## Exercice 1. Évaluations de Complexités

Donner les complexités temporelles des fonctions suivantes :

```
def egal(L1, L2):
    """
    L1, L2: list[int]
    Returns: bool
    ----------------
    Teste si L1 == L2
    """
    if len(L1) != len(L2):
        return False
    for i in range(len(L1)):
        if L1[i] != L2[i]:
            return False
    return True
```

```
def minMax(L):
    """
    L: list[float]
    Returns: (float, float)
    ----------------
    Renvoie le min et le max de L
    """
    assert len(L) != 0
    mini = L[0]
    maxi = L[0]
    for i in range(1,len(L)):
        if L[i] < mini:
            mini = L[i]
        elif L[i] > maxi:
            maxi = L[i]
    return mini, maxi
```

```
def test(N):
    """
    N: int
    Returns: bool
    ----------------
    Indique si f1(n) == f2(n) pour tout n dans [|1,N|].
    Hypothèse: f1 et f2 s'exécutent en temps linéaire en n.
    """
    for n in range(1,N+1):
        if f1(n) != f2(n):
            return False
    return True
```

## Exercice 2. Manipulations de dictionnaires

Dans cet exercice, nous allons créer des dictionnaires à partir de chaînes de caractères. Les caractères qui ne sont pas des lettres (comme les espaces, les ponctuations ...) seront ignorés. La fonction « `ord(c: str) -> int` » établit une correspondance entre l'ensemble des caractères et l'ensemble des entiers. Par exemple, pour vérifier si un caractère `c` est une lettre minuscule, les deux programmes ci-dessous conviennent :

```
print(ord('A'))  # Affiche 65
print(ord('Z'))  # Affiche 90
print(ord('a'))  # Affiche 97
print(ord('z'))  # Affiche 122
print(ord('$'))  # Affiche 36
print(ord(' '))  # Affiche 32
print(ord('\n')) # Affiche 10
```

```
if ord('a') <= ord(c) and ord(c) <= ord('z'):
    # c contient une lettre minuscule
else:
    # ...
```

```
if 'a' <= c and c <= 'z':
    # c contient une lettre minuscule
else:
    # ...
```

Notez que dans le deuxième programme, la comparaison « `'a' <= c` » n'a à priori pas de sens. Ce sont les créateurs de Python qui ont fait en sorte que ce soit un raccourci pour « `ord('a') <= ord(c)` ».

1. Écrire une fonction « `histogramme(s: str) -> dict[str: int]` » de complexité $\mathcal{O}(\texttt{len(s)})$ qui renvoie un dictionnaire `d` tel que :
   → Les clés de `d` sont les lettres présentes dans `s`. Une lettre est soit une lettre minuscule (de `a` à `z`), soit une lettre majuscule (de `A` à `Z`).
   → La valeur associée à une lettre `c` est son nombre d'occurrences dans `s`.
2. Étant donnée une chaîne de caractères `s`, on souhaite déterminer la fréquence d'apparition de chaque lettre dans `s`. Par exemple, dans la chaîne de caractères `"Test de la fonction stat !"`, la fréquence d'apparition de la lettre `e` est $2/20 = 0.1$.
   Soit `d1` le dictionnaire renvoyé par l'appel `histogramme(s)`. À l'aide d'une compréhension (et éventuellement d'une autre boucle `for`), écrire une fonction `stat` qui prend en entrée `d1` et renvoie un dictionnaire `d2` tel que pour chaque clé `c` de `d1`, `d2[c]` est la fréquence d'apparition de `c` dans `s`.
3. Écrire une fonction « `egal(d1: dict, d2: dict) -> bool` » qui indique si `d1` et `d2` sont identiques. Bien sûr, on pourrait utiliser `d1 == d2`, mais on attend ici que vous réécriviez entièrement ce test. On vérifiera en particulier que `{}` et `{"a": 1}` sont différents.

Soient `s1` et `s2` deux chaînes de caractères ne contenant que des lettres. On dit que `s1` et `s2` sont des ***anagrammes*** lorsqu'on peut obtenir `s2` en permutant les lettres de `s2`. Par exemple `"aimons"` et `"maison"` sont des anagrammes, mais `"managera"` et `"anagramme"` ne le sont pas.

4. À l'aide des fonctions précédentes, écrire une fonction « `sont_anagrammes(s1: str, s2: str) -> bool` » qui teste si `s1` et `s2` sont des anagrammes. On pourra supposer sans le vérifier que `s1` et `s2` ne sont composées que de lettres minuscules et majuscules. On vérifiera en particulier que `"a"` et `"aa"` ne sont pas des anagrammes.

## Exercice 3. Séquences génétiques

En biologie, les séquences génétiques sont des suites de nucléotides représentées par les lettres `"A"`, `"C"`, `"G"` et `"T"`. Soit `s` une séquence génétique et $\texttt{n} \in \mathbb{N}^*$ un entier. On souhaite déterminer tous les mots de `n` lettres qui apparaissent dans `s` ainsi que le nombre d'apparitions de chacun de ces mots. Par exemple, dans la séquence `"GAACTGGCTAGCTG"`, le nombre d'apparitions des mots de `n = 2` lettres est :

```
{"GA": 1, "AA": 1, "AC": 1, "CT": 3, "TG": 2, "GG": 1, "GC": 2, "TA": 1, "AG: 1"}
```

1. Écrire une fonction « `sous_mots(s: str, n: int) -> dict[str: int]` » qui renvoie un dictionnaire qui associe à chaque mot de `n` lettres présent dans `s`, son nombre d'apparitions dans `s`. Dans l'exemple précédent, on vérifiera en particulier que `"TG"` est bien associée à `2`.

Pour tester les fonctions, nous allons utiliser la séquence génétique de la bactérie responsable de la tuberculose. Vous pouvez la télécharger à l'adresse :

`https://www.ncbi.nlm.nih.gov/nuccore/AL123456.3?report=fasta`

en cliquant sur `Send To > Complete Record > File > Format = FASTA > Create File`. L'extension du fichier obtenu est `.fasta`, il s'agit en réalité d'un simple fichier texte.

2. À l'aide de la fonction `open` de Python, récupérer la séquence présente dans le fichier téléchargé. Vérifier que la lettre `A` apparaît `758 552` fois et que le mot `"TACTAC"` apparaît `311` fois.
3. Vérifier que les plus petits mots n'apparaissant pas dans le génome de la bactérie sont de taille 7 et qu'il s'agit de `"TAAAATA"`, `"TATAATG"` et `"TATGTTA"`. Dans cette question, on attend une fonction qui calcule cela automatiquement, pas que vous le vérifiez à la main.

Étant données deux séquences génétiques `s1` et `s2`, on souhaite déterminer le plus grand mot qui apparaît à la fois dans `s1` et dans `s2`. Pour cela, on pourrait tester tous les mots de taille 1, puis tous les mots de taille 2, puis tous les mots de taille 3 ... mais cette stratégie n'est pas très rapide. À la place, on va utiliser une ***recherche par dichotomie***.

Soit $N$ la taille du plus grand mot qui apparaît à la fois dans `s1` et dans `s2`. Le calcul de $N$ se déroule en deux temps :

- ⋆ Premièrement, on détermine un encadrement de $N$. Pour cela, on vérifie s'il existe un mot de taille 1 présent dans `s1` et dans `s2`. Si c'est le cas, on recommence avec les mots de taille 2, puis les mots de taille 4, puis les mots de taille 8, ... À la fin de cette procédure, on obtient le plus petit entier $k$ tel que `s1` et `s2` n'ont pas de mot de taille $2^k$ en commun, c'est à dire que $2^{k-1} \leqslant N < 2^k$.
- ⋆ Grâce au point précédent, on sait que $N$ appartient à l'intervalle $[\![2^{k-1}; 2^k - 1]\!]$. Soit $m$ le milieu de cet intervalle, en testant s'il existe un mot de taille $m$ appartenant à `s1` et à `s2`, on peut savoir si $N$ appartient à la première moitié de l'intervalle ou bien à la deuxième moitié. En répétant ce procédé, la taille de l'intervalle de recherche est divisé par 2 (approximativement) à chaque étape. Lorsque l'intervalle de recherche ne contient plus qu'un seul élément, on sait que c'est la valeur de $N$.

Lorsqu'on teste avec la bactérie de la tuberculose et l'une des bactéries de la salmonellose :

`https://www.ncbi.nlm.nih.gov/nuccore/NC_003197.2?report=fasta`

l'intervalle de recherche lors de la seconde étape évolue de la manière suivante :

$$[\![32; 63]\!] \to [\![48; 63]\!] \to [\![56; 63]\!] \to [\![60; 63]\!] \to [\![62; 63]\!] \to [\![62; 62]\!]$$

Le plus grand mot commun aux deux séquences est donc de taille 62.

4. Écrire une fonction qui prend en entrée deux séquences génétiques `s1` et `s2` et détermine la valeur de $N$ comme décrit ci-dessus.

## Exercice 4. Critères de divisibilité

Soient $(n, d) \in \mathbb{N} \times \mathbb{N}^*$ deux entiers. Le but de cet exercice est de tester si $d$ divise $n$.

1. À l'aide de l'opérateur modulo de Python, écrire une fonction `div` qui prend en entrée $n$ et $d$, qui renvoie `True` si $d$ divise $n$ et `False` sinon.
2. Écrire une fonction `decomp` qui prend en entrée un entier $n \in \mathbb{N}$, et renvoie le couple $(a, b)$ où $a$ est le chiffre des unités de $n$ et $b = \frac{n-a}{10}$. Les nombres $a$ et $b$ devront être de type `int`. Par exemple :

`decomp(8)` vaut `(8,0)`, `decomp(1234)` vaut `(4,123)`.

Dans toute la suite de l'exercice, on s'interdit d'utiliser l'opérateur modulo de Python, la division entière de Python ainsi que la fonction `div`. En revanche vous pouvez utiliser la fonction `decomp`.

**Critère de divisibilité par 2.** Un nombre est divisible par 2 si et seulement si son chiffre des unités est 0, 2, 4, 6 ou 8.

3. À l'aide de ce critère de divisibilité, écrire une fonction `div2` qui prend en entrée un entier $n \in \mathbb{N}$, qui renvoie `True` si $n$ est divisible par 2 et `False` sinon.

**Critère de divisibilité par 3.** Un nombre est divisible par 3 si et seulement si la somme de ses chiffres est divisible par 3. On en déduit la procédure suivante pour tester si un nombre $n_0 \in \mathbb{N}$ est divisible par 3 :

- Si $n_0 \leqslant 9$ alors $n_0$ est divisible par 3 si et seulement si $n_0 \in \{0, 3, 6, 9\}$.
- Sinon, on calcule $n_1$ la somme des chiffres de $n_0$. Si $n_1 \leqslant 9$ alors $n_0$ est divisible par 3 si et seulement si $n_1 \in \{0, 3, 6, 9\}$.
- Sinon, on calcule $n_2$ la somme des chiffres de $n_1$. Si $n_2 \leqslant 9$ alors $n_0$ est divisible par 3 si et seulement si $n_2 \in \{0, 3, 6, 9\}$.
- etc ...

Par exemple, pour tester si 876 477 est divisible par 3 :

- On calcule la somme des chiffres de 876 477 qui vaut 39.
- On calcule la somme des chiffres de 39 qui vaut 12.
- On calcule la somme des chiffres de 12 qui vaut 3.
- On a $3 \leqslant 9$ et $3 \in \{0, 3, 6, 9\}$, donc on conclut que 876 477 est divisible par 3.

4. À l'aide de ce critère de divisibilité, écrire une fonction `div3` qui prend en entrée un entier $n \in \mathbb{N}$, qui renvoie `True` si $n$ est divisible par 3 et `False` sinon.

**Critère de divisibilité par 6.** Un nombre est divisible par 6 si et seulement si il est divisible par 2 et divisible par 3.

5. À l'aide de ce critère de divisibilité, écrire une fonction `div6` qui prend en entrée un entier $n \in \mathbb{N}$, qui renvoie `True` si $n$ est divisible par 6 et `False` sinon.

**Critère de divisibilité par 7.** Soit $n$ un entier. On définit $f(n) = b + 5a$ où $a$ est le chiffre des unités de $n$ et $b = (n - a)/10$. Alors $n$ est divisible par 7 si et seulement si $f(n)$ est divisible par 7. Comme pour le critère de divisibilité par 3, on réutilise ce critère jusqu'à obtenir un nombre inférieur ou égal à 49. Par exemple pour $n = 17625$ :

- $f(17625) = 1762 + 5 \times 5 = 1787$.
- $f(1787) = 178 + 7 \times 5 = 213$.
- $f(213) = 21 + 3 \times 5 = 36$.
- On a $36 \leqslant 49$ et 36 n'est pas divisible par 7. On en conclut que 17625 n'est pas divisible par 7.

6. À l'aide de ce critère de divisibilité, écrire une fonction `div7` qui prend en entrée un entier $n \in \mathbb{N}$, qui renvoie `True` si $n$ est divisible par 7 et `False` sinon.

**Critère rapide de divisibilité par 7.** Pour accélérer le test de divisibilité d'un grand entier $n$ par 7, on décompose $n$ en tranches de trois chiffres et on calcule $s$ la somme alternée de ces tranches. En d'autres termes, on note $a_0$ les trois chiffres les plus à droite de $n$, $a_1$ les trois chiffres suivants, $a_2$ les trois chiffres suivants et ainsi de suite. On pose alors $s = a_0 - a_1 + a_2 - a_3 + \ldots$ Étant donné $s$, l'entier $n$ est divisible par 7 si et seulement si $|s|$ est divisible par 7. On réutilise ensuite ce critère jusqu'à obtenir un nombre inférieur ou égal à 999. Finalement, pour tester si un nombre $n' \leqslant 999$ est divisible par 7, on utilise le critère de divisibilité par 7 précédent.
Par exemple pour $n = 56\,249\,794\,107\,674\,111$, on calcule :

- $|111 - 674 + 107 - 794 + 249 - 56| = |-1057| = 1057$
- $|-1 + 57| = 56$

On enchaîne ensuite avec le critère de divisibilité par 7 précédent :

- $f(56) = 35$.
- On a $35 \leqslant 49$ et 35 est divisible par 7 donc on conclut que $n$ est divisible par 7.

7. À l'aide de ce critère de divisibilité, écrire une fonction `div7Rapide` qui prend en entrée un entier $n \in \mathbb{N}$, qui renvoie `True` si $n$ est divisible par 7 et `False` sinon.

**Test des fonctions.** On souhaite maintenant tester les fonctions écrites dans cet exercice.

8. Écrire une fonction `CenMilleTests` (sans argument) qui choisit cent-mille nombres aléatoires dans l'intervalle $[\![0, 10^{20}]\!]$, et qui pour chacun de ces nombres vérifie les valeurs `div2(n)`, `div3(n)`, `div6(n)`, `div7(n)` et `div7Rapide(n)` à l'aide de la fonction `div`. Si toutes les valeurs sont cohérentes, votre fonction renverra `True`, sinon elle renverra `False`. Pour choisir les nombres aléatoires, utilisez la fonction `randint` du module `random` qui prend en entrée deux entiers $(a, b)$, et renvoie un nombre aléatoire de l'intervalle $[\![a, b]\!]$.

## Exercice 5. Temps de recherche d'une clé dans un dictionnaire

Le but de cet exercice est de mettre en évidence le fait que la recherche d'une clé dans un dictionnaire est beaucoup plus rapide que la recherche d'un élément dans une liste.

**Rappels de cours :**

→ La syntaxe « `if c in d` » pour tester l'appartenance d'une clé `c` à un dictionnaire `d` s'exécute en temps constant, c'est à dire que le temps d'exécution ne dépend pas du nombre d'éléments dans `d`.

→ Lorsqu'on veut rechercher un élément `e` dans une liste `L`, on doit examiner tous les éléments qui précèdent `e` dans `L`. Ainsi, une telle recherche s'exécute en temps linéaire en $i$ où $i$ est l'indice de `e` dans `L`.

→ En Python, on peut tester l'appartenance d'un élément `e` à une liste `L` grâce à la syntaxe « `if e in L` ». Toutefois, le temps d'exécution n'est pas constant, il est linéaire en $i$ où $i$ est l'indice de `e` dans `L`. De plus, cette syntaxe n'étant pas au programme de MPSI, évitez de l'utiliser le jour du DS ou du concours.

→ Il n'est pas possible de tester l'appartenance d'une valeur à un dictionnaire en temps constant (contrairement aux clés). En particulier, la commande « `if v in d.values()` » s'exécute en temps linéaire en `len(d)`.

**Énoncé de l'exercice.** Soit $n \in \mathbb{N}$ un entier. On commence par générer $n$ chaînes de caractères contenant des mots de la langue française. Pour cela, téléchargez le fichier `TP06-dico.txt` disponible sur la page du cours et définissez `L_dico` à l'aide du programme ci-dessous :

```
fichier = open("TP06-dico.txt", mode = 'r', encoding = "utf8")
L_dico = [s.strip() for s in fichier.readlines()]
fichier.close()
```

Puisque la variable `L_dico` est globale, vous pouvez l'utiliser dans une fonction `f` sans la donner en argument.

1. (a) Écrire une fonction « `make_dict(n: int) -> dict[str: int]` » qui renvoie un dictionnaire `d` de taille `n` tel que :
   - Les clés de `d` sont des chaînes de caractères choisies aléatoirement dans `L_dico`. On utilisera la fonction « `random.randint(a: int, b: int) -> int` » qui renvoie un entier aléatoire de l'intervalle $[\![$`a; b`$]\!]$.
   - La valeur associée à une clé `c` est l'indice de `c` dans `L_dico`. En d'autres termes, pour toute clé `c` de `d`, on doit avoir `L_dico[d[c]] = c`.

   Par exemple, avec `n = 5`, on peut obtenir :

   ```
   {'réparer': 18579, 'qualités': 17559, 'contrefait': 5259,
    'mécontents': 13675, 'Suédois': 20631}
   ```

   (b) Dans la question précédente, avez-vous pensé à traiter le cas où deux chaînes de caractères ajoutées dans le dictionnaire sont identiques ? Si ce n'est pas le cas, modifiez votre code.

2. Supposons qu'un dictionnaire `d` ait été généré par un appel à `make_dict(n)`. Écrire une fonction « `dict_to_list(d: dict[str: int]) -> list[str, int]` » qui renvoie une liste `L` de taille `n` contenant tous les couples `(c,v)` où `c` est une clé de `d` et `v = d[c]`. Par exemple, avec le dictionnaire ci-dessus, on obtient :

   ```
   [('réparer', 18579), ('qualités', 17559),
    ('contrefait', 5259), ('mécontents', 13675), ('Suédois', 20631)]
   ```

Soit `s` une chaîne de caractères. On souhaite comparer le temps nécessaire pour rechercher `s` dans une liste et dans un dictionnaire.

3. Dans cette question, on suppose que le dictionnaire `d` a été généré par un appel à `make_dict(n)` et que la liste `L` a été générée par un appel à `dict_to_list(d)`.
   (a) Écrire une fonction « `find_list(s: str, L: list[str, int]) -> int` » qui renvoie l'entier associé à `s` dans `L`. Si `s` n'apparaît pas dans `L`, votre fonction renverra `None`. Avec l'exemple ci-dessus `find_list("contrefait", L)` vaut `5259`.
   (b) Écrire une fonction « `find_dict(s: str, d: dict[str: int]) -> int` » qui renvoie la valeur associée à `s` dans `d`. Vous devez rechercher `s` directement dans le dictionnaire, sans passer par une liste. Si `s` n'est pas une clé de `d`, votre fonction renverra `None`. Avec l'exemple ci-dessus `find_dict("contrefait", d)` vaut `5259`.

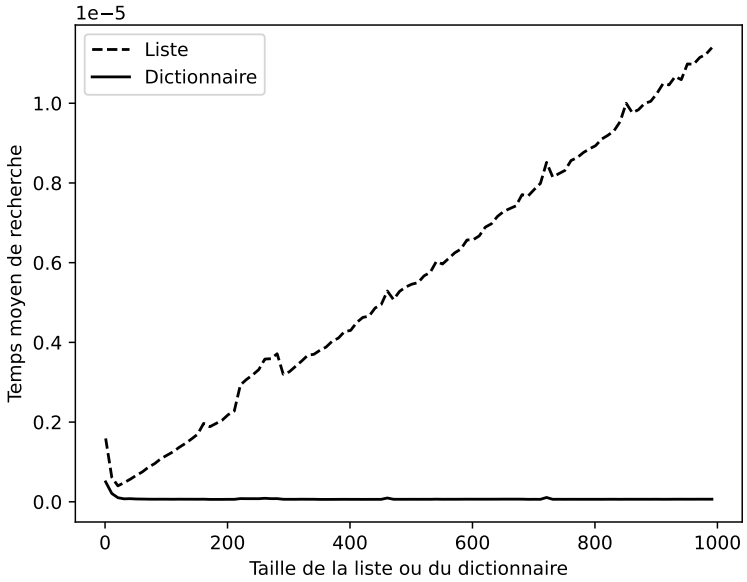
Pour terminer, on va comparer le temps d'exécution des fonctions `find_list` et `find_dict`. La fonction « `perf_counter() -> float` » du module `time` renvoie le nombre de secondes écoulées depuis le démarrage de l'ordinateur. Pour obtenir le temps d'exécution d'un programme, on peut donc écrire :

```
tps_ini = time.perf_counter()
# Programme dont on veut mesurer le temps d'exécution
tps_exec = time.perf_counter() - tps_ini
```

4. À l'aide de tests pertinents, montrer que le temps d'exécution de la fonction `find_dict` est beaucoup plus court que celui de la fonction `find_list`. Bien sûr, comme ce qui nous intéresse est le temps d'exécution de `find_dict` et `find_list`, vous ne devez pas mesurer le temps consacré à la création du dictionnaire et de la liste. Lorsque $n$ augmente, l'écart entre les deux temps d'exécution doit également augmenter (voir le graphique ci-dessous).

**Question facultative :**

5. À l'aide du module `matplotlib`, écrire une fonction « `tracer() -> NoneType` » qui trace le temps nécessaire pour rechercher un élément (resp. une clé) dans une liste (resp. un dictionnaire) de taille $n$ en fonction de $n$. Puisque la complexité est en $\mathcal{O}(n)$ pour les listes et en $\mathcal{O}(1)$ pour les dictionnaires, vous devez obtenir un graphique qui ressemble à :




On pourra s'inspirer du programme ci-contre et utiliser l'aide en ligne du module `matplotlib.pyplot`.

```
import matplotlib.pyplot as plt
def parabole():
    X = [x/100 for x in range(-100, 101)]
    Y = [x**2 for x in X]
    plt.figure()
    plt.plot(X ,Y)
    plt.show()
```

## Exercice 6. Recherche du second maximum dans une liste

Soit `L` une liste d'entiers. On souhaite trouver le second maximum de `L`. Par exemple, le second maximum de `[1,5,2]` est `2`, le second maximum de `[1,5,5]` est `5` et le second maximum de `[2,6,4,6]` est `6`. En particulier, si le maximum apparaît plusieurs fois dans `L`, on considère que le second maximum est égal au maximum.

1. Écrire une fonction « `snd_max(L: list[int]) -> int` » qui renvoie le second maximum de `L`. Dans le cas où il n'y a pas de second maximum, votre fonction déclenchera une erreur.

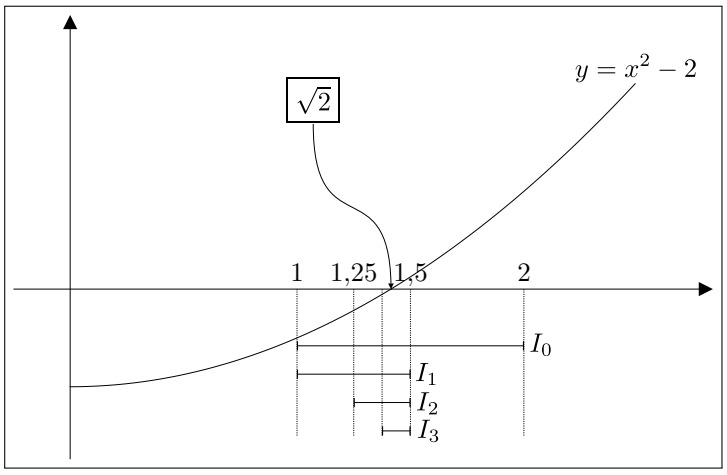
## Exercice 7. Nombres opposés

Écrire une fonction « `nb_opp(L: list[int]) -> int` » qui prend en entrée une liste `L` d'entiers non nuls et renvoie le nombre d'entiers strictement positifs $x$ tels que $x$ et $-x$ apparaissent dans `L`. Par exemple, la fonction renvoie `2` pour la liste `[-4,5,3,2,4,4,-2,5,-4]`. En effet, les entiers strictement positifs `2` et `4` ainsi que leurs opposés apparaissent dans la liste. **La complexité de votre fonction devra être linéaire en la taille de `L`.** Si vous n'y arrivez pas, vous pouvez lire les indications qui suivent.

**Indications (essayez de faire l'exercice sans lire ce qui suit).** On pourra créer un dictionnaire `d` dont l'ensemble des clés est égal à l'ensemble des éléments des `L` (les valeurs de `d` sont quelconques). Ainsi, grâce au dictionnaire `d`, pour chaque élément `e` de `L`, on pourra tester en temps constant si `-e` est un élément de `L`. Pensez à organiser votre code avec des fonctions intermédiaires.

## Exercice 8. Approximation de $\sqrt{2}$ par dichotomie

Soit $\varepsilon > 0$ un réel. Notre but est de donner une approximation de $\sqrt{2}$ à $\varepsilon$-près, c'est à dire de calculer un nombre $x$ tel que $|x - \sqrt{2}| \leqslant \varepsilon$. La difficulté est que les seules opérations autorisées sont l'addition, la soustraction, la multiplication et la division (en particulier, vous ne pouvez pas utiliser l'opérateur `**0.5` de Python ou la fonction `sqrt` du module `math`). Pour cela, on va construire des intervalles $I_0, I_1, I_2, \ldots$ tels que pour tout $k \in \mathbb{N}$, on ait $\sqrt{2} \in I_k$ et $I_{k+1}$ soit au moins deux fois plus petit que $I_k$.




Soit $f$ la fonction $x \mapsto x^2 - 2$. Les intervalles $I_k$ sont définis par récurrence :

→ On pose $I_0 = [\,1\,;2\,]$.

→ Soit $k \in \mathbb{N}$. Supposons que l'intervalle $I_k$ ait été construit et construisons l'intervalle $I_{k+1}$. Pour cela, on note $a_k$ et $b_k$ les bornes de $I_k$, c'est à dire que $I_k = [\,a_k\,;b_k\,]$. Comme $\sqrt{2} \in I_k$, on a :

$$f(a_k) \leqslant 0 \qquad \text{et} \qquad f(b_k) \geqslant 0.$$

On pose $m_k = \dfrac{a_k + b_k}{2}$, alors :

- Si $f(m_k) = 0$, $I_{k+1}$ est défini par $I_{k+1} = [\,m_k\,;m_k\,]$.
- Si $f(m_k) > 0$, comme $f(a_k) \leqslant 0$ et que la fonction $f$ est continue, elle s'annule sur l'intervalle $[\,a_k\,;m_k\,]$. On pose donc $I_{k+1} = [\,a_k\,;m_k\,]$ ce qui garantit que $\sqrt{2} \in I_{k+1}$.
- Si $f(m_k) < 0$, comme $f(b_k) \geqslant 0$ et que la fonction $f$ est continue, elle s'annule sur l'intervalle $[\,m_k\,;b_k\,]$. On pose donc $I_{k+1} = [\,m_k\,;b_k\,]$ ce qui garantit que $\sqrt{2} \in I_{k+1}$.

Soit $k \in \mathbb{N}$ le plus petit entier tel que l'intervalle $I_k = [\, a_k \,;\, b_k \,]$ soit de taille inférieure ou égale à $2\varepsilon$ ; alors le réel $x_0 = \dfrac{a_k + b_k}{2}$ est une approximation de $\sqrt{2}$ à $\varepsilon$-près.

1. Écrire une fonction `approx_sqrt2` qui prend en entrée un flottant `epsilon` et renvoie le réel $x_0$ défini ci-dessus.
2. **Facultatif.** Donner en le justifiant le nombre exact de tours de boucle en fonction de $\varepsilon$. On pourra supposer que la condition $f(m_k) = 0$ n'est jamais vérifiée. On attend une réponse sous la forme d'une formule mathématiques, pas d'un programme.

## Exercice 9. Rallonges audio (exercice facultatif)

Cet exercice est issu du site France-ioi :

`http://www.france-ioi.org/algo/task.php?idChapter=761&idTask=874`

Vous souhaitez raccorder votre nouvel ordinateur à l'amplificateur de votre super chaîne hi-fi afin de pouvoir mettre à fond l'album "Wish You Were Here" (du groupe Pink Floyd). Vous espérez que la mélodie des guitares électriques fera venir en vous l'inspiration nécessaire pour résoudre des sujets d'algorithmique.

En même temps, vous souhaitez vous rapprocher le plus possible de la borne wifi par laquelle vous accédez à Internet afin de pouvoir télécharger tous les autres albums des Pink Floyd avec un débit maximal (téléchargé légalement, bien sûr). Cette borne wifi est commune à votre immeuble et peut théoriquement vous fournir une très bonne bande passante. Malheureusement, elle se situe un peu loin de chez vous et cela est un facteur limitant votre débit. Vous avez déjà remarqué qu'il vous était possible d'améliorer le débit rien qu'en vous déplaçant de quelques mètres en direction de la borne wifi.

Votre but va donc être de combiner les rallonges audio dont vous disposez afin d'obtenir un maximum de longueur, et pouvoir ainsi écouter de la musique tout en étant aussi près que possible de la borne wifi.

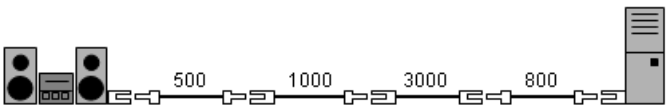
Vous avez à votre disposition un certain nombre de rallonges. Chaque rallonge est caractérisée par sa longueur, et par la nature des connecteurs à ses deux extrémités : chaque connecteur peut être soit mâle, soit femelle. Remarque technique : on ne peut pas connecter des fiches mâles entre elles, ni deux fiches femelles entre elles. Notez enfin que la carte son de votre ordinateur ainsi que l'amplificateur de votre chaîne hi-fi portent tous deux des connecteurs femelles (il faut donc arriver dessus avec des connecteurs mâles).

**Entrée.** Votre fonction `dist_max` prend en entrée une liste dont chaque élément est un triplet $(C_1, C_2, L)$. $C_1$ et $C_2$ décrivent les connecteurs (1 pour mâle, 0 pour femelle), et $L$ donne la longueur de la rallonge. Notez que le signal peut passer dans une rallonge dans le sens que l'on veut. Ainsi le signal peut traverser la rallonge de $C_1$ vers $C_2$ ou de $C_2$ vers $C_1$.

**Sortie.** Vous devez renvoyer un seul entier : la distance maximale à laquelle vous pouvez placer votre ordinateur de la chaîne hi-fi tout en lui envoyant le signal musical à travers les rallonges. S'il est impossible de relier les deux appareils avec les rallonges dont vous disposez, renvoyez -1.

**Exemple.** Entrée : `[(1,0,1000), (1,1,500), (0,0,2000), (1,1,800), (0,0,3000)]`
Sortie : `5300`

**Commentaires.** Utilisez la séquence suivante par exemple :




Notez que la seconde rallonge a été retournée par rapport à sa description dans l'entrée. La longueur totale obtenue est `500 + 1000 + 3000 + 800 = 5300`. Vous pouvez utiliser la commande `L.sort()` qui permet de trier une liste `L` et la commande `sum(L)` qui renvoie la somme des éléments de `L`.