Dictionnaires

Table des matières

I. Révisions sur les types composés en langage Python

1) Rappel de quelques définitions

a) Expression

Définition

Une **expression** est une *suite de caractères* qui possède une **valeur**. Parler de la valeur d'une expression n'a de sens que dans le contexte d'un programme informatique donné : celui-ci "lit" l'expression et lui attribue une valeur ; on dit que le programme **évalue l'expression**.

Dans la suite nous nous intéresserons aux expressions et à leurs valeurs pour le langage Python. Voici des exemples d'expressions qu'on peut utiliser en Python :

$$42 \quad 2+5 \quad 2*3.0 \quad 'Bonjour' \quad x+3 \quad [1,2,5] \quad 2*[1,2] \quad 2==5$$

Pour connaître la valeur d'une expression, on l'écrit dans l'interpréteur de l'éditeur utilisé et on appuie sur la touche <Entrée>. La valeur est alors affichée dans l'interpréteur. *Si Python ne peut pas lui attribuer de valeur, c'est un message d'erreur qui s'affiche.*

Exemples :

-
-
-
-
-
-
-

Les expressions utilisées par Python sont regroupées par **types**. Formellement un **type** est un sous-ensemble de l'ensemble de toutes les expressions permises par le langage de programmation (Python ici). La connaissance du **type** permet de définir les **opérations** qu'on peut faire avec les expressions appartenant à ce **type**.

Le langage Python reconnaît les types suivants (la liste n'est pas exhaustive mais elle suffit pour notre usage) :

type	Description sommaire	Exemples de valeurs
`int`	ensemble des entiers	134 -12
`float`	ensemble des flottants	3.14 -12.3e-6
`bool`	ensemble des booléens	uniquement deux valeurs possibles : `True` et `False`
`NoneType`	ensemble à un élément	uniquement une expression et une valeur : `None`
`str`	ensemble des chaînes de caractères	`"Coucou c'est moi"` 2*"Bonjour"
`tuple`	ensemble des n-uplets	`(2, "chien", 45.67)`
`list`	ensemble des listes Python	`[1, 2.45, "framboise"]`
`dict`	ensemble des dictionnaires	`{ "pseudo" : "Momo123", 2.34 : "ok", "tél" : 0645}`

Pour afficher le type d'une expression, on utilise le mot clé `type` suivi de l'expression entre parenthèses dans l'interpréteur et on appuie sur la touche <Entrée>.

b) Variable

Définition

Dans un langage de programmation, une **variable** est constituée de l'association :

- d'une *zone* de la mémoire vive de l'ordinateur dans laquelle on peut stocker une valeur (de type entier, flottant, booléen, etc ...).
- de la *valeur qui est stockée dans cette zone*. Cette valeur stockée peut changer au cours du programme. Cette valeur est aussi appelée *contenu de la variable*.

Une variable possède un **nom de variable** aussi appelé **identifiant de variable**. Il sert à désigner la zone de la mémoire (on dit que c'est une référence à la zone de la mémoire) de la variable.

Cependant, par abus traditionnel de langage, il nous arrivera souvent d'écrire : " soit x une variable ..." au lieu de dire " soit une variable d'identifiant x ..."

Affectation d'une variable

En Python, une variable est créée ou modifiée lorsqu'on écrit dans l'interpréteur ou dans le fichier du programme :

```
nom_variable = expression
```

Ceci est une **instruction** appelée *affectation*. Cette instruction a pour effet :

1. de *réserver* une zone de la mémoire à laquelle le programme va se référer grâce à l'identifiant `nom_variable` dans le cas où c'est la première fois qu'on écrit cette instruction (il y a alors création de la variable) ;
2. d'y stocker la valeur de l'expression indiquée à droite de l'égalité ou bien de remplacer l'ancienne valeur qui y était déjà stockée si la variable existait déjà avant. L'opérateur = est appelé **opérateur d'affectation**.

Le type d'une variable est celui de l'expression qui y est stockée. Le langage Python autorise que ce type soit changé au cours du programme.

Exemple :

```
x = 3
...
x = "Bonjour"
```

2) Les n-uplets ou tuples

Définition

Un n-uplet (tuple en anglais) est une suite finie de n expressions e_0, e_1, ..., e_{n-1} séparées par des virgules et *délimitée par deux parenthèses*. On écrit donc :

$$(e_0, e_1, ..., e_{n-1})$$

La valeur de ce n-uplet (objet de type `tuple`) s'obtient en remplaçant toutes les expressions e_i par leurs valeurs. Par exemple :

Les différentes expressions peuvent être de *n'importe quel type* : `int`, `float`, `str`, `bool` ... mais aussi `tuple` (et `list`, voir plus loin) :

Chaque expression e_i d'un tuple s'appelle un **élément** de ce tuple. On peut construire un tuple avec un seul élément mais il ne faut alors **pas oublier de mettre une virgule** :

Opérations permises :

- La concaténation + :

- Opération du type $n*$ où $n \in \mathbb{N}$:

On peut bien sûr déclarer - intialiser ou réaffecter des variables de type tuple :

```
t1 = (1,2.2)
t2 = ( 2*3, 'Hello', (1,2) )
```

Dans l'expression à droite de l'opérateur `=`, les parenthèses sont *facultatives*. On peut donc parfaitement écrire :

```
t2 = 2*3, 'Hello', (1,2)
```

Nombre d'éléments d'un tuple :

Si `t` est une expression de type `tuple`, pour connaître son nombre d'éléments on écrit `len(t)` Par exemple :

```
len( (1,2.2) )                    len(t2)
```

Accès aux éléments et définition d'un sous-tuple :

- Chaque élément d'un `tuple` est caractérisé par son *indice* : cet indice va de 0 (premier élément à gauche) jusqu'à $n-1$ pour le dernier élément à droite. Si `t` est unevariable de type `tuple` alors `t[i]` est l'élément d'indice i.

- Si `t` est une variable de type `tuple` alors `t[i:j]` est le *sous-tuple* extrait de `t` dont les éléments vont de `t[i]` à `t[j-1]`.

Accès à un élément d'un tuple élément d'un autre tuple :

Si on dispose de la variable `t = (1,2,(4,5))` comment fait-on pour accéder à l'élément `4` du tuple élément `(4,5)` de `t` ?

Déconstruction d'un tuple :

On peut déconstruire un tuple en affectant ses éléments à différentes variables *en une seule instruction* :

```
x, y = (1,2.2)
```

De même pour la séquence de deux instructions suivantes :

```
t = (1,2.2)
x, y = t
```

Il est important de savoir que l'expression située à droite de l'opérateur `=` est toujours **évaluée avant de réaliser l'affectation**. On peut donc s'en servir pour échanger les valeurs de deux variables en un seul coup :

Test d'appartenance :

On peut tester si une valeur appartient à un tuple grâce à l'opérateur `in` . Par exemple :

`3 in (1,2,3)` $\longrightarrow$ `'Pomme' in (2.2, 3, "Poire")` $\longrightarrow$

Parcours des éléments d'un tuple dans une boucle for :

```
for elt in (1,2.2, "Bonjour") :
   print(elt)
```

Un tuple est immuable (on dit aussi non mutable) :

Si `t` est une expression de type `tuple`, par exemple une variable `(t : (1,2.2,'Hello') )`, alors l'instruction suivante est *interdite* par Python : elle provoque une erreur. On dit qu'un tuple est *immuable* (ou *non mutable*) :

```
t[0] = 4
```

3) Les listes

Définition

Une liste est une suite finie de n expressions e_0, e_1, ..., e_{n-1} séparées par des virgules et délimitée par deux **crochets** `[` et `]`. On écrit donc une liste de la façon suivante :

$$[e_0, e_1, ..., e_{n-1}]$$

La valeur de cette expression de type `list` s'obtient en remplaçant toutes les expressions e_i par leurs valeurs.

Par exemple :

`[ 2*3, 'Pomme' + 'Poire', 2.3 ]` $\longrightarrow$ `[ 6, 'PommePoire', 2.3 ]`

`type( [ 2*3, 'Pomme' + 'Poire', 2.3 ] )` $\longrightarrow$ `< class 'list' >`

Une liste est un tuple dont on a *remplacé les parenthèses par des crochets*. Tout ce qui a été dit pour les tuples est valable pour les listes à l'exception de deux choses :

1. Dans une instruction simple de déclaration initialisation ou de réaffectaction on ne peut pas omettre les crochets à droite de l'opérateur `=`

   ```
   L = [1,2,3]
   ```

   mais :

   ```
   L = 1,2,3
   ```

2. Une liste est **mutable** (très gros avantage) :

   ```
   L = [1, 2.2, 'Pomme']
   L[0] = 2
   print(L)
   ```

Copie de liste : attention danger !

Si on dispose d'une variable de type `list`, par exemple `L = [1,2,3]` on a souvent besoin d'en faire une copie. On peut alors vouloir écrire l'instruction :

```
L2 = L
```

.

Méthodes de Listes

La notion de méthode est issue directement de la programmation orientée objet. Pour faire simple on va se contenter pour le moment de dire qu'une variable de type `list`, par exemple `( MaListe : [1,2,3] )`, possède des **méthodes**. Une méthode est caractérisée par une **action** (= ce qu'elle fait) et une **valeur** (= ce qu'elle vaut). Voyons des méthodes de listes souvent utilisées :

1. **append** : `MaListe.append(4)`

2. **pop** : `MaListe.pop()`

3. **reverse** : `MaListe.reverse()`

4. **sort** : `MaListe.sort()`

5. **copy** : `MaListe.copy()`

4) Les dictionnaires

Un **dictionnaire** généralise la structure de liste. Dans une liste, chaque élément est repéré par son indice i dans la liste, i étant un entier. Dans un dictionnaire les indices sont remplacés par des **clés** qui peuvent être des chaînes de caractères, des entiers, de flottants, ...

Un dictionnaire est une **collection de couples** `(clé,valeur)`

Les contraintes et autorisations sont les suivantes :

- Chaque `clé` ne peut être *présente qu'une seule fois* dans le dictionnaire et ne peut être que d'un type **non mutable** (on dit aussi *immuable*). Ainsi une clé peut être du type :
  - entier (`int`) ou flottant (`float`)
  - chaîne de caractères (`str`)
  - tuple (`tuple`)
- À chaque clé est associée une `valeur` qui peut être de *n'importe quel type* : entier, flottant, chaîne de caractères, tuple, liste, fonction, ... Une valeur peut être aussi un autre dictionnaire, ce qui permet de créer des emboîtements.
- Une `valeur` donnée peut éventuellement être associée à plusieurs `clés` différentes.

On exige qu'un dictionnaire possède toutes les opérations suivantes :

- Ajouter un couple `(clé,valeur)`
- Supprimer un couple `(clé,valeur)`
- Rechercher la `valeur` associée à une `clé` donnée

En langage Python on déclare - initialise une variable de type **dictionnaire** de la façon générale suivante :

```
d = { clé1 : valeur1,
      clé2 : valeur2,
      clé3 : valeur3,
           ...
      clén : valeurn
    }
```

Exemples :

```
d = { "pseudo" : "Momo123", "age" : 26, 1 : 324 }
```

Un dictionnaire est de type `dict` :

```
type(d)
```

```
<class 'dict'>
```

On peut créer un dictionnaire vide :

```
dvide = {}
```

On accède ensuite à une `valeur` du dictionnaire grâce à sa `clé`. Cet accès peut se faire aussi bien en lecture qu'en écriture :

```
v = d["pseudo"]    # accès en lecture
print(v)
```

```
Momo123
```

```
d["age"] = 18    # accès en écriture
print(d)
```

```
{"pseudo" : "Momo123", "age" : 18, 1: 324}
```

De cette façon, une valeur associée à une clé peut être modifiée à tout moment ; on dit qu'un dictionnaire est **mutable**. Avec l'instruction précédente, si la `clé` n'existe pas, elle est tout simplement créée et on lui associe sa `valeur` :

```
d[(1,2)] = "kesako ?"     # insertion d'un couple (clé,valeur)
print(d)
```

```
{"pseudo" : "Momo123", "age" : 18, 1: 324, (1,2) : "kesako ?"}
```

Il est tout aussi facile de supprimer un couple `(clé,valeur)`. Cela se fait encore par l'intermédiaire de la `clé` au moyen de :

```
del d[(1,2)]    # suppression d'un couple (clé,valeur)
print(d)
```

```
{"pseudo" : "Momo123", "age" : 18, 1: 324}
```

L'ensemble des couples (clé,valeur), l'ensemble des clés et l'ensemble des valeurs présentes dans un dictionnaire `d` s'obtiennent respectivement grâce aux **méthodes** `items`, `keys` et `values`, c'est à dire :

```
d.items()   d.keys()   d.values()
```

Ces méthodes permettent de tester l'appartenance à un dictionnaire donné (grâce à l'pérateur `in`) et de faire des boucles `for` :

```
("pseudo","Momo123") in d.items()    # expression de type booléen
```

```
True
```

```
"age" in d.keys()    # expression de type booléen
```

```
True
```

```
47 in d.values()    # expression de type booléen
```

```
False
```

La négation de `in` s'écrit `not in` ; on peut alors facilement tester si un couple (clé, valeur), une clé ou une valeur ne sont pas présentes dans un dictionnaire donné :

```
"Charlie" not in d.values()    # expression de type booléen
```

```
True
```

Pour les boucles, cela se passe de la manière suivante :

```
for (c,v) in d.items() :  # boucle sur les couples (clé, valeur) de d
    ...
```

```
for c in d.keys() :  # boucle sur les clés de d
    ...
```

```
for v in d.values() :  # boucle sur les valeurs de d
    ...
```

Le nombre de couples (clé, valeur) présentes dans un dictionnaire `d` s'obtient grâce à `len(d)`

Enfin, la duplication d'un dictionnaire se fait comme pour les listes grâce à la méthode `copy`. Pour le dictionnaire `d` cela donne :

```
nouveauDico = d.copy()
```

Exercices :

1. On dispose d'une chaîne de caractères `"Barbara ou barbe à papa"`. Créer un dictionnaire dont chaque clé est un des caractères de cette chaîne et dont les valeurs sont les nombres de fois où le caractère est présent dans la chaîne. On commencera par :

   ```
   ch = "Barbara ou barbe à papa"
   analyse = {}
   ```

2. On considère la suite (u_n) définie de façon récursive de la façon suivante :

   $$u_0 = 2 \quad \text{et} \quad \forall\, n \in \mathbb{N}^*,\ u_n = 3\,u_{n-1} - 1$$

   Écrire une fonction `suite(n)` qui renvoie u_n en organisant les calculs de façon itérative

   a) en utilisant une liste ;

   b) en utilisant un dictionnaire.

II. La technique de mémoïsation

Nous allons développer deux exemples pour présenter cette technique qui repose sur l'utilisation d'un dictionnaire.

1) Le problème de la suite de Fibonacci

On souhaite calculer les termes de la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ en utilisant la définition suivante :

$$F_0 = 0 \ ; \ F_1 = 1 \quad \text{et} \quad \forall\, n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$$

a) Méthode itérative

Algo1 :

Si $n = 0$ ou $n = 1$ renvoyer n. Sinon créer un tableau T d'entiers de taille $n + 1$ servant à stocker les F_i de 0 à n. Y placer F_0, F_1 et initialiser tous les autres éléments à 0. Pour i variant de 2 à n, faire $T[i] = T[i-1] + T[i-2]$. Renvoyer $T[n]$.

En langage Python, un tableau peut être implémenté avec une **liste** ou bien avec un **tableau numpy unidimensionnel**.

Mise en œuvre :

1. Écrire l'algo1 en langage Python sous la forme d'une fonction `Fibo1(n)`.
2. Évaluer sa complexité temporelle et spatiale

Algo2 : version avec un dictionnaire

L'utilisation d'un dictionnaire apporte un peu de souplesse puisqu'on n'a pas besoin de créer au début un tableau de taille $n + 1$. Le dictionnaire joue ce rôle et grossit au fur et à mesure que progresse l'algorithme.

Mise en œuvre : Écrire l'algo2 en langage Python sous la forme d'une fonction `Fibo2(n)`.

b) Méthode récursive

On peut toujours adopter une méthode récursive mais il faut alors se méfier de la complexité temporelle.

Algo3(n) : version récursive naïve

Si $n = 0$ ou bien $n = 1$, renvoyer n (condition de terminaison des appels récursifs). Sinon calculer val = Algo3($n-1$) + Algo3($n-2$) et renvoyer val.

Mise en œuvre : Écrire l'algo3 en langage Python sous la forme d'une fonction `Fibo3(n)`.

Malheureusement, avec cette version le temps d'attente de la solution devient très long, même avec des valeurs de n assez petites : plusieurs secondes à partir de $n = 30$, de l'ordre de la minute pour $n = 40$, etc ... Le problème avec cette version est que :

- Le nombre d'appels récursifs devient "exponentiel" lorsque n est grand. Si Nb(n) est le nombre d'appels d'une des fonctions Fibo3(i) nécessaires pour trouver Fibo3(n), on peut montrer que :

$$\mathrm{Nb}(n) \underset{\infty}{\simeq} \frac{2}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^{n+1}$$

Avec un ordinateur cadencé à 2 GHz, chaque instruction prend au moins un temps $\tau = 5.10^{-10}$ s. Comme l'exécution de Fibo3(i) prend forcément un temps strictement supérieur à τ (et même largement supérieur !), la durée d'exécution de Fibo3(100) vérifie :

$$\Delta t > \frac{2}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{101} \times 5.10^{-10} = 1{,}15.10^{21} \text{ s} = 36\,324 \text{ milliards d'années}$$

- Un des problèmes de cette version est que l'ordinateur **refait plusieurs fois le même calcul**. Cela se voit bien en dessinant l'arbre des appels récursifs. La Figure 1 en donne l'exemple avec Fibo3(5).

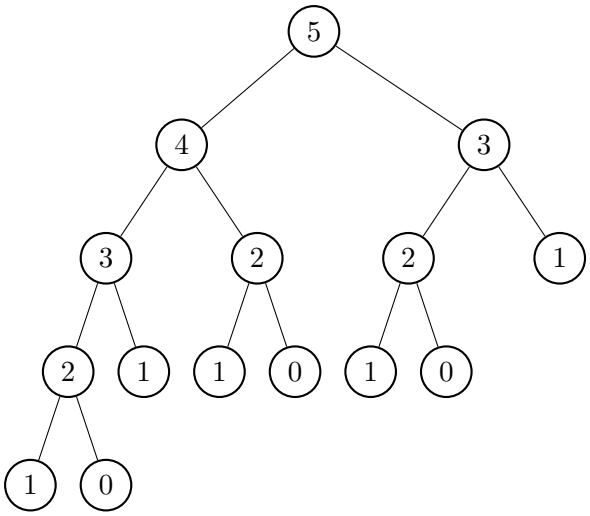

FIGURE 1 – Arbre des appels récursifs pour Fibo3(5). Fibo3(3) est appelé deux fois et Fibo3(2) trois fois

On parle de **chevauchement** des sous-problèmes. Une bonne idée pour réduire le nombre d'appels récursifs est de *stocker les résultats intermédiaires dans un tableau ou bien un dictionnaire*, afin de ne pas avoir à les recalculer : c'est la **mémoïsation** [1].

Algo4(n) : version récursive avec mémoïsation par utilisation d'un dictionnaire

Si Algo4(n) déjà présent dans le dictionnaire, renvoyer Algo4(n). Sinon si $n = 0$ écrire 0 dans le dictionnaire et renvoyer 0. Sinon si $n = 1$, écrire 1 dans le dictionnaire et renvoyer 1. Sinon calculer val = Algo4($n-1$) + Algo4($n-2$), l'écrire dans le dictionnaire, puis renvoyer val.

Mise en œuvre : Écrire l'algo4 en langage python sous la forme d'une fonction `Fibo4(n)`. On déclarera le dictionnaire comme variable globale.

Conclusion :

Fibo3(20) conduit à 21 930 appels récursifs tandis que Fibo4(20) n'en provoque que 39. Le gain de temps est significatif.

1. Ce n'est pas une faute d'orthographe. Le mot est bien *mémoïsation* et non *mémorisation*.

2) La formule du binôme de Newton

Prenons la célèbre formule du binôme :

$$\boxed{\forall\,(a,b)\in\mathbb{R}^2,\ \forall\,n\in\mathbb{N},\ (a+b)^n=\sum_{p=0}^{n}\binom{n}{p}\,a^p\,b^{n-p}}$$

Les coefficients $\binom{n}{p}$ peuvent être calculés au moyen d'une récurrence ; c'est ce qu'exprime le triangle de Pascal mais une façon plus commode de représenter cela est de créer une matrice M d'ordre $(n+1)\times(p+1)$ et d'y mettre des zéros lorsque les coefficients sont sans intérêt.

$$\begin{array}{l}1\\1\quad 1\\1\quad 2\quad 1\\1\quad 3\quad 3\quad 1\\\text{etc ...}\end{array}\qquad M=\begin{bmatrix}1&0&0&0&\dots&0\\1&1&0&0&\dots&0\\1&2&1&0&\dots&0\\1&3&3&1&\dots&0\\1&\dots&\dots&\dots&\dots&0\end{bmatrix}$$

Le coefficient de M situé à l'intersection de la ligne i $(0\leqslant i\leqslant n)$ et de la colonne j $(0\leqslant j\leqslant n)$ est :

$$\binom{i}{j}\quad\text{et on convient que }\binom{i}{j}=0\text{ si }j>i$$

La relation de récurrence s'écrit :

$$\boxed{\forall\,(i,j)\in[\![1,n]\!]\times[\![1,p]\!],\ \binom{i}{j}=\binom{i-1}{j}+\binom{i-1}{j-1}}$$

avec les conditions :

$$\boxed{\forall\,i\in[\![0,n]\!],\ \binom{i}{0}=1\quad\text{et}\quad\forall\,j\in[\![1,p]\!],\ \binom{0}{j}=0}$$

a) Méthode itérative

Algo1(n,p) :

Si $p>n$ renvoyer 0. Sinon si $p=0$ renvoyer 1. Sinon créer une matrice M de taille $(n+1)\times(p+1)$ servant à stocker les $\binom{i}{j}$, remplir la première colonne de 1 et initialiser tous les autres coefficients à 0. Pour i variant de 1 à n faire, pour j variant de 1 à p, faire $M[i,j]=M[i-1,j]+M[i-1,j-1]$. Renvoyer $M[n,p]$.

En langage Python, un matrice peut être **une liste de listes** ou bien une **matrice numpy**. Nous allons travailler ici avec une liste de listes.

Mise en œuvre :

1. Traduire l'algo1 en langage python à l'aide d'une fonction `BinomeNewt1(n,p)`

2. Évaluer sa complexité temporelle et sa complexité spatiale.

   Une version alternative de cet algo. utilise un dictionnaire qui se construit au fur et à mesure du déroulement du processus. Les clés du dictionnaire sont les couples (i, j) (tuples) et les valeurs associées sont les $\binom{i}{j}$.

3. Écrire une version de l'algo1 sous la forme d'une fonction `BinomeNewt2(n,p)` utilisant un dictionnaire à la place de la liste de listes.

b) Méthode récursive

Algo3(n,p) : commençons par une version récursive naïve.

Si $p > n$ renvoyer 0. Sinon si $p = 0$ renvoyer 1. Sinon calculer val = Algo3$(n-1, p)$ + Algo3$(n-1, p-1)$. Renvoyer val.

Mise en œuvre : traduire l'algo3 en langage python à l'aide d'une fonction `BinomeNewt3(n,p)`.

On rencontre alors les mêmes types de problèmes qu'avec la version récursive naïve de la suite de fibonacci. Le problème de cette méthode est que lorsque n augmente et que p est proche de $\lfloor n/2 \rfloor$, le temps de calcul devient long et que le nombre d'appels récursifs explose.

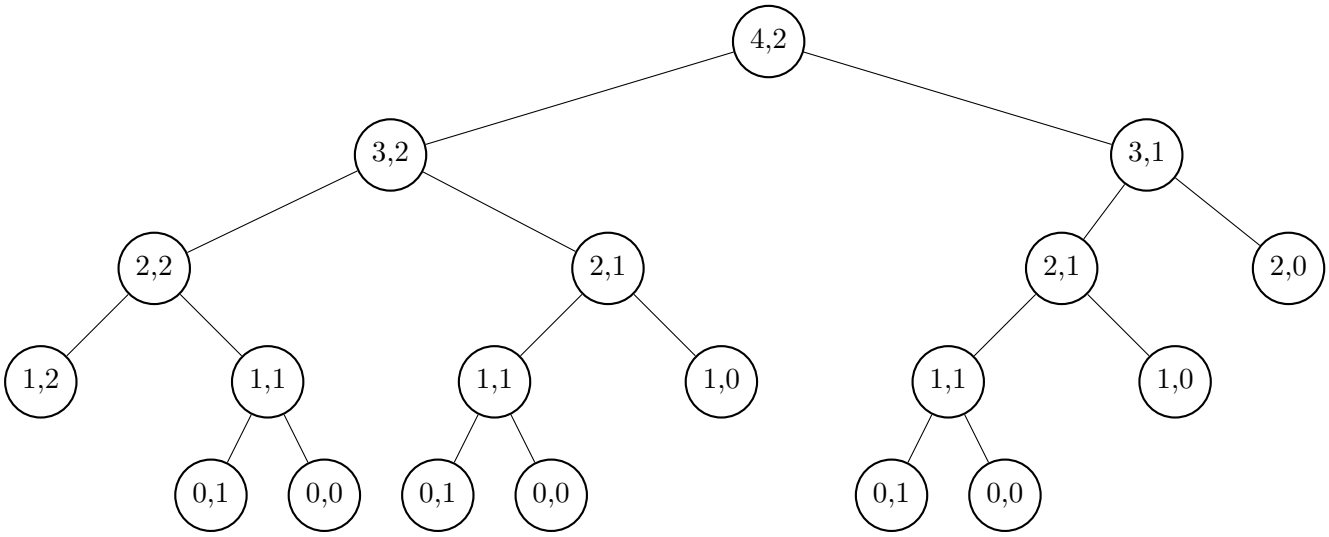

FIGURE 2 – Arbre des appels récursifs provoqués par BinomeNewt3(4,2)

En effet, le nombre d'appels $\mathrm{Nb}(n, p)$ de `BinomeNewt3(n,p)` vérifie l'équation :

$$\mathrm{Nb}(n, p) = 1 + \mathrm{Nb}(n-1, p) + \mathrm{Nb}(n-1, p-1)$$

et on montre facilement que :

$$\boxed{\forall\, n \in \mathbb{N},\ \forall\, p \in \mathbb{N},\ \mathrm{Nb}(n, p) \geqslant \binom{n}{p} \quad \text{en convenant que} \quad \binom{n}{p} = 0 \text{ si } p > n}$$

Il suffit par exemple de montrer par récurrence sur n que l'assertion :

$$A(n) : \forall\, p \in \mathbb{N},\ \mathrm{Nb}(n, p) \geqslant \binom{n}{p}$$

est toujours vraie.

En outre, un même calcul peut être effectué plusieurs fois. On peut le voir facilement en dessinant l'arbre des appels récursifs de `BinomeNewt3(4,2)` : il y a à nouveau un **chevauchement** des sous-problèmes.

Prenons le cas critique où $n = 2p$ et utilisons la formule de Stirling pour donner un ordre de grandeur asymptotique du nombre d'appels lorsque p devient très grand. La formule est :

$$n! \underset{\infty}{\simeq} \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

Ici de même une *technique de mémoïsation* permet de réduire le nombre de calculs. On *stocke les résultats intermédiaires dans un dictionnaire* dont les clés sont les couples (i, j) (tuples) et les valeurs $\binom{i}{j}$. Cela donne :

Algo4(n,p) : version récursive avec mémoïsation (à l'aide d'un dictionnaire)

Si Algo4(n, p) déjà présent dans le dictionnaire, renvoyer Algo4(n, p). Sinon si $p > n$, écrire 0 dans le dictionnaire et renvoyer 0. Sinon si $p = 0$ écrire 1 dans le dictionnaire et renvoyer 1. Sinon calculer val = Algo4$(n-1, p)$ + Algo4$(n-1, p-1)$, l'écrire dans le dictionnaire, puis renvoyer val.

Mise en œuvre : traduire l'algo4 en langage python à l'aide d'une fonction `BinomeNewt4(n,p)`. On déclarera un dictionnaire `dico` en tant que variable globale.

Exemple :

Le nombre d'appels effectués par `BinomeNewt3(16,8)` est de 48 619 avant d'afficher le résultat (12 870). Avec mémoïsation, ce nombre d'appels est réduit à 145.

III. Implémentation d'un dictionnaire en mémoire vive. Fonction de hâchage.

Nous allons étudier dans cette section quelques principes généraux d'implémentation des dictionnaires dans la mémoire vive de l'ordinateur et les conséquences que cela a sur les temps d'accès à une valeur du dictionnaire.

1) Mémoire vive

La mémoire vive d'un ordinateur (RAM) peut être représentée comme un empilement vertical de *cases* (ou *cellules mémoire*) pouvant contenir des valeurs. Pour simplifier les choses, nous allons adopter un modèle de mémoire vive dans lequel chaque cellule peut contenir n'importe quel type de valeur : entier, flottant, chaine de caractère. [2]

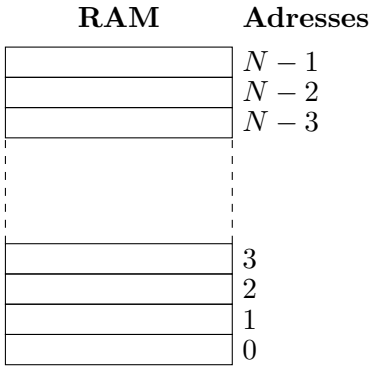

FIGURE 3 – Représentation de la mémoire vive (RAM) par un empilement de cellules mémoire numérotées de bas en haut. Le numéro d'une cellule est son **adresse**.

La mémoire vive est constituée de N cellules (dans les ordinateurs actuels N est de l'ordre de 10^9-10^{10}), numérotées par ordre croissant, en commençant par celle tout en bas qui reçoit le numéro 0, jusqu'à la dernière tout en haut de la pile qui reçoit le numéro $N-1$.

Ce numéro caractérisant chaque cellule de la mémoire s'appelle **adresse**. Le microprocesseur de l'ordinateur est capable de lire ou d'écrire une valeur dans une case mémoire donnée grâce à la connaissance de son adresse (Figure 1).

2) Tableau et liste chaînée

Définition 1

Soit $m \in \mathbb{N}^*$. Un **tableau** de taille m est une zone de la RAM constituée de m cellules consécutives (c'est à dire empilées verticalement les unes à la suite des autres) et ne contenant que des *valeurs de même type*. Chaque cellule du tableau s'appelle une **alvéole**.

On parlera ainsi de tableaux d'entiers, de tableaux de flottants, etc...

Chaque alvéole d'un tableau est caractérisée par un **indice entier** qui commence à 0 pour l'alvéole dont l'adresse est la plus petite et finit à $m-1$ pour l'alvéole d'adresse la plus grande (Figure 2).

2. En réalité, chaque cellule possède une taille fixe de 1 octet (8 bits) et toutes les valeurs codées sur plus d'un octet doivent occuper plusieurs cellules consécutives.

L'adresse d'un tableau est celle de sa première alvéole, c'est à dire celle dont l'indice est 0 (c'est donc le nombre entier a sur la Figure 2).

Une représentation schématique très commode pour un tableau de taille m est donnée sur la Figure 3.

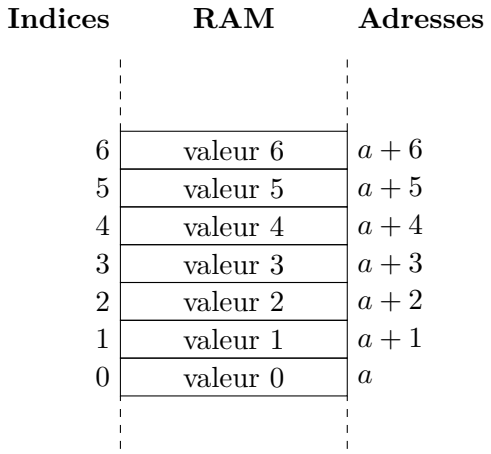

FIGURE 4 – Tableau constitué de 7 alvéoles. L'alvéole la plus basse (adresse a) possède l'indice 0 et l'alvéole la plus haute (adresse $a+6$) possède l'indice 6 et, plus généralement, l'indice $m-1$ dans le cas d'un tableau de taille m.

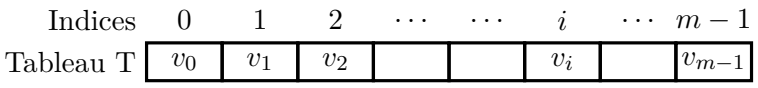

FIGURE 5 – Représentation schématique usuelle d'un tableau. Les alvéoles sont représentées horizontalement et chaque alvéole est repérée par son indice.

Ainsi, si on appelle T un tableau alors $T[i]$ désignera la valeur contenue dans l'avéole d'indice i, $0 \leqslant i \leqslant m-1$.

De l'implémentation en mémoire vive il résulte que la complexité temporelle d'une opération de lecture ou d'écriture d'une valeur dans unes des alvéoles d'un tableau est $O(1)$, c'est à dire qu'elle se fait à temps constant, quelle que soit l'indice de l'alvéole et donc quelle que soit la taille du tableau. En effet, pour ce faire le microprocesseur doit :

- charger dans ses registres l'adresse du tableau (entier a) ;
- lui ajouter l'indice i de l'alvéole visée, afin d'obtenir son adresse ;
- lire ou écrire la valeur contenue dans cette alvéole d'adresse $a+i$.

Ces trois opérations prennent un temps totalement indépendant de la taille du tableau.

Définition 2

Une **liste chaînée** est une *collection de paires de cellules mémoire*, chaque paire étant formée de deux *cellules mémoire consécutives* :

- la première cellule contient une valeur
- la seconde cellule contient l'adresse de la prochaine paire de cellules (Figure 4). On dit que la seconde cellule contient un **pointeur** sur la prochaine paire

Chaque paire de cellules mémoire s'appelle un **élément** de la liste chaînée.

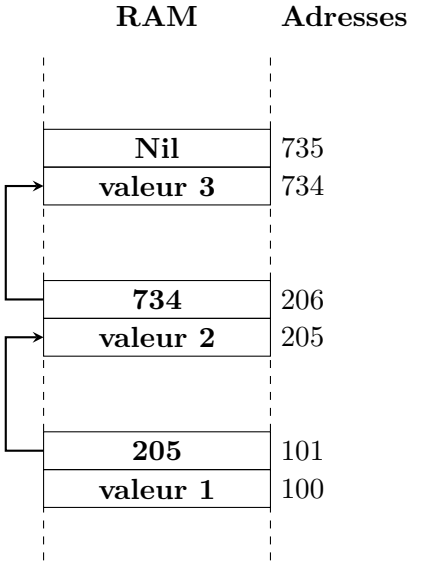

FIGURE 6 – Liste chaînée. Chaque paire de cellules mémoire est un élément de la liste chaînée ; chaque élément contient une valeur et l'adresse de l'élément suivant.

Pour indiquer la dernière paire de la liste chaînée, on met une *adresse impossible* pour la paire suivante, par exemple une adresse négative qu'on désigne par **Nil**. On pourra par exemple définir :

```
Nil = -1
```

L'adresse d'une liste chaînée est celle de son premier élément (c'est à dire l'adresse dela première cellule du premier élément, 100 sur l'exemple de la Figure 4).

La représentation schématique usuelle d'une liste chaînée est donnée sur la Figure 5.

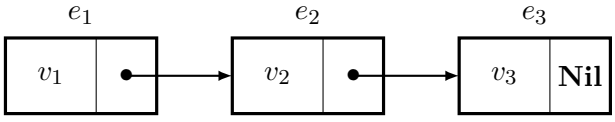

FIGURE 7 – Représentation usuelle d'une liste chaînée. Ici il y a 3 éléments, notés e_1, e_2 et e_3 et dont les valeurs respectives sont v_1, v_2 et v_3.

Quelle est la complexité temporelle d'une opération de lecture/écriture d'une valeur dans une liste chaînée ?

Écriture :

- Si la liste chaînée n'existe pas encore, le microprocesseur doit :
  - Réserver une zone mémoire de 2 cellules ;
  - écrire la valeur dans la première ;
  - écrire Nil dans la seconde.

  La complexité est alors $O(1)$.
- Si elle existe déjà et qu'elle contient n éléments, il doit :
  - charger l'adresse de la liste chaînée dans ses registres ;
  - parcourir toute la liste de proche en proche pour arriver au dernier élément ;
  - réserver un espace mémoire de 2 cellules ;
  - inscrire l'adresse de cet espace à la place du Nil ;
  - écrire la valeur ;
  - écrire Nil dans la deuxième cellule.

  La complexité est alors en $O(n)$

Lecture :

Dans ce cas le microprocesseur doit :

- charger l'adresse de la liste dans ses registres ;
- parcourir la liste de proche en proche jusqu'à arriver à l'élément ciblé ;
- lire la valeur de l'élément.

Dans le **meilleur des cas** la complexité est en $O(1)$ (l'élément ciblé est le premier élement de la liste). Dans le **pire des cas** elle est $O(n)$ (l'élement ciblé est le dernier d'une liste de n éléments).

3) Fonction de hachage

Un dictionnaire étant un ensemble de couples (clé,valeur), appelons U *l'ensemble de toutes les clés possibles* (univers des clés, qui peut être très gros). Rappelons qu'une clé peut être un entier, un flottant, une chaîne de caractères, un tuple. Dans la suite une clé sera notée c.

La création d'un dictionnaire entraîne celle d'un *tableau d'entiers* dans la RAM, de taille $m \in \mathbb{N}^*$, appelé **table d'adressage**, notée TA.

De plus, une application $h : U \longmapsto \{0, 1, ..., m-1\}$, $c \longmapsto h(c)$ est choisie pour associer à chaque clé possible l'une des alvéoles de la table d'adressage *via son indice*. h est appelée **fonction de hachage**.

Notons $K \subset U$ l'ensemble des clés *effectivement présentes* dans un dictionnaire donné. C'est une partie de U telle que $K = \varnothing$ lorsque le dictionnaire est vide. Dans ce cas, toutes les alvéoles de la TA contiennent la même valeur **Nil**, valeur qui indique que l'alvéole est disponible.

Lorsqu'un couple (c, v) est inséré dans le dictionnaire, l'indice $h(c)$ dans la TA est calculé et, si l'alvéole est disponible, on crée un premier élément d'une liste chaînée dont on met *l'adresse dans l'alvéole*. La valeur v est alors inscrite dans cet élément de liste chaînée.

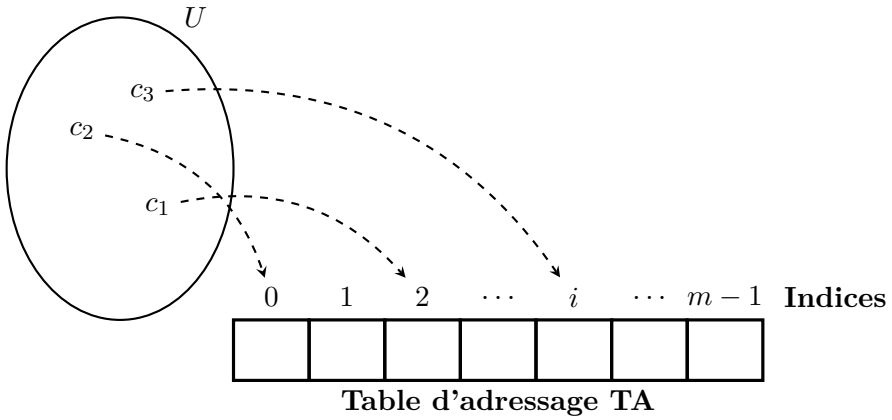

FIGURE 8 – Table d'adressage (tableau d'entiers) et fonction de hachage h qui fait correspondre à chaque clé possible un des indices de cette table et donc une alvéole de celle-ci.

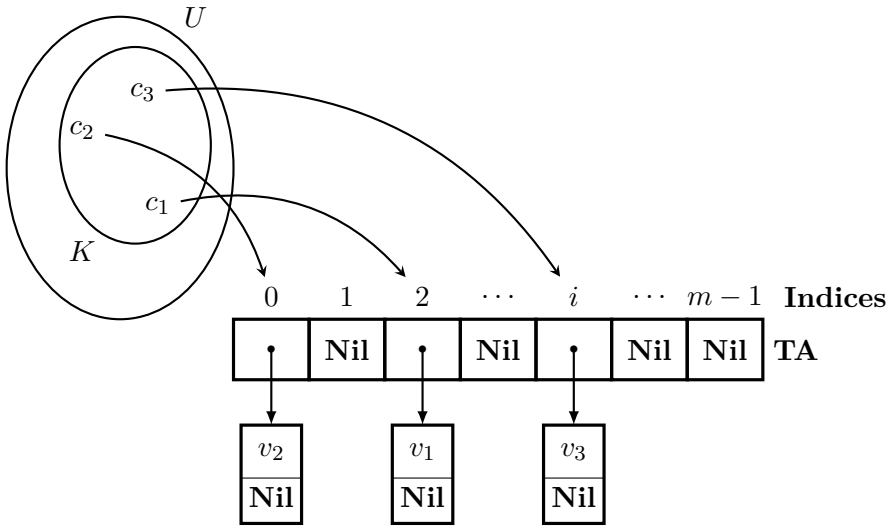

FIGURE 9 – Insertion de 3 premières valeurs, dans 3 alvéoles différentes. $d = \{(c_1, v_1), (c_2, v_2), (c_3, v_3)\}$. $K = \{c_1, c_2, c_3\}$.

La Figure 7 représente le résultat de 3 insertions (c_1, v_1), (c_1, v_1) et (c_1, v_1) dans un dictionnaire initialement vide, en supposant que $h(c_1) \neq h(c_2)$, $h(c_2) \neq h(c_3)$ et $h(c_1) \neq h(c_3)$.

L'idéal serait que $m = \text{Card}(U)$ et que h soit une *bijection*. Dans ce cas, à chaque clé possible on pourrait associer une et une seule alvéole dans la TA. Le problème est que :

- La taille de la TA serait vraiment trop grosse.
- La plupart des alvéoles contiendraient **Nil**.

Cela conduit donc à un gâchis d'espace mémoire.

De plus, ceci est tout simplement impossible : supposons par exemple que les clés possibles soient des chaînes de caractères d'au plus 10 caractères à choisir dans l'alphabet des 26 lettres minuscules, 26 lettres majuscules, plus 5 caractères accentués "é", "è", "à", "ù" et "ç" ; cela fait 57 caractères.

En pratique on a $m \ll \text{Card}(U)$, ce qui fait que la fonction de hachage ne sera *pas injective*. Autrement dit :

$$\boxed{\exists\,(c,c') \in U^2,\ h(c) = h(c') \qquad (1)}$$

Dans ce cas, si deux clés c et c' vérifiant (1) sont choisies dans le dictionnaire, elles seront dirigées vers la même alvéole de la TA, ce qui va provoquer une **collision**.

Cela n'est pas bien grave car si une collision se produit, on inscrit les deux valeurs v et v' des couples (c, v) et (c', v') dans une liste chaînée partant de l'alvéole d'indice $h(c) = h(c')$ (Figure 8).

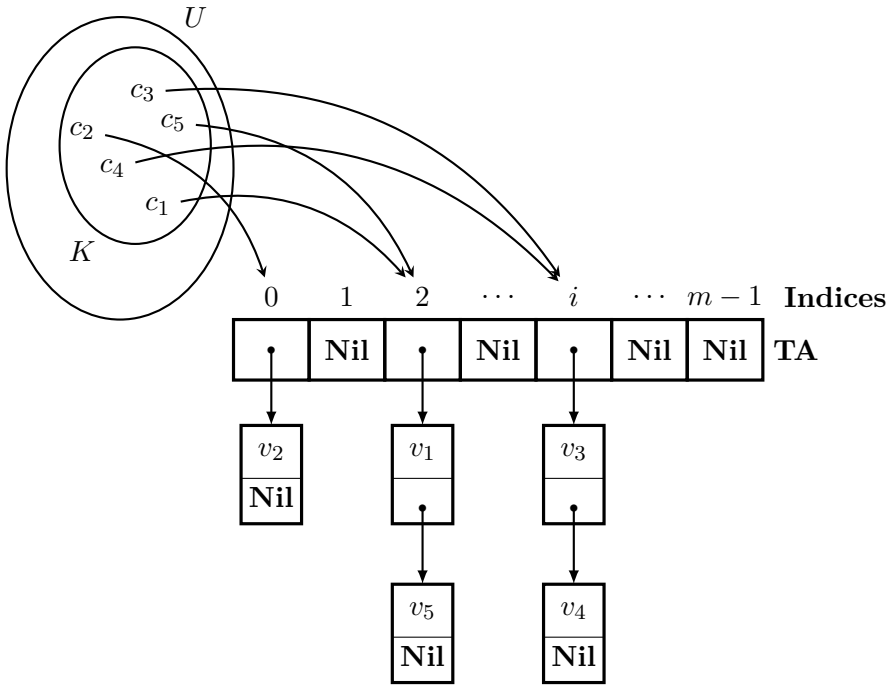

FIGURE 10 – Table d'adressage avec 2 collisions sur les indices 2 et i. $K = \{c_1, c_2, c_3, c_4, c_5\}$.

Quelles sont les propriétés que doit satisfaire une bonne fonction de hachage ?

- Elle doit provoquer le moins de collisions possibles. Idéalement si $\text{Card}(K) \ll m$ il ne doit pas y avoir de collision.
- Les collisions devenant inévitables lorsque $\text{Card}(K)$ se rapproche de m, il faut que celles-ci soient distribuées le plus uniformément possible sur tous les indices de la TA. Cela évite d'avoir une alvéole contenant un liste chaînée très longue alors que d'autres alvéoles sont peu remplies, voire sont vides.

Toute la stratégie consiste donc à choisir h et m de sorte que le nombre d'éléments des listes chaînées ne soit pas trop important. Si on fixe à $n_{\max}$ le **nombre maximal** d'éléments d'une liste chaînée, alors, dans le pire des cas, la complexité temporelle d'un accès en lecture/écriture d'un dictionnaire sera $O(n_{\max})$ et dans le meilleur des cas, elle sera toujours $O(1)$.

Dans le cas où la taille d'un dictionnaire grossit tellement que certaines listes chaînées deviennent de taille supérieure au $n_{\max}$ fixé, il peut être judicieux de créer une nouvelle TA de taille m supérieure à la première, de changer de fonction de hachage h et de recopier toutes les valeurs du dictionnaire dans la nouvelle TA.

En conclusion :

Une fonction de hachage partitionne l'univers des clés en sous-ensembles. Toutes les clés d'un sous-ensemble donné seront dirigées vers la même alvéole d'une table d'adressage et les valeurs associées à ces clés seront placées dans une liste chaînée partant de l'alvéole. On dit que les clés sont hachées dans les alvéoles.

Si C est le nombre de couples dans le dictionnaire et m est la taille de la TA, alors la ***longueur moyenne attendue*** *des listes chaînées est C/m et une bonne fonction de hachage tendra à produire m listes chaînées équitablement réparties dans toutes les alvéoles, chaque liste étant de longueur voisine de C/m.*

Exemple :

Un dictionnaire contient 2000 clés. Si on les hache dans 700 alvéoles, la longueur moyenne attendue d'une liste chaînée sera de $2000/700 \approx 3$.

4) TP étude de fonctions de hachage

Nous allons étudier en TP deux exemples particuliers de fonctions de hachage et en tester les caractéristiques.