

Correction et complexité d'un algorithme

Exercice 1 *Invariants de boucle*

Les problèmes suivants prennent en entrée une liste contenant des entiers.

Pour chaque problème, écrire un algorithme en Python et démontrer sa correction en se basant sur un invariant de boucle.

1. Retourner la somme des éléments de la liste.
2. Retourner le maximum de la liste.
3. Retourner la position du dernier élément non nul de la liste, ou 0 si tous les éléments sont nuls.

Exercice 2 *Opérations sur les matrices*

Dans cet exercice les matrices sont codées par des listes de listes. On suppose qu'on dispose d'une fonction `matricenulle(n, p)` qui retourne une matrice remplie de 0 de taille $n \times p$ où n et p sont donnés en entrée, de complexité $O(np)$. On suppose que les opérations arithmétiques sur les éléments de ces matrices se font en temps constant.

1. Écrire un algorithme calculant la somme de deux matrices de même taille et donner la complexité en fonction du nombre de lignes et de colonnes de ces matrices.
2. Écrire un algorithme calculant le produit de deux matrices de tailles compatibles et donner la complexité en fonction du nombre de lignes et de colonnes de ces matrices.
3. Écrire un algorithme pour tester si une matrice carrée donnée en entrée est triangulaire supérieure. Démontrer la correction de votre algorithme et donner sa complexité.

Exercice 3 *Écriture d'un nombre dans une base*

Dans cet exercice $B \geq 2$ est un entier naturel fixé.

1. Montrer qu'un entier strictement positif N s'écrit de manière unique

$$N = a_n B^n + a_{n-1} B^{n-1} + \dots + a_0 B^0$$

avec $a_i \in \{0, \dots, B-1\}$ pour tout i et $a_n \neq 0$.

On appelle écriture en base B de l'entier N la suite (a_n, \dots, a_0) . On dira que le codage de l'entier N en base B est de taille $n+1$.

L'écriture d'un nombre en base 2 est ce qu'on appelle l'écriture binaire.

2. Écrire en binaire les entiers 5, 12 et 23.
3. Quelle est l'écriture en binaire de 2^p ? Et $2^p - 1$?

On s'intéresse maintenant à la taille de l'écriture en base B d'un entier N .

4. Si N s'écrit (a_n, \dots, a_0) en base B , montrer que

$$B^n \leq N < B^{n+1}.$$

En déduire que $n + 1 = \lfloor \log_B N \rfloor + 1$ (on rappelle que $\log_B N = \frac{\log N}{\log B}$).

Notons $\ell(N)$ la taille de l'écriture en base B de N . D'après ce qu'on a montré ci-dessus, il existe des constantes $\alpha, \beta > 0$ tel que pour tout N assez grand,

$$\alpha \cdot \log N \leq \ell(N) \leq \beta \cdot \log N,$$

ou de manière équivalente il existe $\gamma, \delta > 0$ tel que

$$\gamma \cdot \ell(N) \leq \log N \leq \delta \cdot \ell(N).$$

Ainsi, $\log N$ et la taille en base B de N sont du même ordre de grandeur.

Exercice 4 *Tester si un nombre est premier*

On rappelle qu'un entier $N > 1$ est premier s'il n'a pas de diviseur autre que 1 et lui-même.

Dans cet exercice on suppose qu'on dispose des opérations arithmétiques usuelles sur des entiers arbitrairement grands.

1. Écrire un algorithme qui, en testant successivement tous les diviseurs potentiels, renvoie *vrai* si un nombre N donné en entrée est premier et *faux* sinon.
2. Montrer que votre algorithme est correct.
3. Évaluer la complexité de votre algorithme : on comptera le nombre d'opérations arithmétiques qu'on exprimera en fonction de N .
4. Exprimer la complexité de l'algorithme ci-dessus en fonction de la taille du codage en binaire de l'entier N . L'algorithme est-il polynomial en la taille du codage de N ?
5. Écrire une variante de l'algorithme qui ne teste les diviseurs que jusqu'à environ \sqrt{N} . Démontrer la correction et évaluer la complexité de cet algorithme modifié.
6. En pratique, les opérations arithmétiques ne se font pas en temps constant sur des entiers arbitrairement grands. Si le coût d'une addition ou d'un test sur des entiers de taille n est $O(n)$, et le coût d'une multiplication ou d'une division est $\tau(n) = O(n \log n \log \log n)$, exprimer le coût total du test de primalité ci-dessus.

Exercice 5 *Détection d'un vainqueur dans un tournoi*

On considère un tournoi, c'est-à-dire le résultat de matchs entre toutes les paires de joueurs. On note $\{0, \dots, n-1\}$ l'ensemble des joueurs. Pour chaque paire $\{i, j\}$ avec $i \neq j$, le joueur i bat le joueur j ou le joueur j bat le joueur i (il n'y a pas de match nul). Un joueur i est vainqueur s'il a battu tous les autres joueurs.

1. Montrer qu'il existe au plus un vainqueur dans un tournoi.
2. Pour tout $n \geq 3$, donner un exemple de tournoi sans vainqueur.

On peut coder les résultats d'un tournoi à n joueurs dans une matrice M de taille $n \times n$ de la façon suivante : pour $i \neq j$, $M_{i,j} = 1$ si le joueur i bat le joueur j , et $M_{i,j} = -1$ sinon. Et pour tout $i \in \{0, \dots, n-1\}$, $M_{i,i} = 0$.

Dans la suite, le but est d'écrire un algorithme qui étant donné la matrice M d'un tournoi, retourne i si i est vainqueur du tournoi et -1 s'il n'y a pas de vainqueur.

3. Écrire un algorithme qui prend en entrée un tournoi donné par la matrice décrite ci-dessus et retourne le vainqueur s'il en existe un, et -1 sinon.
4. Quel est la complexité de votre algorithme (en fonction de n). Quelle est la taille de l'entrée? Comment qualifier la complexité de votre algorithme?

Un algorithme a une complexité *sous-linéaire* si le nombre d'étapes de calcul $C(\ell)$ sur les entrées de taille ℓ (dans le pire cas) vérifie $\frac{C(\ell)}{\ell} \rightarrow 0$ quand $\ell \rightarrow \infty$.

5. Écrire un algorithme sous-linéaire qui retourne le vainqueur d'un tournoi (ou -1 s'il n'y a pas de vainqueur). On montrera la correction de cet algorithme.

Il n'existe pas toujours d'algorithme sous-linéaire comme le montre l'exemple suivant.

6. Montrer qu'il n'existe pas d'algorithme sous-linéaire pour décider si un tableau T d'éléments dans $\{0, 1\}$ contient au moins un élément égal à 1.

Exercice 6 Recherche d'un élément majoritaire

On dit qu'un élément x est majoritaire dans une liste L si son nombre d'occurrences dans L est strictement supérieur à $n/2$ où n est la longueur de la liste L . Par exemple, 5 est majoritaire dans la liste $[5, 3, 3, 5, 5, 5, 1]$ car 5 apparaît 4 fois dans ce tableau longueur 7.

Bien sûr si une liste L possède un élément majoritaire il est unique, mais toute liste ne possède pas d'élément majoritaire : c'est le cas par exemple de la liste $[1, 2, 3]$.

On dira qu'un algorithme calcule l'élément majoritaire s'il renvoie l'élément majoritaire s'il existe, et rien sinon (on suppose que la liste ne contient pas l'élément `None`).

On propose l'algorithme suivant `majoritaire(L)` pour le calcul de l'élément majoritaire :

```
def nombre_occurences(L, x):
    c=0
    for y in L:
        if y == x:
            c = c+1
    return c

def majoritaire(L):
    m = None
    v = 0
    for x in L:
        if v == 0:
            m = x
            v = 1
        elif m == x:
            v = v+1
        else:
            v = v-1
    if nombre_occurences(L, m) > len(L)//2:
        return m
    return None
```

Le but de cet exercice est de démontrer la correction de cet algorithme, et de déterminer sa complexité.

Déroulement de l'algorithme sur un exemple

On s'intéresse à l'exécution de la fonction `majoritaire` sur la liste `[5, 3, 3, 5, 5, 1, 5]`.

1. Donner les valeurs des variables m et v après chaque itération de la boucle principale.
2. Que retourne l'algorithme sur cette liste ?

Correction

3. Montrer que l'algorithme `majoritaire(L)` retourne `None` si la liste L n'a pas d'élément majoritaire.
4. On suppose maintenant que L possède un élément majoritaire M . À n'importe quel moment de l'algorithme (par exemple début ou fin d'itération de la boucle principale de la procédure principale) définissons

$$w = \begin{cases} v & \text{si } m = M \\ -v & \text{si } m \neq M \end{cases}$$

Montrer qu'après la dernière itération de la boucle `for` on a $w > 0$.

En déduire que si L a un élément majoritaire M , alors $m = M$ après la boucle `for`.

5. Conclure que l'algorithme proposé est correct.

Complexité

6. Déterminer la complexité de cet algorithme de calcul de l'élément majoritaire.