Chapitre 1 : Piles et Files.

MP Lycée Camille Jullian

septembre 2021

L'an dernier, vous avez étudié en informatique les principes de base de l'algorithmique, ainsi que quelques types de données usuels en Python (notamment les listes). Le but du programme de cette année est d'approfondir un peu ces notions, en donnant quelques outils de programmation supplémentaires (programmation récursive, un peu de programmation orientée objets), et en évoquant quelques « nouveaux » types de données comme les piles (qui sont en fait une nouvelle façon d'exploiter les listes que vous connaissez déjà). Mais on mettra aussi (et même surtout) l'accent sur un point qui avait été laissé de côté dans un premier temps (tout à fait logiquement d'ailleurs) : la « qualité » des programmes que nous écrivons. Qu'est-ce qui fait qu'un programme est meilleur qu'un autre ?

1 Analyse d'algorithmes et complexité.

1.1 Définition de la complexité d'un algorithme.

Analyser un programme revient à calculer (approximativement) les ressources nécessaires à l'exécution de ce programme. Ces valeurs dépendent bien entendu de contingences matérielles impossibles à maîtriser, nous nous contenterons donc d'estimer un ordre de grandeur de ces valeurs dépendant uniquement de la taille des données nécessaires pour faire tourner le programme.

Définition 1. La **complexité** d'un programme informatique est une mesure de son efficacité. Plus précisément :

- la complexité **spatiale** mesure la quantité de mémoire nécessaire à l'exécution de l'algorithme.
- la complexité **temporelle** mesure le temps d'exécution de l'algorithme, c'est-à-dire une estimation de ce temps d'exécution en fonction de la taille des données manipulées.

Nous allons surtout nous intéresser à la complexité temporelle, la plus importante pour nous, et surtout la plus facile à mesurer. Pour cela, il est indispensable de simplifier le mesure du temps d'exécution, en le ramenant à un décompte du nombre d'opérations élémentaires effectuées par l'ordinateur lors de l'exécution du programme. Il faut bien avoir conscience que, lors d'une telle exécution, on effectue énormément de « petites » manipulations (calculs élémentaires, affectations de variables, etc). Il est essentiel que les manipulations que nous allons considérer comme « élémentaires » le soient réellement, c'est-à-dire qu'elles s'effectuent en un temps fixe indépendant des données manipulées. Ainsi, une instruction Python comme *sort* n'est sûrement pas satisfaisante de ce point de vue (le tri d'une liste prend d'autant plus de temps que la liste est longue), mais une addition ou une comparaison de flottants (codés sur un nombre fixe de bits). Une addition de variables entières poserait déjà nettement plus de problèmes...

Exemple : Pour donner un premier exemple concret qui sera un fil rouge de tout le cours d'informatique cette année, prenons le cas des différents algorithmes de tris d'une liste. Il en existe plein (celui utilisé pour le *sort* de Python est l'un des plus efficaces d'entre eux), la plupart étant basés sur des séries de comparaisons entre éléments de la liste. En considérant les autres manipulations (affectations et autres) comme négligeables en termes de temps d'exécution par rapport aux comparaisons, on mesurera la complexité d'un algorithme de tri en comptant simplement le nombre de

comparaisons nécessaires pour terminer le tri (quantité qui dépendra du nombre d'éléments total de la liste).

1.2 Quelques notations mathématiques.

Si on reprend l'exemple précédent, compter **exactement** le nombre de comparaisons effectuées peut s'avérer fastidieux et finalement assez peu intéressant. Ce qui nous intéressera nettement plus sera d'avoir un **ordre de grandeur** de ce nombre d'opérations, par exemple « proportionnel à n » ou « proportionnel à n^2 » (on va voir juste après un vocabulaire associé à ce type d'ordres de grandeur). Pour matérialiser cela, on utilisera principalement la notation suivante :

Définition 2. Une suite u_n est majorée en ordre de grandeur par une autre suite v_n si $\exists A \in \mathbb{R}$, $\forall n \in \mathbb{N}$, $u_n \leqslant A \times v_n$. On le notera $u_n = O(v_n)$.

Ainsi, dire que la complexité d'un algorithme de tri est « en $O(n^2)$ » signifie que son temps d'exécution est majoré par un temps proportionnel à n^2. Pourquoi seulement majoré et pas égal ? D'une part parce que c'est plus facile à démontrer, mais surtout parce que parfois, un algorithme peut avoir un temps d'exécution notablement plus faible dans certaines configurations particulières. Ainsi, un algorithme de tri « en $O(n^2)$ » peut nécessiter un nombre de comparaisons seulement proportionnel à n si on l'applique dans le cas particulier où la liste est déjà triée. On parlera d'ailleurs également de « complexité dans le meilleur cas » ou de « complexité dans le pire cas » pour évoquer ces différences. Bien entendu, c'est la complexité dans le pire cas qui sera conservée pour évaluer l'efficacité de l'algorithme.

Définition 3. Principaux types de complexité. Un algorithme est dit :

- **linéaire** si le nombre d'opérations est en $O(n)$ (donc un temps d'exécution sensiblement proportionnel à la taille des données).
- **quadratique** si le nombre d'opérations est en $O(n^2)$.
- **polynômial** si le nombre d'opérations est en $O(n^k)$ pour un certain entier $k \geqslant 1$.
- **logarithmique** si le nombre d'opérations est en $O(\log n)$.
- **semi-linéaire** si le nombre d'opérations est en $O(n \log n)$.
- **exponentiel** si le nombre d'opérations est en $O(a^n)$ pour un certain réel $a > 1$.

Pour donner un ordre d'idée des temps d'exécutions correspondants, imaginons qu'un algorithme tournant sur une certaine machine mette $1\mu s$ (soit $10^{-6}s$) à trier une liste contenant 100 éléments. On aura les temps d'exécutions suivants pour un tri d'une liste de 1 000 éléments ou de 10^6 éléments selon le type de complexité :

- s'il est logarithmique, $1.5\mu s$ pour trier 1 000 éléments, et $3\mu s$ pour un million d'éléments.
- s'il est linéaire, $10\mu s$ pour 1 000 éléments, et $0.01s$ pour un million.
- s'il est semi-linéaire, $15\mu s$ pour 1 000 éléments, et $0.03s$ pour un million.
- s'il est quadratique, $100\mu s$ pour 1 000 éléments, et $100s$ (soit un peu moins de 2 minutes) pour un million.
- s'il est cubique, $1ms$ pour 1 000 éléments, et $10^6 s$ (soit une bonne dizaine de jours) pour un million.
- s'il est exponentiel de base $a = 1.1$, $1.8 \times 10^{31} s$ (soit un peu plus de 5×10^5 milliards de millards d'années) pour 1 000 éléments (on n'évoquera pas ici le cas d'un million d'éléments...).

1.3 Quelques structures de données en Python.

Vous avez commencé à manipuler l'an dernier quelques types de données classiques en Python, dont le plus complexe (et le plus important) était celui de listes. Ce dernier a un mode de fonctionnement assez particulier puisque les listes sont manipulées à l'aide de « fonctions » particulières appelées méthodes, dont la syntaxe typique est **objet.methode()**. Cela a un lien avec le fait que

les listes représentent une des incursions du langage Python dans le domaine de ce qu'on appelle en informatique la programmation orientée objet, et plus généralement avec la façon dont sont organisés les différents types de données en Python, et en particulier les listes.

De façon générale, lorsqu'on définit une structure de données dans un langage informatique (comme les listes en Python), il est indispensable de gérer la façon dont on va accéder aux données contenues dans la structure, mais aussi la façon dont le stockage des données de cette structure est organisé en mémoire, ce qui a une incidence sur la complexité des commandes de base permettant d'effectuer des opérations élémentaires sur la structure. Nous allons évoquer plus précisément ici les structures de données **linéaires**, c'est-à-dire celles qu'on peut visualiser comme des suites finies d'objets. Les listes en font parties, ainsi que les piles et files que nous allons ensuite étudier.

Parmi celles que vous avez déjà croisées :

- les **tableaux** sont constitués d'une suite de variables de même type stockés sur des emplacements consécutifs dans la mémoire de l'ordinateur. C'est le cas du type **array** de la bibliothèque numpy en Python. Il s'agit d'une structure **statique** : la quantité de mémoire allouée au stockage d'un tableau est fixée au moment de sa création (on réserve $n \times k$ unités d'espace mémoire, n étant le nombre d'éléments du tableau et k la place nécessaire pour stocker une donnée du type représenté dans le tableau). Pour accéder au contenu d'une des données du tableau, il suffit de connaitre l'emplacement d du début du tableau dans la mémoire et de se déplacer jusqu'à l'amplacement $d + kp$, où p est la position de l'élément dans le tableau. Cette opération s'effectue en **coût constant** (la complexité de l'algorithme correspondant est constante et ne dépend pas de la taille du tableau).
- les **listes** sont nettement plus complexes car il s'agit d'une structure de données **dynamique** (la taille mémoire est modifiable en permanence, c'est une des raisons pour lesquelles on peut sans problème créer en Python des listes contenant des données de types incohérents). En pratique, chaque élément de la liste est stocké en mémoire en deux parties : la donnée elle-même, et un pointeur vers l'emplacement mémoire de la donnée suivante (ou vers une valeur particulière indiquant la fin de la liste). L'accès à un élément de la liste demande donc a priori un temps d'autant plus grand qu'il se sirue loin dans la liste (il faut passer par tous les éléments précédents), on a un algorithme d'accès qui est linéaire. Par ailleurs, les listes sont des structures de données pour lesquelles on a besoin de méthodes élémentaires autres que celle permettant simplement l'accès ou l'écriture dans une « case » de la liste. Il faut une méthode d'élimination d'un élément de la liste (**del l[i]** en Python), d'ajout d'un élément en fin de liste (**l.append**), de suppression de la première occurrence d'une valeur (**l.remove(x)**), voire d'insertion d'un élément en milieu de liste (**l.insert(i,x)**), et de suppression et renvoi de la valeur d'un élément en milieu de liste (**l.pop(i)**). Sans rentrer dans le détail de la gestion par Python de la structure de liste (qui est plus sophistiquée que celle présentée ci-dessus, une liste Python étant en gros un tableau de pointeurs dont la taille est « dilatable » en cas de besoin), on admettra que les opérations de modification d'un élément de liste ou d'ajout via append se font en coût constant en Python, alors qu'un insert, un del ou un remove sont en coût linéaire.

2 Piles et Files.

2.1 La structure de Pile.

Une **pile de données** est une structure fonctionnant sur le principe **LIFO** : Last In, First Out. Autrement dit, il s'agit d'une structure linéaire dans laquelle seulement deux opérations sont imposées : l'**empilage** (**push** en anglais) consistant à ajouter un élément à la Pile, et le **dépilage** (**pop** en anglais) consistant à supprimer un élément de la Pile, sachant que cex deux opérations doivent respecter le fameux principe LIFO : le premier élément qui sera « sorti » de la Pile sera celui qu'on y aura inséré en dernier. On peut visualiser une telle structure comme une pile (d'où le nom)

d'objets entassés verticales (assiettes par exemple). On ne peut ajouter un nouvel objet qu'en haut de la pile, et enlever un objet de la pile que s'il est situé sur le dessus (et donc s'il est le dernier objet empilé). En général, quand on créée une structure de Pile dans un langage informatique donnée, on demande également à avoir une fonctionnalité supplémentaire permettant de vérifier si la Pile est vide ou non.

2.2 Implémentation d'une Pile en Python utilisant un tableau numpy.

Les piles ne forment pas à proprement parler un **type** de données en Python, on peut toutefois créer des structures de Piles en Python en utilisant des structures de données déjà existantes. Par exemple, à l'aide d'un tableau numpy, on peut créer une Pile de taille maximale n, pour un entier n fixé à l'avance (c'est l'inconvénient d'utiliser une structure de données statique). Si on essaye d'empiler un nouvel élément alors que la Pile est pleine, il faudra gérer l'erreur qui va nécessairement se produire (on dit que la Pile **déborde** dans ce cas). En pratique, on va créer un tableau de taille $n+1$ dont la première case servira à stocker l'emplacement du dernier élément de la Pile, et les autres contiendront les données. Ainsi, on peut écrire les fonctions suivantes :

```
def creaPile(n) :
    return np.zeroes(n+1)

def vide(P) :
    return P[0]==0

def empile(P,x) :
    i=P[0]
    if i!=len(P) :
        P[i]=x
        P[0]+=1
        return P
    raise IndexError

def depile(P) :
    i=P[0]
    if i==0 :
        raise IndexError
    a=P[i]
    P[i]=0
    P[0]-=1
    return a
```

2.3 Implémentation d'une Pile en Python utilisant une liste.

Il est en fait beaucoup plus pratique d'utiliser des listes Python pour créer des structures de type Pile, puisqu'on dispose déjà de toutes les fonctionnalités nécessaires : la création de pile se fait simplement en retournant une liste vide, l'empilement correspond à un append, et le dépilement à un pop.

2.4 Création d'une classe Pile en Python.

On peut également créer nous-même en Python un nouveau type de données (s'appuyant sur un ou des types déjà existants) en exploitant la structure de classe (c'est la partie de Python qui fonctionne en mode « orienté objet », si vous apprenez un jour à programmer en Java, vous ferez ce genre de choses en permanence). La syntaxe de la définition de classe en Python ressemble à ceci :

```
class(Pile) :                    (on définit et on nomme une nouvelle classe)
    def __init__(self) :
        self.donnees=[]          (la fonction __init__ sera automatiquement appelée
```
pour créer une liste via la commande P=Pile(), on va en fait créer une liste théoriquement accessible via l'appel P.donnees mais qu'il est en pratique déconseillé d'utiliser ainsi, on doit se serivir des fonctionnalités que nous allons maintenant définir)

```
    def vide(self) :
        return self.donnes==[]

    def empile(self,x) :
        self.donnees.append(x)

    def depile(self) :
        if self.vide() :
            raise IndexError
        return self.donnees.pop()
```

Il est traditionnel de nommer **self** l'objet manipulé dans une définition de classe mais on peut bien sûr le remplacer par à peu près n'importe quoi d'autre.

2.5 Exercices et exemples.

Exercice : écrire un programme Python qui vérifie si une expression arithmétique (donnée sous forme de chaîne de caractères) est bien parenthésée (autre ment dit s'il y a autant de parenthèses ouvrantes que de parenthèses fermantes dans la chaine de caractères, et que ces parenthèses sont correctement placée (par exemple, si la première parenthèse croisée est une parenthèse fermante, il y a un problème !). On supposera pour simplifier que seules les parenthèses sont utilisées pour gérer les priorités d'opérations (pas de crochets par exemple).

L'emploi d'une Pile est ici particulièrement adaptée : on empile à chaque parenthèse ouvrante croisée, et on dépile à chaque parenthèse fermante (sauf bien sûr si la Pile est vide, auquel cas on se contente de tout arrêter et de renvoyer False). si on voulait gérer deux types de caractères (parenthèses et crochets par exemple), on ferait pareil avec deux Piles au lieu d'une seule.

```
def verifiepar(s) :
    P=Pile()
    for i in s :
        if i=='(' :
            P.empiler(i)
        elif i==')' :
            if P.vide() :
                return False
            else :
                P.depiler()
    return True
```

Exemple : Les Piles sont en pratique utilisées par les machines (pas spécialement par Python dans l'exemple qui va suivre, mais par n'importe quelle calculatrice ou autre ordinateur effectuant du calcul mathématique) pour gérer les calculs arithmétiques. Une expression arithmétique est une chaine de caractères constituée d'une suite de symbole appartenant aux catégories suivantes : nombres (on supposera pour simplifier que les nombres se réduisent ici à un seul chiffre), opérateurs (pour simplifier à nouveau, on se contentera des opérateurs binaires $+$, $-$, $\times$ et $/$) et séparateurs (parenthèses). On sait bien que le parenthésage de telles expressions est indispensable pour savoir dans quel ordre

effectuer les opérations en respectant les règles usuelles de priorité. Mais il existe en fait plusieurs façons d'écrire ces expressions :

- la **forme infixe**, qui consiste à placer les deux opérandes (valeurs qu'on combine à l'aide d'une opération) de part et d'autre de l'opération (par exemple $2+3$ pour indiquer la somme de 2 et de 3. C'est bien entendu la forme utilisée en pratique par des calculateurs humains !
- la **forme préfixe** consiste au contraire à placer les deux opérandes après le symbole d'opération. Avec cette méthode, le calcul de $(2+3)*5-4$ s'écrirait $-*+2354$. Aucun besoin de parenthèses sous cette forme, il ne peut pas y avoir d'ambiguité ! Cette notation a d'ailleurs été utilisée par certaines calculatrices il y a maintenant quelques décennies (elle est aussi appelée notation polonaise inverse).
- la **forme postfixe** où on place au contraire les opérandes avant l'opération. La même expression $(2+3)*5-4$ s'écrirait ainsi $23+5*4-$. Là aussi, il n'y a plus aucune ambiguité possible.

En pratique, comment un ordinateur effectue-t-il le calcul à partir d'une expression arithmétique du type précédent ? Eh bien, c'est très simple : il utilise une Pile ! Imaginons que l'expression soit donnée sous forme postfixe : on parcourt alors la chaîne de caractères, et à chaque fois qu'on croise une valeur numérique, on l'empile, à chaque fois qu'on croise un opérateur, on dépile deux fois, on effectue l'opération et on rempile la valeur obtenue. Bien sûr, en pratique, il faudrait également gérer les nombres à plusieurs chiffres et décimaux, les opérateurs unaires (qui ne s'appliquent qu'à une seule valeur, comme le signe $-$ des nombres négatifs ou la racine carrée par exemple) et détecter les erreurs de syntaxe. Mais l'idée générale est bien celle-là.

Un autre domaine dans lequel des Piles sont naturellement utilisées par les ordinateurs est celui des fonctions récursives, rendez-vous au prochain chapitre pour aborder ce sujet !