end{cases}$$

Il est possible de calculer les termes de cette suite à l'aide d'un algorithme récursif.

```
def u(n):
    if n == 0:
        return(2) # la valeur de u0
    else :
        return(0.5 * (u(n - 1) + 3 / u(n - 1)) # la formule de l'énoncé
```

• Dans cet exemple, nous allons mettre les additions et les multiplications sur un pied d'égalité en considérant qu'il y a trois opérations élémentaires dans le calcul de l'expression $\frac{1}{2}\left(u_{n-1}+\frac{3}{u_{n-1}}\right)$. Notons $C(n)$ le nombre d'opérations élémentaires effectuées par la fonction u avec la donnée n. Pour calculer $u(n)$, il y a deux appels à la fonction $u(n-1)$, ainsi nous avons la relation de récurrence :

$$C(n) = 3 + 2C(n-1)$$

On est en présence d'une suite arithmético-géométrique. On peut démontrer par récurrrence que pour tout $n \in \mathbb{N}$:

$$C(n) = 3(-1 + 2^n)$$

La complexité est en $O(2^n)$: **c'est une complexité exponentielle.**

Il est clair que cette façon de procéder n'est pas optimale car nous effectuons deux appels récursifs à $u(n-1)$. Voici une version améliorée :

```
1  def u(n):
2      if n == 0:
3          return(2)
4      else:
5          v = u(n - 1) # on stocke la valeur de u(n-1) pour ne pas la calculer deux fois
6          return(0.5 * (u(n - 1) + 3 / u(n - 1))
```

• Avec les mêmes notations, nous avons à présent :

$$\forall n \in \mathbb{N}^*,\ C(n) = 3 + C(n-1)$$

C'est une suite arithmétique, on trouve immédiatement $C(n) = 3n$. La complexité est en $O(n)$: **c'est une complexité linéaire !**

Remarque : La perte en espace mémoire due au stockage de $u(n-1)$ est négligeable par rapport au gain en temps.