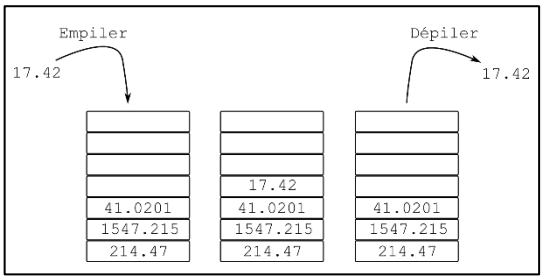
P-uplets et tableaux : Cours

I. Introduction : Un problème = une structure de données adaptée

En informatique, une structure de données est une manière d'organiser des données (entiers, booléens etc.) afin de les traiter efficacement en fonction de ce qu'on souhaite faire avec.



Exemple de la pile :

Nous avons déjà rencontré la notion de "pile de données" lors du TP sur l'assembleur. Une pile est une structure de données qui ne supporte que deux opérations : "Empiler" et "Dépiler".

Une pile n'est pas très pratique car on ne peut pas accéder facilement aux données en bas de la pile.

Si on souhaite concevoir un annuaire téléphonique, une pile n'est sans doute pas adaptée : il serait très difficile de récupérer le premier nom écrit dans la pile.

En revanche, si on souhaite mémoriser un historique de navigation (empiler = "nouvelle page visitée" et dépiler = "afficher page précédente") une pile est parfaitement adaptée et surtout une pile est très rapide et efficace (d'où son utilisation dans les processeurs, voir TP assembleur).

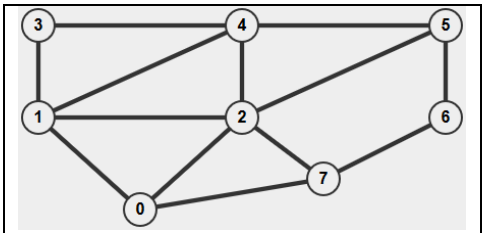
Généralisation à toutes les structures de données :

Nous venons de voir qu'une pile de données n'est pas efficace dans toutes les situations.

En informatique, beaucoup de structures de données vérifient ce schéma : elles ne sont pas efficaces pour toutes les utilisations. C'est pourquoi on a au fil du temps inventé un grand nombre de structures plus ou moins adaptées à tel ou tel problème. Et c'est au concepteur d'algorithme ou au programmeur de choisir la structure de données adaptée à son problème qui lui permettra d'obtenir un programme efficace

Pour prendre conscience de la diversité des structures de données disponibles, voici celles qui sont au programme de NSI :

- p-uplets (première)
- tableaux (première)
- p-uplets nommés (première)
- listes, piles et files (terminale)
- arbres (terminales)
- graphes (terminale)



Une structure de graphe : adaptée pour représenter un réseau (social ou routier)

II. Les p-uplets

1) Définition (point de vue algorithmique)

Le nom vient de couplet, triplet, quadruplet, quintuplet etc. Il s'agit donc d'une séquence ordonnée de p éléments.

Exemple : les coordonnées d'un point du plan pour lesquelles l'ordre est important, ainsi $(3.2\,,5) \neq (5\,,3.2)$.

On peut aussi créer des p-uplets avec des éléments de type différent : $("AEYU"\,,4\,,True)$ est un 3-uplet.

En résumé le cahier des charges d'un p-uplet est donné ci-contre.

p-uplet

- séquence ordonnée
- de p éléments
- de types différents ou pas

2) Les p-uplets en python (point de vue programmation)

♦ on les définit à l'aide de parenthèses, en séparant les éléments par des virgules :

```
>>> mon_triplet = (1024, 99, True)
```

♦ on peut lire les différents éléments grâce à des crochets en commençant à l'indice 0 :

```
>>> mon_triplet[0]
```

```
1024
>>> mon_triplet[1]
99
>>> mon_triplet[2]
True
```

♦ on peut aussi les définir sans mettre les parenthèses, mais le code est moins lisible :

```
>>> mon_doublet = 77, 88
>>> mon_doublet
(77, 88)
```

♦ En python les p-uplets sont de type `'tuple'` :

```
>>> type(mon_doublet)
<class 'tuple'>
```

♦ En python les p-uplets ne sont PAS modifiables (on dit *immutables*) :

Si on veut modifier le premier élément de `mon_doublet` pour lui donner la valeur 10101 au lieu de 77 :

```
>>> mon_doublet[0]=10101
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    mon_doublet[0]=10101
TypeError: 'tuple' object does not support item assignment
```

On verra que si on souhaite modifier des données il faut utiliser un tabelau à la place d'un p-uplet.

♦ En python, la taille d'un p-uplet s'obtient avec la fonction `len()` :

```
>>> len(mon_doublet)
2
```

3) 3 utilisations classiques des tuples en python

Utilisation en python n°1 : affectation multiple

♦ Les p-uplets sont utiles pour faire plusieurs affectations en une seule instruction :

```
>>> a, b, c = 1024, 99, True
>>> a
1024
>>> b
99
>>> c
True
```

On aurait aussi pu faire l'un des trois autres cas suivants :

```
>>> a, b, c = (1024, 99, True)
>>> (a, b, c) = (1024, 99, True)
>>> (a, b, c) = 1024, 99, True
```

Utilisation en python n°2 : échange de variables

♦ L'affectation multiple permet d'échanger très facilement les valeurs de plusieurs variables. Ainsi pour échanger a et b :

```
>>> a = 8
>>> b = 555
>>> a, b = b, a
>>> a
555
>>> b
8
```

Utilisation en python n°3 : Retourner plusieurs valeurs

♦ Il arrive souvent qu'une fonction calcule plusieurs valeurs et qu'on doive toutes les retourner …

```
>>> def moyenne_et_ecart(a, b):
      m = (a+b)/2
      e = abs(b-a)
      return (m, e)

>>> moy, ec = moyenne_et_ecart(12, 16)
>>> moy
```

```
14.0
>>> ec
4
```

Pour récupérer les valeurs on aurait aussi pu faire :

```
>>> stats = moyenne_ecart(12, 16)
>>> stats
(14.0, 4)
```

Remarque : abs retourne la valeur absolue (c-à-d la version "positive") d'un nombre : abs(4) = 4, abs(-6) = 6

III. Les tableaux

1) Définition (point de vue algorithmique)

Un tableau est une séquence de taille fixe et indexée d'éléments de même type.
De plus, l'accès à un élément (lecture ou écriture) doit se faire en temps constant, quelle que soit la taille du tableau et la position de l'élément dans le tableau.

Tableau

- séquence indexée
- de p éléments
- de même type
- accès lecture/écriture en temps constant

Voici une représentation d'un tableau d'entiers de taille 9 : la ligne du dessus donne les éléments, la ligne du dessous donne les indices (qui débutent toujours toujours toujours à zéro pour le premier élément).

12	15	16	8	19	9	14	15	11
0	1	2	3	4	5	6	7	8

Un point important est que d'un point de vue algorithmique les tableaux ne supportent pas l'insertion ou la suppression d'éléments.

2) Quel est le problème de l'insertion/suppression d'éléments dans un tableau ?

Un tableau indexé n'est pas fait pour supporter efficacement ces opérations. Supposons par exemple qu'on veuille insérer le nombre 7 entre les nombres 12 et 15 du tableau ci-dessus. Voilà de façon simpliste ce qui doit être effectué sur la mémoire de la machine pour y parvenir :

a) ajout d'un indice

12	15	16	8	19	9	14	15	11	
0	1	2	3	4	5	6	7	8	9

b) Recopie des 8 valeurs de la ligne supérieure dans la case voisine (en commençant par 11 puis 15 …)

12		15	16	8	19	9	14	15	11
0	1	2	3	4	5	6	7	8	9

c) Copie de la valeur 7 à l'endroit désiré

12	**7**	15	16	8	19	9	14	15	11
0	1	2	3	4	5	6	7	8	9

On voit tout de suite le problème : l'étape b) coûte très très cher. Il faut imaginer avec un tableau de plusieurs millions de valeurs : on se retrouve à effectuer des millions d'opérations pour insérer … un seul élément. L'insertion dans un tableau peut donc coûter beaucoup plus cher que la modification d'un élément dans un tableau. Donc pour maîtriser l'efficacité d'un algorithme utilisant des tableaux on évite l'insertion ou la suppression d'éléments.

Exemple en python sur un PC familial (l'insertion en début de tableau coûte très cher) :

- Si on effectue 7 777 fois de suite une **insertion à l'indice 0** sur un tableau de 1 000 000 nombres entiers

(qui finit donc avec une taille 1 007 777) on a un temps d'exécution d'environ 8 s.

- Si on effectue 7 777 fois de suite une **insertion à l'indice 1000 000** sur un tableau de 1 000 000 nombres entiers (qui finit donc avec une taille 1 007 777) on a un temps d'exécution d'environ 24 ms.

- Si on effectue 7 777 fois de suite une **modification à l'indice 0** sur un tableau de 1 000 000 nombres entiers (qui reste donc avec une taille 1 000 000) on a un temps d'exécution d'environ 9 ms.

3) Tableaux en python (point de vue programmation) : les bases

♦ On les définit à l'aide de crochets, en séparant les éléments par des virgules (on parle de création en extension) :

```
>>> mon_tableau = ["Alfonso", "Chiara", "Lino"]
```

♦ On peut lire les différents éléments grâce à des crochets en commençant à l'indice 0 :

```
>>> mon_tableau[0]
"Alfonso"
>>> mon_tableau[1]
"Chiara"
>>> mon_tableau[2]
"Lino"
```

♦ En python les tableaux sont de type `'list'` :

```
>>> type(mon_tableau)
<class 'list'>
```

♦ On modifie naturellement les éléments d'un tableau :

Si on veut modifier le premier élément de `mon_tableau` pour lui donner la valeur "Paola" :

```
>>> mon_tableau[0]="Paola"
>>> mon_tableau
['Paola', 'Chiara', 'Lino']
```

♦ En python, la taille d'un tableau s'obtient avec la fonction `len()` :

```
>>> len(mon_tableau)
3
```

♦ En python, on peut aussi accéder aux derniers éléments d'un tableau avec les indices -1, -2 … :

```
>>> tab_lettres = ["a", "b", "c", "d", "e", "f", "g", "h"]
>>> tab_lettres[7]
'h'
>>> tab_lettres[-1]
'h'
>>> tab_lettres[-2]
'g'
```

♦ En python on peut facilement créer un tableau vide :

```
>>> T = []
>>> T
[]
```

IV. Construction de tableaux en python par compréhension ou .append()

1) Construction de tableaux par compréhension

On a souvent besoin de construire des tableaux sans écrire tous les éléments un à un. La méthode par compréhension consiste à construire un tableau en parcourant un autre objet (range, chaîne de caractères, autre tableau, etc…).

Rappels :

On peut parcourir un `range` avec `for` :

```
>>> for nbre in range(5, 15, 2):
        <instructions>
```

Ou parcourir une chaîne de caractères :

```
>>> for car in "Barbapapa":
        <instructions>
```

Voici quelques exemples classiques :

```
>>> T1 = [car + car for car in "PYtHoN"]
>>> T1
['PP', 'YY', 'tt', 'HH', 'oo', 'NN']
```

```
>>> T2 = [nbr for nbr in range(5, 15)]
>>> T2
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

```
>>> T3 = [car for car in "hallucinatoire" if car not in "aeiouy"]
>>> T3
['h', 'l', 'l', 'c', 'n', 't', 'r']
```

```
>>> T4 = [x*x for x in range(11)]
>>> T4
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Et puisqu'on peut aussi parcourir un tableau on a aussi l'exemple suivant :

```
>>> tab_mots = ["ici", "la", "le", "ceci", "cela", "oui", "non", "tic", "tac", "pif"]
>>> garder_c = [mot for mot in tab_mots if "c" in mot]
>>> garder_c
['ici', 'ceci', 'cela', 'tic', 'tac']
```

2) Construction quand on ne connait pas à l'avance le nombre d'éléments du tableau

On est alors obligé d'utiliser la méthode .append() qui rajoute un élément à la fin d'un tableau :

```
>>> T = [10, 9, 8, 7]
>>> T.append(6)
>>> T
[10, 9, 8, 7, 6]
```

Exemple d'utilisation avec une boucle (tant qu'on a pas un 6 au tirage au sort, on remplit le tableau) :

```
>>> import random
>>> tirages_du_de = []
>>> de = -1
>>> while de != 6:
      de = random.randint(1,6)
      tirages_du_de.append(de)

>>> tirages_du_de
[1, 3, 2, 3, 2, 5, 6]
```

V. Le cas des matrices

Une matrice est un tableau de tableaux de même longueur comportant tous des éléments du même type. C'est un objet extrêmement utilisé en informatique (représentation de graphes, représentation d'images, outil de calcul mathématique etc.)

Voici une représentation d'une matrice M comportant m = 3 lignes et n = 4 colonnes. On peut voir la matrice M comme un tableau contenant 3 tableaux de 4 éléments :

0	2	4	6
30	32	37	39
47	48	49	49

```
>>> M = [ [0, 2, 4, 6], [30, 32, 37, 39], [47, 48, 49, 49] ]
>>> M[2]
[47, 48, 49, 49]
>>> M[2][1]
48
>>> M[1][2]
37
```

Ainsi on représente aisément une matrice M comme un tableau de lignes. Dans ce cas :
- M[i] désigne le i-ème tableau de M c'est-à-dire la i-ème ligne
- M[i][j] désigne le j-ème élément du i-ème tableau de M c'est-à-dire la valeur située à (ligne i, colonne j)

REMARQUE : c'est une convention quasi-universellement partagée de représenter une matrice comme un tableau de lignes. Il serait très maladroit de représenter une matrice comme un tableau de colonnes.

VI. "Copie" de tableaux : ATTENTION !

Nous ne nous attarderons pas sur ce point mais sachez que l'affectation de tableaux se fait par "étiquetage" et non par "valeur".

Lorsque vous manipulez des entiers, vous faites des *affectations par valeur* : lorsque vous "copiez" une variable a dans une variable b, vous faites une copie de a dans b. Ainsi vous avez deux "exemplaires" de votre valeur en mémoire : l'exemplaire a et l'exemplaire b qui sont indépendants. Si vous modifiez l'un des deux, l'autre n'est pas impacté :

```
>>> a = 7
>>> b = a
>>> a = 9
>>> a
9
>>> b
7
```

Si on fait la même manipulation avec a et b des tableaux, tout va se passer comme si a et b étaient deux "étiquettes" attachées au même tableau (comme deux badges avec deux prénoms différents sur la même personne).

Conséquence : si par la suite vous modifiez le tableau avec l'étiquette a, vous modifiez aussi le tableau avec l'étiquette b (et pour cause : c'est le MÊME tableau !).

```
>>> a = [7, 77, 777]
>>> b = a
>>> a[2] = 999
>>> a
[7, 77, 999]
>>> b
[7, 77, 999]
```

Il y a une très bonne raison : un tableau est supposé a priori long, lourd et contenant des milliers de valeurs. Donc au lieu de faire une vraie copie (qui serait longue pour le programme), le langage crée juste une 2e petite étiquette attachée au premier tableau … c'est moins lourd.

Ce comportement par défaut est vrai dans la majorité des langages de programmation ET en algorithmique. Si on veut vraiement faire une vraie copie d'un tableau, il faut utiliser une instruction explicite telle que .deepcopy() en python. Dans ce cas on crée en mémoire un "clone" du premier tableau qui est donc indépendant du premier tableau.

```
>>> a = [7, 77, 777]
>>> b = a.deepcopy()
>>> a[2] = 999
>>> a
[7, 77, 999]
>>> b
[7, 77, 777]
```

Remarque : cette partie est surtout là pour vous aider en cas de bug prolongé et incorrigible dans un programme. Si vous manipulez des tableaux, que tout vous semble correct mais que "ça ne marche pas", l'erreur peut venir de ce problème de copie par "étiquette" (on dit par *référence*). Donc si à un endroit vous copiez un tableau … méfiance !