Chapitre 2

Tableaux

1. Représentations d'un tableau en Python

1.1 Structures de données

Dans son acceptation la plus générale, une *structure de données* spécifie la façon de représenter en mémoire machine les données d'un problème à résoudre en décrivant :

- la manière d'attribuer une certaine quantité de mémoire à cette structure ;
- la façon d'accéder aux données qu'elle contient.

Dans certains cas, la quantité de mémoire allouée à la structure de donnée est fixée au moment de la création de celle-ci et ne peut plus être modifiée ensuite ; on parle alors de structure de données *statique*. Dans d'autres cas l'attribution de la mémoire nécessaire est effectuée pendant le déroulement de l'algorithme et peut donc varier au cours de celui-ci ; il s'agit alors de structure de données *dynamique*. Enfin, lorsque le contenu d'une structure de donnée est modifiable, on parle de structure de données *mutable*, et dans le cas contraire de structure de données *immuables*.

Exemples. Python propose un certain nombre de structures de données (pas toutes au programme), telles :

- la classe `list`, dynamique et mutable ;
- la classe `numpy.array`, statique et mutable ;
- les classes `str` et `tuple`, statiques et immuables ;
- la classe `set`, dynamique et immuable.

et bien d'autres encore.
La classe `list` est dynamique car la méthode `append` permet d'augmenter la taille d'une liste. Elle est mutable car l'instruction `s[i] = x` permet de remplacer la valeur de `s[i]` par la valeur de `x`.
La classe `numpy.array` est elle aussi mutable, mais elle est immuable car la taille d'un tableau doit être définie au moment de sa création, et ne peut plus être modifiée ensuite.
Les classes `tuple` (pour représenter les n-uplets) et `str` (pour les chaînes de caractères) sont statiques et immuables car ni leur longueur, ni leur contenu ne peuvent être modifiées une fois ces structures de données crées.
Enfin, la classe `set` (hors programme) qui modélise les ensembles mathématiques, est dynamique pour permettre l'ajout de nouveaux éléments à un ensemble et immuable car elle interdit de modifier un élément rangé dans un ensemble.

Attention. Une erreur fréquente consiste à confondre deux opérations sur les listes, qui pourtant fournissent le même résultat apparent :

```
s.append(x)
```

```
s = s + [a]
```

La première instruction est efficace : la classe `list` étant dynamique, cette opération ajoute un élément en queue de liste. En revanche, la seconde opération *crée une nouvelle liste*, augmentée d'un nouvel élément, dans un nouvel espace mémoire, qu'on lie ensuite à l'identifiant `s`.
Même si les deux instructions conduisent à un résultat analogue, la seconde est à rejeter car dans le cas de listes de tailles importantes, ou lorsque cette opération est réalisée de nombreuses fois, les performances du script peuvent s'en ressentir fortement.

En revanche, les chaînes de caractères étant de nature statique, il n'y a pas d'autre alternative que de créer une nouvelle chaîne si on souhaite ajouter un caractère à une chaîne existante.

1.2 Représentation des tableaux en Python

En informatique, on appelle *tableau* une suite de variables de même type associées à des emplacements consécutifs de la mémoire.

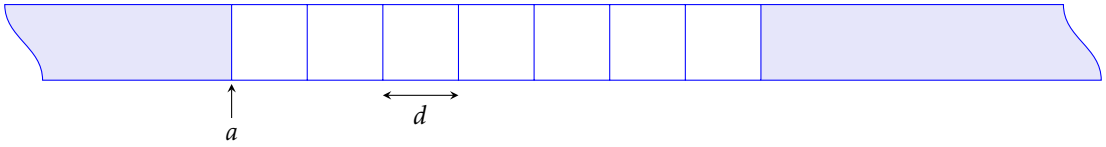


FIGURE 1 – Une représentation d'un tableau en mémoire.

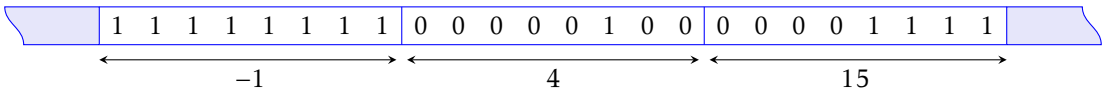
Puisque tous les emplacements sont de même type, ils occupent tous le même nombre d de cases mémoire ; connaissant l'adresse a de la première case du tableau, on accède *en coût constant* à l'adresse de la case d'indice k en calculant l'adresse $a + kd$. En revanche, ce type de structure est statique : une fois un tableau créé, la taille de ce dernier ne peut plus être modifiée faute de pouvoir garantir qu'il y a encore un espace mémoire disponible au delà de la dernière case. En résumé :

- un tableau est une structure de donnée statique et mutable ;
- les éléments du tableau sont accessibles en lecture comme en écriture en temps constant O(1).

L'implémentation stricte des tableaux est réalisée en Python par la classe `numpy.array`. Par exemple, le script ci-dessous :

```
T = numpy.array([-1, 4, 15], dtype=numpy.int8)
```

réserve un espace de trois octets pour y stocker les représentations en complément à 2 des entiers −1, 4 et 15 :



La classe `list` permet elle aussi de représenter les tableaux informatiques ; il s'agit cependant d'une structure de donnée hybride qui, en plus des opérations usuelles des tableaux, possède certaines propriétés qui les enrichissent. Nous n'enutiliserons principalement que deux, données dans l'encadré ci-dessous.

Si `L` est une instance de la classe `List`, alors :

- `L.append(x)` ajoute en temps constant un élément en queue de liste ;
- `x = L.pop()` supprime et affecte à x en temps constant le dernier élément de la liste.

On observera que ces deux opérations indiquent que la classe `list` est une structure de donnée dynamique, ce que ne sont pas en principe les tableaux. Nous ne décrirons pas l'implémentation machine de cette classe, hors programme.

Remarque. D'autres fonctions et méthodes existent sur les listes : `del`, `extend`, `insert`, `remove`, etc, mais il est déconseillé d'en user dans une épreuve d'algorithmique car elles sont en général de complexité non constante et de ce fait peuvent fausser l'évaluation de la complexité algorithmique des algorithmes qui les utilisent. Elles sont d'ailleurs parfois explicitement interdites dans certaines épreuves de concours.

■ Tableaux bi-dimensionnel

La classe `numpy.array` permet la manipulation aisée d'un tableau bi-dimensionnel : l'instruction ci-dessous crée un tableau $n \times p$ initialisé avec l'entier 0 dans chacune de ses np cases :

```
T = numpy.zeros((n, p), dtype=int)
```

L'élément de rang (i, j) peut être obtenu ou modifié par la syntaxe `T[i, j]` ou `T[i][j]`.
Si on souhaite définir un tableau bi-dimensionnel avec la classe `List` c'est un peu plus malcommode : il faut créer un tableau uni-dimensionnel de n cases dans lesquelles se trouve un tableau de p cases.

```
L = [[0 for j in range(p)] for i in range(n)]
```

Cette fois, seule la syntaxe `L[i][j]` permet d'accéder à l'élément de rang (i, j).

1.3 Recherche au sein d'un tableau

Nous allons maintenant nous intéresser au parcours d'un tableau déjà créé. Puisqu'ils ne concernent pas la création d'un tableau, ces algorithmes peuvent s'appliquer indifféremment aux éléments de la classe `list` ou de la classe `numpy.array`.

■ Calcul du maximum

Intéressons nous maintenant au problème du calcul de la valeur maximale d'une liste non vide d'entiers. On peut procéder par énumération des éléments :

```
def maximum(t):
    m = t[0]
    for x in t:
        if x > m:
            m = x
    return m
```

ou par énumération des indices du tableau (nécessaire si on désire obtenir l'indice de la case où se situe le maximum) :

```
def indice_du_max(t):
    i = 0
    for k in range (1, len(t)):
        if t[k] > T[i]:
            i = k
    return i
```

Exercice 1. Rédiger une fonction `secondMaximum(t)` qui prend pour argument un tableau contenant au moins deux éléments entiers et qui renvoie la valeur du deuxième plus grand élément de ce tableau. On pourra supposer les éléments du tableau deux-à-deux distincts.

■ Parcours incomplet dans un tableau

Les algorithmes de recherche dans une liste correspondent le plus souvent à des parcours incomplets, puisqu'on stoppe la recherche une fois l'élément trouvé. On utilise traditionnellement une boucle conditionnelle, mais le langage Python incite plutôt à privilégier une sortie prématurée de boucle énumérée.

Il y a deux façons d'interrompre une énumération en cours :
- en utilisant `return` au sein de la définition d'une fonction ;
- en utilisant `break` au sein d'une énumération.

Un problème fréquent consiste à trouver un élément au sein du tableau, s'il en existe, vérifiant une certaine propriété. Pour formaliser cette situation nous allons supposer posséder une fonction `prop(x)` qui renvoie la valeur `True` lorsque x vérifie la propriété qui nous intéresse, et `False` dans le cas contraire. La fonction de recherche va suivre le schéma suivant :

```
def cherche(prop, t):
    for x in t:
        if prop(x):
            return x
```

Exercice 2. Rédiger une fonction `chercheSecond(prop, t)` qui renvoie le second élément vérifiant la propriété, s'il en existe.

1.4 Complexité temporelle

Mesurer la complexité temporelle d'une fonction agissant sur une structure de données consiste à évaluer son temps d'exécution $C(n)$ en fonction de la taille n de la structure de données. Pour ce faire on détermine quelles sont les opérations élémentaires utilisées par cette fonction et on en compte le nombre d'utilisation dans le pire des cas.

On appelle *opération élémentaire* toute instruction pour lesquelles il est raisonnable de penser que son temps d'exécution est indépendant des paramètres qu'elle utilise.

Par exemple, les opérations arithmétiques, les comparaisons entre nombres, la lecture ou l'affectation dans un tableau sont des opérations élémentaires.
En revanche les fonctions `sum` (calcul de la somme des éléments d'un tableau) ou `sorted` (trier les éléments d'un tableau) ne sont pas des opérations élémentaires : il est raisonnable de penser que leur temps d'exécution varie en fonction du nombre n d'éléments du tableau.

Considérons l'exemple du calcul du maximum des éléments d'un tableau non vide d'entiers :

```
def maximum(t):
    m = t[0]
    for i in range(1, len(t)):
        if t[i] > m:
            m = t[i]
    return m
```

Cette fonction réalise $n-1$ comparaisons et un nombre d'affectations de la variable `m` compris entre 1 (lorsque l'élément maximal est dans la première case) et n (lorsque le tableau est trié par ordre croissant). Si on note τ_1 le temps d'exécution d'une comparaison et τ_2 le temps d'exécution d'une affectation de la variable `m`, le temps d'exécution total (en négligeant le temps d'execution des autres instructions) s'exprime sous la forme $C(n) = (n-1)\tau_1 + k\tau_2$ avec $k \in [\![1, n]\!]$.
Puisque l'entier k dépend du la façon dont les entiers sont rangés dans le tableau, on se place dans la situation la plus défavorable : $C(n) \leqslant (n-1)\tau_1 + n\tau_2$.
Cette formule n'est pas encore satisfaisante, car, à moins d'être un électronicien chevronné, nous n'avons aucune idée de l'ordre de grandeur de τ_1 et τ_2 : ces valeurs dépendent de nombreux paramètres qui nous sont inconnus (nature de la machine sur laquelle est exécuté le code, le traducteur de code utilisé, etc). La seule chose que l'on sache, *c'est qu'il s'agit de valeurs constantes*. C'est la raison pour laquelle on utilise les notations de Landau pour faire disparaître ces constantes qui ne nous apportent aucune information, et on traduira ce résultat en écrivant que $\boxed{C(n) = O(n).}$ On parlera de :

- complexité *logarithmique* lorsque $C(n) = O(\log n)$;
- complexité *linéaire* lorsque $C(n) = O(n)$;
- complexité *quasi-linéaire* lorsque $C(n) = O(n \log n)$;
- complexité *quadratique* lorsque $C(n) = O(n^2)$;
- complexité *exponentielle* lorsque $C(n) = O(a^n)$ avec $a > 1$.

Considérons maintenant la fonction suivante, qui calcule l'image miroir d'un tableau de type `numpy.array` :

```
def miroir(t):
    n = len(t)
    m = numpy.zeros_like(t)  # création d'un tableau de n cases
    for i in range(n):
        m[i] = t[n-1-i]
    return m
```

Sa complexité temporelle est à l'évidence un $O(n)$: on réalise n affectations et de l'ordre de n opérations arithmétiques sur les indices. Mais elle possède aussi une complexité spatiale $C(n)$: la création d'un nouveau

tableau de taille n. On a $C(n) = n\sigma$, où σ est le nombre d'octets utilisé pour représenter les objets du tableau. On aura par exemple $\sigma = 1$ si t est un tableau d'entiers `int8` ou $\sigma = 8$ si t est un tableau de flottants `float64`. Cette fonction s'appliquant à tout tableau de type `numpy.array`, on utilisera là encore les notations de Landau pour exprimer la complexité spatiale à savoir ici un $O(n)$.

Exercice 3. Recherche d'un mot dans une chaîne de caractères.
On souhaite écrire une fonction qui détermine si un mot m est présent dans une chaîne de caractères s. On pose $m = m_0 m_1 \cdots m_{k-1}$ et $s = s_0 s_1 \cdots s_{n-1}$.

a) Rédiger une fonction `egal(i, m, s)` qui renvoie la valeur `True` lorsque $i + k \leqslant n$ et $m_0 m_1 \cdots m_{k-1} = s_i s_{i+1} \cdots s_{i+k-1}$, et `False` dans les autres cas.

b) En déduire une fonction `sousMot(m, s)` qui renvoie la valeur `True` lorsque m est présent dans s, et `False` dans le cas contraire.

c) Évaluer la complexité temporelle des fonctions `egal` et `sousMot`, qu'on exprimera en fonction de n et de k.

1.5 Recherche dichotomique

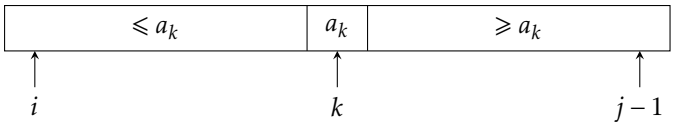
Lorsque le tableau est *trié* (d'une façon ou d'une autre) et qu'il est possible en temps constant de déterminer si l'élément recherché se trouve dans la partie gauche ou la partie droite du tableau, on peut employer une méthode de recherche dichotomique.

Pour fixer les idées supposons que l'on cherche à déterminer la présence ou non d'un élément x dans un tableau trié par ordre croissant $t = [a_0, \ldots, a_{n-1}]$. Le principe de la recherche dichotomique consiste à comparer x et a_p où $p = \left\lfloor \frac{n}{2} \right\rfloor$ puis :

- si $x < a_p$ alors x, s'il se trouve dans t, ne peut se trouver qu'au sein de $[a_0, \ldots, a_{p-1}]$;
- si $x = a_p$, la recherche est terminée ;
- si $x > a_p$ alors x, s'il se trouve dans t, ne peut se trouver qu'au sein de $[a_{p+1}, \ldots, a_{n-1}]$.

La mise en œuvre pratique utilise deux variables entières i et j délimitant le domaine $[a_i, \ldots, a_{j-1}]$ dans lequel on cherche x. Les valeurs initiales de ces variables sont donc $i = 0$ et $j = n$.

On compare ensuite x à l'élément médian d'indice $k = \left\lfloor \frac{i+j}{2} \right\rfloor$ et si nécessaire on poursuit la recherche en remplaçant j par k ou i par $k+1$.



La recherche s'achève (par une réponse négative) lorsque $i = j$ car le domaine de recherche est vide dans ce cas.

```
def rechercheDichotomique(x, t):
    i, j = 0, len(t)
    while i < j:
        k = (i + j) // 2
        if t[k] == x:
            return True
        elif t[k] > x:
            j = k
        else:
            i = k + 1
    return False
```

• Évaluation du nombre de comparaison
Notons $C(n)$ le nombre de comparaison que cet algorithme effectue entre x et un élément $t[k]$ du tableau. Le principe dichotomique sépare tableau en trois parties de respectivement $\left\lfloor \frac{n-1}{2} \right\rfloor$, 1 et $\left\lfloor \frac{n}{2} \right\rfloor$ cases, et une comparaison permet de restreindre la recherche à l'une de ces trois sous-listes.
Dans le pire des cas, on a donc : $C(n) = 1 + C(\lfloor n/2 \rfloor)$. On démontre alors par récurrence sur n que $C(n) \leqslant \log n + 1$.

– Si $n = 1$ à l'évidence on a $C(1) = 1$.
– Si $n \geqslant 2$ supposons le résultat acquis jusqu'au rang $n-1$. En particulier on a donc $C(\lfloor n/2 \rfloor) \leqslant \ln\lfloor n/2 \rfloor + 1$ et ainsi $C(n) \leqslant \ln\lfloor n/2 \rfloor + 2 \leqslant \ln(n/2) + 2 = \ln(n) + 1$.

La complexité temporelle de l'algorithme de recherche dichotomique est donc un $\boxed{O(\log n),}$ à comparer à la complexité linéaire que nécessite l'algorithme de recherche lorsque le tableau n'est pas trié.

2. Exercices

Exercice 4 On appelle *rotation* d'un tableau t le fait de décaler tous les éléments d'une place vers la droite, à l'exception du dernier qui est placé en première place. Par exemple, la rotation du tableau $[1,2,3,4]$ est le tableau $[4,1,2,3]$.

a) Rédiger une fonction `rotation(t)` qui renvoie un *nouveau* tableau égal à la rotation du tableau initial.

b) Rédiger une fonction `rotation2(t)` qui *modifie* le tableau t pour le remplacer par sa rotation.

c) Rédiger une fonction `rotation3(k, t)` qui renvoie un nouveau tableau égal à k rotations du tableau t.

Exercice 5 À partir d'un tableau t d'entiers naturels, rédiger une fonction `entierManquant(t)` qui renvoie le plus petit entier naturel absent du tableau. Par exemple, pour $t = [1,3,7,6,4,1,2,0]$ cette fonction devra renvoyer l'entier 5.

Exercice 6 Un tableau non vide t de n entiers relatifs étant donné, on cherche la valeur maximale de la quantité $\Delta_k = (t[0] + \cdots + t[k]) - (t[k+1] + \cdots + t[n-1])$ lorsqu'on fait varier k dans $[\![0, n-2]\!]$.

a) Rédiger une fonction `delta(k, t)` qui calcule la quantité Δ_k et en déduire une fonction `equilibre(t)` qui résout le problème posé. Évaluer la complexité temporelle de cette dernière fonction.

b) Trouver une fonction `equilibre2(t)` qui résout ce problème en temps linéaire.

Exercice 7 Une petite grenouille se trouve face à une rivière. Initialement située sur une des deux rives (position 0), elle veut se rendre sur la rive opposée (position $x+1$) mais ne peut réaliser que des sauts d'une unité. Heureusement pour elle, des feuilles tombent sur la surface de la rivière et peuvent lui permettre de passer de feuille en feuille.

On donne un tableau t composé de n entiers représentant les feuilles qui tombent : $t[k]$ représente la position où une feuille tombe à l'instant k. L'objectif est de trouver le moment le plus précoce où la grenouille pourra passer d'une rive à l'autre, c'est-à-dire la date où toutes les positions de 1 à x seront couvertes par une feuille.
Rédiger une fonction `grenouille(x, t)` qui résout ce problème. par exemple, pour $x = 5$ et $t = [1,3,1,4,5,3,2,4]$ cette fonction devra renvoyer l'entier 6.
Évaluer la complexité temporelle et spatiale de votre fonction.

Exercice 8 **Recherche dans un tableau bi-dimensionnel**
On considère un tableau bi-dimensionnel t à n lignes et p colonnes vérifiant les propriétés suivantes :

(i) pour tout $i \in [\![0, n-1]\!]$, $\forall j \in [\![0, p-2]\!]$, $t[i,j] < t[i,j+1]$ (chaque ligne est triée par ordre croissant) ;

(ii) pour tout $j \in [\![0, p-1]\!]$, $\forall i \in [\![0, n-2]\!]$, $t[i,j] < t[i+1,j]$ (chaque colonne est triée par ordre croissant).

1	3	9	11	13	14
3	5	10	12	15	17
4	7	11	15	16	21
7	10	16	17	20	24
13	14	19	23	25	29

FIGURE 2 – Un exemple de tableau vérifiant les hypothèses *(i)* et *(ii)*.

On considère la fonction de recherche d'un élément x dans t :

```
def recherche(x, t):
    n, p = len(t), len(t[0])
    i, j = n-1, 0
    while i >= 0 and j < p:
        if x == t[i, j]:
            return True
        elif x < t[i, j]:
            i = i - 1
        else:
            j = j + 1
    return False
```

a) Démontrer la validité de l'algorithme ci-dessus.

b) Quelle est sa complexité, exprimée en fonction de n et de p ?