INF 109 : DEVOIR MAISON n° 1

Consignes. — Ce devoir porte sur le chapitre 1. Les copies réponses ne sont ni ramassées, ni notées. Vous pouvez demander de l'aide lors des heures de permanence (*office hours*) ou en discuter avec vos camarades de classe. Des éléments de ce devoir pourront être repris dans l'examen final.

Conseils. — Il est recommandé d'implémenter les algorithmes et de jouer avec pour se donner une intuition de leur fonctionnement. Il est recommandé de rédiger les réponses entièrement pour s'entraîner à la rédaction d'arguments informatiques et de ne pas se contenter de chercher les solutions approximativement et au brouillon.

1.1. Les grains de café de David Gries

Une boîte de café contient N grains noirs et B grains blancs ($N + B \geq 2$). On répète le processus suivant aussi longtemps que possible.

> Sélectionner au hasard deux grains dans la boîte.
> S'ils ont la même couleur, les jeter, mais mettre un autre grain noir dedans.
> (On dispose d'un stock illimité de grains noirs.)
> S'ils sont de couleurs différentes, remettre le grain blanc dans la boîte et jeter le grain noir.

1. Écrire un programme Python qui simule le processus.
   On pourra utiliser la fonction `randint` du module `random`.
2. Démontrer que l'algorithme s'arête.
3. Donner une spécification de l'algorithme. La démontrer.

1.2. Élément majoritaire dans un tableau et élection de Boyer-Moore

Soit $\mathscr{T}$ un multi-ensemble à n éléments, c'est-à-dire une collection non ordonnée de n éléments qui peuvent se répéter. Nous disons que le multi-ensemble $\mathscr{T}$ possède

un élément majoritaire s'il existe un élément $x \in \mathscr{T}$ dont la multiplicité $\mathsf{mult}(x, \mathscr{T})$ est strictement supérieure à $\dfrac{n}{2}$.

Nous nous intéressons au problème de décision (Π) suivant :

Entrée : Tableau T possédant n cases ($n > 0$).
Sortie : Booléen $b \in \mathbb{B} = \{\mathsf{Vrai}, \mathsf{Faux}\}$.
Postcondition : La valeur de vérité de b vaut Vrai si le multiensemble

$$\mathscr{T} = \{T[x]; x \in [\![0, n-1]\!]\}$$

admet un élément majoritaire et elle vaut Faux sinon.

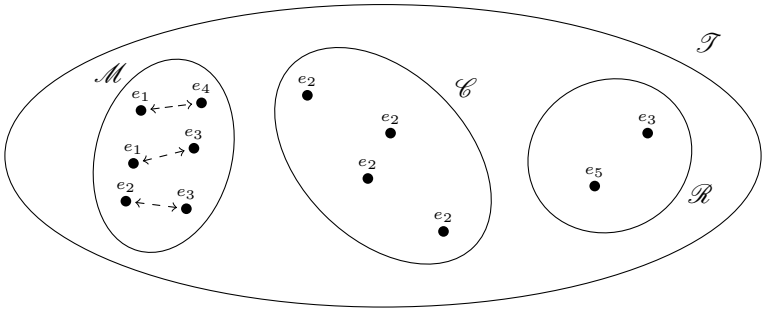
Usuellement, nous introduisons des invariants après avoir écrit l'algorithme pour en justifier la correction. Dans cet exercice, nous inversons la perspective : nous partons d'invariants pour deviner un algorithme efficace.

1. Quelle serait la complexité d'un algorithme résolvant (Π) par force brute ?
2. En supposant que l'on peut préalablement trier les éléments de T, donner un algorithme résolvant (Π) ? Quelle est la complexité (tri inclus) de cet algorithme ?

Nous souhaitons identifier un *élément candidat* à être l'élément majoritaire dans un multi-ensemble $\mathscr{T}$. Nous notons (✠) la propriété suivante : le multi-ensemble $\mathscr{T}$ est divisé en trois groupes dont $\mathscr{T}$ est la réunion :

— le multi-ensemble $\mathscr{M}$ des éléments « mariés » : dans $\mathscr{M}$, les éléments peuvent être chacun appariés à un autre élément au sein d'une paire où les composantes sont l'une distincte de l'autre ;
— le multi-ensemble $\mathscr{C}$ des éléments « célibataires » : soit $\mathscr{C}$ est vide, soit $\mathscr{C}$ ne contient qu'un seul élément de multiplicité quelconque ;
— le multi-ensemble $\mathscr{R}$ des éléments « restants ».

Voici un exemple, avec $\mathscr{T} = \{e_1, e_1, e_2, e_2, e_2, e_2, e_2, e_3, e_3, e_3, e_4, e_5\}$.



3. Dans cette question, on suppose que $\mathscr{T}$ est divisé comme dans (✠) et que $\mathscr{R}$ et $\mathscr{C}$ sont les multi-ensembles vides. Montrer que $\mathscr{T}$ ne possède pas d'élément majoritaire.
4. Montrer qu'un multi-ensemble $\mathscr{T}$ ne possède pas plus d'un élément majoritaire.
5. Dans cette question, on suppose $\mathscr{T}$ est divisé comme dans (✠), que $\mathscr{R}$ est vide et que $\mathscr{C}$ ne l'est pas. Notons c l'élément répété dans $\mathscr{C}$. Montrer que, si $\mathscr{T}$ possède un élément majoritaire, il s'agit forcément de c.

6. Donner les détails d'un algorithme constitué d'une boucle « tant que » dont un invariant est donné par l'existence la décomposition ci-dessus et dont un variant est le cardinal $|\mathscr{R}|$.

Nous supposons avoir écrit un programme `candidat(T)` dont la valeur de retour c est soit non définie (valeur `None`) soit un élément du tableau T. Postcondition : s'il existe un élément majoritaire dans T, alors il vaut c ; si la valeur de retour est `None`, il n'y a pas d'élément majoritaire dans T.

7. Donner le principe d'un algorithme aussi simple que possible qui résout le problème de décision (Π) et le programmer en Python. Quel en est la complexité en temps ?

1.3. Le voyageur dans le désert

Un voyageur veut aller d'une oasis à une autre sans mourir de soif. Il connaît la position des puits sur la route (numérotés de 1 à n, le puits n° 1 (resp. n) étant l'oasis de départ (resp. d'arrivée)). Le voyageur sait qu'il consomme exactement un litre d'eau au kilomètre. Il est muni d'une gourde pleine à son départ. Quand il arrive à un puits, il choisit entre deux possibilités : a) poursuivre sa route, ou b) remplir sa gourde. S'il fait le second choix, il vide sa gourde dans le sable avant de la remplir entièrement au puits afin d'avoir de l'eau fraîche. À l'arrivée, il vide la gourde.

1. Le voyageur veut faire le moins d'arrêts possible. Mettre en évidence une stratégie gloutonne optimale atteignant cet objectif.
2. Le voyageur veut verser dans le sable le moins de litres d'eau possible. Montrer que la stratégie gloutonne précédente est toujours optimale.
3. À chaque puits, y compris celui de l'oasis d'arrivée, un gardien lui fait payer autant d'unités de la monnaie locale que le carré du nombre de litres d'eau qu'il vient de verser à l'arrivée du tronçon qu'il a parcouru. Le problème est de choisir les puits où il doit s'arrêter pour payer le moins possible. Montrer avec les données de l'exemple ci-dessous que la stratégie gloutonne précédente n'est plus optimale.

   La gourde contient jusqu'à dix litres, les puits sont situés à 8, 9, 16, 18, 24 et 27 km de l'oasis de départ, l'arrivée est située à 32 km de l'oasis de départ.
4. Construire une solution fondée sur la programmation dynamique dont les éléments sont :
   - $\sigma(i)$: somme minimale payée au total depuis le puits n° 1 (l'oasis de départ) jusqu'au puits n° i, étant donné que le voyageur vide sa gourde au puits n° i
   - $d(i, j)$: nombre de kilomètres entre le puits n° i et le puits n° j
   - D : volume de la gourde

   dont on donnera la récurrence, la structure tabulaire utilisée, l'évolution de son remplissage et les complexités temporelle (en nombre de conditions évaluées) et spatiale du programme qui en résulte (le code n'est pas demandé).
5. Appliquer cette solution avec les éléments de la question 3.

1.4. La boîte à outils

Dans une boîte à outils, il y a en vrac m boulons, avec $m > 0$, constitués de l'ensemble N de m écrous (en anglais *nuts*) de diamètres tous différents et de l'ensemble B des m vis correspondantes (en anglais *bolts*). La différence de diamètre est si faible qu'il est impossible de comparer visuellement la taille de deux écrous ou de deux vis entre elles. Pour savoir si une vis correspond à un écrou, la seule façon est d'essayer de les assembler. La tentative d'appariement vis-écrou est l'opération élémentaire choisie pour l'évaluation de la complexité de ce problème. Elle fournit l'une des trois réponses suivantes : soit l'écrou est plus large que la vis, soit il est moins large, soit ils ont exactement le même diamètre. Le problème consiste à établir la bijection qui, à chaque écrou, associe sa vis.

Nous attribuons un numéro entre 0 et $m-1$ à chacun des écrous ; nous faisons de même pour chacune des vis. Nous modélisons le problème par la classe `Box` suivante :

Listing 1.1. Modélisation de la boîte à outils par la classe `Box`

```
1  from random import shuffle
2
3  class Box():
4      def __init__(self,m):
5          self.nut = [ x for x in range(m)]
6          shuffle(self.nut)
7          self.bolt = [ x for x in range(m)]
8          shuffle(self.bolt)
9      def screw(self, n, b):
10         if self.nut[n] < self.bolt[n]:
11             return -1 # Ecrou trop étroit
12         if self.nut[n] == self.bolt[n]:
13             return 0  # Ecrou et vis s'ajustent
14         if self.nut[n] > self.bolt[n]:
15             return 1  # Ecrou trop lâche
```

1. Écrire une fonction Python naïve `assemble`. La valeur de retour de `assemble(box)` est un dictionnaire d contenant des associations $n \mapsto b$ telles que, pour tout indice $n \in [\![0, m-1]\!]$, on a `box.nut[n] == box.bolt[b]`.
   La fonction `assemble` ne peut que utiliser `box.screw` pour explorer la boîte.
2. Quelle est la complexité de `assemble` dans le pire des cas et dans le meilleur des cas ?
3. Montrer que n'importe quel algorithme obéissant à la spécification de `assemble` appelle $\Omega(m \log m)$ fois `box.screw`.
4. Proposer un algorithme probabiliste obéissant à la spécification de `assemble` et utilisant la stratégie « diviser pour régner », dont la complexité moyenne est $O(n \log n)$. Dire s'il s'agit d'un algorithme de type *Las Vegas* ou *Monte Carlo*.