Structures linéaires

1. Structures de données et complexité

Il existe plusieurs types de *structures de données*, c'est-à-dire de façons de ranger et d'ordonner un ensemble de données. Ces types se distinguent par la façon d'accéder aux éléments de l'ensemble considéré et par la façon dont on peut, ou pas, modifier cet ensemble, en supprimant ou en ajoutant des éléments, ou en modifiant les liens existant entre les différents éléments de cet ensemble.

Le choix d'une structure de données est fondamental car il conditionne la complexité des algorithmes que l'on va programmer. Ainsi, selon la structure choisie, la recherche d'un élément dans un « ensemble » de taille n peut prendre un temps $\mathcal{O}(1)$ (coût dit constant) ou $\mathcal{O}(n)$ (coût linéaire). On notera qu'en matière de complexité, on ne s'intéresse qu'à des ordres de grandeur, et que les termes *constant* ou *linéaire* qualifient donc ces ordres de grandeurs et non le temps effectivement nécessaire à l'exécution du programme.

En matière de complexité, on utilisera deux notations mathématiques.

Définition 1. Si $f, g\colon \mathbb{N} \to \mathbb{R}_+$, on a les deux comparaisons suivantes :

$$f(n) = \mathcal{O}(g(n)) \iff \exists C > 0,\ \exists n_0, \forall n \geqslant n_0\ :\ f(n) \leqslant Cg(n)$$
$$f(n) = \Theta(g(n)) \iff \exists B, C > 0,\ \exists n_0, \forall n \geqslant n_0\ :\ Bg(n) \leqslant f(n) \leqslant Cg(n).$$

En d'autres termes, $f(n) = \mathcal{O}(g(n))$ si la croissance de f est contrôlée par celle de g et $f(n) = \Theta(g(n))$ si $f(n)$ et $g(n)$ sont du même ordre de grandeur au voisinage de l'infini. Ainsi, $f(n) = \Theta(g(n))$ est équivalent (au sens logique...) à $g(n) = \Theta(f(n))$.

Exercice 1. Soient deux polynômes unitaires P et Q à coefficients réels. Donner une condition nécessaire et suffisante sur P et Q pour que $P(n) = \mathcal{O}(Q(n))$. Même question pour $P(n) = \Theta(Q(n))$.

Exercice 2. Évaluer la complexité du calcul de 3^n, de 4^n.

Dans certains cas, la quantité de mémoire allouée à l'objet représenté par la structure de données est fixée au moment de la création de l'objet. On parle alors de *structure de données statique*. Dans d'autres cas, l'attribution de mémoire évolue au cours de l'exécution de l'algorithme en fonction des besoins. On a affaire à une *structure de données dynamique*. Enfin, quand l'objet est modifiable, on parle de *structure de données mutable*, ce qui est un anglicisme, mais bon. Pour « non mutable », on dit souvent *immuable*. Le terme *modifiable* signifie bien sûr modifiable sans changement d'adresse mémoire ; autrement, tout objet serait modifiable.

2. Principales structures linéaires

Les structures de données classiques sont les suivantes :

(1) Les structures linéaires (ou plates). Il s'agit essentiellement de structures représentables par des suites finies ordonnées. On y trouve les listes (`list`), les tableaux (matrices unidimensionnelles en Python), les tuples, les chaînes de caractère, les piles, les files. La liste n'est pas exhaustive, mais ce sont celles que nous traiterons. La définition des structures linéaires n'est pas importante ; en revanche, il faut avoir bien compris la différence entre structures mutables et immuables.
(2) Les matrices ou tableaux multidimensionnels (`ndarray` de Numpy).
(3) Les structures arborescentes, arbres enracinés ou non, en particulier les arbres binaires. Ces structures ne sont pas au programme.
(4) Les structures relationnelles, sous la forme de graphes ou de bases de données.

2.1. **Tableaux.** Les *tableaux* forment une suite unidirectionnelle de variables de même type associées à des emplacements consécutifs de la mémoire. Connaissant l'adresse de l'une des cases, on accède en temps constant à n'importe quelle autre case du tableau par une translation codée par une addition car toutes les cases sont de la même taille ; en particulier, tous les éléments du tableau doivent être du même type. Ce type de données est statique. On ne peut l'agrandir, mais on peut en modifier les éléments. En Python, c'est le modèle utilisé par les tableaux Numpy de dimension 1. On les utilisera plus tard dans l'année, notamment en ingénierie numérique.

2.2. **Listes chaînées.** Une liste chaînée est formée d'une première case et d'une suite de pointeurs, chacun d'entre eux pointant vers la case suivante. C'est une structure dynamique et très souple, mutable, qui a l'inconvénient que l'accès à une case ne se fait pas en temps constant, mais linéaire, vu qu'il faut pour l'atteindre passer par toutes les cases précédentes. Cette structure n'est pas implémentée en Python.

2.3. **Le type `list` de Python.**

2.3.1. *Structure.* C'est sans doute la structure principale ; en tout cas celle que nous utiliserons le plus. Ce type est conçu pour concilier les avantages des tableaux et des listes chaînées. Les listes de Python sont une structure dynamique et mutable, où chaque case est accessible en temps constant. L'idée est simple : une liste est un tableau de pointeurs renvoyant vers des emplacements de la mémoire appropriés. Si les pointeurs sont évidemment de même type, il n'en va pas de même de ce vers quoi ils pointent, ce qui fait qu'une liste peut contenir des éléments de natures différentes. Puisqu'il s'agit d'un tableau, chaque pointeur est accessible en temps constant, de même que la modification de n'importe quel élément de la liste.

Exercice 3. On dispose d'une liste L. À ce stade de la description, quel est le coût des opérations suivantes ?

(1) `L[4] = 6`

(2) `L[4] = 3.14`

(3) `L.append(-3)`

(4) `L.pop()`

(5) `L.insert(3,'a')`

(6) `L.remove(8)`

(7) `del L[4]`

2.3.2. *Redimensionnement.* On constate qu'il y a une difficulté quand on supprime des cases ou qu'on en ajoute. Python gère ce problème avec un redimensionnement du tableau. Cette opération dépend dans ses détails de l'implémentation de Python mais le principe est le suivant : à la création d'une liste de longueur n, Python ne crée pas un tableau de longueur n, mais de longueur λn (pour fixer les idées la valeur de λ en Cpython est de $9/8$). Cela laisse de la marge pour ajouter des éléments. Si le tableau devient trop petit, Python le dilate du même facteur λ. Il supprime de même des cases du tableau.

Exercice 4. Quel est le coût d'un redimensionnement ? Que change le redimensionnement aux résultats de l'exercice précédent ?

2.3.3. *Opérations et méthodes.* On a vu ci-dessus quelques-unes des méthodes et commandes les plus importantes sur les listes. En voici quelques autres :

```
L[2:19:3]
[3,4] + [9,1] = [3,4].extend([9,1])
[3,4]*3
len(L)
L.index('a')
L.count('a')
L.sort()
L.reverse()
4 in L
```

2.4. **Les tuples.** C'est une structure qui ressemble à celle de liste, mais qui n'est pas mutable. Elle est repérée en outre avec des parenthèses et non avec des crochets et l'on n'est pas obligé d'utiliser les parenthèses. Ses éléments, qui ne sont pas nécessairement de même type, sont séparés par des virgules, comme pour les listes. La longueur d'un tuple est obtenue, comme pour les listes, par la commande `len`. Le test d'appartenance fonctionne comme avec les listes. Attention, on définira `t = (5,9,2)` mais `t = (6,)`. Cela s'explique par le fait que Python interprète l'écriture `(6)` comme une écriture algébrique et lui affecte le type `int`. Enfin, il est facile de convertir une liste en tuple et réciproquement : `L = list(t)` et `t = tuple(L)`.

Python code certaines informations sous forme de tuples. Un exemple en est la dimension des tableaux sous Numpy : `np.shape(M)`. Il n'y a rien d'autre à connaître sur la structure de tuple. Un autre exemple utile est donné par la commande `enumerate` (*cf.* le paragraphe sur les itérateurs).

Les tuples sont une structure pratique d'affectation et de réaffectation de variables. Voici deux codes équivalents pour échanger `a` et `b` :

```
c = b
a = b
b = c
```

```
a, b = b, a
```

On peut aussi donner des noms de variables aux éléments d'une liste (*unpacking*) :
`L = [12, 5, 9, 1.4]`
`a, b, c, d = L` revient à `a = 12, b = 5, c = 9, d = 1.4`.

2.5. **Les chaînes de caractères.**

2.5.1. *Structure.* La première chose à savoir sur les chaînes de caractères est qu'en anglais, cela se dit *string*. C'est une structure immuable, sans séparateur, délimitée par des guillemets simples ' ou doubles ". Cette latitude permet d'utiliser des guillemets ou des apostrophes comme caractères à l'intérieur de la chaîne. Par exemple, on peut définir `"c'est comme ça"`. Une dernière façon, plus anecdotique, de définir une chaîne de caractères est de la situer entre des guillemets triples. En ce cas, Python prend en compte automatiquement les retours chariot. En pratique, cela sert surtout à écrire une documentation. De manière plus élémentaire, les guillemets triples permettent d'ajouter des commentaires de plus d'une ligne dans un programme Python. Un commentaire ne dépassant pas la ligne est simplement introduit par le symbole `#`.

2.5.2. *Opérations et méthodes de base.* L'addition de deux chaînes et la multiplication par un entier naturel fonctionne comme pour les listes, le test d'appartenance, l'accès par indice et le slicing également. Dans les TP et à l'oral, on est régulièrement amené à importer des fichiers texte, qui sont lus comme des chaînes de caractère. Il est alors utile de connaître les codages suivants :
`\n` : saut de ligne
`\r` : retour chariot
`\t` : tabulation horizontale
`\\` : \
Attention à ne pas laisser d'espace après une commande de ce type.
Quelques commandes utiles, mais pas indispensables en première lecture (`C` est une chaîne de caractères) :

```
C.upper()
C.lower()
C.replace('AB','TB')
C.find('AB')
C.count('AB')
```

Exercice 5. Quelle est la différence de nature entre `L.append(2)` et `C.upper()` ?

2.5.3. *Conversions.* Les chaînes de caractères peuvent se convertir en listes.

Si `C` est une chaîne, la commande `L = list(C)` va créer une liste dont les éléments seront eux-mêmes des chaînes de caractères de longueur 1 correspondant, dans l'ordre, aux caractères de `C`.

La commande `L = C.split()` va créer une liste des « mots » de `C` en séparant la chaîne selon les espaces. On peut ajouter un argument, du genre `L = C.split('-')` pour que la séparation se fasse selon un autre caractère (ou une autre chaîne).

Inversement, les éléments d'une liste de chaînes de caractères peuvent être regroupés en une chaîne unique. La commande `C = "-".join(L)` relie les éléments de `L` par le caractère (ou la chaîne de caractères) précisée entre guillemets - ici, un trait d'union.

2.5.4. *La méthode `format`.* C'est une méthode qui sert à afficher de manière agréable les résultats d'une fonction. Un exemple vaut mieux qu'un long discours.

```
def eucl(a,b):
    q = a//b; r = a%b
    print("le quotient de la division euclidienne de {} par {} est {};
        le reste vaut {}".format(a,b,q,r)
```

Exercice 6. Réécrire la fonction précédente en utilisant la commande `divmod`.

2.6. Les dictionnaires.

2.6.1. *Tables de hachage.* Les listes sont très efficaces en matière d'accessibilité à condition de connaître la position de l'élément auquel on veut accéder. Dans le cas contraire, il faut parcourir la liste et l'on passe d'une complexité constante à une complexité linéaire. C'est notamment le cas pour le test d'appartenance. Un autre défaut des listes est le caractère pas très parlant de leur indiciation. Par exemple, si l'on dresse une liste de personnes, chacune étant répertoriée par un nom, un prénom et une date de naissance, on aimerait pouvoir accéder rapidement aux informations concernant une personne par son nom et non par son numéro dans la liste, que l'on va ignorer en général. Cette problématique est résolue de manière pointue par les structures relationnelles et les bases de données, mais cela ne dit pas comment accéder en temps constant à une instance de la base.

Les *tables de hachage (hash table)* sont une réponse à ce problème. On dispose d'un tableau contenant des couples `(clef, valeur)` et d'une *fonction de hachage* qui, à la clef, associe en temps constant l'indice du tableau correspondant.

En fait, c'est un peu plus compliqué que ça. Les fonctions de hachage ne sont pas injectives (c'est fait exprès : simplicité, compression de fichiers, vérification de signatures...) et ne renvoient pas un indice, mais une liste chaînée d'indices, qu'il reste à parcourir (un itérateur sur la liste correspond au cas trivial d'une fonction de hachage constante). Il n'en reste pas moins que l'accès est extrêmement efficace.

En Python, les dictionnaires sont une implémentation des tables de hachage. Un dictionnaire est une collection mutable non ordonnée de couples `(clef, valeur)`, la clef étant nécessairement un objet immuable et la valeur un objet quelconque. Ils sont optimisés pour la relation d'appartenance. Il n'est pas difficile de comprendre pourquoi la clef ne peut être mutable. Imaginons l'inverse : on crée une entrée du dictionnaire. La fonction de hachage appliquée à la clef détermine en quel endroit de la mémoire cette entrée est stockée. On modifie ensuite la clef, ce qui possible car le dictionnaire, lui, est mutable. Il s'ensuit qu'on ne peut plus retrouver l'entrée correspondant, celle-ci étant rangée au mauvais endroit.

Cette structure n'est pas au programme et les commandes ne sont donc pas à connaître, mais elle peut faire son apparition à l'écrit et rien n'empêche de l'utiliser dans un exercice si on la maîtrise.

2.6.2. *Définition et opérations.* On peut principalement définir un dictionnaire de deux manières.
`d = {'marc': 21, 'alice': 20, 'pierre': 20}` ou
`L = [('marc', 21), ('alice', 20), ('pierre', 20)]` ; `d = dict(L)`.

`len(d)` compte le nombre de couples
`'alice' in d` teste l'appartenance de la clef `'alice'` au dictionnaire (ça ne marche pas pour les valeurs) ;
`d['marc']` permet d'accéder à la valeur associée ; `del d['pierre']` efface le couple ;
`d['nathalie'] = 19` en crée un nouveau (ou modifie la valeur si la clef existe déjà). ;
`d.keys()` donne la liste des clefs ;
`d.values()` donne la liste des valeurs ;
`d.items()` donne la liste des couples.
`d.update(d2)` met à jour le dictionnaire `d` avec le dictionnaire `d2`.

3. Références partagées

On parle de référence partagée quand plusieurs variables référencent le même objet. Il n'y a aucune difficulté quand l'objet est immuable, mais, pour les objets mutables, cela entraîne des *effets de bord*.

3.1. **Cas immuable.** Quand on rentre `pes = 9.8`, Python crée l'objet `9.8`, puis une variable `pes` et, enfin, un référence qui pointe de la variable `pes` sur l'objet `pes = 9.8`. Si l'on déménage sur la lune, on va modifier la valeur de la pesanteur en posant `pes = 1.6`. Cela conduit Python à créer l'objet `1.6`, à effacer la première référence et à en créer une nouvelle, pointant de la variable `pes` sur l'objet `1.6`. Si l'on pose maintenant `acc = pes`, Python va créer la variable `acc` et une référence vers la valeur actuelle de `pes`, soit l'objet `1.6`. Si l'on revient maintenant sur Terre et que l'on pose `pes = 9.8`, on va revenir à l'état précédent en ce qui concerne la variable `pes`, ce qui sera sans incidence sur `acc`, qui pointera toujours sur `1.6`. Tout cela se déroule naturellement.

3.2. **Cas mutable.** La situation est différente si l'on quitte les objets flottants, qui sont immuables, pour des listes, qui sont mutables. Voyons ce qui se passe sur des exemples

Instructions : `a = [1, 2]; b = a; a[0] = 0`.

Instructions : `a = [1, 2]; b = a[:]; a[0] = 0`.

Instructions : `a = [1, [2]]; b = a[:]; a[0] = 0; a[1][0] = 0`.

3.3. **Copies superficielles et profondes.** L'instruction `b = a` ne fait pas de copie du tout ; elle crée une référence.
L'instruction `b = a[:]` crée une copies superficielle (*shallow copy*). On comprend dans le dernier exemple ce que cette copie a de superficiel.
Faire une copie profonde (*deep copy*), c'est créer une seconde variable complètement indépendante de la première. Quand les objets sont immuables, c'est ce que se passe automatiquement. Dans le cas d'objets mutables, on procède de la façon suivante :

```
import copy
b = copy.deepcpy(a)
```

Une copie profonde procède récursivement sur les niveaux d'imbrication de la variable. On s'en servira quand on étudiera les tris, selon que ceux-ci se font *en place*, c'est-à-dire en modifiant la liste de départ, ou en créant de nouvelles listes. De manière plus générale, il faut bien avoir compris que les méthodes appliquées sur les objets mutables modifient ceux-ci définitivement, même se ces objets n'interviennent que comme des variables locales dans l'exécution d'une fonction et que si l'on a besoin de garder une trace de l'objet de départ, il faut commencer par en faire une copie profonde.

3.4. **Égalité.** Il y a deux notions d'égalité de variables en Python. L'égalité de valeur, testée en Python par `a == b` et l'égalité d'adressage mémoire (donc égalité parfaite des variables), testée par `a is b`. Il est évident que celle-ci entraîne celle-là, mais la réciproque est fausse. Et cela peut être assez piégeur. Puisqu'on en parle, l'adresse mémoire de la variable `a` est donnée par `id(a)`. Ainsi, `a is b` est-il équivalent à `id(a) == id(b)`.
`a = 5; b = 5; c = a` conduit à `a is b, a is c -> True, True`, mais
`u = [1,2]; v = [1,2]; w = u` conduit à `u is v, u is w -> False, True`.
Ça a l'air loufoque, mais il y a une explication rationnelle.

Exercice 7. 1. Si l'on a rentré `a = b = [1,2]; b[1] = 3`, que donne `print(a)` ? **2.** On pose `L = [0]; L[0] = L`. Que vaut `L[0][0][0] is L` ?

3.5. **Application aux dictionnaires.** Une clef doit non seulement, comme on l'a vu, être immuable, mais l'être globalement. Un exemple pour comprendre ce dont il s'agit : `d={}` crée un dictionnaire vide ;
`cle1=(1,2)` crée une clef de bon aloi, immuable ;
`d[cle1] = "Allez l'OM"` crée une entrée dans le dictionnaire ; toujours aucun problème ;
`cle_bidon = (1,[2,3])` crée une clef immuable, mais non globalement immuable ;
`d[cle_bidon] = "Paris est magique"` ;
`cle_bidon[1].append(4)` fiche tout par terre.

4. Itérateurs

4.1. **Un parcours sans indices.** On a vu que les structures linéaires étaient liées sous le capot à une structure chaînée permettant de les parcourir. L'instruction `for` permet de parcourir toute instance des types Python introduites ci-dessus. C'est important car, selon ce que l'on voudra faire, on pourra choisir avantageusement entre deux possibilités pour parcourir, disons, une liste : ou bien le faire par ses indices, ou bien utiliser directement ses éléments. Ainsi, pour imprimer un par un les éléments d'une liste, on pourra écrire indifféremment :

```
for i in range(len(L)):        | for x in L:
    print(L[i])                |     print(x)
```

La commande `range` crée un itérateur, ce qui prend beaucoup moins de place en mémoire que la liste elle-même. Aussi la ligne de commande `print(range(20)` renvoie-t-elle à une adresse mémoire. Pour créer la liste, il faut écrire `L = list(range(20))` ou `L = [x for x in range(20)]`.
La commande `enumerate` crée un intérateur sur la liste de tuples `(i, L[i])` et peut être utile. Pour en faire quelque chose, il faut parcourir cet itérateur avec une boucle `for`.

Pour parcourir un dictionnaire `d`, on peut passer par la liste `d.items()`, mais il est préférable de passer par l'itérateur :

```
for clef, valeur in dico.iteritems():
    print("clef: {}, valeur: {}".format(clef, valeur)
```

4.2. **Listes par compréhension.** La liste `U = [x**2 for x in range(10)]` affiche la liste des carrés des dix premiers entiers.
`V = [x**2 for x in range(100) if x%5 == 1]` affiche la liste des carrés des nombres x au plus égaux à 99 de la forme $x = 1 + 5k$.
Le temps d'exécution d'une liste en compréhension est beaucoup plus faible que le même résultat obtenu avec une boucle `for`.

Exercice 8. 1. Engendrer la liste `U` avec une boucle `for`.
2. Engendrer la liste `V` avec une boucle `for`, puis avec une boucle `while`.

Comme pour les listes, on peut définir des dictionnaires par compréhension. On peut par exemple définir `d = {i: i**2 for i in range(10)}`.
Les dictionnaires permettent un accès rapide aux clefs, mais pas aux valeurs. Pour trouver une valeur, il faut *a priori* parcourir les clefs ; c'est comme trouver un élément dans une liste, linéaire. Toutefois, on peut, en utilisant la compréhension, on peut renverser un dictionnaire et assurer ainsi un accès rapide aux valeurs. Imaginons un dictionnaire associant des numéros de sécurité sociale à des noms.
`SS = {'Jean' : 1881213435678, 'Marie' : 2900875867549,...}`. On retrouve le numéro de SS de Marie en tapant SS['Marie']. Pour retrouver l'individu ayant pour numéro de SS 1881213435678, on crée le dictionnaire inversé, avant de l'interroger : `SS2 = {SS[k]: k for k in SS}`
`SS2[1881213435678]`

Exercice 9. On dispose d'une liste de dictionnaires de la forme
`Annuaire = [{'nom': 'Pocquelin', 'prénom': 'Jean-Baptiste', 'naissance': (15,1, 1622)}, {'nom': 'Racine', 'prenom': 'Jean', 'naissance': (22,12,1639)}, ..., {...}]`. Créer un index sous la forme d'un dictionnaire de dictionnaires dont les clefs sont les noms et les valeurs les entrées de l'annuaire.

5. Exercices

Exercice 10. Les structures suivantes en Python sont-elles statiques, dynamiques, mutables ?
- les entiers (`int`) ;
- les listes (`list`) ;
- les tuples (`tuple`) ;
- les chaînes de caractère (`string`) ;
- les tableaux de numpy (`ndarray`) ;
- les dictionnaires (`dict`).

Exercice 11. On dispose de deux fonctions f et g. Écrire une fonction `equal` d'arguments f, g et L (une liste), qui renvoie une liste de booléens testant l'égalité $f(x) = g(x)$ pour les éléments x de L. Par exemple, si $f(x) = x$ et $g(x) = x^2$, alors `equal(f,g,list(range(-2,2))) = [False, False, True, True]`.

Exercice 12. Écrire une fonction `elt` dressant la liste des éléments des éléments d'une liste de listes. Par exemple, `elt([[0,1,2],[3,0]]) = [0,1,2,3,0]`.

Exercice 13. Écrire une fonction `alter` d'arguments deux listes et dressant une liste des éléments de ces deux listes en alternance. Par exemple, `alter([0,1,2,5],[3,0]]) = [0,3,1,0,2,5]`.

Exercice 14.
```
def Puissance(a,n):
    A,N,R=a,n,1
    while N>0 :
        if N%2==0 :
            A,N=A**2,N/2
        else :
            R,N=R*A,N-1
    return R
```
Pour ce programme :

(1) Montrer à l'aide d'un invariant de boucle que cet algorithme calcule a^n

(2) Quel est sa complexité ?