Cours inspiré par celui de mon collègue Pierre Duclosson.
Carnet Capytale avec les exercices et leurs corrigés : https://capytale2.ac-paris.fr/web/c/894d-5867775.

Introduction
L'étude de la complexité des algorithmes s'attache à mesurer leur efficacité. Lorsqu'on s'intéresse au temps d'exécution on parle de **complexité temporelle** et lorsqu'il s'agit de la mémoire utilisée, on parle de **complexité spatiale.**

1 Méthodologie de l'évaluation de la complexité

1.1 Une approche empirique

Exercice 1 ***Nombres triangulaires***
Soit n un entier naturel, le nombre triangulaire $t(n)$ représente le nombre d'étoiles dans un triangle de n lignes avec une étoile sur la première ligne, deux sur la deuxième etc ... Ainsi, $t(1) = 1$, $t(2) = 3$, $t(3) = 6$ et $t(4) = 10$ ...

```
*
**
***
****
```

1. On considère la fonction naïve ci-dessous :

```
def triangle1(n):
    """
    Renvoie la valeur du nombre triangulaire t(n) comptant
    le nombre d'* dans le triangle de n lignes :
    *
    **
    ....
    *****

    Méthode 1 : calcule avec 2 boucles imbriquées
                on compte les * une à une !

    Parametre: n (int) : nombre de lignes >= 0

    Retour:(int)
    """
    assert n >= 0
    t = 0
    for i in range(1, n + 1):
        for j in range(1, i + 1):
            t = t + 1
```

```
    return t
```

Mesurer l'évolution du temps d'exécution avec la fonction :

```
import time

def doubling_ratio(triangle, nb_iter):
    """
    Mesure l'évolution du temps d'exécution de triangle(n)
    en doublant nb_iter fois le nombre initial n = 10
    """
    temps_preced = 0
    n = 10
    for _ in range(nb_iter):
        debut = time.perf_counter()
        triangle(n)
        temps = time.perf_counter() - debut
        if temps_preced != 0:
            ratio = temps/temps_preced
        else:
            ratio = 1
        print(f"n = {n} <-> temps (s) = {temps}
              <-> temps/temps_preced = {ratio}" )
        temps_preced = temps
        n = 2 * n

doubling_ratio(triangle1, 9)
```

Que peut-on remarquer ? Est-il raisonnable de tester `doubling_ratio(triangle1, 15)` et `doubling_ratio(triangle1, 45)` ?

.....

.....

.....

.....

2. Écrire une autre fonction `triangle2(n)` qui calcule le nombre triangulaire $t(n)$ à l'aide d'une seule boucle `for`.

Tester `doubling_ratio(triangle2, 15)`.

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

3. On peut démontrer (comment?) que $t(n) = \dfrac{n(n+1)}{2}$. En déduire une troisième Fonction `triangle3(n)`.

Tester `doubling_ratio(triangle3, 1000)`. Que peut-on remarquer?

Comparer l'efficacité des trois algorithmes implémentés par `triangle1`, `triangle2` et `triangle3` pour résoudre le problème du calcul du nombre triangulaire $t(n)$.

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

1.2 Ordre de complexité

Méthode

Il est très difficile de répondre à la question « *Quelle sera la durée d'exécution d'un programme?* », car ce temps dépend de plusieurs paramètres : certains déterministes comme le nombre d'opérations élémentaires effectuées (affectation, test, calcul ...), le langage utilisé, le jeu d'instructions du microprocesseur, la vitesse du microprocesseur voir le nombre de coeurs et d'autres non comme le contexte d'exécution. En effet le contexte change la façon dont le processeur est alloué à chaque programme en cours d'exécution, sachant que ces programmes sont en concurrence pour l'accès aux différents coeurs de calcul.
Le paramètre le moins contingent est le nombre d'opérations élémentaires effectuées. Il dépend juste de **l'algorithme** implémenté. Avec une constante qui dépend de la machine et du langage on a ($\approx$ car contexte non déterministe) :

Temps d'exécution $\approx$ Constante $\times$ Coût de l'algorithme en opérations élémentaires

Si on s'abstrait de la machine, la question de la performance d'un programme peut donc s'abstraire en « *Quel est le coût de cet algorithme pour résoudre ce problème ?* ». Plus facile à dire qu'à faire, surtout avec des instructions conditionnelles et des boucles non bornées !
Heureusement les mathématiques vont nous aider ! Par exemple, la fonction `triangle1(n)` du premier exercice nécessite $\frac{n(n+1)}{2}$ additions pour calculer le n^e nombre triangulaire. On peut remarquer que le coût de l'algorithme est exprimé en fonction de la taille de l'entrée : ici le rang n. Mais en général le coût de l'algorithme nous intéresse surtout pour des valeurs de n grandes et en particulier on aimerait exprimer l'évolution de ce coût à l'aide de fonctions mathématiques usuelles. Expérimentalement on observe que le temps d'exécution est multiplié par quatre lorsque la taille double, ce qui indique que $\frac{n(n+1)}{2}$ varie comme $k \times n^2$ avec k constante. En mathématiques, on dira que le terme dominant dans $\frac{n(n+1)}{2}$ est celui de plus haut degré $\frac{1}{2}n^2$: c'est celui qui nous intéresse pour mesurer le coût de l'algorithme. Dans les livres, on utilise plutôt le terme de **complexité** et pour cet algorithme de *complexité quadratique*.
On a vu dans l'exercice 1 que le même problème de calcul de nombre triangulaire peut se traiter avec d'autres algorithmes qui ont une meilleure complexité : *complexité linéaire* pour `triangle2` et même *complexité constante* pour `triangle3`. On vient de voir la **complexité temporelle** mais un algorithme est aussi caractérisé par son empreinte mémoire : la **complexité spatiale**.

Exercice 2 *source : Nicolas Réveret*

Un programme traite des données dont la taille peut être mesurée à l'aide d'une variable n.
Si $n = 100$, le programme retourne un résultat en 8 ns.
On admet que le temps d'exécution de ce programme évolue proportionnellement à une certaine puissance de n. Par exemple si le temps évolue proportionnellement à n^2, lorsque l'on triple la valeur de n, le temps est multiplié par $3^2 = 9$.
Compléter le tableau de durées approximatives ci-dessous :

Valeur de n	n^1	n^2	n^3
$n = 100$	8 ns	8 ns	8 ns
$n = 200$	16 ns		
$n = 300$		72 ns	
$n = 400$			512 ns
$n = 500$			
$n = 1000$			
$n = 10000$			
$n = 1000000$			

Définition 1 ***(Ordre de complexité)***
On dit qu'un algorithme est d'une complexité de l'ordre de $f(n)$ si il existe une constante positive K telle que, quelle que soit la taille n de l'entrée, le nombre d'opérations élémentaires est plus petit que $K \times f(n)$. On dit alors que l'algorithme est en $\mathcal{O}(f(n))$

En pratique, on ne rencontrera qu'un petit nombre de complexités dont on peut faire la liste de la plus petite (algorithme rapide) à la plus grande (algorithme très lent) :

☞ $\mathcal{O}(1)$: **Complexité constante.**

Le temps d'exécution est indépendant de n.

Exemples : accéder à un élément d'une liste de longueur n, ajouter un élément en fin de liste (méthode `.append()`*.*

☞ $\mathcal{O}(\ln(n))$: **Complexité logarithmique.**

Le temps d'exécution est augmenté d'une quantité constante lorsque la taille de l'entrée est doublée.

L'ordre de grandeur est celui du nombre de chiffres de taille n de l'entrée (fonction logarithme noté ln ou log, en base 2 si on considère le nombre de chiffres en binaire).

Exemples : Recherche dichotomique dans une liste triée. Algorithme d'exponentiation rapide.

☞ $\mathcal{O}(n)$: **Complexité linéaire.**

Le temps d'exécution est proportionnel à la taille n de l'entrée, ainsi il est doublé lorsque la taille est doublée.

Exemples : Calcul de la somme des éléments d'une liste. Recherche d'un élément dans une liste non triée (recherche séquentielle) dans le pire des cas (l'élément est en fin de liste).

☞ $\mathcal{O}(n\ln(n))$: **Complexité log-linéaire ou linéarithmique**

Le temps d'exécution n'est pas proportionnel à la taille de l'entrée mais la c'est à peine moins bien (n multiplié par son nombre de chiffres), on parle parfois de complexité quasi-linéaire.

Exemple : Le tri fusion.

☞ $\mathcal{O}(n^2)$: **Complexité quadratique.**

Le temps d'exécution est multiplié par 4 lorsque la taille de l'entrée est doublée. C'est le cas des algorithmes qui sont construit avec deux boucles imbriquées.

Exemples : Le tri par sélection, le tri par insertion, le tri par bulles.

☞ $\mathcal{O}(n^k)$: **Complexité polynomiale.**

Le temps d'exécution est majorée par une expression polynomiale en n. Plus k est grand plus l'algorithme sera lent.

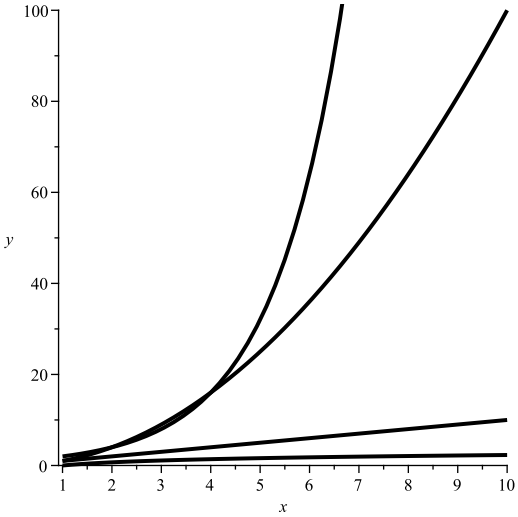
Exemple : Le calcul du produit de deux matrices de taille n est en $\mathcal{O}(n^3)$

☞ $\mathcal{O}(k^n)$: **Complexité exponentielle.**

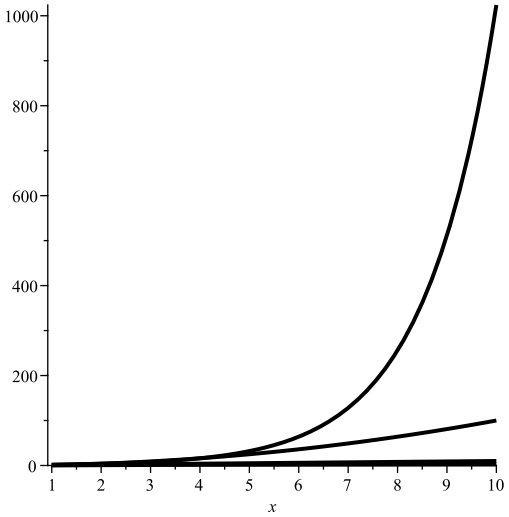
Le temps d'exécution est multiplié par une constante lorsque la taille augmente d'une unité.

Le temps d'exécution croît tellement vite que l'algorithme est impraticables sauf pour des données de petites tailles. Pour résoudre certains problèmes, on ne sait parfois pas faire mieux.

Exemple : Problème du voyageur de commerce (voir exercice 3).



graphe de $\ln(x), x, x^2, 2^x$



idem mais en changeant l'échelle

D'un point de vue pratique, pour un processeur capable d'effectuer un million d'instructions élémentaires par seconde :

	n	$n\log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n=10$	<1 s	<1 s	<1 s	<1 s	<1 s	<1 s	4 s
$n=30$	<1 s	<1 s	<1 s	<1 s	<1 s	18 m	10^{25} a
$n=50$	<1 s	<1 s	<1 s	<1 s	11 m	36 a	∞
$n=10^2$	<1 s	<1 s	<1 s	1s	12.9 a	10^{17} a	∞
$n=10^3$	<1 s	<1 s	1s	18 m	∞	∞	∞
$n=10^4$	<1 s	<1 s	2 m	12 h	∞	∞	∞
$n=10^5$	<1 s	2 s	3 h	32 a	∞	∞	∞
$n=10^6$	1s	20s	12 j	31710 a	∞	∞	∞

- Notations: ∞ = le temps dépasse 10^{25} années, s= seconde, m= minute, h = heure, a = an.

Exercice 3 *source : Nicolas Réveret*

Un « *voyageur de commerce* » doit réaliser un circuit qui passe une et une seule fois par 4 villes A, B, C et D et revienne à la ville de départ

On donne les distances entre deux villes distinctes :

	A	B	C	D
A	0	10	5	25
B	10	0	22	15
C	5	22	0	35
D	25	15	35	0

Problème d'optimisation : Déterminer le circuit le plus court passant une fois et une seule par chaque

ville.

1. Quelle est la longueur du circuit $A-B-C-D-A$? et celle de $D-A-B-C-D$? et celle de $A-D-C-B-A$?

.....

.....

.....

2. Justifier que 4×2 circuits (en comptant $A-B-C-D-A$) ont la même longueur que $A-B-C-D-A$.

.....

3. Déterminer la longueur du circuit $A-C-B-D-A$.

.....

4. Déterminer la longueur du plus court circuit.

.....

5. Si on avait 5 villes combien faudrait-il calculer de longueurs de circuits pour déterminer le plus court ? Et si on avait n villes ?

.....

.....

6. On suppose qu'un ordinateur met 1 μs à calculer la longueur d'un circuit. Combien de temps mettrait-il pour calculer toutes les longueurs de circuit nécessaires à la recherche du circuit le plus court avec 20 villes ? Un tel algorithme est-il utilisable en pratique ?

.....

.....

.....

.....

.....

2 Calculs de complexité en Python

2.1 Principes

Méthode

Opérations sur les types usuels

- **Flottants :** toutes les opérations de base sur les flottants sont en temps constant (sauf élever à une puissance entière, temps logarithmique en la puissance).
- **Entiers :** toutes les opérations de base (sauf élever à une puissance entière) sont en temps constant si les entiers ont une taille raisonnable (jusqu'à 10^{20} environ). Sinon c'est plus compliqué.
- **Listes :**
  - `t[i] = x` ou `x = t[i]` : temps constant.
  - `t.append(x)` : temps constant.
  - `t = u + v` : temps proportionnel à `len(u) + len(v)`.
  - `u = t[ : ]` (copie) : temps proportionnel à `len(t)`.
  - `u = t` : temps constant (mais ce n'est pas une copie, bien sûr).
  - `x in t` (qui renvoie `True` si l'un des éléments de `t` vaut `x`, `False` sinon) : temps proportionnel à `len(t)` (dans le pire des cas).

Boucles `for`

Le nombre d'opérations effectué au total dans une boucle `for` est la somme des nombres d'opérations à chaque itération. On veillera à bien distinguer les boucles successives des boucles imbriquées.

Boucles `while`

Le cas des boucles `while` est similaire à celui des boucles `for`, sauf qu'il est plus délicat de déterminer combien de fois on passe dans la boucle. Notez qu'on a le même problème avec une boucle `for` contenant un `return` ou un `break`.

Recommandation

Dans les questions de devoir, lorsqu'il est demandé de déterminer la complexité de l'algorithme que vous proposez, il est préférable de n'utiliser que des opérations élémentaires.

Exercice 4

On considère quatre programmes :

Programme 1

```
for i in range(20):
    print("*")
for j in range(20):
    print("*")
```

Programme 2

```
for i in range(20):
    for j in range(20):
        print("*")
```

Programme 3

```
for i in range(20):
    for j in range(i, 20):
        print("*")
```

Programme 4

```
def f(n):
    while n > 0:
        print("*")
        n = n // 2

f(2 ** 10)
```

Pour chaque programme déterminer le nombre de fois où le caractère `"*"` est affiché en faisant apparaître le calcul effectué.

1. **Programme 1** :

   .....

2. **Programme 2** :

   .....

3. **Programme 3** :

   .....

   .....

4. **Programme 4** :

   .....

   .....

Exercice 5

On donne ci-dessous les temps d'exécution d'un programme mesurés en fonction de la taille de l'entrée.

Taille de l'entrée	Temps d'exécution en secondes
1000	2
2000	16
4000	128
8000	1024

1. Quelle conjecture peut-on faire sur l'ordre de grandeur de la complexité temporelle de cet algorithme ? On exprimera la complexité en fonction de la taille n de l'entrée et on justifiera sa réponse.

.....

.....

2. Dans cette question on utilisera l'ordre de grandeur $10^3 \approx 2^{10}$.

a. Justifier que d'après le tableau précédent le temps d'exécution du programme considéré sur une entrée de taille 10^6 sera de l'ordre de 2^{31} secondes.

.....

b. L'ordre de grandeur d'une année est de 2^{25} secondes. Combien d'années faudra-t-il attendre pour que le programme s'exécute complètement sur une entrée de taille 10^6 ? On donnera le résultat arrondi à l'entier le plus proche.

.....

.....