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.

Correction 1

1. On propose l'algorithme suivant :

```
def somme(T):
  n = len(T)
  s = 0
  for i in range(n):
    s = s + T[i]
  return s
```

On montre la correction en se basant sur l'invariant de boucle suivant :

"Au début de l'itération i : $s = \sum_{j<i} T[j]$."

— C'est vrai pour $i = 0$ car s est initialisé à 0 ;

— Par récurrence, on vérifie que l'invariant de boucle est bien vérifié à chaque étape ;

— Au début de l'itération $i = n - 1$ on a donc $s = \sum_{j=0}^{n-2} T[j]$ et après la dernière itération s contient bien la somme des éléments de T. L'algorithme est correct.

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 $B \geqslant 2$ est un entier naturel fixé.

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

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

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

On appelle écriture en base B de l'entier N la suite $(a_n, \ldots, a_0)$. 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 $(a_n, \ldots, a_0)$ en base B, montrer que

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

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,

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

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

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

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 arith–métiques 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.

Correction 4

3. On effectue au plus N opérations arithmétiques.
4. Soit n la taille du codage en binaire de l'entier N. On a vu dans l'exercice précédent que $N < 2^{n+1}$, on effectue donc au plus 2^{n+1} opérations arithmétiques. Si on considère que ces opérations sont en $O(1)$, alors la complexité est en $O(2^n)$. Cette borne est optimale, puisque pour un nombre premier on va devoir tester tous les diviseurs et qu'il existe des nombres premiers arbitrairement grands. Or pour tout k, la fonction $n \mapsto 2^n$ n'est pas un $O(n^k)$, donc la complexité n'est pas polynomiale en la taille du codage.
5. Si $N = ij$, alors $i \leqslant \sqrt{N}$ ou $j \leqslant \sqrt{N}$. En effet dans le cas contraire on aurait $N = ij > (\sqrt{N})^2 = N$ ce qui est absurde. Ceci implique qu'il suffit de tester les diviseurs $\leq \sqrt{N}$. Le nombre d'opérations sera alors en $O(\sqrt{N}) = O(\sqrt{2^n}) = O(2^{n/2})$.
6. On aurait un coût en $O(n \log n \log\log n 2^{n/2})$, en particulier en $O(2^{\alpha n})$ pour tout $\alpha > 1/2$ (car alors $n \log n \log\log n = O(2^{(\alpha - \frac{1}{2})n})$).

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 $\{0, \ldots, n-1\}$ 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 $n \geqslant 3$, 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 $i \in \{0, \ldots, n-1\}$, $M_{i,i} = 0$.

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 $\frac{C(\ell)}{\ell} \to 0$ quand $\ell \to \infty$.

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.

Correction 5

5. *Indication :* L'algorithme ci-dessous retourne l'unique vainqueur potentiel du tournoi.
   Voir la suite des valeurs de (i, j) comme un chemin dans la matrice partant du coin supérieur gauche. Remarquer qu'on reste dans la partie triangulaire supérieure, donc la boucle se termine bien avec $i = i_0$ et $j = n$.
   — Un joueur $i < i_0$ ne peut être vainqueur car on a fait un déplacement de type $(i, k) \to (i + 1, k)$. Donc i est battu par au moins un joueur.
   — Un joueur d'indice $j > i_0$ ne peut être vainqueur car on a fait un déplacement de type $(k, j) \to (k, j + 1)$ (avec nécessairement $k \neq j$). Le joueur j est donc battu par au moins un joueur.

```
def candidat(T):
  n = len(T)
  i = 0
  j = 0
  while j < n:
    if T[i][j] >= 0:
      j = j + 1
    else:
      i = i + 1
  return i
```

   Il suffit ensuite de tester le joueur retourné par la fonction *candidat* bat tous les autres.
   La complexité totale est $O(n) = O(\sqrt{\ell(M)})$ où $\ell(M) = n^2$ désigne la taille de l'entrée. L'algorithme est bien sous-linéaire.

6. Supposons qu'il existe un algorithme correct en temps sous-linéaire. Sur l'entrée composée de n zéros il renvoie *faux*. En temps sous-linéaire il ne peut pas inspecter toutes les cases du tableau (pour n assez grand). En changeant l'entrée pour mettre 1 à la place de 0 dans une case non inspectée, l'algorithme retourne encore *faux*. Il n'est donc pas correct.

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
```

```python
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.

Correction 6

1. Initialement, $m = None$ et $v = 0$. Après chacune des étapes, on a les valeurs suivantes :

étape	élément traité	m	v
1	5	5	1
2	3	5	0
3	3	3	1
4	5	3	0
5	5	5	1
6	1	5	0
7	5	5	1

2. À la dernière étape, l'algorithme calcule le nombre d'occurrences de $m = 5$ dans la liste. Comme il y a 4 occurrences et que $4 > \lfloor 7/2 \rfloor$, l'algorithme retourne 5.
3. Si la liste ne contient pas d'élément majoritaire, le nombre d'occurrences de m à la fin ne peut être strictement plus grand que $\lfloor n/2 \rfloor$ et l'algorithme retourne *null*.
4. Pour $k \in \{0, \ldots, n\}$ notons

$$\delta_k = |\{i \leqslant k \ : \ T[i] = M\}| - |\{i \leqslant k \ : \ T[i] \neq M\}|.$$

   Notons $w_0, \ldots, w_{n-1}$ les valeurs de w au début des différentes itérations de la boucle et w_n la valeur de w après la dernière itération.

   On montre par récurrence que $w_k \geqslant \delta_k$. Pour $k = 0$ c'est clair car $w_0 = \delta_0 = 0$. Étape de récurrence :

   — Si $T[k] = M$, δ est incrémenté et w aussi (distinguer les deux cas $m = M$ et $m \neq M$) ;

   — Si $T[k] \neq M$, δ est décrémenté et w est incrémenté ou décrémenté.

   Comme $\delta_n > 0$ car M majoritaire, on en déduit $w_n > 0$. Ceci implique que $m = M$ à la fin de la boucle car si on avait $m \neq M$, on aurait $w_n \leqslant 0$ (car $v \geqslant 0$ tout au long de l'algorithme, en particulier à la fin).
5. Si la liste ne possède pas d'élément majoritaire alors l'algorithme renvoie la bonne réponse d'après la question 3.

   Si la liste possède un élément majoritaire M alors la valeur de m à la fin de la boucle est M d'après la question 4. Comme le nombre d'occurrences de $m = M$ est strictement supérieur à $\lfloor n/2 \rfloor$, l'algorithme retourne M, ce qui est correct.
6. La boucle `for` de la fonction `majoritaire` a un coût $O(n)$. Le calcul du nombre d'occurrences a aussi un coût $O(n)$. En tout, la complexité de cet algorithme est $O(n) + O(n) = O(n)$.