Chapitre 6

Preuve et Complexité de programme

Le cours en questions

Nous voyons dans ce chapitre deux concepts fondamentaux d'Informatique théorique, que nous allons introduire sur des exemples simples. Elle permettent d'analyser l'efficacité d'un programme :

- La *terminaison et correction* de programme (*Preuve* de programme).
  Elle consiste à justifier rigoureusement qu'un programme s'arrête (c'est la *terminaison* de programme) et retourne bien le résultat attendu (c'est la *correction* de programme). C'est un domaine actif de recherche en Informatique. Nous en voyons quelques méthodes élémentaires dans le cas d'un programme constitué d'une seule boucle.
- La *complexité* de programme.
  Elle permet de mesurer la rapidité d'exécution d'un programme, non pas dans l'absolu (puisque cela dépendra de l'ordinateur et de son état au moment de l'exécution), mais plutôt comment évoluera le temps d'exécution en fonction de la taille de la donnée.

En application nous étudierons l'algorithme de recherche d'un élément dans une liste triée par la méthode de dichotomie, qui nous permettra d'illustrer l'importance de la notion de complexité, et de rencontrer pour la première fois l'approche de "diviser pour régner".

1 Terminaison d'une boucle

Il y a deux types de boucles :

- La boucle `for` où une variable parcourt une séquence. Elle ne pose aucun problème de terminaison à condition de respecter quelques règles simples.
- La boucle `while` qui s'exécute tant qu'une condition est satisfaite. Elle pose le problème de sa terminaison et doit être employée avec précaution.

1.1 Pourquoi privilégier quand c'est possible une boucle `for` à une boucle `while` ?

Une boucle `for` finit toujours par s'arrêter sauf si l'on fait n'importe quoi :

Il faut s'interdire de rajouter des éléments à une séquence parcourue à l'intérieur de la boucle :

```
L = [1,2,3]
for i in L:
    print(i)
    L.append(0) # Ajout de l'élément 0 en fin de liste
```

Produit le résultat suivant et boucle indéfiniment :

```
1
2
3
0
0
etc...
```

boucle sans fin. Il faut forcer l'arrêt du programme.


Il faut s'interdire au sein d'une boucle `for` d'ajouter un élément à la séquence parcourue.

Il faut éviter aussi de modifier la valeur de la variable qui parcourt. Si ça ne pose aucun problème de terminaison en `python`, ce n'est pas le cas dans d'autres langages. Et on peut toujours s'en passer.

```
for i in range(3):
    i=0
    print(i)
```

Produit le résultat suivant en `python` sans boucler. Mais ce n'est pas le cas en langage $C/C++$ par exemple.

```
0
0
0
```

On retiendra que :

- Pour une boucle `for` bien écrite le problème de la terminaison ne se pose pas.
- On s'interdit au sein d'une boucle `for` de modifier la séquence parcourue ou la variable qui parcourt.

C'est une raison pour préférer une boucle for à une boucle while lorsque c'est possible (c'est à dire lorsque le nombre d'itérations est connue à l'avance).


Une boucle `for` est préférable à une boucle `while` lorsque le nombre d'itérations de la boucle est connu à l'avance.

1.2 Terminaison d'une boucle `while`

Démontrer qu'une boucle `while` finit par s'arrêter :

```
while condition:
    ...
    corps de la boucle
    ...
```

consiste à démontrer que l'expression booléenne `condition` finit par ne plus être vérifiée (c'est à dire par valoir `False`).

1.2.1 Exemples

Confrontons les deux exemples suivants. Les boucles `while` se terminent-elles ?

Exemple 1

```
S = 0
n = 0
while (S <= 2):
    S += 1/(2**n)
    n += 1
print("n=",n,'\n',"S=",S)
```

Exemple 2

```
S = 0
n = 1
while (S <= 2):
    S += 1/n
    n +=  1
print("n=",n,'\n',"S=",S)
```

• Dans l'exemple 1, la boucle ne s'arrête par, elle boucle infiniment. En effet, après n itérations de la boucle, la variable S vaudra :

$$S=\sum_{k=0}^{n}\frac{1}{2^k}=\frac{1-\left(\frac{1}{2}\right)^{n+1}}{1-\frac{1}{2}}=2-\frac{1}{2^n}\quad\nearrow_{n\to\infty}\quad 2^-$$

et n'atteindra jamais la valeur 2. La condition demeure satisfaite.

• Dans l'exemple 2, la boucle finit par s'arrêter. En effet, après n itérations, la variable S vaut :

$$S = \sum_{k=1}^{n} \frac{1}{k} \underset{+\infty}{\sim} \ln(n) \underset{n\to\infty}{\to} +\infty$$

On peut le vérifier par :

$$\ln(n) = \int_1^n \frac{1}{t}dt = \sum_{k=1}^{n-1} \int_k^{k+1} \frac{1}{t}dt$$

La fonction $t \mapsto \frac{1}{t}$ étant décroissante sur $\mathbb{R}_+^*$, on en déduit l'encadrement :

$$\forall k \in \mathbb{N}^* \ \frac{1}{k+1} \leqslant \int_k^{k+1} \frac{1}{t}dt \leqslant \frac{1}{k} \implies \sum_{k=2}^{n} \frac{1}{k} \leqslant \ln(n) \leqslant \sum_{k=1}^{n-1} \frac{1}{k} \implies \ln(n) \leqslant \sum_{k=1}^{n} \frac{1}{k} \leqslant \ln(n)+1$$

$$\implies \boxed{\forall n \in \mathbb{N}^* \smallsetminus \{1\} : \quad 1 \leqslant \frac{1}{\ln(n)} \times \sum_{k=1}^{n} \frac{1}{k} \leqslant 1 + \frac{1}{\ln(n)}}$$

qui donné l'équivalent souhaité et montre que la condition `(S <= 2)` finit par ne plus être satisfaite. On vérifie en effet à l'exécution :

```
n= 5
S= 2.083333333333333
```

On a montré Mathématiquement que, pour tout réel A il existe un entier n tel que $\sum_{k=1}^{n} \frac{1}{k} \geqslant A$. En effet, puisque $\ln(n) \leqslant \sum_{k=1}^{n} \frac{1}{k}$, il suffit de prendre $n = \exp(A)$.

Cependant, Informatiquement le problème se complique.

```
def harmonique(A):
    S = 0
    n = 1
    while (S <= A):
        S += 1/n
        n +=  1
    print("n=",n,'\n',"S=",S)
```

Le code de la fonction précédente ne se terminera que pour des valeurs relativement petites de A. En effet, on se rappelle qu'on ne peut par représenter en flottant des nombres arbitrairement petits. Le plus petit nombre que l'on puisse représenter en flottant sur 64 bits est 2^{-1074}.

Par exemple `harmonique(1076)` tournera indéfiniment. En effet :

$$\sum_{k=1}^{n} \frac{1}{k} \leqslant \ln(n)+1 \quad \text{donc} \quad \sum_{k=1}^{n} \frac{1}{k} \geqslant 1076 \implies n \geqslant e^{1075} \geqslant 2^{1075}$$

Or pour $k > 2^{1074}$ le calcul en flottant de $\frac{1}{k}$ donnera la valeur 0. La variable S finira par être stationnaire et n'atteindra jamais la valeur 1076.

1.2.2 Montrer la terminaison d'une boucle `while` à l'aide d'un variant de boucle

Théorème 1 *Il n'existe pas de suite infinie $(u_n)_{n\in\mathbb{N}}$ d'entiers naturels qui soit strictement décroissante.*

Preuve. Par l'absurde : supposons que $(u_n)_{n\in\mathbb{N}}$ soit une suite d'entiers naturels strictement décroissante. Soit $U = \{u_n \mid n \in \mathbb{N}\}$; c'est un sous-ensemble de $\mathbb{N} \subset \mathbb{R}$ qui est minoré par 0. Il admet donc une borne inférieure l, qui est le plus grand des minorants de U. Nécessairement $l \in U$, car autrement $\lfloor l \rfloor + 1$ serait aussi un minorant de U, supérieur à l. Donc $l = \min(U)$; soit $k \in \mathbb{N}$ tel que $l = u_k$. Puisque (u_n) est strictement décroissante $u_{k+1} < u_k = l$ ce qui contredit le fait que l soit minorant de U. □

Définition : On appelle **variant de boucle** une quantité entière et positive qui décroît strictement à chaque passage dans la boucle.

• Ainsi si l'on exhibe un variant de boucle, nécessairement la boucle finit par s'arrêter. Autrement la suite des valeurs prises serait une suite d'entiers naturels strictement décroissante.


Un **variant de boucle** est une expression prenant des valeurs entières et positives et qui décroît strictement à chaque passage dans la boucle. Un variant de boucle prouve qu'une boucle finit par s'arrêter.

• Exemple 1 :

```
n = 10
while n >= 0:
    print('Bonjour')
    n -= 1
```

Ici `n` est un variant de boucle. Ses valeurs prises sont entières et positives et décroissent strictement à chaque passage dans la boucle.

• Exemple 2 :

```
n = 0
while n <= 10:
    print('Bonjour')
    n += 1
```

Ici `10-n` est un variant de boucle. Ses valeurs prises sont entières et positives et décroissent strictement à chaque passage dans la boucle.

• Exemple 3 : Terminaison de l'algorithme d'Euclide pour le calcul du $pgcd$ de deux entiers naturels.

```
def pgcd(a,b):
    while b != 0:
        a, b = b, a % b
    return a
```

On a toujours $0 \leqslant$ `a % b` $<$ `b` puisque `a % b` est le reste dans la division euclidienne de `a` par `b`.
Donc `b` est un entier positif dont la valeur décroît strictement à chaque passage dans la boucle (avec l'instruction `b = a % b`). Ainsi `b` est un variant de boucle. La boucle s'arrête et on a prouvé la terminaison de l'algorithme d'Euclide (pour `a` et `b` des entiers positifs).

2 Correction d'une boucle

Prouver qu'un programme finit par s'arrêter est une première étape. Prouver qu'il retourne le résultat escompté est une deuxième étape.

Concentrons nous sur un programme simple, constitué d'une boucle.

Prouver que le résultat calculé dans une boucle est bien le résultat attendu peut souvent se faire grâce à un *Invariant de boucle*.

Définition Pour une boucle, un **invariant de boucle** est

- une propriété qui est vraie avant l'entrée dans la boucle et qui reste vraie après chaque passage dans la boucle.
- ou une quantité numérique qui demeure inchangée après chaque passage dans la boucle.

Ainsi l'invariant (comme son nom l'indique) reste inchangé à la sortie de la boucle.

```
# Invariant I avant la boucle
boucle for ou while :
```

```
    ...
    # Invariant I inchangé après chaque passage
# => Invariant I inchangé à la sortie de  la boucle
```


Un **invariant de boucle** reste inchangé après chaque passage dans la boucle.

Dans un invariant de boucle, on peut utiliser comme paramètre le nombre `k` de passages dans la boucle. En effet, on pourrait très bien ajouter dans le corps du programme un compteur `k` indépendant qui s'incrémente à chaque passage dans la boucle et compte le nombre de passage, cela ne changerait rien à la terminaison et correction du programme.

2.1 Exemple fondateur : l'algorithme d'Euclide

Si la définition d'un invariant de boucle ne pose aucune difficulté, il n'est pas clair à priori comment l'utiliser pour prouver la correction d'un programme, c'est à dire qu'il retourne bien ce qu'on en attend. Voici un exemple fondateur.

• Algorithme d'Euclide pour le calcul du pgcd.

Nous avons déjà démontré que pour deux paramètres entiers positifs a et b, non tous deux nuls, l'algorithme d'Euclide finit par d'arrêter. Il reste à prouver qu'il retourne bien le plus grand commun diviseur de a et b.

```
def pgcd(a,b):
    while b != 0:
        a, b = b, a % b
    return a
```

Ici pour prouver la correction de l'algorithme d'Euclide, on montre que la quantité mathématique $pgcd(a,b)$ (le plus grand commun diviseur des valeurs prises par a et b durant l'exécution) reste inchangée à chaque passage dans la boucle.

Proposition 1 *Soient $(a,b) \in \mathbb{N} \times \mathbb{N}^*$, et r le reste dans la division euclidienne de a par b. Alors $pgcd(a,b) = pgcd(b,r)$.*

Preuve. Soit r le reste dans la division euclidienne de a par b, et q le quotient, alors :

• $a = q.b + r$ donc tout diviseur de b et r est aussi diviseur de a et b.
En effet : soit d une diviseur de b et r : $r = dr'$ et $b = db'$. Alors $a = qdb' + dr' = d(qb' + r')$. Donc d divise aussi a (et b).

• $r = a - q.b$ donc tout diviseur de a et b est aussi diviseur de b et r.
En effet : soit d une diviseur de a et b : $a = da'$ et $b = db'$. Alors $r = a'd - qb'd = (a' - qb')d$ donc d divise aussi r (et b).

Ainsi a, b et b, r ont mêmes diviseurs communs :

$\Longrightarrow$ En particulier a, b et b, r ont même PLUS GRAND diviseur commun : $pgcd(a,b) = pgcd(b,r)$. □

• Déduisons-en maintenant que $pgcd(a,b)$ est un invariant de boucle :
Soient a_k, b_k et r_k les valeurs de `a`,`b` et du reste `r` au k-ième passage dans la boucle.
Avec l'instruction `a, b = b, a%b`, on a les relations de récurrence : $a_{k+1} = b_k$, $b_{k+1} = r_k$ $(*)$.
Donc :

$$pgcd(a_{k+1}, b_{k+1}) \underset{(*)}{=} pgcd(b_k, r_k) \underset{Prop.1}{=} pgcd(a_k, b_k)$$

Ainsi la valeur de $pgcd(a,b)$ est invariante. C'est un **invariant de boucle**

• Déduisons-en la correction du programme :

La boucle s'arrête (déjà démontré) après n itérations, lorsque $b_n = 0$. Le programme retourne a_n qui égale $pgcd(a_n, 0) = pgcd(a_n, b_n)$.

• Puisque $pgcd(a,b)$ est un invariant de boucle, $pgcd(a,b) = pgcd(a_0, b_0) = pgcd(a_n, b_n)$ et le résultat retourné est bien le pgcd des paramètres `a` et `b`. □

On en déduit :

Théorème 2 *Avec pour paramètres deux entiers naturels a et b non tous deux nuls, l'algorithme d'Euclide retourne le plus grand commun diviseur $pgcd(a,b)$ de a et b.*

3 Complexité de programme

3.1 Introduction au concept de complexité

De nombreux systèmes informatiques manipulent des données de très grandes : Météorologie, gestion du trafic aérien, statistiques de la population, traitement d'image, serveurs, réseau sociaux, etc...

Soit N_{max} la taille maximale des données pour que l'exécution d'un programme s'exécute efficacement dans un temps d'exécution acceptable. De quelle capacité informatique faudrait-il se doter pour doubler la capacité de traitement :$2N_{max}$, ou pour la multiplier par 10 : $10N_{max}$?

Réponse : Cela dépend de comment le temps d'exécution du programme varie en fonction de la taille N des données à traiter :

- Si l'algorithme employé a un temps d'exécution approximativement linéaire en fonction de N : $a.N+b \approx a.N$, il suffit approximativement respectivement de doubler, ou de multiplier par 10 la puissance de calcul de son parc informatique.
- Si l'algorithme employé a un temps d'exécution quadratique $a.N^2+b.N+c \approx a.N^2$, il suffit de multiplier approximativement la puissance de calcul du parc informatique par 4, ou 100.

- Si l'algorithme employé a un temps d'exécution exponentiel, par exemple $a.2^N$ il faudrait élever les capacités du parc informatique au carré, ou à la puissance 10. En effet :

$$2^{N_{max}} \leqslant K_{max} \implies \quad 2^{2N_{max}} \leqslant K_{max}^2 \quad \text{et} \quad 2^{10N_{max}} \leqslant K_{max}^{10}.$$

C'est pourquoi une solution algorithme de complexité de temps de calcul linéaire en fonction de la taille des données présente de considérables avantages par rapport à une solution de complexité quadratique, pire : exponentielle, etc... Dans des proportions qui donnent le vertige.

Comparons les temps d'exécution sur un ordinateur très puissant cadencé à 10 GHz (10 milliards d'opérations à la seconde) pour divers tailles de données N et différentes fonctions dont le nombre d'opérations à exécuter au sein du microprocesseur figure dans la colonne de gauche.

	$N=20$	$N=50$	$N=100$	$N=200$
$1000.N$	0.000002s	0.000005s	0.00001s	0.00002s
$100.N^2$	0.000004s	0.000025s	0.0001s	0.0004s
$10.N^3$	0.000008s	0.000125s	0.001s	0.008s
2^N	0.0001s	1,3 jours	-	-
3^N	0.35s.	23.10^6 années	-	-
$N!$	7,7 ans	-	-	-
- : supérieur à 100 milliards d'années				

Pour un algorithme dont le temps de calcul est exponentiel en fonction de la taille de la donnée N, le temps d'exécution devient rapidement totalement impraticable lorsque N augmente, même si l'on était pourvu de super-ordinateurs des millions de fois plus puissants.

A l'inverse pour des algorithme dont le temps de calcul est approximativement linéaire, le temps d'exécution augment proportionnellement avec la taille de la donnée N.

Pour évaluer comment le temps d'exécution d'un programme évoluera en fonction de la taille N de ses données, on compte le nombre d'opérations durant son exécution dont le temps d'exécution ne dépend pas de N, que l'on exprime comme fonction de N.

Le comportement asymptotique de cette fonction (linéaire, quadratique, exponentiel,...) définit la *complexité de l'algorithme*, et donne une mesure vague de comment le temps d'exécution du programme va varier lorsque la taille de la donnée augmente.

3.2 Opérations élémentaires

Puisque la taille des données numériques est bornée par leur représentation, le temps d'exécution de l'addition de 2 nombres, de leur multiplication, etc..., sont bornés par une constante dépendant de la machine et de l'interpréteur. Il en est de même si l'on effectue n'importe quelle opération mathématiques simple, ou des opérations d'usage courant (lire une donnée, écrire une donnée, etc..). Elles prennent toutes un temps d'exécution qui est variable selon la machine mais qui ne dépassera pas une constante uniforme.

Définition. On appelle opérations élémentaires toutes les opérations consistant en :

- une comparaison, un test, une opération logique,
- une opération arithmétique, une fonction mathématique simple,
- l'affectation de variable,
- l'accès à un élément d'une structure de donnée (liste, chaine, etc..), sa modification,
- saisie/impression/retour d'une donnée à l'écran, ou dans un fichier.
- Plus généralement, toute instruction ou opération prédéfinie, basique, s'effectuant sur une donnée de taille fixée par le système, et dont le temps d'exécution est majoré par une constante ne dépendant que du système.

Par exemple :

- Lire ou modifier un élément dans une liste est une opération élémentaire.
- Le calcul de la longueur d'une liste est une opération élémentaire (l'interpréteur la connait à priori).
- Pour une séquence `seq` de longueur N, les instructions `seq.count('a')` ou `sum(seq)` ne sont pas des opérations élémentaires. Leur temps d'exécution dépend de la longueur N de `seq`.


On appelle opération élémentaire une opération basique de programmation dont le temps d'exécution est majoré par une constante ne dépendant que du système, et non de la taille de la donnée.

Informellement, pour mesurer le temps d'exécution d'un programme, on fait comme si chaque opération élémentaire prenait une seule opération au sein du micro-processeur. C'est bien sûr inexact, mais c'est suffisant pour mesurer comment le temps d'exécution va évoluer lorsque la taille des données augmente.

3.3 Fonctions de complexité d'un algorithme

Définition. Donné un algorithme (ou programme) prenant en paramètre des données de taille N variable.

• Sa **Fonction de complexité (en temps) dans le pire des cas** de l'algorithme, est l'application $F_{max} : \mathbb{N} \longrightarrow \mathbb{N}$ définie par $F_{max}(N)$ est le nombre maximum d'opérations élémentaires nécessaires pour exécuter le programme sur une entrée de taille N (celui dans le cas le plus défavorable).

• Sa **Fonction de complexité (en temps) dans le meilleur des cas** de l'algorithme, est l'application $F_{min} : \mathbb{N} \longrightarrow \mathbb{N}$ définie par $F_{min}(N)$ est le nombre minimum d'opérations élémentaires nécessaires pour exécuter le programme sur une entrée de taille N (celui dans le cas le plus favorable).

• Sa **Fonction de complexité (en temps) en moyenne** de l'algorithme, est l'application $F_{moy} : \mathbb{N} \longrightarrow \mathbb{R}$ définie par $F_{moy}(N)$ est le nombre moyens d'opérations élémentaires effectuées lorsque l'on exécute le programme sur les entrées de taille N.


Les fonctions de complexité (respectivement dans le meilleur des cas, dans le pire des cas, et en moyenne) mesurent le nombre d'opérations élémentaires nécessaires à l'exécution d'un algorithme sur une donnée de taille N (respectivement au maximum, au minimum, et en moyenne). Ce sont des fonctions de N, et elles ne prennent que des valeurs strictement positives.

Remarque On ne s'intéresse pas au détail de la fonction de complexité qui a peu de sens vu la nature des opérations élémentaires, mais à son comportement asymptotique, c'est- à-dire à sa croissance en fonction de N au voisinage de $+\infty$. Précisément on s'intéresse à l'ordre de la fonction de complexité, que nous allons définir dans le prochaine paragraphe.

3.4 Ordre d'une application de $\mathbb{N}$ dans $\mathbb{R}_+^*$

Définition.
Soient f et g deux applications de $\mathbb{N}$ dans $\mathbb{R}$. On note $f = O(g)$, et on dit que f est *dominée par* g, ou plus simplement que f est *un grand 'O' de* g si :

il existe une constante réelle $C > 0$ et $n_0 \in \mathbb{N}$ tels que pour tout $n \geqslant n_0 : |f(n)| \leqslant C \times |g(n)|$.

En particulier :

Si f/g existe pour $n >> 0$ et a une limite finie quand n tend vers $+\infty$, alors $f = O(g)$.

Lorsque $f, g : \mathbb{N} \longrightarrow \mathbb{R}_+^*$, les propositions suivantes sont équivalentes :

- $f = O(g)$,
- $\exists C > 0, \exists n_0 \in \mathbb{N}, \forall n \geqslant n_0 : f(n) \leqslant C \times g(n)$,
- f/g est une application bornée.

Proposition 2 *La relation "être dominée par" est une relation d'ordre partiel sur l'espace des fonctions de $\mathbb{N}$ dans $\mathbb{R}$. Plus précisément :*

- *elle est réflexive :* $\forall f : \mathbb{N} \longrightarrow \mathbb{R}$, $f = O(f)$,
- *elle est transitive :* $\forall f, g, h : \mathbb{N} \longrightarrow \mathbb{R}$, $f = O(g)$ *et* $g = O(h) \implies f = O(h)$.

Preuve. La réflexivité est triviale, puisqu'en prenant $C = 1$ et $n_0 = 0 : \forall n \geqslant n_0 : |f(n)| \leqslant C \times |f(n)|$.
Transitivité. Soient $C_1, C_2 > 0$ et $n_1, n_2 \in \mathbb{N}$ tels que :

$$\forall n \geqslant n_1 : \quad |f(n)| \leqslant C_1 \times |g(n)|$$
$$\forall n \geqslant n_2 : \quad |g(n)| \leqslant C_2 \times |h(n)|$$

Soit $n_0 = max(n_1, n_2)$. Alors pour tout $n \geqslant n_0$:

$$|f(n)| \leqslant C_1 |g(n)| \leqslant C1 \times C_2 \times |h(n)|$$

Et on a bien $f = O(h)$ en posant $C = C_1 \times C_2 > 0$ et $n_0 = \max(n_1, n_2)$. □

Ce n'est pas une relation d'ordre total, comme le montre l'exemple $f(n) = \cos\left(\frac{n\pi}{2}\right)$ et $g(n) = \sin\left(\frac{n\pi}{2}\right)$, dont aucune n'est dominée par l'autre.

Définition.
On note $f = \Theta(g)$, et on dit que f et g ont même ordre si $f = O(g)$ et $g = O(f)$.

Proposition 3 *Avoir même ordre est une relation d'équivalence sur l'espace des applications de $\mathbb{N}$ dans $\mathbb{R}$, c'est-à-dire une relation réflexive, symétrique et transitive.*

Preuve. La réflexivité et la transitivité découlent de la propositition 2 et de la définition.
La symétrie : $f = \Theta(g) \iff g = \Theta(f)$ découle immédiatement de la définition et de la commutativité du et logique. □

Proposition 4 *Soient $f, g : \mathbb{N} \longrightarrow \mathbb{R}$. On vérifie les propriété suivantes :*

- *Si f/g existe pour $n >> 0$ et a une limite finie non-nulle alors $f = \Theta(g)$.*
- *Si $f = o(g)$ alors $f = O(g)$ et $f \neq \Theta(g)$.*
- *Si f est un polynôme de degré d, alors $f = \Theta(n^d)$. Si $e > d$ alors $f = O(n^e)$ et $f \neq \Theta(n^e)$.*
- *Si f est un polynôme de degré $d > 0$, alors $\ln(n) = O(f)$ et $\ln(n) \neq \Theta(f)$.*
- *Si f est un polynôme et $a > 1$ alors $f = O(a^n)$ et $a^n \neq O(f)$.*
- *Pour tout $a \geqslant 0 : a^n = O(n!)$ et $n! \neq \Theta(a^n)$.*
- *$n! = O(n^n)$ et $n^n \neq \Theta(n!)$.*

Preuve (esquisse). Les propriétés s'établissent très facilement. La première découle de la définition et de la propriété : $\lim f/g$ finie $\implies f = O(g)$. Les deuxième et troisième découlent de la première, ainsi que les quatrième et cinquième en utilisant le théorème de comparaison des polynômes avec ln et exp. Les deux dernières en découlent aussi en utilisant la formule de Stirling qui donne l'équivalent $n! \underset{+\infty}{\sim} \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$. □

On a coutume de noter :

- $O(1)$ l'ensemble des applications bornées.
- $\Theta(\ln(n))$ l'ordre de $\ln(n)$.
- $\Theta(n)$ l'ordre des applications linéaires non nulles.
- $\Theta(n^2)$ l'ordre des applications polynomiales de degré 2.
- $\Theta(f)$ l'ordre de f, c.à d. la classe d'équivalence des applications de même ordre que f.

Les propriétés suivantes permettent les calculs avec les notations O et Θ :

Proposition 5 *Soient $f, g, f_1, f_2, g_1, g_2 : \mathbb{N} \longrightarrow \mathbb{R}_+^*$. On a les propriétés suivantes :*

1. $a \in \mathbb{R}$ *et* $f = O(g) \implies a.f = O(g)$
2. $a \in \mathbb{R}^*$ *et* $f = \Theta(g) \implies a.f = \Theta(g)$
3. $\left.\begin{array}{l} f_1 = O(g_1) \\ f_2 = O(g_2) \end{array}\right\} \implies f_1 + f_2 = O(g_1 + g_2)$.
4. $\left.\begin{array}{l} f_1 = \Theta(g_1) \\ f_2 = \Theta(g_2) \end{array}\right\} \implies f_1 + f_2 = \Theta(g_1 + g_2)$.
5. $\left.\begin{array}{l} f_1 = O(g_1) \\ f_2 = O(g_2) \end{array}\right\} \implies f_1 \times f_2 = O(g_1 \times g_2)$.
6. $\left.\begin{array}{l} f_1 = \Theta(g_1) \\ f_2 = \Theta(g_2) \end{array}\right\} \implies f_1 \times f_2 = \Theta(g_1 \times g_2)$.
7. *Si* $f_1 = \Theta(g)$ *et* $f_2 = O(g)$ *alors* $f_1 + f_2 = \Theta(g)$.
8. *Si* $f_2 = O(f_1)$ *alors* $f_1 + f_2 = \Theta(f_1)$.

Preuve.

1) $a \in \mathbb{R}$ et $f = O(g)$. Alors $\exists C > 0$ et $n_0 \in \mathbb{N}$ tel que $\forall\, n \geqslant 0$, $f(n) \leqslant C \times g(n)$, donc, en multipliant l'inégalité par $|a|$: $\forall\, n \geqslant 0$, $|a.f(n)| \leqslant C \times |a.g(n)|$. Donc $a.f = O(a.g)$.

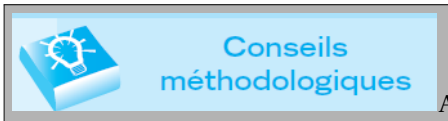
2) D'après 1), $a.f = O(g)$. Par ailleurs $g = O(f)$ donc $\exists C > 0$ et $n_0 \in \mathbb{N}$ tel que $\forall\, n \geqslant n_0 : g(n) \leqslant C \times f(n) = C/|a| \times |a.f(n)|$. Donc $g = O(a.f)$ et ainsi $a.f = \Theta(g)$.

3) et 5) . $f_1 = O(g_1)$ et $f_2 = O(g_2)$. Ainsi $\exists C_1, C_2 > 0$ et $n_1, n_2 \in \mathbb{N}$ tels que $\forall\, n \geqslant \max(n_1, n_2)$, $f_1(n) \leqslant C_1 \times g_1(n)$ et $f_2(n) \leqslant C_2 \times g_2(n)$.
En posant $C = \max(C_1, C_2)$ et $n_0 = \max(n_1, n_2)$, $\forall\, n \geqslant n_0$, $f_1(n) + f_2(n) \leqslant C \times (g_1(n) + g_2(n))$. Donc $f_1 + f_2 = O(g_1 + g_2)$.
En posant $C = C_1 \times C_2$ et $n_0 = \max(n_1, n_2)$, $\forall\, n \geqslant n_0$, $f_1(n) \times f_2(n) \leqslant C \times g_1(n) \times g_2(n)$. Donc $f_1 \times f_2 = O(g_1 \times g_2)$.

4) et 6) découlent immédiatement, respectivement, de 1) et de 3) par symétrie.

7) $f_1 = O(g)$ et $f_2 = O(g)$: alors d'après 1) et 3) $f_1 + f_2 = O(2g) = O(g)$. D'autre part $g = O(f_1)$ donc $\exists C > 0$ et $n_0 \in \mathbb{N}$ tels que $\forall\, n \in \mathbb{N} : g(n) \leqslant C \times f_1(n)$. Or $\forall\, n \in \mathbb{N}$, $f_2(n) > 0$ donc $g(n) \leqslant C \times (f_1(n) + f_2(n))$ Ainsi $f_1 + f_2 = \Theta(g)$.

8) Découle immédiatement de 7) en prenant $g = f_1$. □



Ainsi, pour des suites à valeurs strictement positive (ce qui sera toujours le cas dans les calculs de complexité) on s'autorisera les notations :
$O(f_1) + O(f_2) = O(f_1 + f_2)$: la somme d'une fonction dominée par f_1 et d'une fonction dominée par f_2 est une fonction dominée par $f_1 + f_2$.
$\Theta(f_1) + \Theta(f_2) = \Theta(f_1 + f_2)$: la somme d'une fonction d'ordre f_1 et d'une fonction d'ordre f_2 est une fonction d'ordre $f_1 + f_2$.
Et de même, pour $a \in \mathbb{R}$ et $b \in \mathbb{R}^*$:

$$a.O(f) = O(f), \quad b.\Theta(f) = \Theta(f), \quad O(f_1) \times O(f_2) = O(f_1 \times f_2), \quad \Theta(f_1) \times \Theta(f_2) = \Theta(f_1 \times f_2), \quad f + O(f) = \Theta(f)$$

En particulier on pourra utiliser les notations O et Θ dans les calculs. Nous utiliserons notamment :

$$\sum \Theta(f_k) = \Theta\left(\sum f_k\right)$$

3.5 Complexité d'un algorithme

Définition. La **complexité d'un algorithme**, respectivement **dans le pire des cas**, **dans le meilleur des cas**, et **en moyenne** est définie comme l'ordre de sa fonction de complexité respectivement dans le pire des cas, dans le meilleur des cas, et en moyenne.


La **complexité d'un algorithme** (dans le pire des cas, dans le meilleur des cas, et en moyenne), est l'ordre de sa fonction de complexité.

Si un algorithme a une fonction de complexité (respectivement dans le pires des cas, dans le meilleur des cas, en moyenne) $T(n)$ on dira que l'**algorithme est de complexité** (respectivement dans le pire des cas, dans le meilleur des cas, en moyenne) :

- **bornée** si $T(n)$ est majorée (ordre $O(1)$).
- **logarithmique** si $T(n)$ a pour ordre $\Theta(\ln(n))$.
- **linéaire** si $T(n)$ a pour ordre $\Theta(n)$.
- **quadratique** si $T(n)$ a pour ordre $\Theta(n^2)$.
- **polynomiale** si $\exists k \in N$, $T(n)$ a pour ordre $\Theta(n^k)$.
- **exponentielle** si $\exists a > 1$, $T(n)$ a pour ordre $\Theta(a^n)$.

Plus l'ordre de la fonction de complexité est faible, au sens de la relation d'ordre "être dominée par", meilleur est le comportement asymptotique de la rapidité d'exécution de l'algorithme, et meilleure est la complexité. Il n' y a pas de complexité meilleure que celle en temps borné $O(1)$, qui signifie informellement que le temps d'exécution d'un algorithme est majoré par une constante ne dépendant pas de la taille de la donnée.

Exemples :

• Compter le nombre d'occurrences d'un élément dans une séquence de longueur variable N :

```
def compte(liste,a):
    compteur = 0
    for x in liste:
        if x == a:
            compteur += 1
    return compteur
```

Pour une entrée de taille N le nombre d'opérations élémentaires est égal à $T(N) = 2 + 2.N + k$ où k est le nombre d'occurrence de `a` dans `liste`, $0 \leqslant k \leqslant N$.

La fonction de complexité dans le pire des cas est $F_{max}(N) = 2 + 3N$; l'algorithme est de complexité, dans le pire des cas, linéaire, d'ordre $\Theta(N)$.

La fonction de complexité dans le meilleur des cas est $F_{min}(N) = 2 + 2N$; l'algorithme est de complexité, dans le pire des cas, linéaire, d'ordre $\Theta(N)$.

L'ordre de la fonction de complexité en moyenne étant compris entre le meilleur des cas et le pire des cas, la complexité moyenne est aussi linéaire. Ici il est facile de la déterminer.

Il n'est pas possible de faire mieux qu'une complexité linéaire : pour compter le nombre d'occurrence de `a` il faut lire chaque élément de la liste soit au moins N opérations élémentaires.

• Recherche d'un élément dans une séquence :

```
def find(liste,a):
    for x in liste :
        if x == a:
            return True
    return False
```

Pour une entrée de taille `N=len(liste)`.

Dans le pire des cas : celui où l'élément `a` est absent, ou se trouve en dernière position, la fonction de complexité dans le pire des cas est $F_{max}(N) = 2N+1$; la complexité dans le pire des cas est d'ordre $\Theta(N)$, linéaire.

Dans le meilleur des cas, celui où l'élément `a` est présent en première position, la fonction de complexité dans le meilleur des cas est $F_{min}(N) = 3$; la complexité dans le meilleur des cas bornée, en $O(1)$.

La complexité en moyenne est plus délicate à calculer. Il faudrait d'abord calculer la probabilité que `a` figure à chaque position, ou n'y figure pas. Pour une loi uniforme, elle serait aussi linéaire.

Il n'est pas possible de faire mieux : dans le pire des cas, celui où l'élément est absent, tous les élément doivent être lus au moins une fois, puisqu'on n'a aucune information à priori sur la position que pourrait occuper `a`, ainsi au moins N opérations. Dans le meilleur des cas, on ne peut obtenir meilleure complexité que $O(1)$.

La complexité en moyenne est en général plus délicate à calculer. Ce n'est pas un exigible dans le programme officiel.

3.6 Calcul de complexité

En général on évite pour le calcul de la complexité d'exprimer les fonctions de complexité. On essaie tant que possible de se ramener à des schémas généraux qui permettent facilement de déduire la complexité.

• Tout algorithme de la forme (boucle simple) :

```
for x in liste :
    suite d'opérations élémentaires sans boucle
    ou sortie anticipée (break, return)
```

est de complexité dans le pire des cas, le meilleur des cas et en moyenne linéaire en fonction de la longueur de la liste.

En effet :

$$\sum_{k=1}^{N} \Theta(1) = \Theta(\sum_{k=1}^{N} 1) = \Theta(N)$$

• Pour entrée est de taille N, tout algorithme de la forme (deux boucles imbriquées) :

```
for i in [[1,N]]:
    for j in [[1,N]]:
        suite d'opérations élémentaires sans boucle
        ou sortie anticipée
```

est de complexité quadratique $\Theta(N^2)$, dans le pire, le meilleur des cas, et en moyenne.

$$\sum_{i=1}^{N}\sum_{j=1}^{N} \Theta(1) = \sum_{i=1}^{N} \Theta(N) = \Theta\left(\sum_{i=1}^{N} N\right) = \Theta(N^2)$$

Exemple : Calcul d'une somme double de la forme :

$$\sum_{0\leqslant i,j\leqslant N} a_{i,j}$$

```
S = 0
for i in range(N+1):
    for j in range(N+1):
        S += a[i][j]
print(S)
```

est de complexité quadratique en moyenne et dans le pire et le meilleur des cas.

• Parcours d'un tableau triangulaire (deux boucles `for` imbriquées) :

```
for k in [[1,N]]:
    for j in [[1,k]]:
        Suite d'opérations élémentaires sans boucle
        ou sortie anticipée
```

est de complexité quadratique N^2.

En effet supposons qu'il y ait C opérations élémentaires dans la boucle.

La fonction de complexité vaut dans chaque cas $C+2C+3C+\cdots NC = C\frac{N(N+1)}{2} = \frac{C}{2}(N^2+N)$ qui est d'ordre quadratique.
On peut aussi noter :

$$\sum_{k=1}^{N}\sum_{j=1}^{k}\Theta(1) = \sum_{k=1}^{N}\Theta(k) = \Theta\left(\sum_{k=1}^{N}k\right) = \Theta\left(N\times\frac{N+1}{2}\right) = \Theta(N^2)$$

Ainsi les fonctions de complexité en moyenne, dans le meilleur et dans le pire des cas sont toutes quadratiques.

Exemple : Calcul de la somme double :

$$\sum_{0\leqslant i\leqslant j\leqslant N} a_{i,j}$$

```
S = 0
for j in range(N+1):
    for i in range(j+1):
        S += a[i][j]
print(S)
```

est de complexité quadratique en moyenne et dans le pire et le meilleur des cas.

• Trois boucles `for` imbriquées :

```
for i in [[1,N]]:
    for j in [[1,N]]:
        for k in [[1,N]]:
            suite d'opérations élémentaires sans boucle
            ou sortie anticipée
```

est de complexité cubiquee en moyenne et dans le pire et le meilleur des cas.
Exemple : Produit matriciel de deux matrices carrées A et B d'ordre N.

$$\boxed{\forall\,(i,j)\in[[1,N]]^2, (A\times B)_{i,j} = \sum_{k=1}^{N} A_{i,k}B_{k,j}}$$

```
def prod(A,B):
    prod = []
    for i in range(len(A)):
        prod.append([])
        for j in range(len(B[0])):
            s = 0
            for k in range(len(B)):
                s += A[i][k] * B[k][i]
            prod[i].append(s)
    return prod
```

Quelle est sa complexité ? Dans le pire, le meilleur des cas, et en moyenne :
Cubique $\Theta(N^3)$ pour des matrices carrées d'ordre N.

$$\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k=1}^{N}\Theta(1) = \Theta\left(\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k=1}^{N}1\right) = \Theta\left(\sum_{i=1}^{N}\sum_{j=1}^{N}N\right) = \Theta\left(\sum_{i=1}^{N}N^2\right) = \Theta(N^3)$$

De complexité $\Theta(P\times Q\times R)$ pour le produit de matrices de types (P,Q) et (Q,R).

3.6.1 Complexité en espace

La complexité que nous venons de définir s'appelle la **complexité en temps**. On définit aussi la **complexité en espace** d'un algorithme, dans le pire des cas, dans le meilleur des cas, et en moyenne.

Définition. Donné un algorithme (ou programme) prenant en paramètre des données de taille N variable.

• Sa **Fonction de complexité (en espace) dans le pire des cas** de l'algorithme, est l'application $E_{max} : \mathbb{N} \longrightarrow \mathbb{N}$ définie par $E_{max}(N)$ est la quantité maximale d'espace mémoire nécessaire au stockage des variables utilisées pour exécuter le programme sur une entrée de taille N (celui dans le cas le plus défavorable).

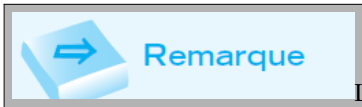
• Sa **Fonction de complexité (en espace) dans le meilleur des cas** de l'algorithme, est l'application $E_{min} : \mathbb{N} \longrightarrow \mathbb{N}$ définie par $E_{min}(N)$ est la quantité minimale d'espace mémoire nécessaire au stockage des variables utilisées pour exécuter le programme sur une entrée de taille N (celui dans le cas le plus favorable).

• Sa **Fonction de complexité (en espace) en moyenne** de l'algorithme, est l'application $E_{moy} : \mathbb{N} \longrightarrow \mathbb{R}$ définie par $E_{moy}(N)$ est la quantité moyenne d'espace mémoire nécessaire au stockage des variables utilisées pour exécuter le programme sur les entrées de taille N.


On ne comptabilise pas l'espace déjà occupé par les paramètres auquel s'applique l'algorithme.

Définition. La **complexité spatiale d'un algorithme**, respectivement **dans le pire des cas**, **dans le meilleur des cas**, et **en moyenne** est définie comme l'ordre de sa fonction de complexité en espace respectivement dans le pire des cas, dans le meilleur des cas, et en moyenne.


La **complexité spatiale d'un algorithme** (dans le pire des cas, dans le meilleur des cas, et en moyenne), est l'ordre de sa fonction de complexité en espace, c'est à dire l'ordre de l'espace mémoire supplémentaire nécessaire à son exécution.



Lorsque on parle de complexité sans préciser s'il s'agit de complexité spatiale ou temporelle, hors de tout contexte, c'est de la complexité temporelle dont on parle.

4 Application : recherche d'un élément dans une liste triée

Nous étudions dans cette partie un exemple important et fondateur : la recherche d'un élément dans une liste (ou tableau) ordonnée.

4.1 Première approche naïve

Supposons que l'on souhaite écrire un algorithme qui recherche si un élément passé en paramètre est présent ou non dans une liste que l'on sait être triée, ordonnée par exemple dans le sens croissant.
On dispose déjà d'un algorithme pour rechercher si un élément apparaît dans une séquence :

```
def recherche1(L,a):
    for x in L:
        if x==a:
            return True
    return False
```

C'est la première solution qu'on pourrait utiliser. Sa complexité est linéaire (en fonction de la longueur N de la liste) dans le pire des cas.

On pourrait aussi utiliser l'instruction prédéfinie `in` :

```
def recherche2(L,a):
    return a in L
```

Sa complexité est-elle meilleure ? Son exécution serait-elle plus rapide ?
Réponses : non, sa complexité n'est pas meilleure. Sans information supplémentaire nous avons déjà vu que l'on ne peut pas faire mieux qu'une recherche en temps linéaire dans le pire des cas : il faut bien déjà accéder à chaque élément de la liste, jusqu'au dernier si nécessaire, d'où N opérations élémentaires de lecture.

Son exécution serait légèrement plus rapide. C'est dû au fait que l'instruction est programmée à plus bas niveau, dans un langage plus proche du langage machine, et donc sensiblement plus rapide, puisque qu'on économise l'interprétation du code. Mais le rapport entre les temps d'exécution des deux fonctions resterait approximativement constant lorsque N augmente, puisque les deux algorithmes ont même complexité.

Mais ces deux solutions ne tiennent nullement compte du fait que la liste soit triée.
Prenons l'exemple de la recherche d'un mot dans un dictionnaire. On comprend tous qu'il est bien pratique que tous les mots soient rangées dans l'ordre alphabétique. Autrement la recherche d'un mot ne pourrait se faire qu'en lisant le dictionnaire page après page, ligne après ligne.
Autre exemple, la recherche d'un numéro de téléphone dans un annuaire (ou botin), où les numéros sont rangés par ordre alphabétique des noms des interlocuteur. On comprend tous que s'il est pratique d'y chercher le numéro d'une personne dont on connait le nom, il est presque impossible d'y rechercher le nom d'une personne à partir de son seul numéro de téléphone.
Pourtant appliquer l'un de ces deux algorithmes reviendrait dans chaque cas à rechercher l'élément requis, page après page, ligne après ligne. On doit pouvoir fait mieux, puisqu'on y parvient très bien dans la pratique. C'est ce que nous allons voir dans la suite, avant d'analyser l'efficacité de l'algorithme que nous obtiendrons.

4.2 Recherche par dichotomie dans une liste triée

4.2.1 Principe de la recherche par dichotomie

Pour schématiser la recherche d'un mot dans le dictionnaire :
– on ouvre le dictionnaire approximativement au milieu. Si on a de la chance on tombe pile sur le mot recherché. La recherche s'interrompt.
– Sinon, on est tombé soit avant, soit après, pour l'ordre alphabétique. Si l'on est tombé avant on poursuit le même procédé dans la deuxième moitié du dictionnaire.
– Si l'on est tombé après on poursuit la recherche dans la première moitié du dictionnaire.
– et ainsi de suite... Jusqu'à trouver le mot recherché ou vérifier qu'il est assurément absent du dictionnaire.

C'est une recherche par dichotomie dans une liste trié. Pour l'implémenter pour la recherche d'un élément `e` dans une liste `L` que l'on sait être triée dans le sens croissant, et de longueur `N` :

- Deux variables `Imin` et `Imax` délimiteront les indices de la partie de la liste dans laquelle on poursuit la recherche. Initialement `Imin` et `Imax` valent respectivement `0` et `N-1`.
- On détermine l'indice du milieu (approximativement) dans la partie délimitée par `[[Imin,Imax]]` de la recherche. C'est l'entier `Imed = (Imin+Imax) // 2`.
- On compare l'élément de `L` à l'indice `Imed` avec `e`. Trois cas peuvent survenir :
  - `L[Imed] == e`. La recherche s'arrête, l'algorithme renvoit `True` : on a trouvé l'élément (on pourrait aussi renvoyer l'indice `Imed` où il se trouve).
  - `L[Imed] < e`. On poursuit la recherche dans la moitié à droite, en posant `Imin = Imed +1`. (Dichotomie à droite).
  - `L[Imed] > e`. On poursuit la recherche dans la moitié à gauche, en posant `Imax = Imed -1`. (Dichotomie à droite).
- On poursuit la recherche tant que la partie délimitée est non vide, c'est à dire tant que `Imax-Imin>=0`.
- A la fin de la recherche on renvoie `False`. Si on ne l'a pas troué à ce stade, c'est que l'élément `e` n'est pas présent dans `L`.

4.2.2 Code de la recherche par dichotomie dans une liste triée dans le sens croissant

```
def rechercheDichotomique(L,e):
    """Recherche par dichotomie dans une liste triée dans le
    sens croissant. Renvoie True ou False selon que l'élément
    e est ou non présent dans L."""

    Imin, Imax = 0, len(L)-1     # Délimitation de la recherche
    while Imax - Imin >= 0:      # Boucle principale
```

```
        Imed = (Imin + Imax)//2 # indice médian dans la zone de recherche
        if L[Imed] == e:        # Cas de succès
            return True
        elif L[Imed] < e:
            Imin = Imed + 1     # dichotomie à droite
        else:
            Imax = Imed - 1     # dichotomie à gauche
    return False                # Cas d'échec
```

Il faut savoir programmer cette méthode de recherche par dichotomie, et si besoin en adapter le code.

Preuve de l'algorithme :
Terminaison. L'expression `iMax-iMin` est un variant de boucle : il ne prend que des valeurs entières et positives, et décroît strictement, puisqu'il est diminué au moins de moitié à chaque passage dans la boucle. Lorsque la zone délimitée de contient qu'un élément `Imax-Imin==0`, à l'étape suivante on aurait, si la boucle ne s'arrêtait pas `Imax-Imin<0`.

Correction. La liste étant triée dans le sens croissant, on a l'invariant suivant de boucle (ce qu'on peut vérifier par une récurrence immédiate) :

"Si e est présent dans L, il est présent entre les indices Imin et Imax"

4.2.3 Test de comparaison

On teste empiriquement les 3 fonctions de recherche, les deux recherches naïves avec la recherche par dichotomie. On effectue trois séries de tests, sur des listes aléatoires de 10^4, 10^5 et 10^6 éléments, construites à l'aide de la fonction `random` du module de même nom avant d'être triées grâce à la méthode `sort` des listes. Dans chaque cas on effectue une recherche par chacun des 3 algorithmes. On recherche l'élément 1. La fonction `random` renvoyant un élément dans $[0,1[$, l'élément ne s'y trouvera pas, et ce qu'on calcule est durée d'exécution dans le pire des cas. Le calcul de la durée se fait grâce à la fonction `clock` du module `time`. Pour chaque calcule on affiche les durées d'exécution ainsi que le rapport entre la plus grande et la plus courte durée. On a effectué le test avec le code suivant :

```
from random import random
from time import clock
# liste de 10^4, 10^5, 10^6 éléments aléatoires dans [0,1[
for N in (10**4,10**5,10**6):
    print("\nComparaison des durées de recherche pour",N,"éléments")
    L = [random() for k in range(N)]
    L.sort()    # Tri du tableau
    R = []
    for f in (recherche1,recherche2,rechercheDichotomique):
        a = clock()
        f(L,1)
        b = clock()
        print("Avec ",f,":",b-a,"secondes")
        R.append(b-a)
    print("Rapport entre les plus lent et plus rapide :",int(max(R)/min(R)))
```

On a obtenu les mesures suivantes :

```
Comparaison des durées de recherche pour 10000 éléments
Avec  <function recherche1> : 0.0011410000000040554 secondes
Avec  <function recherche2> : 0.0008149999999957913 secondes
Avec  <function rechercheDichotomique> : 1.6000000002236447e-05 secondes
Rapport entre les plus lent et plus rapide : 71

Comparaison des durées de recherche pour 100000 éléments
Avec  <function recherche1> : 0.01389999999999958 secondes
```

```
Avec  <function recherche2> : 0.006959999999999411 secondes
Avec  <function rechercheDichotomique> : 1.4999999997655777e-05 secondes
Rapport entre les plus lent et plus rapide : 926

Comparaison des durées de recherche pour 1000000 éléments
Avec  <function recherche1> : 0.19840399999999647 secondes
Avec  <function recherche2> : 0.13552299999999917 secondes
Avec  <function rechercheDichotomique> : 2.800000000036107e-05 secondes
Rapport entre les plus lent et plus rapide : 7085
```

Plusieurs exécutions donnent des résultats sensiblement équivalent : ce n'est pas le fait du hasard. On remarque que :
– Sans conteste, et dans tous les cas, la recherche par dichotomie est de loin la plus rapide.
– La recherche naïve par la fonction `recherche1` est la plus lente. La fonction `recherche2` est environ deux fois plus rapide qu'elle.
– Le temps d'exécution des fonctions `recherche1` et `recherche2` augmente assez proportionnellement au nombre d'éléments N de la liste.
– A mesure que N augmente, le gain de la recherche par dichotomie sur les autres méthodes devient de plus en plus important. Elle est 71 fois plus rapide que `recherche1` pour 10^4 éléments, puis près de 1000 fois plus rapide pour 10^5 éléments, et plus de 7000 fois plus rapide pour 10^6 éléments.

Le test s'avère positif. Notre approche par dichotomie présente un avantage incontestable. C'est une approche par le principe de **diviser pour régner**. Analysons maintenant les raisons de ces performances, la complexité de cet algorithme.

4.2.4 Complexité de l'algorithme

La complexité spatiale est en $O(1)$ (l'algorithme n'utilise que les 3 variables `Imin`, `Imed`, `Imax`). La complexité (temporelle) dans le meilleur des cas est en $O(1)$. Déterminons la complexité (temporelle) dans le pire des cas, celui où l'élément n'est pas présent, ou est trouvé seulement à la dernière étape.

Pour une liste de taille N le nombre d'opérations élémentaires pour rechercher un élément par dichotomie est du même ordre que la profondeur de la dichotomie (chaque étape consistant en un nombre borné d'opérations élémentaires : 1 calcul, 1 affectation, au plus 2 tests, et 1 opération élémentaire).

Déterminer la complexité de l'algorithme revient alors pour une liste de N éléments à déterminer combien faut-il au plus de passage dans la boucle (dichotomies) en fonction de N ?

Il est plus facile de calculer la fonction réciproque : Si pour une liste au plus k dichotomies sont nécessaires pour chercher un élément, combien d'éléments contient cette liste approximativement ?

A chaque dichotomie on sépare la liste en 2 zones de tailles égales ± 1.
Donc si à l'étape $i+1$ la zone de recherche contient N_{i+1} éléments, alors à l'étape précédente i la zone de recherche contient entre $2\times N_{i+1}$ et $2\times N_{i+1}+1$ éléments.
A la dernière étape, dans le pire des cas, la zone de recherche ne contient plus qu'un élément. Ainsi la liste était initialement de taille comprise entre :

$$2^k = \underbrace{2\times 2\times\cdots\times 2}_{k \text{ fois}}\times 1 \qquad \text{et} \qquad \underbrace{2\times(2\times\ldots 2\times(2}_{k \text{ fois}}\times 1+1)+1)\cdots)+1 = \sum_{i=0}^{k} 2^i = 2^{k+1}-1 = \sum_{i=0}^{k} 2^i = 2^{k+1}-1$$

En composant par $\log_2$ (qui est croissante) :

$$\begin{aligned} 2^k \leqslant N \leqslant 2^{k+1} &\implies k \leqslant \log_2(N) \leqslant k+1 \leqslant 2k \\ &\implies k = O(\log_2(N)) \text{ et } \log_2(N) = O(k) \end{aligned}$$

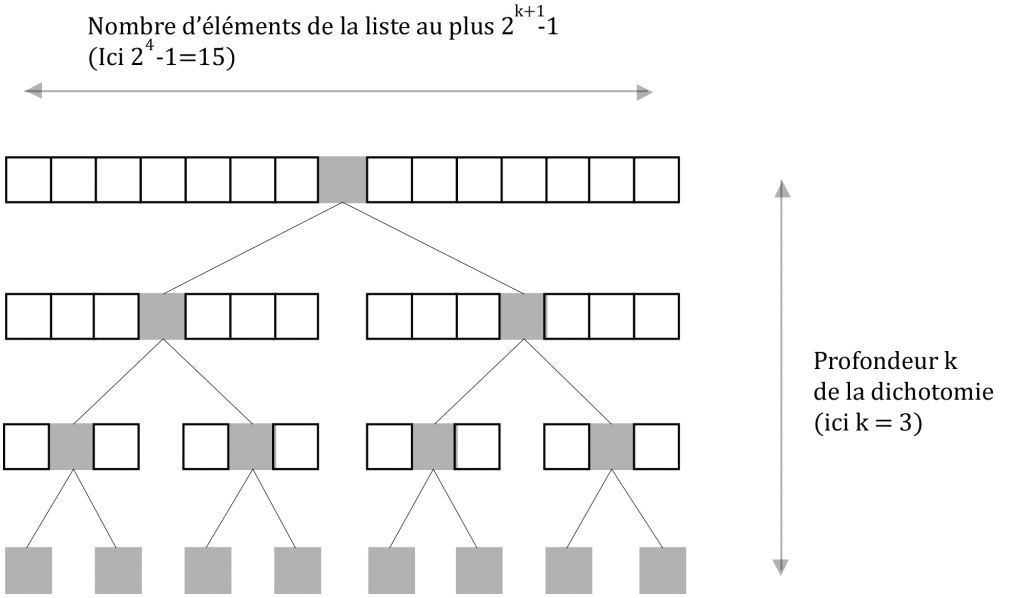
On trouve donc :

$$\boxed{\text{La profondeur de dichotomie } k \text{ a pour ordre } \Theta(\log_2(N))}.$$

Ainsi : la complexité d'une recherche par dichotomie dans une liste triée dans le sens croissant est dans le pire des cas : $\Theta(\log_2(N)) = \Theta(\log(N))$.

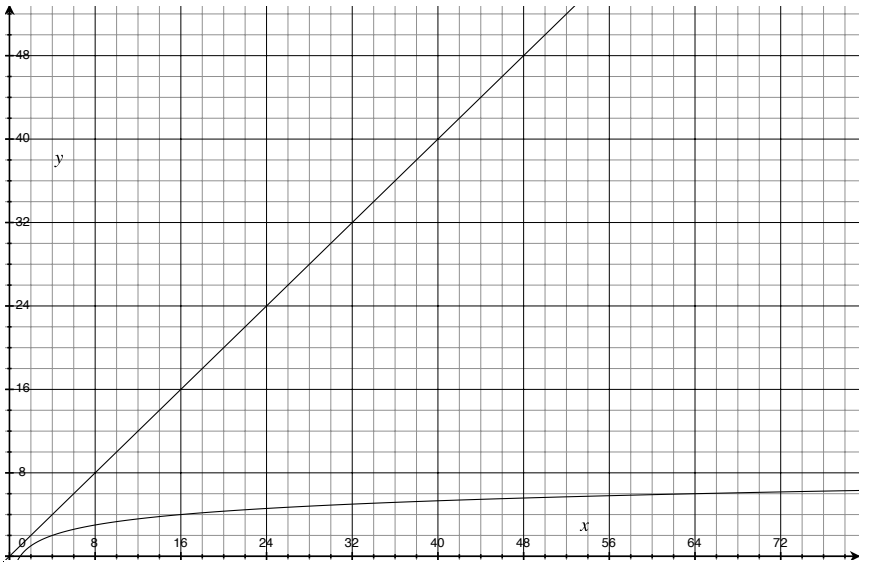
Une recherche par dichotomie dans une liste triée est de complexité dans le pire des cas logarithmique

Une recherche par dichotomie dans une liste ordonnée, se fait en complexité (temporelle) logarithmique dans le pire des cas (et bornée dans le meilleur). Sa complexité spatiale est bornée. C'est bien meilleur que l'algorithme de recherche naïf en temps linéaire.



Les cases grisées représentent les seuls éléments susceptibles d'être comparés avec l'élément recherché aux diverses étapes de la dichotomie. Lors d'une implémentation, seules celles sur une branche (jusqu'en bas dans le pire des cas) seront comparées, alors que l'algorithme naïf peut comparer avec tous les éléments de la liste.

• C'est beaucoup plus rapide par dichotomie qui se fait en complexité logarithmique, que par une recherche naïve qui elle est de complexité linéaire : voici les graphes des fonctions $x \mapsto x$ et $x \mapsto \log_2(x)$:



1) ■ Temps de vol de la suite de Syracuse.

La suite de syracuse est définie par son premier terme $u_0 \in \mathbb{N}$ et par la relation de récurrence :

$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n+1 & \text{si } u_n \text{ est impair} \end{cases}$$

Il est conjecturé (Collatz, 1937) que quelque soit $u_0 \in \mathbb{N}$ la suite de syracuse finit toujours par donner un terme égal à 1. Elle devient alors périodique de période 3 : $1,4,2,1,4,2,etc$....

1. Ecrire une fonction récursive `syracuse` prenant en paramètre un entier positif `u0` et qui retourne le rang du premier terme de la suite de syracuse égal à 1 (pour cette valeur u_0=`u0`).
2. Pourriez-vous prouver la terminaison de votre algorithme ?

1. Temps de vol de la suite de Syracuse :

```
def SyracuseVol(u_0):
    u = u_0
    n = 0
    while u != 1:
        if u%2==0:
            u = u/2
        else:
            u = 3*u+1
        n += 1
    return n
```

2. Réponse : non ! (sauf à résoudre une conjecture vieille de près de 80 ans). La preuve de la terminaison de l'algorithme est équivalente à la conjecture de Collatz.

2) ■ Division euclidienne.

On rappelle le principe de division euclidienne : donnés $(a,b) \in \mathbb{N} \times \mathbb{N}^*$ il existe un unique couple d'entiers naturels q, r tels que

$$a = b \cdot q + r \qquad \text{et } 0 \leqslant r < b$$

1. Sans utiliser les opérateur arithmétiques `//` et `%`, écrire une fonction `divisionEuclidienne(a,b)` qui retourne le couple `q,r)` des quotient et reste dans la division euclidienne de `a` par `b`.
2. Prouver la terminaison de votre programme. Pour cela on cherchera un variant de boucle.
3. Prouver la correction de votre programme. Pour cela on cherchera un invariant de boucles adaptés

1. Le principe de l'algorithme est simple : on retranche `b` à `a` que `a>=b`, en incrémentant un compteur, qui donnera le quotient, à chaque soustraction. Ce qui reste donne le reste.

```
def divisionEuclidienne(a,b):
    q = 0
    while a >= b:
        a -= b
        q += 1
    return (q,a)
```

2. On suppose `a,b` des entiers, et `b` non-nul. Montrons la terminaison de l'algorithme c'est à dire que la boucle `while` s'arrête. Pour cela il suffit de considérer le variant de boucle `a`. En effet `a` ne prend que des valeurs strictement positives et diminue strictement à chaque itération de la boucle.

3. On suppose `a,b` des entiers, et `b` non-nul. Considérons l'expression `q*b+a` et montrons que c'est un invariant de boucle. Notons a_k, b_k, q_k les valeurs prises par les variables `a`, `b` et `q` après k passages dans la boucle. On a alors les relations de récurrence :

$$a_{k+1} = a_k - b \qquad b_{k+1} = b_k \qquad q_{k+1} = q_k + 1$$

Ainsi :

$$q_{k+1} \times b_{k+1} + a_{k+1} = (q_k + 1) \times b_k + a_k - b_k = q_k \times b_k + b_k + a_k - b_k = q_k \times b_k + a_k$$

Ainsi l'expression est bien un invariant de boucle. Pour $k = 0$, $q_0 = 0$ et donc $q_0 \times b_0 + a_0 = a_0$. A la sortie de boucle, après n itérations $a_n < b_n$. Or $b_n = b_0$, aussi en posant $r = a_n$ et $q = q_n$, on a

$$a_0 = q \times b_0 + r$$

Ceci prouve qu'on retourne bien le couple (q, r) donné par la division euclidienne de a par b.

3) ■ Recherche par dichotomie dans une liste triée.

1. Recherche dichotomique en utilisant un slicing.
   (a) Écrire une fonction `rechercheDich(L,e)` qui recherche par dichotomie si le nombre `e` apparaît dans la liste `L` de nombres ordonnés dans le sens croissant, mais qui utilise pour la dichotomie les slicings :
   `L = L[:Imed]` ou `L = L[Imed:]`.
   (b) Calculer dans le pire des cas les complexités temporelles et spatiales de cette version. Présente-t-elle un avantage ?
2. Tri par insertion dichotomique.
   (a) Ecrire une fonction `cherche(L,e)` prenant en paramètre un nombre `e` et une liste `L` de nombres triée dans l'ordre croissant et qui retourne par dichotomie :
   - S'il existe : le plus grand indice `k` d'éléments de `L`, tel que l'élément `L[k]` soit inférieur ou égal à `e`.
   - s'il n'existe pas : -1.

   Quelle est sa complexité dans le pire des cas ?
   (b) A l'aide la fonction décrite en a), écrire une fonction `insere(L,e)` prenant en paramètre une liste de nombres triée dans l'ordre croissant `L` et une nombre `e`, et qui retourne une liste triée dans l'ordre croissant obtenue en insérant l'élément `e` dans la liste `L`.
   (c) A l'aide de la fonction décrite en b), écrire une fonction `tri(L)` prenant en paramètre une liste `L` et qui retourne une liste contenant les mêmes éléments que `L` triés dans le sens croissant.

1. Si `L` a pour longueur N et `Imed` est son indice médian, un slicing `L[:Imed]` ou `L[Imed:]` a un coût en temps et en espace de l'ordre de $N/2$ (car nécessite de l'ordre de $N/2$ copies en mémoire).
2.a. La condition de dichotomie à droite devient `L[iMed] <= e:`.

1.a. Recherche par dichotomie en utilisant un slicing :

```
def rechercheDich(L,e):
    """ Recherche par dichotomie avec slicing"""
    while len(L) > 0:
        iMed = len(L)//2
        if L[iMed] == e:
            return True
        elif L[iMed] < e:
            L = L[iMed+1:]
        else:
            L = L[:iMed]
    return False
```

1.b. Calcul des complexités temporelle et spatiale dans le pire des cas.

Complexité temporelle. Pour une liste de longueur N, le nombre de passages dans la boucle `while` dans le pire des cas est entre $\log_2(N) - 1$ et $\log_2(N)$ (voir calcul de la complexité dans le cours). Au k-ième passage il y a de l'ordre de $N/2^k$ opérations élémentaires (à cause du slicing qui crée une copie d'approximativement la moitié de `L`). Ainsi la fonction

de complexité est encadrée par des applications d'ordres :

$$\begin{aligned}
& \sum_{k=1}^{\log_2(N)-1} \Theta\left(\frac{N}{2^k}\right) \leqslant F_{max}(N) \leqslant \sum_{k=1}^{\log_2(N)} \Theta\left(\frac{N}{2^k}\right) \\
\Longrightarrow \quad & \Theta\left(\sum_{k=1}^{\log_2(N)-1} \frac{N}{2^k}\right) \leqslant F_{max}(N) \leqslant \Theta\left(\sum_{k=1}^{\log_2(N)} \frac{N}{2^k}\right) \\
\Longrightarrow \quad & \Theta\left(\frac{N}{2}\frac{1-\left(\frac{1}{2}\right)^{\log_2(N)-1}}{1-\frac{1}{2}}\right) \leqslant F_{max}(N) \leqslant \Theta\left(\frac{N}{2}\frac{1-\left(\frac{1}{2}\right)^{\log_2(N)}}{1-\frac{1}{2}}\right) \\
\Longrightarrow \quad & \Theta\left(N\left(1-\left(\frac{1}{2}\right)^{\log_2(N)-1}\right)\right) \leqslant F_{max}(N) \leqslant \Theta\left(N\left(1-\left(\frac{1}{2}\right)^{\log_2(N)}\right)\right) \\
\Longrightarrow \quad & \Theta\left(N\left(1-\frac{2}{N}\right)\right) \leqslant F_{max}(N) \leqslant \Theta\left(N\left(1-\frac{1}{N}\right)\right) \\
\Longrightarrow \quad & \Theta(N-2) \leqslant F_{max}(N) \leqslant \Theta(N-1) \\
\Longrightarrow \quad & \boxed{F_{max}(N) = \Theta(N)}
\end{aligned}$$

La complexité temporelle dans le pire des cas est linéaire. Tout comme l'algorithme de recherche naïf n'utilisant pas l'hypothèse que la liste est triée.

Complexité spatiale. Il y a entre $\log_2(N)-1$ *et* $\log_2(N)$ *passages dans la boucle* `while`. Au k-ième passage, l'extraction de liste effectue une copie qui occupe $N/2^k$ espaces en mémoire. L'espace mémoire n'est pas libéré avant la sortie de la fonction. Ainsi le même calcul que précédemment donne une complexité en espace linéaire dans le pire des cas.

Cette approche perd tous les avantages de la recherche dichotomique : les slicings causent une complexité linéaire en temps, comme l'algorithme naïf, mais de plus la complexité en mémoire est aussi linéaire, au contraire de l'algorithme naïf qui nécessite un espace borné. Cette approche est à proscrire.

2.a. Fonction `cherche(L,e)`.

```
def cherche(L,e):
    if e < L[0]:
        return -1
    iMin, iMax = 0, len(L)
    while iMax - iMin > 1:
        iMed = (iMin + iMax) // 2
        if L[iMed] <= e:
            iMin = iMed
        else:  iMax = iMed
    return iMin
```


LA condition de dichotomie à droite est devenu `if L[iMed] <= e:`.

Complexité dans le pire des cas : $O(\log(N))$ où `N=len(L)`

2.b. Fonction `insere(L,e)`.

```
def insere(L,e):
    i = cherche(L,e)
    if i == -1
        L.insert(e,0)
    else:
        L.insert(e,i+1)
    return L
```

2.c) Fonction `tri(L)` de tri par insertion dichotomique.

```
def tri(L):
    Lt = []
    for x in L:
        insere(Lt,x)
    return Lt
```

4) ■ Preuve et complexité de l'algorithme d'écriture binaire d'un nombre et de l'exponentiation rapide.

• **Partie I.** *Préliminaires.*

1. Montrer, sans utiliser les opérateurs `//` et `%` comment obtenir le quotient par 2, et le reste modulo 2 d'un nombre entier positif `n` en temps borné ne dépendant pas de `n`.
2. A l'aide de la méta-commande `%timeit` comparer les temps d'exécution sur `n=10**9` de votre solution avec les commandes `n//2` et `n%2`.

• **Partie II.** *Algorithme d'écriture binaire d'un entier naturel.*

On rappelle l'algorithme prenant en paramètre un entier positif et retournant une chaine de caractère contenant son écriture en binaire :

```
def binaire(n):
    if n == 0:
        return "0"
    ch = ""
    while n>0:
        ch = str(n%2) + ch
        n = n//2
    return ch
```

3. Prouver la terminaison de l'algorithme et calculer sa complexité en fonction de n.
4. Prouver la correction de l'algorithme. Pour cela considérer pour invariant de boucle : la somme de $n \times 2^k$ et de l'entier représenté par l'écriture binaire dans `ch`. (Où k est le nombre de passages dans la boucle effectués)

• **Partie III.** *Algorithme d'Exponentiation rapide.*

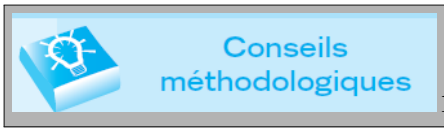
L'élévation à la puissance d'un nombre a par un entier naturel n peut se faire naïvement par n multiplications en appliquant les relations $a^0 = 1$ et $a^{n+1} = a^n \times a$. Mais on obtient un bien meilleur algorithme en appliquant plutôt les relations suivantes :

$$\boxed{a^0 = 1 \quad ; \quad a^{2p} = (a^2)^p \quad ; \quad a^{2p+1} = a.a^{2p}}$$

C'est l'algorithme d'exponentiation rapide, dont voici le code d'une implémentation :

```
def expoRapide(a,n):
    R = 1
    while n>0:
        if n%2==1:
            R *= a
        n = n//2
        a = a**2
    return R
```

5. Montrer la terminaison et la correction du programme ; pour cela utiliser comme invariant de boucle `R * a**n`.
6. Donner la complexité de l'algorithme d'exponentiation rapide et la comparer à celle de l'algorithme naïf. Comparer sa durée d'exécution avec l'algorithme naïf (qu'on aura écrit au préalable) en utilisant la méta-commande `%timeit` et les paramètres $a = 2$ et $n = 10^3,\ 10^4$.



1. Utiliser les opérateurs de "et bit à bit" & et de décalage à droite ».

Partie A.

1. Quotient de `n` par 2 : `n // 2` ou `n >> 1`
Reste de `n` modulo 2 : `n % 2` ou `n & 1`.

Les opérations logiques bits à bits & et > > sont exécutées directement au niveau du microprocesseur, et s'exécutent en $O(1)$ sur des données encodées sur 64 bits.

2. Comparaisons de leur temps d'exécution :

```
In [1]: %timeit 10**9%2  # modulo 2 version 1
10000000 loops, best of 3: 30.9 ns per loop

In [2]: %timeit 10**9&1  # modulo 2 version 2
10000000 loops, best of 3: 31.2 ns per loop

In [3]: %timeit 10**9//2 # quotient par 2 version 1
10000000 loops, best of 3: 31.4 ns per loop

In [4]: %timeit 10**9>>1 # quotient par 2 version 2
10000000 loops, best of 3: 31.2 ns per loop
```

On voit que les opérateurs % et // sont très performants et s'exécutent en temps borné $O(1)$ pour les quotients et restes modulo 2.

Partie B.

3. *Terminaison de l'algorithme.* Lorsque `n==0`, la fonction retourne la chaine `"0"`. Lorsque `n>0` la fonction est constituée d'une boucle, et on a un variant de boucle simple : `n` qui ne prend que des valeurs strictement positives qui décroissent strictement à chaque itération (à l'instruction `n = n//2`).
Complexité en fonction de `n`. Le programme consiste en des opérations s'effectuant en temps borné $O(1)$, et en une boucle `while`. A chaque passage dans la boucle le coût des opérations est en temps borné. La complexité est donc donnée par le nombre de passages nécessaires. Il faut de l'ordre de $\lceil \log_2(n) \rceil + 1 = \Theta(\log(n))$ passages dans la boucle pour que `n` vaille 0. La complexité dans le pire et le meilleur des cas est donc logarithmique.

4. Soit k le nombre de passages effectués dans la boucle. Remarquons qu'après k passages, la chaine `ch` est constituée de k chiffres. Notons n_k la valeur prise par la variable `n` et m_k le nombre entier que représente l'écriture binaire (non signée) dans `ch` après k passages dans la boucle. Montrons que

$$2^k \times n_k + m_k$$

est un invariant de boucle.

• Premier cas. Lorsque n_k est pair : alors `n%2` $= n_k \mod 2$ vaut 0 et `n//2` $= n_k/2$ vaut $n_k/2$. On a alors les relations de récurrence :

$$n_{k+1} = n_k/2 \quad \text{et} \quad m_{k+1} = m_k \quad \Longrightarrow 2^{k+1} n_{k+1} + m_{k+1} = 2^{k+1} n_k/2 + m_k = 2^k n_k + m_k$$

Ainsi : $2^{k+1} n_{k+1} + m_{k+1} = 2^k n_k + m_k$. C'est l'égalité recherchée.

• Deuxième cas. Lorsque n_k est impair : alors `n%2` $= n_k \mod 2$ vaut 1 et `n//2` $= \lfloor n_k/2 \rfloor$ vaut $(n_k - 1)/2$. On a alors les relations de récurrence :

$$n_{k+1} = (n_k - 1)/2 \quad \text{et} \quad m_{k+1} = 2^k + m_k \quad \Longrightarrow 2^{k+1} n_{k+1} + m_{k+1} = 2^{k+1}(n_k - 1)/2 + 2^k + m_k = 2^k n_k - 2^k + 2^k + m_k = 2^k n_k + m_k$$

Ainsi : $2^{k+1} n_{k+1} + m_{k+1} = 2^k n_k + m_k$. C'est l'égalité recherchée. On a bien un invariant de boucle.

Revenons à la correction du programme : Avant la première itération de boucle, $k = 0$ et $2^k n_k + m_k = n_0$ égale la valeur initiale du paramètre passé `n`. Après la dernière itération de boucle, $n_k = 0$, ainsi la valeur du nombre représenté par la chaine `ch` est égale à `n`. C'est la valeur qu'on retourne, et c'est bien l'écriture binaire du paramètre `n`.

5. *Terminaison l'algorithme.* Comme dans l'algorithme d'écriture binaire, la variable `n` est un variant de boucle. D'où la terminaison de l'algorithme.
Correction de l'algorithme. Montrons que `R * a**n` est un invariant de boucle. Après k passages dans la boucle, notons R_k, a_k et n_k les valeurs des variables `R`, `a` et `n`. On a alors les relations de récurrence :

$$\text{Si } n_k \text{ est pair : } \begin{cases} R_{k+1} = R_k \\ n_{k+1} = n_k/2 \\ a_{k+1} = a_k^2 \end{cases} \qquad \text{Si } n_k \text{ est impair : } \begin{cases} R_{k+1} = R_k \times a_k \\ n_{k+1} = (n_k - 1)/2 \\ a_{k+1} = a_k^2 \end{cases}$$

Ainsi, lorsque n_k est pair :

$$R_{k+1} \times a_{k+1}^{n_{k+1}} = R_k \times (a_k^2)^{n_k/2} = R_k \times a_k^{n_k}$$

et lorsque n_k est impair :

$$R_{k+1} \times a_{k+1}^{n_{k+1}} = R_k \times a_k (a_k^2)^{(n_k-1)/2} = R_k \times a_k \times a_k^{n_k-1} = R_k \times a_k^{n_k}$$

Ceci prouve bien que `R * a**n` est un invariant de boucle. Or avant la première itération de la boucle la valeur de l'expression est `a**n` pour les paramètres `a` et `n` passés à la fonction, quand après la dernière itération de boucle la variable `n` vaut 0 et `R * a**n` vaut la valeur `R` que l'on retourne. Ainsi la fonction retourne bien le paramètre `a` élevé à la puissance le paramètre `n` comme attendu.

6. La complexité est la même que celle de l'écriture binaire du nombre `n`, soit $\Theta(\log_2(n))$ (logarithmique) dans le pire et le meilleur des cas. La complexité de l'algorithme naïf est quant à elle linéaire.

```
def puissance(a,n):
    R = 1
    for k in range(n):
        R *= a
    return R
```

La comparaison des durées d'exécution confirme que l'algorithme d'exponentiation rapide est bien préférable à l'algorithme naïf :

```
In [1]: %timeit puissance(2,10**3)
10000 loops, best of 3: 124 µs per loop

In [2]: %timeit expoRapide(2,10**3)
100000 loops, best of 3: 7.89 µs per loop

In [2]: %timeit puissance(2,10**4)
100 loops, best of 3: 3.74 ms per loop

In [3]: %timeit expoRapide(2,10**4)
10000 loops, best of 3: 78.5 µs per loop
```

5) ■ Preuve et complexité d'un algorithme de Tri.

Une opération de grand usage en informatique est le tri d'un tableau de nombres. C'est à dire, donné un tableau de nombres (une liste de nombres en `python`), le modifier pour ordonner toutes ses valeurs (par exemple) dans le sens croissant.

Nous allons voir deux algorithmes de tri simples, prouver leur terminaison et correction, et calculer leur complexité.

A. Le *tri par sélection.*
Son principe est le suivant : donné un tableau de N nombres :
- Chercher le plus grand élément
- L'échanger dans le tableau avec l'élément en dernière position
- Recommencer avec le tableau des N-1 premiers éléments
- Et ainsi de suite...

1. Écrire une fonction `indMax(T,j)` qui retourne l'indice de l'élément maximal dans la liste `T` situé entre les indices `0` et `j`.
2. Écrire une fonction `triSelection(T)` prenant en paramètre une liste de nombres `T` et qui lui applique un tri par sélection.
3. Prouver la terminaison de l'algorithme.
4. Calculer la complexité de l'algorithme en fonction de `N` dans le pire et dans le meilleur des cas.
5. Prouver la correction de l'algorithme. Pour cela considérer pour invariant de boucle : "Après `i` passages dans la boucle principale, tous les élément de `T` à partir de l'indice `N-i` sont les plus grands éléments de `T` et sont triés dans le sens croissant.

B. Le *tri à bulle.*
Son principe est le suivant : donné un tableau de N nombres :
– Parcourir les éléments du tableau à trier du premier au dernier.
– Dès que deux éléments consécutifs ne sont pas dans le bon ordre, échanger leur position.
– Recommencer tant que l'on a changé quelque chose.

1. Après k parcours du tableau que peut-on dire des k derniers éléments du tableau ?
2. En déduire la terminaison et la correction de l'algorithme.
3. Écrire une fonction `triBulle(T)` prenant en paramètre une liste de nombres et qui les ordonne dans le sens croissant par un tri à bulle.
4. Calculer la complexité de l'algorithme en fonction de N dans le pire et dans le meilleur des cas.

A.1. Fonction `indMax(L,j)` qui retourne l'indice du maximum dans `T[:j+1]` :

```
def indMax(T,j):
    # Retourne l'indice du max dans L entres les indices 0 et j
    indMax = 0
    for k in range(1,j+1):
        if T[k] > T[indMax]:
            indMax = k
    return indMax
```

2. Fonction implémentant un tri par sélection. Elle appelle la fonction `indMax`.

```
def triSelection(T):
    """Algorithme du tri par Sélection pour trier
    une liste dans le sens croissant"""
    n = len(T)
    for j in range(n-1,-1,-1):
        iMax = indMax(T, j)
        T[j], T[iMax] = T[iMax], T[j]
    return T
```

A.3. Terminaison de l'algorithme :
• La fonction `indMax(T,j)` s'arrête à cause de sa boucle `for`. Elle retourne bien l'indice du maximum entre les indices

0 et j : pour cela considérer l'invariant de boucle :
"Après k passages dans la boucle for, indMax est l'indice du maximum de T entre les indices allant de 0 à k".

• La fonction triSelection(T) s'arrête à cause de sa boucle for et parce que indMax(T,0,k) s'arrête pour tout k.

A.4. Complexité. La fonction indMax(T,j) a pour complexité $\Theta(j)$ dans le pire et dans le meilleur des cas.

La fonction triSelection est constituée d'une boucle for j, où la variable j va de n-1 à 0, et où à chaque passage il y a $2+\Theta(j)=\Theta(j)$ opérations élémentaires dues à l'appel de indMax(T,j). Ainsi le nombre d'opérations élémentaires est au total, dans le pire et le meilleur des cas, de :

$$\sum_{j=N-1}^{0} \Theta(j) = \Theta(\sum_{j=0}^{N-1} j) = \Theta(N^2)$$

La complexité est quadratique dans le pire des cas, le meilleur des cas, et en moyenne.

A.5. Correction de l'algorithme :
L'invariant de boucle à considérer pour la boucle for de triSelection(T) est la propriété :
"Après i passages dans la boucle, les éléments de T entre les indices n-i et n-1 sont les plus grands éléments de T et sont ordonnés dans le sens croissant".

Par récurrence sur i. La propriété est vraie initialement (liste vide).
Au i+1-ème passage dans la boucle :
Par (HR) les éléments des indices n-i à n-1 sont les plus grands éléments de T et sont ordonnés dans le sens croissant :

$$\boxed{T[0],\quad \cdots,\quad max(T[0:n-i]),\quad \cdots,\quad T[n-(i+1)] \quad\leqslant\quad T[n-i] \quad\leqslant\quad \cdots \quad T[n-1]}$$

On échange alors l'élément d'indice n-(i+1) avec le plus grand élément de T entre les éléments 0 et n-(i+1). La proposition reste vraie à la sortie de boucle : les éléments d'indices allant n-(i+1) à n-1 sont les plus grands et ordonnés dans le sens croissant.

$$\boxed{T[0],\quad \cdots,\quad T[n-(i+1)] \quad,\ldots \quad\leqslant\quad max(T[0:n-i]) \quad\leqslant\quad T[n-i] \quad\leqslant\quad \cdots \quad T[n-1]}$$

Lorsque le programme s'arrête l'invariant de boucle reste vrai et $n-i=0$: Tous les éléments des indices 0 à n-1, c'est à dire tous les éléments de T sont ordonnés dans le sens croissant. La liste T est triée.

B.1. À la fin du premier parcours du tableau, le dernier élément du tableau est sa plus grande valeur : il est donc à sa place définitive. En effet, supposons que le maximum soit disposé à l'indice k, alors durant le premier parcours du tableau il sera successivement déplacé à l'indice $k+1$, $k+2$, ..., jusqu'à l'indice $N-1$ (en effet, toutes les comparaisons T[k] > T[k+1], T[k+1] > T[k+2], ..., T[N-2] > T[N-1] étant successivement vérifiées).
Ainsi après le k-ème parcours du tableau, tous les k derniers éléments sont à leurs places définitives.
En particulier à chaque parcours de tableau, le parcours pourra s'arrêter un indice avant celui du précédent parcours.

B.2. Au vu de cette propriété, on a pour invariant de boucle :
"Après k parcours du tableau, les k derniers éléments du tableau sont à leur place définitive."
Ainsi après N parcours, le tableau est trié. Le parcours suivant ne permutera donc aucun élément et l'algorithme s'arrêtera.

B.3. Code du tri à bulle.

```
def triBulle(T):
    """Algorithme du tri à bulle pour trier une liste
    dans le sens croissant."""
    N = len(T)
    changement = True
    while changement == True:
        changement = False
        for k in range(N-1):
            if T[k] > T[k+1]:
                T[k], T[k+1] = T[k+1], T[k]
```

```
                changement = True
        N = N-1
    return T
```

B.4. Calcul de la complexité.

Dans le pire des cas (celui où la liste est ordonnée dans le sens décroissant), la boucle `while` s'exécutera N fois. Le corps de la boucle est constitué essentiellement d'une boucle `for` (le parcours), qui s'exécute N fois. Ainsi la complexité est quadratique $\Theta(N^2)$.
Dans le meilleur des cas (celui où la liste est ordonnées dans le sens croissant), la boucle s'exécutera une seule fois. Son corps consistera en un parcours (boucle for). La complexité dans le meilleur des cas est linéaire $\Theta(N)$.

1. Notion de complexité ..... **D'après épreuve écrite E.N.A.C. 2015.**

Parmi les assertions suivantes lesquelles sont vraies :

A) La complexité d'un algorithme est une notion qui permet de quantifier la difficulté à concevoir cet algorithme.

B) La complexité d'un algorithme est une notion qui permet de quantifier le coût de cet algorithme tant en terme de temps d'exécution qu'en terme de place mémoire utilisée pendant l'exécution en fonction du nombre et de la taille des données du problème qu'on veut traiter.

C) Deux algorithmes de complexités différentes aboutissent nécessairement à deux résultats différents.

D) Pour un même résultat il existe des algorithmes de complexités différentes. Celui qui est le plus efficace est celui dont la complexité est la plus élevée pour un nombre de données fixé.

A) est faux.

B) est vrai.

C) est faux.

D) est faux. Il n'est pas loin d'être vrai : celui qui est le plus efficace n'est pas celui dont la complexité est la plus élevée, mais celui dont la complexité est la plus **faible** pour une nombre de données fixé.

2. Tri par insertion ..... **D'après épreuve écrite E.N.A.C. 2015.**

On considère la fonction suivante qui s'applique à une liste `L` de nombres :

```
def Ordonnom(L):
    for i, s in enumerate(L):
        j=i
        while 0<j and s<L[j-1]:
            L[j]=L[j-1]
            j=j-1
            L[j]=s
    return L
```

Parmi les assertions suivantes lesquelles sont vraies ?

A) `Ordonnom(L)` renvoie la liste `L` ordonnées dans le sens croissant.

B) `Ordonnom(L)` renvoie la liste `L` ordonnées dans le sens décroissant.

C) La complexité dans le cas le pire de `Ordonnom` est en $O(n^3)$ où n désigne la taille de la liste `L`.

D) La complexité dans le cas le pire de `Ordonnom` est en $O(n^2)$ où n désigne la taille de la liste `L`.

E) La complexité dans le cas le pire de `Ordonnom` est en $O(n)$ où n désigne la taille de la liste `L`.

On rappelle que l'instruction `for i, s in enumerate(L):` permet d'exécuter une boucle for pour laquelle la variable `s` parcourt les éléments de la liste `L`, tandis que la variable `i` parcourt les indices correspondants.

A) est vrai. (C'est ce qu'on appelle le *tri par insertion*).

B) est faux.

C), D) et E). Calcul de la complexité : la boucle `for` s'exécute pour chaque élément de la liste. Pour chacun d'entre eux, on l'insère à la bonne place en le comparant successivement aux éléments qui lui précèdent dans la liste (au sein de la boucle `while`).

Ainsi, dans le pire des cas, celui où la liste est ordonnée dans le sens décroissant, l'élément à l'indice `i` devra être inséré jusqu'en début de liste, soit `i-1` insertion. On a alors pour complexité :

$$\sum_{i=1}^{n} \Theta(i-1) = \Theta\left(\sum_{i=1}^{n}(i-1)\right) = \Theta\left(\sum_{i=0}^{n-1} i\right) = \Theta\left(\frac{n(n-1)}{2}\right) = \Theta(n^2).$$

La complexité dans le pire des cas est quadratique.

Ainsi :

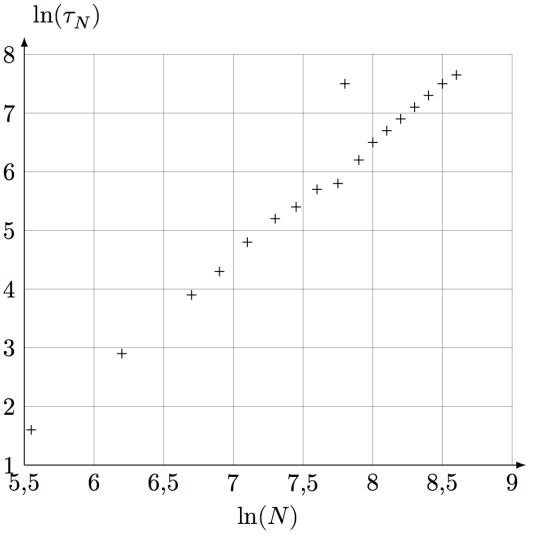
C) est vrai bien qu'imprécis, puisque $n^2 = O(n^3)$.

D) est vrai.

E) est faux. (Cependant la complexité du tri par insertion est linéaire dans le meilleur des cas).

3. Détermination empirique de la complexité. . . . . D'après Centrale-Supélec 2015

Pour un algorithme s'appliquant à une donnée de taille N variable, on a mesuré la durée τ_N d'exécution pour différentes valeurs de N. Cela permis de constituer le graphique suivant où l'on a porté $\ln(N)$ en abscisse et $\ln(\tau_N)$ en ordonnée.



1. Quelle relation simple peut-on établir entre $\ln(\tau_N)$ et $\ln(N)$ à partir de la figure ci-dessus ?
2. Quelle hypothèse peut-on émettre quant à la complexité de l'algorithme étudié ?

1. On a approximativement une relation affine :

$$\ln(\tau_N) = A \cdot \ln(N) + B$$

Puisque les points du nuage obtenus sont proches d'être distribués le long d'une droite.

2. Ainsi, en composant par exp :

$$\tau_N = e^B \cdot N^A$$

Ainsi la complexité est polynomiale.

On mesure sur le graphique donné que A vaut approximativement $(7-3)/(8{,}25-6{,}25) = 2$.
On émet comme hypothèse que la complexité de cet algorithme est quadratique (en $\Theta(N^2)$) en fonction de la taille N de la donnée.

4. Transformée de Fourier discrète. . . . . D'après Sujet 0 - Écrit Banque PT.

Un signal numérique U_f est constitué du tableau de N mesures prises à intervalles réguliers de l'amplitude d'un signal f, dépendant du temps.
Un outil classique d'analyse du signal, permettant de déterminer les fréquences de signaux sinusoïdaux (ou harmoniques) qui le constituent, est la transformée de Fourier discrète du signal numérique. Elle est définie par les N valeurs complexes :

$$TU_f(k) = \sum_{n=0}^{N-1} U_f(n) e^{-2i\pi k \frac{n}{N}} = \sum_{n=0}^{N-1} U_f(n) \cos\left(-2\pi k \frac{n}{N}\right) + i \sum_{n=0}^{N-1} U_f(n) \sin\left(-2\pi k \frac{n}{N}\right)$$

avec $k \in [\![0, N-1]\!]$.

1. Écrire une fonction `trans_fourier_freq(U,k)` retournant la partie réelle et la partie imaginaire de la valeur $TU_f(k)$ de la transformée de Fourier discrète de U_f en `k`.
2. Écrire une fonction `module(a,b)` retournant le module du nombre complexe $a + ib$.

3. Écrire une fonction `trans_fourier(U)` retournant le module de la transformée de Fourier discrète sous la forme d'une liste de taille N contenant les modules des termes $TU_f(k)$ en utilisant notamment des appels aux fonctions précédentes `trans_fourier_freq(U,k)` et `module(a,b)`.
4. Déterminer la complexité du calcul de la transformée de Fourier discrète par la fonction `trans_fourier(U)` en fonction de N.

1. Fonction `trans_fourier_freq(U,k)` :

```
from math import cos, sin, pi

def trans_fourier_freq(U,k):
    N = len(U)
    reel = 0
    imag = 0
    for n in range(N):
        reel += U[n] * cos(-2*pi*k*n/N)
        imag += U[n] * sin(-2*pi*k*n/N)
    return reel, image
```

2. Fonction `module(a,b)` :

```
def module(a,b):
    return (a ** 2 + b ** 2) ** 0.5
```

3. Fonction `trans_fourier(U)` :

```
def trans_fourier(U):
    TFU = [ ]
    for k in range(len(U)):
        reel, imag = trans_fourier_freq(U,k)
        TFU.append(module(reel,imag))
    return TFU
```

4. Complexité de `trans_fourier` en fonction de N.

Soit N la longueur de U. il faut 1 opérations pour créer la liste vide `TFU` par `TFU = []`. A cela s'ajoute une boucle for qui s'implémente N fois. A chaque passage dans la boucle on appelle `trans_fourier_freq(U,k)` qui nécessite $\Theta(N)$ opérations (car constitué d'une boucle for s'implémentant N fois et consistant en un nombre borné d'opérations) et `module()` qui est en temps borné $O(1)$. Donc la complexité de `trans_fourier(U)` est quadratique en $\Theta(N^2)$.

La complexité n'est pas optimale : l'algorithme FFT de calcul rapide de la transformée de Fourier discrète s'effectue en temps $O(N\log(N))$.

5. Nombre maximal de zéros contigus.....D'après Sujet 0 - Oral Banque PT.

Soit un entier naturel n non nul et une liste `t` de longueur n dont les termes valent 0 ou 1. Le but l'exercice est de trouver le nombre maximal de 0 contigus dans `t` (c'est-à-dire figurant dans des cases consécutives). Par exemple, le nombre maximal de zéros contigus de la liste `t1` suivante vaut 4 :

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
t1[i]	0	1	1	1	0	0	0	1	0	1	1	0	0	0	0

1. Écrire une fonction `nombreZeros(t,i)`, prenant en paramètres une liste `t`, de longueur n, et un indice `i` compris entre 0 et $n-1$, et renvoyant :

$$\begin{cases} 0, \text{ si } t[i] = 1 \\ \text{le nombre de zéros consécutifs dans } t \text{ à partir de } t[i] \text{ inclus, si } t[i] = 0 \end{cases}$$

   Par exemple, les appels `nombreZeros(t1,4)`, `nombreZeros(t1,1)` et `nombreZeros(t1,8)` renvoient respectivement les valeurs 3, 0 et 1.

2. Comment obtenir le nombre maximal de zéros contigus d'une liste `t` connaissant la liste des `nombreZeros(t,i)` pour $0 \leqslant i \leqslant n-1$?
   En déduire une fonction `nombreZerosMax(t)`, de paramètre `t`, renvoyant le nombre maximal de 0 contigus d'une liste `t` non vide. On utilisera la fonction `nombreZeros`

3. Quelle est la complexité de la fonction `nombreZerosMax` construite à la question précédente ?

4. Trouver un moyen simple, toujours en utilisant la fonction `nombreZeros`, d'obtenir un algorithme plus performant.

1. Fonction `nombreZeros`.

```
def nombreZeros(t,i):
    n = len(t)
    r = 0
    while i < n and t[i] == 0:
        i += 1
        r += 1
    return r
```

2. Il suffit de constituer la liste du nombre max de zéros débutant aux indices allant de `0` à `len(t)-1`, puis de retourner son maximum :

```
def nombreZerosMax(t):
    n = len(t)
    L = [nombreZeros(t,i) for i in range(n) ]
    return max(L)
```

3. Complexité dans le pire des cas : c'est celui d'une liste constituée uniquement de zéros : `nombreZeros(t,i)` fait de l'ordre de `len(t)-i` opérations.

Ainsi la constitution de `L` effectue de l'ordre de

$$\sum_{i=0}^{len(t)-1} len(t) - i = \sum_{k=1}^{len(t)} k = \Theta(len(t)^2)$$

opérations.
La recherche d'un maximum est linéaire : $O(len(t))$ opérations.
Ainsi la complexité est dans le pire des cas quadratique $O(len(t)^2)$.

4. On peut effectuer la recherche en complexité linéaire : il suffit de ne pas calculer `nombreZeros(t,i)` pour tous les indice $\widehat{i}$: si on a obtenu `a` zéros contigus au dernier calcul on poursuit à l'indice du premier 1 qui suit, soit à l'indice précédent augmenté de $a+1$. En voici une implémentation :

```
def nombreZerosMax1(t):
    ListeNombreZeros = []
    i = 0
```

```
    while i < len(L):
        d = nombreZeros(t,i)
        ListeNombreZeros.append(d)
        i += d+1
    return max(ListeNombreZeros)
```

Dans la boucle `while` chaque élément de la liste `t` est lu une fois : complexité linéaire. Suivi de la recherche du maximum qui est en temps linéaire sur la longueur de la liste `ListeNombreZeros`, soit en temps au plus linéaire sur la longueur de `t`.

6. Tri comptage. ..... **D'après Sujet 0 - Oral Banque PT.**

Soit N un entier naturel non nul. On cherche à trier une liste `L` d'entiers naturels strictement inférieurs à `N`.

1. Écrire une fonction comptage, d'arguments `L` et `N`, renvoyant une liste `P` dont le k-ième élément désigne le nombre d'occurences de l'entier k dans la liste `L`.
2. Utiliser la liste `P` pour en déduire une fonction `tri`, d'arguments `L` et `N`, renvoyant la liste `L` triée dans l'ordre croissant.
3. Tester la fonction `tri` sur une liste de 20 entiers inférieurs ou égaux à 5, tirés aléatoirement.
4. Quelle est la complexité temporelle de cet algorithme ?

3. On pourra utiliser la fonction `randint(0,5)` du module `random` qui retourne un nombre entier pseudo-aléatoire entre 0 et 5 inclus.
4. On exprimera la complexité en fonction du couple (N,n). On commencer par étudier celle de la fonction `comptage` avant de calculer la complexité de la fonction `tri`.

1) Fonction `comptage`.

```
def comptage(L,N):
    P = [0] * N
    for n in L:
        P[n] += 1
    return P
```

2) Fonction `tri`.

```
def tri(L,N):
    P = comptage(L,N)
    i = 0
    for j in range(N):
        x = P[j]
        for k in range(x):
            L[i] = j
            i += 1
    return L
```

3) La fonction `randint(0,5)` du module `random` retourne un nombre entier pseudo aléatoire entre 0 et 5 inclus.
On pourrait aussi utiliser la fonction `random()` qui retourne un float dans $[0,1[$ (loi pseudo-uniforme), et `int(6*random())`.

```
In [1]: from random import randint
In [2]: L = [ randint(0,5) for i in range(20) ]
In [3]: tri(L, 6)
Out[3]:
[0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 3, 3, 5, 5, 5, 5, 5, 5, 5]
```

4) Complexité de `comptage` :

N opérations pour créer le tableau P ; parcours de `L` par la boucle `for` : `len(L)` opérations : incrémentations d'un élément de P (l'écriture d'un élément dans une liste est en $O(1)$).

Pour un tableau de n éléments , constituées de valeurs dans $[[0, N]]$, la complexité temporelle de `comptage()` est en $\Theta(\max(n, N))$ (et spatiale en $O(N)$).
Complexité de `tri` :

C'est la complexité de comptage ($\max(n, N)$) plus la complexité de la reconstitution de la liste `L` triée :
Reconstitution de `L` triée : de l'ordre de :

$$P[0] + P[1] + \cdots + P[N-1] = n \quad \text{opérations}$$

(puisque la somme du nombre d'occurrences des éléments apparaissant dans `L` égale son nombre d'éléments au total).

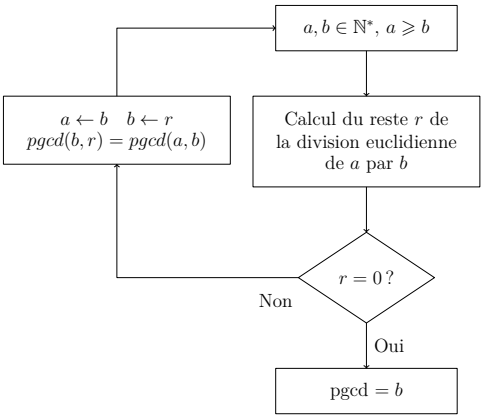
Reconstitution de `L` trié de complexité : $\Theta(n)$.

Ainsi l'algorithme est de complexité : $\Theta(\max(N, n))$ (où n est la longueur de la liste L à trier.

L'algorithme est intéressant seulement lorsque les valeurs sont des entiers positifs peu étalés (N peu grand devant n).
Pour des entiers uniformément bornés, il est intéressant car de complexité linéaire.
Autrement lorsque $N >> n$ il faut l'éviter et privilégier d'autres algorithmes de tri.

7. Complexité de l'algorithme d'Euclide. Théorème de Lamé

On peut retrasnscire l'algorithme d'Euclide de calcul du pgcd par l'organigramme :



Le but de l'exercice est de montrer le Théorème de Lamé :

Théorème 3 (Théorème de Lamé) *Le nombre n de divisions euclidiennes nécessaires au calcul du pgcd de a et b par l'algorithme d'Euclide est inférieur ou égal à 5 fois le nombre c de chiffres dans l'écriture décimale du plus petit de ces deux nombres. Plus précisément soit $\phi = \dfrac{1+\sqrt{5}}{2}$, alors $n < c\left(\dfrac{\ln(10)}{\ln(\phi)}\right) + 1$.*

A. Soit $(f_k)_{k\in\mathbb{N}}$ la suite de Fibonacci définie par :

$$f_0 = f_1 = 1 \qquad \forall\, k \in \mathbb{N},\quad f_{k+2} = f_{k+1} + f_k$$

et soit ϕ le *nombre d'or*, $\phi = \dfrac{1+\sqrt{5}}{2}$.

1. Montrer que $\phi^2 = \phi + 1$; en déduire que pour tout $k \in \mathbb{N}$, $\phi^{k+2} = \phi^{k+1} + \phi^k$.
2. En déduire que pour tout $k \in \mathbb{N}^*$, $f_{k+1} > \phi^k$.

B. On suppose sans perte de généralité que $a \neq b$ car autrement une seule division euclidienne suffit.
Soit n le nombre d'étapes (de divisions euclidiennes) dans l'algorithme d'Euclide. En posant $b = r_{n-1}$, il existe deux suites finies $(r_k)_{0\leqslant k<n}$ et $(q_k)_{0\leqslant k<n}$ d'entiers naturels telles que :

$$\begin{aligned} a &= r_{n-1} \times q_{n-1} + r_{n-2} \\ r_{n-1} &= r_{n-2} \times q_{n-2} + r_{n-3} \\ &\vdots \\ r_2 &= r_1 \times q_1 + r_0 \\ r_1 &= r_0 \times q_0 \end{aligned}$$

avec $r_{n-1} > r_{n-2} > \cdots r_2 > r_1 > r_0 > 0$.

1. Justifier que $q_0 \geqslant 2$ et pour tout $k \in [\![1, n-2]\!]$, $q_k \geqslant 1$.
2. Montrer que pour tout $k \in [\![0, n-1]\!]$, $r_k \geqslant f_{k+1}$. En déduire que $b > \phi^{n-1}$.
3. Soit c le nombre de chiffres dans l'écriture décimale de b. Remarquer que $10^c > b$, et en déduire le théorème.

C. En considérant que le quotient et le reste dans la division euclidienne de 2 nombres sont des opérations élémentaires, quelle est la complexité l'algorithme d'Euclide appliqué à deux entiers naturels non-tous deux nuls a et b ?

A.1. Le calcul donne :

$$\phi^2 = \frac{(1+\sqrt{5})^2}{4} = \frac{6+2\sqrt{5}}{4} = \frac{3+\sqrt{5}}{2} = 1 + \frac{1+\sqrt{5}}{2} = 1 + \phi$$

En multipliant les deux membres de l'égalité par ϕ^n, on en déduit :

$$\forall\, n \in \mathbb{N},\quad \phi^{n+2} = \phi^{n+1} + \phi^n$$

A.2. Par récurrence à deux pas.

Initialisation. $\phi^1 < 2 = f_2$ et $\phi^2 = \phi + 1 < 3 = f_3$. La proposition est vraie.

Hérédité. Soit $k \in \mathbb{N}^*$ tel que $f_{k+1} > \phi^k$ et $f_{k+2} > \phi^{k+1}$. Alors d'après la relation de récurrence :

$$f_{k+3} = f_{k+2} + f_{k+1} > \phi^{k+1} + \phi^k = \phi^{k+2}$$

B.2. Puisque $r_1 > r_0$, nécessairement $q_0 \geqslant 2$. Par ailleurs pour $k \in [[1, n-2]]$, $q_k \neq 0$, car autrement on aurait $r_{k+1} = r_{k-1}$; ainsi $q_k \geqslant 1$.

B.3. D'après la question précédente :

$$\begin{array}{rcl} r_{n-1} & \geqslant & r_{n-2} + r_{n-3} \\ & \vdots & \\ r_2 & \geqslant & r_1 + r_0 \\ r_1 & \geqslant & 2 \\ r_0 & \geqslant & 1 \end{array}$$

Or puisque $1 = f_1$, $2 = f_2$, et pour tout $k \in \mathbb{N}$, $f_{k+2} = f_{k+1} + f_k$, une récurrence immédiate montre que pour tout $k \in [[0, n-1]]$, $r_k \geqslant f_{k+1}$. Puisque $b = r_{n-1}$ et $f_n > \phi^{n-1}$ (d'après A.2), on en déduit que $b > \phi^{n-1}$.

B.4. SI b s'écrit sur c chiffres dans son écriture décimale, alors :

$$10^c > b > \phi^{n-1} \implies 10^c > \phi^{n-1} \implies c \times \ln(10) > (n-1) \times \ln(\phi) \implies \boxed{n < c\frac{\ln(10)}{\ln(\phi)} + 1}$$

Puisque $\dfrac{\ln(10)}{\ln(\phi)} \approx 4{,}875$ et n, c sont des entiers, on en déduit que $\boxed{n \leqslant 5 \times c}$.

C. La version de l'algorithme que nous utilisons habituellement une étape de plus (le reste nul devient b avant arrêt). En considérant quotient et reste comme des opérations élémentaires (puisqu'ils peuvent s'obtenir par une division tronquée, une multiplication et une soustraction), le nombre d'opérations élémentaires dans l'algorithme d'Euclide est dominé par le nombre de chiffre dans l'écriture décimale du plus petit des nombres, disons b.
Si b s'écrit sur c chiffres dans son écriture décimale, alors $10^{c-1} \leqslant b \leqslant 10^c \implies \log_{10}(b) \leqslant c \leqslant \log_{10}(b) + 1$.
Ainsi la complexité de l'algorithme d'Euclide du calcul du pgcd de a et b est dans le pire des cas $\boxed{O(\log(\min(a, b)))}$.