Informatique de tronc commun
première année

Julien Reichert

2021/2022

Ces documents de cours sont organisés en trois parties : le cours du deuxième semestre, les TP du premier semestre et les TP du deuxième semestre.

Le premier chapitre (numéroté 0) du cours du deuxième semestre reprend le chapitre consacré à l'enseignement de Python prévu par la réforme de 2013. Il est maintenu en guise de rappel mais ne fait pas l'objet d'un enseignement, l'essentiel de son contenu étant supposé connu depuis la fin du lycée.

La première section du chapitre 2 est également maintenue depuis l'ancien programme car elle permet de ne pas introduire brutalement le binaire sans passer par les bases numériques en général.

La dernière section du chapitre 3 contient trois algorithmes de recherche de plus court chemin dans un graphe pondéré, dont l'algorithme de Dijkstra, seul mentionné au programme. Les deux autres sont présentés dans les notes de cours mais pas en classe et ne font pas l'objet d'une évaluation.

Mis à part ces passages, du texte orange dans ces notes de cours marque les informations mises pour la culture, mais absentes du programme et non évaluées.

Attention : Si certains codes fournis en TP ne semblent pas marcher, il faut vérifier la version de Python utilisée, une version ultérieure à 3.5 est recommandée, en particulier il faut totalement laisser de côté la version 2, déclarée obsolète avant 2020.

Table des matières

Première partie

Cours

Chapitre 0

Le langage Python

0.1 Bases de la programmation

0.1.1 Les données et leurs types

La notion de *type* est essentielle en programmation : elle indique quel genre de données on manipule. On peut voir un type comme une catégorie d'objets mathématiques sur lesquels des opérations spécifiques peuvent être effectuées. Un exemple issu de la géométrie : un point dans le plan peut être vu comme la donnée de deux réels qui sont ses coordonnées cartésiennes, une droite est la donnée de deux points par lesquels elle passe ; ce qui est possible avec des points, comme déterminer le milieu du segment entre les points, calculer une distance entre autres, ne l'est pas forcément avec des droites, par exemple déterminer une intersection ou calculer un angle.

Types de base

Les types simples de base en informatique sont les entiers, les flottants, permettant d'approcher des nombres réels et que peu de langages rassemblent avec les entiers dans un type « nombre », les *booléens*, ensemble formé des constantes appelées vrai et faux, qui permettent de faire du calcul et de la logique, même si de nombreux langages se permettent de faire de la logique avec des entiers, et les *caractères*. Les séquences de base sont les *n-uplets*, les *listes* et les *chaînes de caractères*.

Le typage de Python est *dynamique*, c'est-à-dire que le type de chaque expression est déterminé lors de l'exécution d'un programme, et non pas lors de l'analyse du code.

Entiers

Commençons par les opérations sur les entiers en Python. L'addition se fait avec `+`, la soustraction avec `-` (qui est aussi utilisé pour obtenir l'opposé, contrairement à la calculatrice qui dispose de deux touches différentes), la multiplication avec `*` **qui ne peut pas être implicité**, le quotient de la division euclidienne avec `//`, le reste de la division euclidienne avec `%` et la puissance avec `**`.

Attention : ne pas utiliser pour la puissance l'opérateur `^`.

Flottants

Pour écrire un flottant, on utilise des chiffres et un point au lieu de la virgule en guise de séparateur décimal, puisque Python applique bien entendu la convention anglo-saxonne. On peut omettre la partie à gauche ou à droite de la virgule si elle vaut zéro, mais zéro ne peut pas s'écrire simplement `.`, ce sera `0.` ou `.0`. Les opérations sur les flottants sont sensiblement les mêmes, on retrouve `+`, `-`, `*` et `**`, et comme la notion de division euclidienne n'a pas d'intérêt (bien que les opérateurs `//` et `%` fonctionnent, avec un comportement intuitif), la division se fait par `/` ; elle retourne un flottant même si elle concerne deux entiers.

Les conversions se font à l'aide des fonctions `int()` (troncature), `float()` (conversion d'un entier en lui-même virgule zéro), `math.floor()` (arrondi par défaut, donc la partie entière) et `math.ceil()` (arrondi par excès), les deux dernières nécessitant de charger le module `math` par l'instruction `import math`. L'arrondi (transformant un flottant en flottant) se fait par la fonction `round`, qui prend deux arguments : le nombre à arrondir et le nombre de chiffres après la virgule. Ce deuxième argument est optionnel et l'omettre donnera un arrondi à l'entier le plus proche, par ailleurs la fonction ne retournera pas un flottant mais un entier. Si le deuxième argument est négatif, on obtient un arrondi à la dizaine la plus proche, ou la centaine, et ainsi de suite.

Les priorités opératoires sont les mêmes que celles appliquées en mathématiques. Les parenthèses peuvent être utilisées et imbriquées, mais on ne peut pas les remplacer par des crochets.

Les flottants étant représentés de manière approchée, il est possible de trouver trois flottants `x`, `y` et `z` tels que le résultat de `x + (y + z)` soit différent de celui de `(x + y) + z`, mais il est toujours vrai que `x + y` est égal à `y + x`.

Booléens

Les booléens sont `True` et `False`. **Attention aux capitales**, qui sont nécessaires, Python se distingue en cela. On les utilise notamment pour des tests d'arrêts de boucles `while` et pour des instructions contenant `if`. Les opérations associées sont la *conjonction* `and`, la *disjonction* `or` le *test de différence* `^` (aussi appelé « ou exclusif » et raccourci "xor" en anglais) et la *négation* `not`.

Les opérateurs `&` et `|` sont les équivalents bit-à-bit pour des entiers de `and` et `or`, respectivement. Ils ne sont pas à connaître et peuvent aussi être utilisés pour les booléens. Leur utilisation est dangereuse, car d'une part ils ne sont pas paresseux, et d'autre part ils sont prioritaires sur les comparaisons.

La négation est prioritaire sur la conjonction qui est prioritaire sur la disjonction. Ainsi, `True or not True and False` se parenthèse automatiquement `True or ((not True) and False)` et donne `True`.

En outre, il est important de souligner que Python, comme l'essentiel des langages, *évalue paresseusement* les expressions booléennes, ce qui signifie que quand l'expression `a and b` est évaluée (avec `a` et `b` deux expressions), si `a` se révèle fausse, `b` n'est même pas évaluée ; donc `False and 1/0 == 42` ne provoque pas d'erreur de division par zéro. Il en va de même pour `a or b`, et c'est notamment problématique de l'oublier si l'expression `b` concerne une fonction avec effet de bord.

Les booléens peuvent s'obtenir comme résultat de comparaisons de deux entiers, ou deux flottants, voire deux objets de type quelconque, avec les opérateurs suivants : `<`, `>`, `<=`, `>=`, `==` (test d'égalité, à ne pas utiliser avec un seul égal) et `!=` (test de différence). On peut enchaîner les comparaisons, ce serait parfois une aberration en mathématiques, mais on le fera là aussi presque exclusivement pour des tests d'intervalles, le résultat étant vrai si toutes les comparaisons isolées sont vraies. Les comparaisons sont prioritaires sur les opérations booléennes.

Quelques résultats à connaître au cas où :
- `42 == 42.0` est vrai, de même pour tous les entiers.
- D'ailleurs, `True == 1` est vrai, de même que `False == 0`, `True == 1.0` et `False == 0.0`.
- Du coup, `True + True` vaut 2 et `False < True` est vrai. En fait, on dit que le type booléen de Python *hérite* du type entier.
- On peut (mais on ne doit pas) utiliser une chaîne de caractères dans les tests. Bien que `False == ""` soit faux, la chaîne vide est la seule qui s'assimile à `False` si on l'utilise dans un test. Quoi qu'il en soit, il faut éviter de confier à Python le soin de comprendre quelque chose qui n'est pas un booléen comme un booléen.

0.1.2 Variables

Les *variables* en programmation sont assimilables à leur pendant mathématique : « Soit x un réel... ». Ici, il faut donner une valeur, de n'importe quel type, à une variable dès sa création (c'est-à-dire sa première mention, qui est donc nécessairement une affectation) afin de l'utiliser.

Le nom d'une variable est composé de lettres, qu'on évitera d'accentuer, de chiffres, à condition que ce ne soit pas par un chiffre que le nom commence, et de caractères de soulignement.

L'*affectation* d'une variable se fait par le symbole `=`, avec nécessairement le nom de la variable à gauche.

Le principe d'une affectation est que la partie à droite du symbole `=` est évaluée, puis le résultat est stocké dans la variable à affecter.

Rien n'interdit que le type d'une variable change lors d'une réaffectation, qui consiste simplement à écraser l'ancienne valeur de la variable et ne plus en tenir compte.

Ainsi, une réaffectation peut impliquer l'ancienne valeur d'une variable. Voir le code ci-dessous, dont la partie après chaque symbole # est appelée *commentaire* et ignorée par Python :
- `a = 4`
- `a = a + 3 # a + 3 est évalué à 7, donc a vaut maintenant 7`
- `a *= 4 # raccourci pour a = a * 4`

Remarque : Pour aller plus loin, les instructions `a *= 4` et `a = a * 4` ne sont pas tout à fait équivalentes, mais la nuance n'est pas apparente et il n'est pas pénalisable de l'oublier. En fait, `a *= 4` n'est pas une réaffectation comme les autres, mais plutôt une mise à jour de l'ancienne valeur par une opération. En outre le raccourci `++` ou `-` pour ajouter ou retirer un n'existe pas en Python.

Un peu de théorie... Dans l'ordinateur, une variable est une zone mémoire accessible en lecture et en écriture. L'état du système est alors l'ensemble des variables définies à un instant donné de l'exécution d'un programme. Toute expression contenant une variable s'évalue en remplaçant les noms de variables par leur valeur dans l'état courant, déclenchant au passage une erreur lorsqu'une des variables n'est pas encore affectée, c'est-à-dire qu'elle n'apparaît pas dans l'état. Déclarer, c'est-à-dire créer et affecter, ou réaffecter une variable modifie donc l'état.

0.1.3 Séquences

Introduction

Les séquences sont des objets en Python qui possèdent une *taille*, sont *itérables*, c'est-à-dire qu'on peut les parcourir, *indexables*, c'est-à-dire qu'on peut accéder à chacun de leurs éléments au moyen d'un indice, et *tranchables*, tentative personnelle de traduction de "sliceable", sachant que la notion de *slice* sera vue ci-après).

Certains objets n'ont pas toutes ces propriétés, par exemple les ensembles ne sont pas indexables, et il existe des itérables infinis donc sans taille. Pour plus d'informations sur ce vocabulaire, voir `http://sametmax2.com/les-trucmuchables-en-python/`.

n-uplets

Un n-uplet ("tuple", en anglais) est une collection d'éléments séparés par des virgules (on utiliserait des points-virgules en français), éventuellement entourée de parenthèses.

Le seul 0-uplet existant est `()`, qu'on retrouve dans les fonctions à zéro argument. Un 1-uplet s'écrit de manière spéciale en ajoutant une virgule après l'élément unique. Son utilité est de pouvoir faire des opérations de n-uplet avec un seul élément.

La fonction `len()` donne la taille du n-uplet, et fonctionne en fait pour toutes les séquences.

On accède au `i`-ième élément du n-uplet `t` (très souvent stocké dans une variable, mais la syntaxe suivante marche dans tous les cas) par `t[i]`, où les indices commencent à 0, avec une erreur si `i` est supérieur ou égal à la taille de `t`.

On peut aussi accéder à un élément à partir de la fin : `t[-1]` est le dernier élément, et ainsi de suite jusqu'à `t[-len(t)]` qui est le premier élément (et là aussi, un indice strictement inférieur provoque une erreur).

Enfin, on peut accéder à une tranche (le fameux *slice*) d'un n-uplet en écrivant `t[i:j]`, ce qui donne un k-uplet formé de `t[i]`, `t[i+1]`, ..., `t[j-1]`, en remplaçant tout nombre négatif par lui-même plus la taille de `t`, à la manière décrite à la phrase précédente. Ce k-uplet est vide si `t[i]` est à droite de `t[j-1]` et tronqué aux extrémités de `t` sans provoquer d'erreur si i ou j débordent des indices autorisés. Il est même possible de préciser le pas dans la tranche, permettant notamment de lire un n-uplet à l'envers. La syntaxe devient `t[i:j:pas]`.

On ne peut accéder aux éléments d'un n-uplet qu'en lecture, il est juste possible de réaffecter un n-uplet dans son ensemble si on veut le modifier.

Les n-uplets peuvent être déconstruits : on écrira `(a, b, c) = (3, 2, 1)`, les parenthèses étant optionnelles des deux côtés s'il n'y a pas de risque de confusion, pour affecter d'un coup trois variables, en faisant tout de même attention à la lisibilité. Le fait que la partie à droite du signe `=` soit évalué avant que les affectations soient effectuées permet de trouver une méthode élégante pour échanger le contenu de deux variables.

Le test d'appartenance se fait à l'aide de l'opérateur infixe, c'est-à-dire qu'on l'insère entre ses deux arguments, `in` : `2 in (1, 4, 2, 5)` donne `True`. Cet opérateur caractérise en fait des objets dits *conteneurs* (en anglais "container").

Les n-uplets peuvent fusionner entre eux avec l'opérateur `+`. Ils peuvent être reproduits avec l'opérateur `*` utilisé avec un entier à gauche ou à droite.

Attention cependant aux parenthèses : `1, 2 + 3, 4` donne `(1, 5, 4)` tandis que `(1, 2) + (3, 4)` donne `(1, 2, 3, 4)` ; de même, `3 * 1, 2` donne `(3, 2)`, tandis que `3 * (1, 2)` donne `(1, 2, 1, 2, 1, 2)`.

Chaînes de caractères

Une chaîne de caractères se manipule de la même manière qu'un n-uplet. En particulier, il est impossible de modifier un caractère isolé d'une chaîne.

Une chaîne de caractères est entourée d'apostrophes ou de guillemets, simples ou triples, l'utilisation de délimiteurs triples permettant d'aller à la ligne en écrivant la chaîne.

En outre, si on veut utiliser une apostrophe dans une chaîne, il suffit d'utiliser un autre délimiteur que l'apostrophe pour ne pas avoir de souci, etc (ou échapper l'apostrophe à l'intérieur de la chaîne).

Le test d'appartenance avec `in` peut détecter des facteurs d'une chaîne, c'est-à-dire des chaînes formés de caractères consécutifs d'une chaîne, à la manière des parties de n-uplets. Comparer les résultats des tests `"23" in "123456"`, `"24" in "123456"`, `(2, 3) in (1, 2, 3, 4, 5, 6)` et `(2, 3) in (1, (2, 3), 4, (5, 6))`. Il est intéressant de constater au passage que certaines parenthèses sont importantes en retirant celles à gauche de `in`.

Listes

Remarque : Il faut éviter de dire « tableau », car en Python, contrairement à d'autres langages, on appelle « tableau » une structure, introduite dans la bibliothèque `numpy`, ressemblant à une liste mais contenant préférablement uniquement des entiers ou uniquement des flottants.

Une liste est une collection d'éléments, similaire à un n-uplet, à la différence notable près que cette fois-ci les éléments peuvent être modifiés individuellement.

On crée une liste en donnant ses éléments séparés par des virgules et en entourant le tout de crochets au lieu des parenthèses du n-uplet.

Contrairement à d'autres langages, Python ne permet pas de créer un élément dans une liste à un indice trop élevé. Par exemple, si on crée la liste `a = [1, 2, 3]`, elle est de taille 3 et on ne peut pas créer un quatrième élément en écrivant `a[3] = 4`. Il reste cependant possible d'écrire `a += [4]`, et on préfèrera même `a.append(4)` pour un élément et `a.extend([4, 5, 6])` pour une séquence regroupant un nombre arbitraire d'éléments.

Pour retirer un élément, la façon la plus classique consiste à écrire `a.pop()`, cette méthode enlève et retourne le dernier élément de la liste `a`. Il s'avère que la méthode `pop` supporte la précision d'un argument, qui est l'indice d'où l'élément est retiré, mais cette dernière opération a un coût car tous les éléments au-delà subissent un décalage (et donc l'utilisation d'un argument n'est pas à connaître).

Tout ce qu'on a présenté sur les n-uplets fonctionne encore sur les listes.

Il est également possible de créer une liste *en compréhension*, ce qui correspond en mathématiques à la description d'ensembles à partir d'une fonction. Là où le mathématicien écrit $\{f(x) \mid x \in E\}$, Python permet d'écrire `[expr for x in seq]`, engendrant ainsi la liste des valeurs de l'expression `expr`, dépendant a priori d'une variable nommée `x`, et ce pour toutes les valeurs de la séquence `seq`, stockées dans la variable `x`, dans l'ordre de cette séquence. Nous n'entrerons cependant pas ici dans les détails de cette syntaxe, avec la possibilité d'imbriquer des boucles ou de glisser des tests.

Conversions

Passer d'une séquence à une autre peut se faire à l'aide d'un algorithme parcourant les éléments, mais il existe aussi des fonctions pour cela : `list(a)` retourne la liste des éléments de la séquence `a`, donc soit la liste des caractères d'une chaîne soit la liste des éléments d'un n-uplet, et `tuple(a)` retourne le n-uplet des éléments de la séquence `a`.

En revanche, si `a` est une liste ou un n-uplet `str(a)` entoure simplement `a` d'apostrophes, sachant que si `a` est composé de chaînes de caractères, Python gère les conflits de délimiteurs, un algorithme (ou la méthode `join`) est nécessaire pour convertir une liste ou un n-uplet de caractères en la chaîne correspondante.

La méthode `split` est une méthode de séparation d'une chaîne de caractères en liste de chaînes de caractères avec la donnée d'un déparateur. Il s'agit de la réciproque de la méthode `join` évoquée plus haut. On l'utilise par l'instruction `chaine.split(separateur)`. Toutes les occurrences de la chaîne de caractères `separateur` disparaissent alors, occasionnant autant de ruptures de la chaîne de départ et la réorganisation des morceaux dans la liste proprement dite.

0.1.4 Instructions composées

Séquence d'instructions

Enchaîner deux instructions se fait naturellement, en allant à la ligne entre les deux. Il serait valide d'enchaîner deux instructions sur une même ligne, en les séparant d'un point-virgule, mais on évitera pour la propreté du code de recourir à cela.

Disjonction de cas (if) À NE SURTOUT PAS QUALIFIER DE BOUCLE

Une *disjonction de cas*, ou test, correspond à l'instruction de type **Si ... Alors ... Sinon ...**. Elle s'introduit en Python par le mot-clé `if`, suivi de la condition, qui est interprétée comme un booléen, puis d'un **double-point en fin de ligne**.

La partie correspondant au **Alors**, qui est exécutée lorsque la condition est remplie, est délimitée par ce qu'on appelle **l'indentation** : toutes les lignes dans le corps du test doivent commencer par le même nombre, strictement positif, que les recommandations stylistiques fixent à quatre, d'espaces par rapport à la ligne du `if`. **On dit que Python est un langage à indentation signifiante.** Sans cela, Python déclenche une erreur de syntaxe. Par ailleurs, indenter un programme est recommandé dans d'autres langages pour des raisons de lisibilité.

La partie correspondant au **Sinon** peut s'introduire par le mot-clé `else`, suivi d'un **double-point**, avec là aussi une indentation spécifique pour le corps de cette partie, ou par le mot-clé `elif`, suivi d'une autre condition et d'un **double-point**, toujours avec une indentation spécifique.

Un bloc avec `elif` est exécuté si aucune des conditions précédentes n'est remplie alors que la condition actuelle l'est ; un bloc avec `else`, nécessairement en toute fin s'il existe, est exécuté si aucune des conditions n'est remplie.

Par exemple, si on veut affecter à la variable `couleur` la valeur `"rouge"` si une autre variable `valeur` est strictement inférieure à 5, `"jaune"` si `valeur` vaut 5 exactement et `"vert"` sinon, on écrira :

```
if valeur < 5:
    couleur = "rouge"
elif valeur == 5:
    couleur = "jaune"
else:
    couleur = "vert"
```

Boucle inconditionnelle (for)

La *boucle inconditionnelle*, correspondant à **Pour ... de ... à ...**, s'écrit en Python comme un parcours de séquence, c'est-à-dire qu'on aura toujours un objet itérable associé, dont une variable, créée pour l'occasion et qu'on n'a pas besoin d'initialiser par ailleurs, vaudra successivement chacun des éléments dans le corps de boucle.

La syntaxe est `for i in a:`, où `i` est un exemple de nom de variable et `a` est la séquence parcourue. Bien entendu, le double-point est indispensable, ainsi que l'indentation du corps de boucle.

L'exemple le plus typique est celui d'une boucle où `i` prend les valeurs entières de 0 inclus à n exclu, ce qui s'écrit en utilisant la fonction `range`, qui génère une séquence de nombres en fonction de son (ou ses) argument(s). Noter que cette séquence est un « objet-range » et non pas une liste.

On écrira alors `for i in range(n):`.

Au passage, si on veut parcourir une liste `l`, il est tout aussi acceptable d'introduire la boucle par `for x in l:` que `for i in range(len(l)):` en commençant le bloc par `x = l[i]`.

Le premier code, de par sa concision, est à privilégier si la connaissance des indices n'est pas essentielle. Il est par ailleurs possible de créer à la fois l'indice et la valeur en écrivant `for i, x in enumerate(l)`. Pour tout dire, il n'est pas exclu que l'utilisation systématique de la fonction `enumerate` limite les confusions entre les indices et les valeurs.

Remarque : Python ne permet pas de perturber une boucle inconditionnelle par modification de la variable contenant la séquence à parcourir ni par une modification de la variable de boucle, mais elle peut être perturbée si on la fait muter. Constater par exemple que le code suivant imprime effectivement les nombres de 1 à 3 (alors même qu'à la fin la liste `a` est bien vide). Tester également les autres codes.

```
a = [1,2,3]
for x in a:
    a = []
    print(x)

a = [1,2,3]
for x in a:
    a[2] = 42 # Qui sera imprimé
    print(x)

a = [1,2,3]
for x in a:
    del a[:] # On retire la tranche contenant tout
    print(x)

a = [1,2,3]
for x in a:
    del a # Destruction totale de la variable, plantage.
    print(x)

a = [1,2,3]
for x in a:
    a.append(x) # Oh, une boucle infinie !
    print(x)

for i in range(10):
    i = 10 # Imprimera certes uniquement des 10, mais il y en aura dix.
    print(i)

a = [1,2,3]
for x in a:
    a.remove(x) # Suppression
    print(x) # Et le résultat risque de surprendre tout le monde !
```

Boucle conditionnelle (while)

La *boucle conditionnelle*, correspondant à **Tant que ... faire ...**, utilise là aussi une condition qui est interprétée comme un booléen.

On écrit `while cond:`, où `cond` est la condition, on n'oublie toujours pas le **double-point** ni l'indentation dans le corps de la boucle, qui est exécuté tant que la condition est remplie, cette condition étant testée entre la fin de chaque passage dans la boucle et le début de l'éventuel passage suivant (mais pas pendant).

Bien souvent, la condition est une inégalité faisant intervenir une variable. L'écueil classique consiste à oublier de mettre à jour la variable dans le corps de la boucle, ce qui fait que le programme y est piégé. C'est l'une des premières choses à vérifier en cas de problème.

Une boucle conditionnelle peut aisément remplacer une boucle inconditionnelle, à savoir :

```
for x in a:
    (blablabla)
```

est équivalent (aux mutations, réaffectations et autres perturbations près) à :

```
i = 0
while i < len(a):
    x = a[i]
    i += 1
    (blablabla)
```

Bien entendu, il est possible d'imbriquer des boucles et tests l'un(e) dans l'autre à un niveau de profondeur arbitraire.

0.1.5 Fonctions

Il est possible de voir les *fonctions* en programmation comme le pendant des fonctions mathématiques : si on dispose d'une fonction telle que $f(x) = x^4 - x^3 + 2x^2 - 3x + 7$, on ne va pas écrire $\pi^4 - \pi^3 + 2\pi^2 - 3\pi + 7$ puis $e^4 - e^3 + 2e^2 - 3e + 7$ pour les images respectives de π et de e par f, mais bien $f(\pi)$ et $f(e)$.

En programmation, définir une fonction permet entre autres de ne pas avoir à recopier des morceaux de code plusieurs fois, et surtout c'est l'essence même de la programmation que de définir des procédures qui vont nécessiter le moins possible l'intervention de l'utilisateur.

Une fonction a un certain nombre d'*arguments*, appelés « paramètres » par certains, qui sont des valeurs d'entrée sur lesquelles elle est appelée, comme π et e dans notre exemple.

En Python, on définit une fonction ainsi : `def f (arg1, arg2, ...):`, où `f` est le nom de la fonction, et les arguments sont les noms arbitraires de variables dans la parenthèse.

Comme pour les instructions composées, on n'oublie pas le **double-point** ni l'indentation ensuite. À l'intérieur du bloc où l'on définit la fonction, les arguments définis dans la parenthèse peuvent être utilisés tels quels.

Une fonction a aussi une *valeur de retour*, qui peut être de n'importe quel type. Cette valeur est renvoyée par la fonction quand on utilise l'instruction `return`, qui interrompt toutes les exécutions de la fonction. C'est normal : une fonction ne peut retourner quelque chose qu'une fois.

Une fois la fonction définie, elle s'invoque en écrivant `f(a, b, ...)` où tous les arguments doivent s'évaluer en une constante, c'est-à-dire que s'il s'agit d'expressions, les variables qu'elles contiennent doivent toutes être définies.

Une fonction sans argument exécute simplement du code, en modifiant éventuellement des variables dites globales et en ayant éventuellement une valeur de retour (constante, aléatoire voire obtenue par des interactions). On l'appelle avec des parenthèses vides.

La fonction définie précédemment s'écrit et s'appelle alors ainsi sur l'entier 3 :

```
def f(x):
    return x**4-x**3+2*x**2-3*x+7

f(3)
```

0.1.6 Entrées et sorties

L'interaction fait partie des fonctionnalités nécessaires pour qu'un langage de programmation puisse faire tout ce qu'on attendait d'un ordinateur à l'origine.

L'instruction `input(message)` affiche `message` et renvoie ce que l'utilisateur a écrit en réponse, en tant que chaîne de caractères qu'il faut convertir soi-même en cas de besoin, ce pour quoi la fonction `eval` est particulièrement utile à connaître.

Attention : pour les fonctions présentées ci-avant, il ne faut pas oublier les guillemets autour du message, s'il s'agit d'une chaîne de caractères ; bien entendu, si `message` est une variable, sa valeur sera imprimée par "pretty-print" (si la valeur n'est pas une chaîne de caractères).

En ce qui concerne la lecture d'un fichier, nous allons omettre pour le moment la notion de chemin et considérer que tous les fichiers sont dans le même répertoire, chargé par l'interpréteur Python.

Lorsqu'on ouvre un fichier, que ce soit comme ici en mode lecture ou en mode écriture (ou ajout), on utilise la fonction `open` : `fichier = open(nom_du_fichier, "r")`, ce qui crée un « objet-fichier », où `nom_du_fichier` est une chaîne de caractères qui contient bien entendu l'extension également.

L'argument `"r"` indique que le fichier est ouvert en lecture seule ("read"), donc il n'est pas modifié. D'autres modes de lecture sont possibles, et sans précision du mode c'est celui-là qui est utilisé par défaut.

Il n'est jamais trop tôt pour signaler qu'une fois qu'on a fini de traiter un fichier on le ferme par habitude et pour éviter les problèmes. Cela se fait simplement par `fichier.close()`, où `fichier` est la variable définie précédemment. Faute de fermeture du fichier, certains programmes refuseront temporairement d'afficher le nouveau contenu du fichier, et des conflits en écriture sont possibles.

La syntaxe est peut-être troublante, elle provient de la programmation orientée objet, à l'instar de `l.append(x)`.

Pour récupérer le contenu d'un fichier, on utilise `fichier.read()`, qui retourne la totalité du fichier. Si on ajoute un argument, cela définit le nombre maximal de caractères à lire. Utiliser `readline` au lieu de `read` permet de s'arrêter en fin de la ligne actuelle, en incluant le caractère de fin de ligne, et non en fin de fichier, et `readlines` retourne la liste des lignes restantes. Dans tous les cas, le contenu total, ou chaque ligne le cas échéant, est de type chaîne de caractères.

La gestion de ce contenu n'est pas à connaître par cœur. On signalera simplement l'existence de `fichier.seek(position, mode)` qui place le pointeur de lecture (déplacé par `fichier.write(n)` de `n` caractères et initialement au début du fichier) au `position`-ième caractère (si `mode` vaut `0`), `position` caractères plus loin (si `mode` vaut `1`) ou au `position`-ième caractère en partant de la fin (si `mode` vaut `2`).

L'instruction `print(x)` imprime `x`, quel que soit son type. Après l'impression de `x`, un retour à la ligne est produit, ce qu'on peut modifier en ajoutant un argument `end=y` après ce qui est imprimé, de sorte que le retour à la ligne est remplacé par `y`.

Il est possible d'imprimer plusieurs choses à la fois, avec des types totalement arbitraires, qui seront autant d'arguments de la fonction `print`. L'impression se fait alors successivement en séparant deux arguments par une espace, ce qu'on peut également modifier en ajoutant, également à la fin, un argument `sep=z`.

Par exemple, tester le code suivant :

```
print("A", "B", "C", sep="-", end="!")
```

L'impression dans un fichier nécessite que celui-ci soit ouvert en écriture (auquel cas son contenu original est définitivement supprimé) ou en ajout (auquel cas ce qu'on imprime est mis à la suite).

On utilise encore la fonction `open` : `fichier = open(nom_du_fichier, "w")`
ou `fichier = open(nom_du_fichier, "a")`, respectivement, ce qui crée le fichier s'il n'existait pas encore.

Une fois le fichier ouvert, on y écrit ainsi : `fichier.write(contenu)`, la fonction ayant par ailleurs la particularité d'avoir une valeur de retour, à savoir le nombre de caractères écrits. Cela peut toujours servir...

0.2 Compléments

Cette partie du polycopié présente des notions qui ne sont pas forcément à maîtriser, ni forcément déjà vues en lycée, mais éventuellement utiles pour les TP, et surtout utiles dans la perspective d'un approfondissement important de la programmation dans la suite de la scolarité ou en-dehors des études.

0.2.1 Variables globales et variables locales, etc.

Dans un programme en Python, toutes les variables qui sont créées en-dehors de la portée d'une fonction sont appelées *variables globales*. Ces variables sont considérées comme existantes et donc utilisables à partir du moment où l'instruction où elles sont déclarées est exécutée.

Comme une fonction peut être vue comme une variable, cela permet de revenir sur le fait qu'il faille exécuter le contenu de l'éditeur pour pouvoir appeler dans la console les fonctions qui y sont définies. Par ailleurs, cela vaut aussi et surtout pour la prise en compte des corrections d'erreurs.

Ainsi, dans le code suivant, la fonction `f` est d'abord définie, puis la variable `a`, de sorte que lors de l'appel ultérieur à `f(4)`, `n+a` peut effectivement être calculé et vaut `7`, qui est imprimé.

```
def f(n):
    print(n+a)

a = 3
f(4)
```

On peut ainsi écrire une fonction qui en fait intervenir une autre, définie plus tard, à condition que la fonction ne soit pas appelée avant que l'autre ne soit définie.
Attention : Ceci ne serait dans tous les cas pas autorisé dans d'autres langages, notamment en Caml.

À l'intérieur du code définissant une fonction, toutes les variables qui sont définies, de même que les arguments d'une fonction, sont appelées *variables locales* : elles n'ont pas d'existence en-dehors de la fonction, sauf si elles partagent le nom d'une variable globale, auquel cas cette dernière est localement oubliée, c'est-à-dire qu'on ne peut pas accéder à sa valeur.

Ainsi, le code suivant imprimera la valeur `42`, puis `0`, ce qui témoigne du fait que la valeur de `a` n'est pas affectée en-dehors du code de la fonction `f` :

```
def f(a):
    a = 42
    print(a)

a = 0
f(4)
print(a)
```

Certaines fonctions ont un effet de bord modifiant leur argument, lorsqu'il s'agissait de listes par exemple.

En pratique, lorsqu'une liste est un argument d'une fonction, la variable locale correspondante est un alias pour cette liste, et toute modification de l'un entraîne une modification de l'autre, ce qui est exactement le même comportement que lorsqu'on écrit `b = a`.

Ces effets de bord peuvent s'obtenir pour d'autres variables globales à condition de les déclarer comme telles dans la fonction. Pour autant, mieux vaut éviter d'avoir recours à cela.

Ainsi donc, le code suivant imprimera deux fois la valeur `42`, ce qui signifie que `a` a été modifiée.

```
def f():
    global a
    a = 42
    print(a)

a = 0
f()
print(a)
```

En pratique, si `a` avait été un argument de la fonction, Python aurait provoqué une erreur car un argument et une variable globale, censés coexister, ne peuvent pas avoir le même nom.

La question qui demeure est celle de variables ni locales ni globales pour une fonction qui est elle-même définie dans une autre (voir la sous-section suivante).

Dans ce cas, le mot-clé est `nonlocal`, et le code suivant imprimera successivement `42` puis `0`, car la variable `a`, locale à `f` mais pas à `g`, est modifiée par cette dernière, sans conséquence cependant pour la variable globale également nommée `a`.

```
def f():
    a = 73
    # Premier contact avec une fonction locale
    def g():
        nonlocal a
        a = 42
    g()
    print(a)

a = 0
f()
print(a)
# noter qu'ici g n'est pas définie
```

On retiendra tout de même que ces pratiques sont esthétiquement douteuses.

Au passage, un tel souci permet d'expliquer pourquoi en exécutant les deux codes, pourtant semblables, Python déclenche une erreur dans le deuxième cas seulement :

```
def f1(a):
    print(b)
b = "hello world"
f1(b)

def f2(a):
    b = b + "coucou"
b = "bonjour"
f2(b)
```

En résumé, à l'intérieur d'une fonction, lorsqu'une expression fait référence à une variable, Python cherche la valeur définie à l'intérieur de la fonction et à défaut la valeur dans l'espace global du module. La notion en jeu est celle de *portée lexicale*.

Dans le même ordre d'idée, lorsque Python définit une fonction `f` qui dépend d'une autre, disons `g`, puisque chaque appel à `f` exécute son code, on pourrait se demander si c'est au moment de la définition de `f` que le code de `g` est copié ou si Python appelle `g` au moment de l'appel de `f`.

La réponse peut être anticipée de ce qui précède : puisque `g` peut ne pas encore avoir été définie sans que des problèmes se posent, la deuxième réponse est retenue.

En fait, Python fait ce que l'on appelle des *liaisons dynamiques*, de sorte que redéfinir `g` après coup modifie le comportement de `f` sans qu'on n'ait besoin de relancer la définition de celle-ci.

Ainsi, le code suivant imprimera successivement `1` puis `2`, signifiant que le code de la fonction `f` aura effectivement changé par la redéfinition de `g` :

```
def g():
    print(1)

def f():
    g()

f()

def g():
    print(2)
f()
```

Sans entrer dans le détail des mécanismes qui régissent Python, on peut donner un principe qui donnera l'idée de son comportement : une variable d'un type *mutable* sera susceptible de subir un effet de bord dans une fonction, une variable *immutable* non. Jusque là, les listes sont les seuls objets mutables que nous avons vus, ce qui rejoint la discussion précédente.

En Python, lors d'un appel d'une fonction, on passe les variables en argument *par affectation*, c'est-à-dire que les arguments d'une fonction sont simplement des noms, et la fonction travaille directement sur les variables en question.

Ceci étant, une variable immutable ne sera pas modifiée par la fonction, et des réaffectations éventuelles n'auront pas d'effet en-dehors de la fonction, car on considèrera les variables mises en jeu comme locales.

Deux autres possibilités en programmation (dans d'autres langages, en pratique) sont le *passage par valeur*, ce qui signifie qu'on procède à une évaluation et c'est la valeur qui est transmise à la fonction, en tant qu'objet virtuel tout neuf et en omettant totalement la provenance de ladite valeur, et le *passage par référence*, ce qui signifie que ce qui est manipulé est une sorte de pointeur vers une adresse mémoire où se trouve la valeur en question, qui sera éventuellement modifiée, ce qui peut réaffecter des variables même stockant un entier (alors qu'en Python les entiers sont immutables, et ce phénomène nécessite par exemple d'utiliser le mot-clé `global`).

En outre, de la même façon que dans le cas du passage par valeur, avant d'exécuter une fonction, les arguments sont évalués, notamment lorsqu'il s'agit d'expressions. En cas d'effet de bord, cela peut changer beaucoup de choses...

0.2.2 Fonctions locales, fonctions anonymes

De même qu'on utilise des variables locales dans le code de fonctions, puisqu'on a dit que les fonctions étaient des variables, il est possible de définir ce qu'on appelle des *fonctions locales*, c'est-à-dire des fonctions qui n'existent que dans le code d'une fonction.

Sans entrer dans les considérations sur l'efficacité en termes d'utilisation de mémoire (ni du fait qu'à chaque appel les fonctions locales soient redéfinies) dans certains langages, voici un exemple :

```
def coeff_bin(n, m):
    def fact(x):
        s = 1
        for i in range(1, x+1):
            s *= x
        return s
    return fact(n)//(fact(m)*fact(n-m))
```

Bien entendu, les variables locales à la fonction locale n'ont pas d'existence dans la fonction principale.

En fait, les fonctions locales définissent un niveau d'imbrication supérieur par rapport aux fonctions, et rien n'exclut d'aller encore plus profondément en créant une fonction locale à une fonction locale.

Pour les fonctions locales que l'on n'utiliserait qu'une fois et dont le code serait court, il n'est pas nécessaire de faire une définition sur plusieurs lignes, et on peut avoir recours à des *fonctions anonymes* à l'aide du constructeur `lambda` provenant du lambda-calcul.

La syntaxe est la suivante : `lambda x_1, ..., x_n : resultat`, ce qui est équivalent à :

```
def f(x_1, ..., x_n):
    return resultat # en fonction des arguments
f # noter ce f ici qui fait partie de l'expression en lambda
```

Le `f` final est l'objet de type fonction obtenu par l'instruction avec `lambda`. Ainsi, pour appeler directement une fonction anonyme, par exemple une fonction qui élève au carré, de même qu'on écrirait `carre(10)`, on écrira `(lambda x : x * x)(10)`.

Par conséquent, pour obtenir la somme des carrés des entiers de `1` à `n`, on donne les quatre versions équivalentes (la quatrième étant réservée aux casse-cous) :

```
def somme_carres1(n):
    s = 0
    for i in range(n+1):
        s += i * i
    return s

def somme_carres2(n):
    def carre(x):
        return x * x
    return sum(map(carre, range(n+1)))

def somme_carres3(n):
    return sum(map(lambda x: x * x, range(n+1)))

somme_carres4 = lambda n: sum(map(lambda x: x * x, range(n+1)))
```

L'exemple précédent permet d'introduire la fonction `map`, qui applique son premier argument, nécessairement une fonction, à chacun des éléments de son deuxième argument, nécessairement un itérable mais de type quelconque, formant un objet itérable qui est renvoyé et qu'on convertit généralement en liste (ou en n-uplet, voire en chaîne de caractères à l'aide de la méthode `join`).

0.2.3 Exceptions

Cette sous-section peut être lue en parallèle du TP sur le debug, où de nombreux messages d'erreur sont présentés.

Déclencher une *erreur* peut se faire facilement, voire involontairement. Outre les messages renvoyés par Python quand on lui demande par exemple de diviser par zéro, ce qu'on appelle *exception* peut être de nature très variée.

L'assertion est la seule erreur à connaître dans le programme officiel, elle est détaillée au chapitre 1.

Les exceptions ont un type, qui est précisé en début de message d'erreur, et la valeur des exceptions est donnée entre guillemets ensuite.

Ainsi, quand on demande la racine d'un nombre strictement négatif, la syntaxe est correcte, le type aussi, l'erreur réside dans la valeur : Python renvoie `ValueError: math domain error`.

De même, si on veut accéder à un élément inexistant dans une liste (ou dans un itérable en général), c'est une erreur d'indice : `IndexError: list index out of range`.

Déclencher une exception manuellement se fait par le mot-clé `raise`. On écrira donc `raise ValueError("Je voulais 42")` pour déclencher une erreur personnalisée.

Tout cela serait bien inutile si rencontrer une erreur faisait irrémédiablement cesser l'exécution d'un programme. Par chance, toute exception peut être rattrapée.

La syntaxe est la suivante :

```
try:
    # on se met dans un contexte où chaque exception est rattrapée
    code_avec_exceptions_possibles
    # il s'agit bien entendu d'un bloc, donc indentation et double-point
except (un_type_d_exception, un_autre):
    autre_code_sans_que_les_nouvelles_exceptions_soient_rattrapees
    # le code ci-dessus est exécuté si une exception d'un des types est soulevée
except un_autre_type:
    autre_code_idem
    # à l'instar du elif, ce code est exécuté si rien de ce qui précède ne l'est
    # mais qu'une exception de ce nouveau type est soulevée
except:
    # toutes les autres exceptions sont rattrapées ici
    encore_un_autre_code_idem
else:
    # si aucune exception n'est déclenchée, ce code est exécuté
    code_pas_idem
finally:
    # ce code est exécuté à la fin dans tous les cas
    code_final
# étonnamment, un return dans le finally remplacerait tout autre return
```

Dans ce qui précède, seuls les blocs introduits par `try` et par `except` sont obligatoires.

La valeur associée à une exception peut être récupérée dans une variable par une précision dans la syntaxe : `except type as valeur`.

En imbriquant les `try`, on peut rattraper une exception soulevée au moment où une exception était rattrapée, mais trouver un programme faisant naturellement ceci semble très artificiel.

Ce qui est en revanche envisageable est de rattraper une erreur afin de transmettre un message à l'utilisateur avant de redéclencher l'erreur en écrivant simplement `raise`.

La fonction `exc_info` du module `sys`, qui n'a pas d'argument, renvoie un triplet dont le premier élément est le type de l'exception déclenchée et le deuxième élément est sa valeur. Ceci s'illustre par le code suivant :

```
import sys
def kamikaze(a):
    try:
        i = 0
        while True:
            1 / a[i]
            i += 1
    except:
        print("Une erreur a été déclenchée pour i =",i,": son type est",
sys.exc_info()[0],"et le détail est \"",sys.exc_info()[1],"\"")
        raise

kamikaze([1, 2, 3])
kamikaze([1, 2, 3, 0])
kamikaze([1, 2, 3, "0"])
```

Il est possible de mettre la valeur des exceptions dans une variable, pour s'en servir au moment de rattraper les erreurs :

```
def kamikaze2(a):
    try:
        kamikaze(a)
    except IndexError as valeur:
        if str(valeur) == "list index out of range":
        # car valeur est du type de l'exception
            print("a est une liste de nombres tous non nuls")
        else:
            print("""a n'est pas une liste
mais tous ses éléments sont des nombres non nuls""")
    except:
         print("le parcours de a n'a pas pu se finir")

kamikaze2([1, 2, 3])
kamikaze2([1, 2, 3, 0])
kamikaze2([1, 2, 3, "0"])
```

0.3 Petit détour par les bibliothèques

Un *module* Python est en quelque sorte un ensemble d'instructions qu'on va pouvoir importer. En un sens, tout fichier Python peut être vu comme un module et son code récupéré, pour un effet similaire au fait de l'exécuter (en ne tenant pas compte des réinitialisations de la console). Importer un module sert usuellement à définir des variables et fonctions, et même des classes en programmation objet, pour gagner du temps.

Une *bibliothèque* peut à première vue se confondre avec un module, mais elle doit être vue comme une boîte à outils, chargée dans un but spécifique, toujours en gardant l'idée de ne pas avoir à recréer ses outils *ex nihilo*.

En pratique, les fonctions usuelles qui ont été vues au chapitre précédent font eux-mêmes partie d'une bibliothèque, appelée bibliothèque standard (*core library*) et qui est importée systématiquement par Python.

Pour information ou rappel, un module (ou une bibliothèque, donc) peut être importé en entier ou un sous-ensemble de ses fonctions peut être importé. La syntaxe de l'importation change, de même que celle des appels aux fonctions (et aux variables en général) importées.

Les morceaux de code suivants produisent la même chose mais on ne peut pas échanger une ligne d'un morceau avec une ligne de l'autre sous peine de déclencher une erreur de nom :

```
import mon_module # tout le module est disponible
mon_module.ma_fonction(mon_module.ma_variable)

import mon_module as momo
# momo sera un alias, et mon_module sera inconnu
momo.ma_fonction(momo.ma_variable)

from mon_module import ma_fonction, ma_variable
# le reste du module est inaccessible
ma_fonction(ma_variable)

from mon_module import * # tout est importé
ma_fonction(ma_variable)
```

En utilisant un environnement de développement intégré, on est amené à voir apparaître des rectangles expliquant le principe d'une fonction dont on commence à écrire le nom. C'est de la documentation de code, et ces « *docstrings* » peuvent bien entendu être écrits par n'importe qui : par exemple, au moment d'écrire une fonction, juste en-dessous de la ligne de définition, écrire une chaîne de caractères (généralement sur plusieurs lignes, et donc avec des triples guillemets comme délimiteurs) fera comprendre à Python qu'on a précisé un message pour l'utilisateur, et ceci sera imprimé dès lors qu'on demandera des informations à l'aide de la fonction `help`, qui prend en argument un nom de variable, usuellement une fonction, que cette variable soit issue de la bibliothèque standard, d'un module ou d'une bibliothèque importée, en préfixant le cas échéant, ou définie par l'utilisateur.

Bien entendu, documenter une fonction fait partie des critères rendant un code agréable à lire, de même que des noms de variable et de fonction pertinents, des commentaires sur des passages délicats et une indentation rigoureuse (de toute façon Python ne laisse pas grandement le choix sur ce dernier point).

Pour aller encore plus loin, il est possible de faire des annotations au niveau des fonctions, qui contribuent encore plus à leur lisibilité. Il est possible d'en faire un réflexe pour les utilisateurs de Caml, et pour tout le monde en fait.

Les annotations ont été présentées dans le sujet de Centrale en 2017, disponible sur le site du concours. Un autre lien utile à ce sujet : `http://sametmax2.com/les-annotations-en-python-3/`.

Des bibliothèques et modules peuvent contenir eux-mêmes des sous-modules, qu'il faut parfois importer même si la bibliothèque ou le module l'a été.

Chapitre 1

Méthodes de programmation et analyse des algorithmes

1.1 Compléments théoriques

En complément du chapitre précédent, rappelant les bases de Python dans l'optique d'apporter plutôt une aide technique, il convient d'ajouter ici des précisions théoriques.

La notion d'*expression*, bien qu'intuitive, mérite une définition détaillée. Une expression est une combinaison de valeurs, certaines pouvant être obtenues par des appels de fonctions, utilisant éventuellement des opérateurs, permettant d'obtenir une nouvelle valeur lors de l'exécution du fragment de code ainsi formé.

Au contraire, une *instruction* est un fragment de code dont l'exécution produit un effet, ce qui fait en général intervenir l'évaluation d'une expression pour se servir de la valeur alors calculée : elle peut être stockée en mémoire ou affichée, par exemple.

Ainsi, *affecter* une variable à une valeur **est une instruction en Python**. Il est intéressant de constater que ce n'est pas le cas dans tous les langages : en laissant de côté le langage Caml, où la notion d'instruction serait en toute rigueur absente, on peut signaler que dans le langage C une affectation est une expression, et donc `x = 3` a pour valeur ce qui est dans `x` à la fin de l'évaluation. Il ne faut alors pas croire que c'est le cas en Python, malgré la possibilité d'écrire `x = y = 3` pour créer en même temps deux variables `x` et `y` initialisées à la valeur `3`. Cette syntaxe abusive induit en erreur, et il est en revanche normal que `x = (y = 3)` déclenche une erreur, au contraire du C, où on peut même se permettre d'écrire `x = (y = 3) + 1` pour que `y` soit affecté à `3` et `x` à `4`.

Pour mieux comprendre ce qui se passe au niveau d'une affectation, il est bon de garder à l'esprit qu'en Python, tout est objet, avec une gestion particulière de la mémoire.

L'instruction `x = qqch` ne lie pas `x` à `3` mais associe à `x` une adresse mémoire temporairement, celle où `qqch` vient d'être écrit, voire l'était déjà, ce qui est le cas de certaines valeurs en cache comme des entiers proches de zéro, et pour les objets mutables.

Cette adresse mémoire peut être amenée à changer facilement, contrairement au cas du C, ne serait-ce que parce que le changement de type d'une variable est possible et que la taille de la mémoire utilisée peut alors changer, même quand le type ne change pas, par ailleurs.

Une manipulation recommandée pour mieux comprendre ce principe est de créer puis réaffecter des variables, notamment des entiers et des listes, et de regarder l'endroit où elles sont stockées dans la mémoire, ce qui se fait en utilisant la fonction `id`. Cette même manipulation permet également de comparer les diverses façons de copier une liste. Ceci fait l'objet du TP 9.

Puisqu'une affectation n'est pas une expression, il est également interdit de s'en servir en tant que condition, ne permettant pas d'écrire du code tel que `while ((c = getchar()) != EOF)` utilisé classiquement en C. Pour pallier cette absence, une récente version de Python a ajouté un opérateur noté `:=`, au grand dam de beaucoup de programmeurs [1]. Cet opérateur n'est bien entendu pas à connaître et mieux vaut éviter de l'utiliser dans une copie si on n'est pas certain que le correcteur le connaît.

De même, pour ceux qui utilisent l'opérateur `lambda`, sa partie droite doit être une expression et ne supporte donc pas les affectations.

Pour autant, ces deux conséquences ne sont pas liées à des difficultés techniques, notamment puisque l'opérateur `:=` a pu être introduit sans mal, il s'agissait simplement d'un choix au moment de la création de Python.

Terminons par définir la notion d'*effet de bord*. Ils ont déjà été rencontrés, et même mentionnés, notamment dans le TP sur les tris, où ceux-ci pouvaient être amenés à créer une nouvelle liste, version triée de la liste en argument, ou modifier la liste en argument. Le deuxième choix est précisément l'application d'un effet de bord, c'est-à-dire la modification, notamment par l'application d'une fonction, de l'état du système, que ce soit par la mutation de variables mutables ou, dans d'autres cas, l'affichage d'une valeur ou toute autre interaction.

Le terme « effet de bord » est un choix malheureux selon certains auteurs, en tant que traduction du terme anglais "side effect". Il est peut-être plus approprié de dire « effet secondaire », évitant par ailleurs l'homonymie avec les effets de bord en sciences physiques.

L'effet de bord est surtout évoqué lorsqu'il est inattendu et cause des erreurs en raison de la négligence de l'auteur d'un programme, le débogage peut alors être axé sur sa recherche, en cas de doute. Parmi les exemples de mauvaises surprises en programmation éventuellement avancée : l'utilisation de variables mutables comme arguments optionnels d'une fonction ou la mutation d'une liste en plein parcours (car l'effet de bord n'est pas limité aux fonctions).

1. dont le créateur du langage :
`https://pythonsimplified.com/the-most-controversial-python-walrus-operator/`

1.2 Spécification, justification

1.2.1 Spécification, syntaxe des annotations

Lorsqu'on écrit un programme, pour peu qu'il ne provoque pas d'erreur de syntaxe, il n'est pas faux de dire qu'il donne un résultat, même si ce résultat est une erreur d'indexation ou n'importe quel calcul inutile.

Cependant, dire que le programme fait bien ce qu'on attend de lui implique évidemment d'avoir défini à l'avance ce qu'on attend de lui. Il s'agit de la notion de *spécification*. À notre niveau, il s'agira simplement d'écrire pour chaque fonction ce qu'elle attend comme type d'argument, ce qu'elle renvoie, ses effets de bord et toute information utile d'ordre général, à compléter par des commentaires sur des endroits particuliers.

On l'aura compris, la spécification s'écrit dans la documentation de code, qu'il est recommandé de toujours rédiger en l'absence de contraintes de temps trop restrictives. Cette documentation de code a déjà été traitée dans le TP 3, et la syntaxe des annotations y a été évoquée. C'est le moment de l'illustrer par un exemple. Considérons un début de création de fonction :

```
def insere_dicho(l : list, x) -> list:
```

Ici, la ligne traditionnelle `def insere_dicho(l, x):` a été quelque peu étoffée. Il est indiqué que la variable `l` est de type `list`, et la flèche précise le type de la valeur de retour. Quant à `x`, il semble être de type quelconque, vu l'absence d'informations. Ces informations se retrouveront, comme la documentation de code, dans le texte imprimé par l'appel de la fonction `help`. Cependant, **les types ainsi annoncés ne forment pas une contrainte au niveau de l'exécution**, au contraire de la syntaxe analogue utilisable en Caml. On appelle ceci la *signature* de la fonction.

Ensuite, la spécification de la fonction. Comme le choix du nom est pertinent, une autre pratique recommandée, on comprend qu'il s'agit d'une insertion dichotomique. Il faut encore apporter des clarifications.

```
def insere_dicho(l : list, x) -> list:
    """Crée la liste obtenue par insertion de x dans l. La liste l n'est pas modifiée.
    On procède par dichotomie pour chercher l'endroit où x est à insérer,
    ce qui suppose la liste de départ croissante.
    Aucune vérification n'est faite, ce qui peut donner des résultats aberrants
    quand la liste de départ n'est pas croissante.
    L'insertion est par la suite en temps linéaire."""
```

Tout ce qu'on voulait savoir est désormais annoncé. Il n'y a pas d'effet de bord, une hypothèse doit être vérifiée par l'utilisateur pour une bonne utilisation, et la dichotomie se retrouve au moment de chercher la position, sachant que le calcul de complexité précis sera vu ultérieurement.

Le programme peut alors se lire plus facilement. Précisons pour autant que l'écriture de la documentation peut se faire une fois celle de la fonction terminée, voire s'ébaucher en premier puis s'étoffer avec d'éventuelles corrections à la fin.

```
def insere_dicho(l : list, x) -> list:
    """Crée la liste obtenue par insertion de x dans l. La liste l n'est pas modifiée.
    On procède par dichotomie pour chercher l'endroit où x est à insérer,
    ce qui suppose la liste de départ croissante.
    Aucune vérification n'est faite, ce qui peut donner des résultats aberrants
    quand la liste de départ n'est pas croissante.
    L'insertion est par la suite en temps linéaire."""
    avant, apres = 0, len(l)
    while avant < apres:
        milieu = (avant + apres) // 2 # ne sera jamais len(l) dans la boucle
        if l[milieu] == x:
            avant = apres = milieu # fin au tour suivant
        if l[milieu] < x:
            avant = milieu+1
        else:
            apres = milieu
    return l[:avant] + [x] + l[avant:]
```

1.2.2 Préconditions, postconditions

Restons sur l'exemple précédent. Il se lit certes plus facilement, mais peut-être n'est-il pas immédiat de se convaincre qu'il fonctionne. Nous allons utiliser des *préconditions et postconditions*, qui sont à la base de la logique de Hoare. [2]

Pour ce faire, des passages techniques du code bien cernés vont s'accompagner de commentaires supplémentaires avant et après chaque passage, de sorte que le commentaire avant contienne une information supposée vraie à ce moment de l'exécution, la *précondition*, et que le commentaire après contienne une information qui se déduit de la précondition ainsi que de l'exécution du passage. L'exemple le plus simple est le suivant :

```
# ici x vaut 3
y = 2*x # ici y vaut 6... et x continue de valoir 3
```

Appliquons ceci à la fonction d'insertion dichotomique. Notons qu'une postcondition d'un passage peut devenir une précondition d'un autre passage, et un code entièrement balisé aurait les hypothèses sur les arguments en précondition et la spécification déductible de la postcondition.

Par choix personnel, qui n'est nullement une convention, les préconditions seront introduites par une flèche vers la gauche dans un programme et les postconditions d'une flèche vers la droite. Dans l'écriture sur papier d'un programme, on utilise usuellement des accolades pour les deux.

Ces préconditions et postconditions ont tout à fait leur place sur une copie quand une preuve est demandée ou quand le programme est si obscur que le correcteur pourrait se fâcher sans explications. L'autre cas d'utilisation est la recherche de bug, parfois même utile quand le bug est une bête coquille.

2. Il s'agit d'une méthode particulièrement efficace pour prouver des programmes. Elle n'est pas à maîtriser en détail mais n'en est pas moins abordable.

```
def insere_dicho(l : list, x) -> list:
    """Crée la liste obtenue par insertion de x dans l. La liste l n'est pas modifiée.
    On procède par dichotomie pour chercher l'endroit où x est à insérer,
    ce qui suppose la liste de départ croissante.
    Aucune vérification n'est faite, ce qui peut donner des résultats aberrants
    quand la liste de départ n'est pas croissante.
    L'insertion est par la suite en temps linéaire."""
# <- RAS
    avant, apres = 0, len(l)
# -> les élements strictement à gauche de l'indice avant sont < x (il n'y en a pas...)
# -> les élements à droite de l'indice apres sont > x (il n'y en a pas...)
# -> la liste est vide ou avant < après, on note h l'écart
    while avant < apres:
# <- postconditions précédentes
        milieu = (avant + apres) // 2 # ne sera jamais len(l) dans la boucle
        if l[milieu] == x:
            avant = apres = milieu # fin au tour suivant
        if l[milieu] < x:
            avant = milieu+1
        else:
            apres = milieu
# -> avant ou apres s'est déplacé mais dans les deux cas
#        les deux premières postconditions précédentes restent vraies,
#        en remplaçant les inégalités strictes par des larges dans le premier cas
# -> apres-avant < h (nul dans le premier cas), on le renote h pour la suite
# <- avant = apres, les éléments à gauche sont <= x et les éléments à droite sont >= x
    return l[:avant] + [x] + l[avant:]
# -> liste croissante avec l augmentée x renvoyée
```

Si la place le permet (ce qui n'était pas le cas ici), indenter les préconditions et postconditions est une bonne idée.

1.2.3 Justification du programme

Avant le programme, il y a d'ordinaire le problème nécessitant de programmer. Parfois, il s'agit même d'une spécification, en tant que cahier des charges afin de cadrer ce qui est à réaliser.

Par conséquent, le programmeur a la liberté, qu'on pourrait qualifier d'embarras du choix, dans le style de programmation à employer, les structures de données à utiliser, et bien entendu l'agencement des opérations au niveau de l'algorithme.

Le principe à garder en tête est qu'il n'y a pas forcément un seul programme qui répond à la question, ni même un seul programme qui y réponde le mieux. L'efficacité et la simplicité peuvent être à équilibrer, l'optimisation des cas particuliers rares peut être retenue au détriment d'autres avantages ou non, cette optimisation pouvant être par exemple une accélération des calculs ou l'évitement que le programme ne perde trop de temps.

Cependant, il peut y avoir des programmes directement meilleurs que d'autres, et il faut pouvoir les reconnaître en ce sens, pour être en mesure de les écrire à terme.

En tout état de cause, un programmeur professionnel, et même un étudiant dans l'idéal, doit pouvoir faire les choix évoqués ci-avant et expliquer les raisons qui l'ont motivé. Par la réflexion impliquée, les meilleures options se dessinent plus facilement.

1.3 Preuves, complexité

Pour compléter la section précédente, l'utilisation des préconditions et postconditions permettait de se convaincre que les portions de code faisaient bien ce qu'on attendait d'elles, mais il ne s'agissait pas encore de preuves en bonne et due forme.

Par ailleurs, la mention de l'écart entre `avant` et `apres` n'était pas innocente : un élément garantissant que le programme faisait ce qu'il fallait avait été laissé délibérément de côté pour le moment, à savoir s'assurer que le programme ne provoque pas de boucle infinie, plus généralement qu'il termine.

Les *preuves de terminaison et de correction* sont souvent proches de raisonnements par récurrence, et la plupart du temps elles ne se compliquent que lorsque le programme fait intervenir des boucles, puisqu'une suite d'instructions simples peut simplement être assortie de préconditions et postconditions.

1.3.1 Preuves de terminaison

Commençons par une bonne nouvelle : une boucle inconditionnelle (`for`) termine toujours en Python, sauf en cas de mutation de la séquence à parcourir ou si celle-ci est de taille infinie, ce qui constitue généralement une faute de goût.

Par ailleurs, écrire par exemple `for i in range(2, 5, -1)` donnera une boucle qui n'est jamais exécutée (l'itérable est en fait vide) et `for i in range(42, 45, 0)` donnera une erreur car Python n'accepte pas zéro en troisième argument de `range`.

Remarque : Ce n'est pas le cas de tous les langages, le code PHP suivant produit une boucle infinie : `$n = 1; for ($i = 0 ; $i < $n ; $i++) $n++;`, et son pendant en C a le même effet.

En pratique, nous allons nous concentrer sur les preuves de terminaison de programmes utilisant une boucle conditionnelle (`while`), et de programmes récursifs.

La théorie mathématique que l'on peut utiliser dans pour ainsi dire tous les cas est la suivante : l'ensemble $\mathbb{N}$ est bien fondé, c'est-à-dire qu'il n'existe pas de suite infinie strictement décroissante d'entiers naturels. Pour prouver qu'une boucle conditionnelle termine, il suffit de trouver une expression dépendant des variables du programme et qui s'évalue en un entier décroissant strictement à chaque tour dans la boucle.

Une telle expression, appelée *variant*, est souvent assez rapide à trouver.

Prenons pour exemple un morceau du programme, écrit plus tôt dans ce chapitre, qui simule une boucle inconditionnelle (ça tombe bien...) :

```
i = 0
while i < len(a):
    x = a[i]
    i += 1
```

On prend pour variant `len(a)-i`. Puisque la boucle est quittée dès que `i` ne sera plus strictement inférieur à `len(a)`, on peut considérer que cette expression est toujours un entier naturel. Ensuite, puisque `a` n'est jamais modifié, sa taille non plus à plus forte raison, alors que `i` est augmenté d'un à chaque tour dans la boucle. Par conséquent, le variant décroît strictement à chaque tour, ce qui garantit la terminaison du programme.

Un deuxième exemple, venu du futur, en l'occurrence, la section suivante, est la fonction qui calcule la somme des chiffres d'un entier naturel représenté en base 10 :

```
def somme_chiffres(n):
    n = abs(n) # je n'aime pas les négatifs !
    s = 0
    while n > 0:
        s = s + (n % 10)
        n = n // 10
    return s
```

Cette fois-ci, on prend simplement pour variant `n`, qui est positif d'après la première ligne et dont la décroissance est garantie puisqu'on procède à une division euclidienne par 10 d'un nombre strictement positif (par hypothèse) en fin de boucle.

Les variants peuvent dépendre de variables qui ne sont pas des entiers, comme des réels dans la preuve de terminaison de la recherche de zéro d'une fonction, issue d'un TP du premier semestre, ou des séquences, mais auquel cas le variant concerne souvent leur taille.

Dans le cas de fonctions récursives, le terme de variant n'est pas utilisé en principe, mais la preuve s'organise de la même façon : on cherche une expression qui dépend des variables du programme et qui s'évalue en un entier décroissant strictement à chaque appel récursif **imbriqué**.

Ainsi, la fonction suivante termine, car chaque appel pour le paramètre strictement positif `n` déclenche deux appels sur `n-1`, et ce même si un appel sur `n-1` est suivi d'un autre.

```
def stupide(n):
    if n == 0:
        return 1
    a = stupide(n-1)
    b = stupide(n-1)
    return a + b
```

1.3.2 Preuves de correction

Une fois qu'on a prouvé que le programme terminait, et même dans les cas où il ne termine pas, en fait, il faut encore établir qu'il fait ce pour quoi on l'a écrit. La *preuve de correction partielle* revient à établir que le programme respecte sa spécification lorsqu'il termine, et la *preuve de correction totale* ajoute la terminaison dans tous les cas.

Tester sur plusieurs entrées, notamment des cas limites, est une solution assez efficace en pratique, mais très peu rigoureuse du point de vue scientifique. On préfèrera écrire une preuve formelle du programme, ce qui sera relativement facile dans les cas étudiés dans ce cours, mais qui peut être très contraignant dans la vie réelle. Pour aller plus loin, une troisième approche existe : la construction de modèles, donnant lieu à une branche de l'informatique appelée *model-checking*.

Comme pour la terminaison, un programme sans boucle est relativement aisé à prouver. Cette fois-ci, en revanche, les boucles inconditionnelles devront être étudiées, et on les traitera comme les boucles conditionnelles.

L'outil présenté ici est l'*invariant de boucle*. Il s'agit d'une propriété dont on veut prouver qu'elle est vraie en entrant dans une boucle et qui le demeure à chaque tour jusqu'à la sortie de la boucle. L'invariant devra être pertinent, de sorte qu'en le prouvant on aura prouvé que le programme est correct. La mention des invariants se fait séparément ou, comme les préconditions et postconditions, au niveau de la boucle traitée.

Prenons encore l'exemple de la fonction `somme_chiffres`. L'invariant de boucle choisi est :

La variable `s` contient la somme des `i` derniers chiffres de l'argument de la fonction à la fin du `i`-ième tour.

Cet invariant sera difficile à prouver tel quel, donc nous allons l'étendre en précisant :

... et de plus la variable `n` contient $[\frac{|x|}{10^i}]$ (où `x` est l'argument de la fonction) à la fin du `i`-ième tour (... existant si `n` ne valait pas encore `0` au tour précédent, pour être totalement rigoureux).

Comme ce sera souvent le cas, il s'agit de faire une récurrence.

On suppose l'argument de la fonction non nul, pour que la boucle soit parcourue au moins une fois.

- À la fin du 0^e tour, c'est-à-dire avant d'entrée dans la boucle, la variable `s` contient bien `0` et la variable `n` contient bien le nombre en entrée.
- Supposons l'invariant vrai à la fin du `k`-ième tour dans la boucle. Alors à la fin du tour suivant on divise `n` par `10`, ce qui fait que la deuxième partie de l'invariant est héréditaire, et de plus on a entre temps ajouté à `s` le chiffre des unités de `n`, qui est le `(k+1)`-ième chiffre de l'argument en partant de la fin, d'après la deuxième partie de l'invariant et la définition du `i`-ième chiffre d'un nombre écrit dans n'importe quelle base. Par conséquent, la variable `s` contient bien la somme des `k+1` derniers chiffres de l'argument de la fonction à la fin du `(k+1)`-ième tour.

— En sortie de la boucle, `n` vaut `0` et donc le nombre de tours est le nombre de chiffres de l'argument de la fonction, qui est un de plus que la partie entière du logarithme décimal de ce nombre. Ainsi, `s`, qui est aussi la valeur retournée, vaut bien la somme des chiffres de l'argument, une fois la boucle terminée, ce qui prouve la correction du programme.

Enfin, la preuve de correction de fonctions récursives revient classiquement à établir par une récurrence pertinente que la fonction fait bien ce qu'on attend d'elle, cette récurrence s'écrivant presque en paraphrasant le code. C'est normal, car les fonctions récursives sont connues pour leur simplicité d'écriture, qui n'est parfois qu'une transcription du principe mathématique à simuler.

1.3.3 Complexité

Il est temps de discriminer les algorithmes en fonction de leur efficacité, c'est-à-dire leur coût en opérations élémentaires et l'espace mémoire qu'ils utilisent. Nous introduisons pour cela ici la notion de *complexité*, respectivement en temps et en espace.

En fait, dans la mesure où les ordinateurs ont une capacité de plus en plus grande, la complexité en espace est un critère moins prépondérant que la complexité en temps. Les problèmes viennent surtout du temps exponentiel, l'espace restant souvent raisonnable.

Prenons un exemple pratique avec la méthode de Horner pour déterminer l'image d'un réel par une fonction polynomiale.

En écrivant un programme qui calcule $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_0$, le nombre d'additions est de n et le nombre de multiplications est de $n + (n-1) + \cdots + 0$, soit $\frac{n(n+1)}{2}$.

La complexité en espace est la taille de la réponse finale, ou plus précisément la plus grande taille d'une valeur intermédiaire.

On peut améliorer un tel programme en se rendant compte que la valeur de x^i est recalculée pour tous les monômes de degré supérieur. En stockant les valeurs de tous les x^i, de sorte de ne les calculer qu'une fois par $n-1$ multiplications, on fait alors passer le nombre de multiplications à $2n-1$. Cette amélioration nette de la complexité en temps nécessite malheureusement un coût en espace supplémentaire pour le stockage des valeurs des x^i.

La méthode de Horner consiste à se rendre compte que

$$a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 = \underbrace{(\ldots((}_{n-1\ \text{'('}} a_n x + a_{n-1})x + a_{n-2})\ldots)x + a_1)x + a_0,$$

et donc à réaliser seulement n multiplications et n additions, sans utiliser d'espace mémoire supplémentaire.

L'exemple ci-dessus permet de voir que la complexité se fait en fonction des paramètres du problème (ici n, mais aussi les coefficients du polynôme qui détermineront la taille des nombres stockés, qui influe sur le temps demandé pour faire les opérations arithmétiques), et elle soulève la question de l'unité exacte employée : on ne peut pas parler de temps alors même que les performances de l'ordinateur peuvent causer des écarts de temps bien plus importants entre deux programmes.

La première définition mentionnait les opérations élémentaires, or une multiplication est toujours plus coûteuse qu'une addition pour un processeur. Ainsi, il s'agit de retenir un critère pertinent et adapté au problème.

Pour de nombreux algorithmes, notamment ceux qui sont vus en option et les tris, on distingue la complexité dans le pire des cas et la complexité dans le meilleur des cas (la complexité en moyenne ou le coût amorti ne figurant pas dans le programme).

Cette nuance s'illustre par un programme naïf déterminant si un entier n est premier en testant la divisibilité par tous les entiers entre 2 et $\sqrt{|n|}$, avec une réponse immédiate pour des nombres proches de 0 (discutable).

Dans le meilleur des cas, notre nombre est pair (et $\neq \pm 2$). Alors la réponse est `False` dès le premier test de divisibilité.

Dans le pire des cas, notre nombre est premier, ce qui nécessite $\sqrt{|n|}-1$ tests de divisibilité (critère retenu).

On dira alors que la complexité du programme est dans le meilleur des cas un test de divisibilité et dans le pire des cas $\sqrt{|n|}-1$ tests de divisibilité, ce qui correspond conventionnellement à un programme fonctionnant en temps exponentiel, car le calcul de complexité considère la taille de n, donc le nombre de bits utilisés dans sa représentation, qui est $\lceil \log_2(n+1) \rceil$.

Remarque : Certains informaticiens considèrent que les nombres sont représentés en « unaire », donc par autant de symboles `0` que leur valeur, ce qui donne dans le cas présent la complexité attendue en racine de n.

Le calcul de la complexité en temps d'un algorithme se fait par une analyse de sa structure, ainsi :

- Une instruction de base, par exemple un calcul arithmétique ou logique, une affectation, etc. aura un coût négligé ou retenu comme une unité (comme dit précédemment).
- Le coût de plusieurs instructions qui se suivent, notamment au sein d'un bloc, est la somme des coûts des instructions. Cela vaut bien entendu si les instructions qui se suivent sont eux-mêmes des blocs.
- Le coût d'un test conditionnel est au plus le maximum des coûts des blocs qui le composent, auquel on ajoute au plus le coût d'évaluation de tous les booléens dans la portée du `if` et des `elif` éventuels.
- Le coût d'une boucle inconditionnelle est la somme des coûts des instructions du bloc pour chaque tour dans la boucle. Dans les nombreux cas où les coûts sont tous identiques, on peut procéder à une multiplication du coût commun par le nombre de tours.
- Le coût d'une boucle conditionnelle est également la somme des coûts des instructions du bloc pour chaque tour dans la boucle, plus le coût de chaque vérification de la condition. Malheureusement, cette fois-ci, le nombre de tours dans la boucle n'est pas forcément connu. En règle générale, on cherche juste une bonne majoration, ceci étant.
- Dans le cas de fonctions récursives, on écrit une relation de récurrence entre les coûts pour différentes valeurs de l'argument, et on résout mathématiquement la suite récurrente ainsi obtenue. Pour simplifier, de nombreux cas peuvent être rassemblés, avec la solution déjà connue.

Lorsque des boucles sont imbriquées, il s'agit de faire une somme double, qui se limite dans les meilleurs cas à une double multiplication.

Par exemple, le programme suivant a un coût de $mn + 1$ affectations, mn impressions et $2mn$ multiplications :

```
s = 1
for i in range(n):
    for j in range(m):
        print(42*s)
        s *= -1
```

Dans le programme précédent, la rigueur imposait d'ajouter 1 au coût en raison de la première affectation, mais il est évident que ceci est négligeable quand m et n sont assez grands. En pratique, la complexité ne se donne presque jamais exactement mais toujours de manière asymptotique, c'est-à-dire en négligeant tout ce qui peut être négligé afin de donner un ordre de grandeur concis.

Pour ce faire, on utilise les notations de Landau : soient f, g des fonctions, de $\mathbb{N}$ dans $\mathbb{N}$ pour nous. On dit que :

- $f = o_{+\infty}(g)$ si la limite en $+\infty$ de $\frac{f}{g}$ est 0.
- $f = \mathcal{O}_{+\infty}(g)$ si $\frac{f}{g}$ est bornée au voisinage de $+\infty$.
- $f = \Omega_{+\infty}(g)$ si $g = \mathcal{O}_{+\infty}(f)$ (cette notation est utilisée plus rarement).
- $f = \Theta_{+\infty}(g)$ si $f = \mathcal{O}_{+\infty}(g)$ et $g = \mathcal{O}_{+\infty}(f)$.
- $f \sim_{+\infty} g$ si la limite en $+\infty$ de $\frac{f}{g}$ est 1, donc si $g - f = o_{+\infty}(g)$.

On s'autorise à écrire $f = \mathcal{O}(g)$ pour aller plus vite : on ne signale pas comme en mathématiques que la comparaison se fait en l'infini. En fait, il faudrait même en toute rigueur écrire $f \in \mathcal{O}(g)$.

Ainsi, on dira que la complexité en temps de notre programme est un $\mathcal{O}(mn)$. L'inconvénient de notre notation est que pour nous, $10^9 n = \mathcal{O}(n)$, ce qui n'est pas pertinent au niveau d'un ordinateur. Toutefois, de telles extrémités n'interviennent pas si fréquemment que cela.

Les algorithmes étudiés ici auront généralement une complexité en temps qui est un $\Theta(1)$ (temps constant), un $\Theta(\log n)$ (temps logarithmique), un $\Theta(n)$ (temps linéaire), un $\Theta(n^2)$ (temps quadratique), un $\Theta(n^3)$ (temps cubique) ou un $\Theta(a^n)$ (temps exponentiel, pour un $a > 1$). Les algorithmes de tri efficaces du TP 8 ont une complexité qui est un $\Theta(n \log n)$.

Remarque : Dans le cas du temps logarithmique, la base du logarithme n'importe pas, vu que le rapport entre deux logarithmes se retrouvera dans la constante multiplicative.

Signalons pour finir qu'en théorie de la complexité, l'imprécision va encore plus loin : on se contente de dire par exemple qu'un algorithme est en temps polynomial, sans déterminer a priori pour quel $k \in \mathbb{N} \setminus \{0\}$ la complexité en temps est un $\mathcal{O}(n^k)$ sans être un $\mathcal{O}(n^{k-1})$, et on cherche parfois même des k non entiers.

Quelques rapides informations sur la complexité en espace : elle se calcule à peu près comme la complexité en temps, mais il faut tenir compte de la réutilisation ou non de la mémoire au fur et à mesure de l'exécution. En quelque sorte, la complexité en espace est la taille maximale de la mémoire utilisée à tout moment de l'exécution.

Les deux programmes suivants, tous deux de complexité linéaire en temps en nombre d'instructions élémentaires, et retournant la même chose, auront ainsi des complexités en espace respectives de $\mathcal{O}(\log n)$ et $\mathcal{O}(n \log n)$ bits, en raison de la taille nécessaire pour écrire les entiers en binaire :

```
def somme1(n):
    s = 0
    for i in range(n):
        s += i
    return s

def somme2(n):
    l = list(range(n)) # !?
    s = 0
    for x in l:
        s += x
    return s
```

Attention : La complexité d'un programme n'est pas la complexité du problème auquel il a répondu, qui correspond à la complexité du meilleur programme imaginable, exprimée asymptotiquement en pratique.

Terminons par les complexités les plus classiques dans le cas de fonctions récursives vues en classes préparatoires, à partir de la relation de récurrence déterminée en analysant le programme, en notant c_n le coût pour une valeur n d'une expression dépendant des arguments, en pratique souvent l'un d'entre eux.

- $c_n = c_{n-1} + \mathcal{O}(1) : c_n = \mathcal{O}(n)$.
- $c_n = ac_{n-1} + \mathcal{O}(1)$, $a > 1 : c_n = \mathcal{O}(a^n)$.
- $c_n = c_{n-1} + \mathcal{O}(n) : c_n = \mathcal{O}(n^2)$.
- $c_n = nc_{n-1} + \mathcal{O}(1) : c_n = \mathcal{O}(n!)$.
- $c_n = c_{\frac{n}{2}} + \mathcal{O}(1) : c_n = \mathcal{O}(\log n)$.
- $c_n = c_{\frac{n}{2}} + \mathcal{O}(n) : c_n = \mathcal{O}(n)$.
- $c_n = 2c_{\frac{n}{2}} + \mathcal{O}(1) : c_n = \mathcal{O}(n)$.
- $c_n = 2c_{\frac{n}{2}} + \mathcal{O}(n) : c_n = \mathcal{O}(n \log n)$.
- $c_n = ac_{\frac{n}{b}}$, a et b étant des entiers strictement positifs : $c_n = \Theta(n^{\log_b a})$.

Dans toutes ces formules, apporter une précision en remplaçant $\mathcal{O}$ par Θ dans la relation de récurrence fait que le Θ se maintient dans la complexité qui en résulte.

1.4 Tests de programmes

Pour de petites fonctions, faciles à écrire mais sensibles aux fautes de frappe, ou plus généralement dans les cas où la rédaction d'une preuve ne s'envisage pas, l'absence de problèmes évidents est confortée, après la relecture usuelle, par les tests au niveau de chaque fragment important d'un fichier. On parle alors de *tests unitaires*.

En pratique, si un algorithme a été prouvé puis réalisé dans un langage, la transcription peut avoir donné naissance à des erreurs que la preuve ne peut pas permettre de détecter à ce stade, donc les deux méthodes sont complémentaires.

Il ne faut pas négliger les tests. Même s'ils ne sont plus visibles dans les corrections publiées, des exemples d'exécutions de toutes les fonctions dans les corrections des TP ont été effectués et ont donné lieu dans certains cas à du débogage. Il faut savoir que dans les métiers associés à la programmation, une grande part du budget est allouée aux tests des logiciels développés.

Dans cette section, nous allons parler des tests les plus classiques, d'un module pour automatiser les tests, ainsi que des assertions, seules exceptions au programme.

1.4.1 Jeux de tests

Pour tester une fonction, le plus simple est de l'appeler pour diverses valeurs possibles de ses arguments. Autant cette méthode est intuitive et normalement déjà utilisée depuis bien plus tôt dans l'année, autant quelques suggestions d'optimisations peuvent être apportées.

Tout d'abord, il n'est bien entendu pas utile que les tests effectués soient exhaustifs, mais ceux-ci doivent cependant être ciblés pour ne pas manquer parmi les cas possibles celui qui, en raison de propriétés particulières des arguments, fera échouer la fonction. Cela correspond à la notion de *partitionnement des données d'entrée*.

Dans le cas de l'insertion dichotomique vue plus tôt dans ce chapitre, cela revient à tester la fonction :

— Dans un cas où l'élément à insérer n'est pas dans la liste.
— Dans un cas où l'élément à insérer est dans la liste (une fois puis plusieurs).
— Dans un cas où l'élément à insérer est le premier de la liste.
— Dans un cas où l'élément à insérer est le dernier de la liste.
— Dans un cas où la liste est vide.

Pour engendrer plus de tests lorsque les cas pathologiques ne s'identifient pas aussi facilement, il ne faut pas hésiter à créer une fonction qui engendre les arguments de manière aléatoire. Il faudra alors imprimer les valeurs ainsi obtenues et comparer avec la résolution à la main.

Enfin, comme évoqué dans le TP 4, Python dispose de modules permettant de tester automatiquement toutes les fonctions, pour peu que des cas de tests aient été fournis dans la documentation de code dans le cas de `doctest`. À force d'utiliser ces méthodes, elles deviennent plus rapides à appliquer que l'écriture et l'exécution manuelles des tests.

1.4.2 Assertions

Une *assertion*, comme pourrait l'expliquer un professeur de français, est une affirmation que l'on fait. Qu'elle soit vraie ou non, elle est censée être vraie pour celui qui l'affirme.

En informatique, il s'agit donc d'une ligne de code introduite par le mot-clé `assert` suivi d'une expression de type booléen ou qui sera convertie en booléen, mais avec les mêmes considérations esthétiques que pour l'utilisation de non-booléens avec `if` ou `while`. À la suite de cela, il est possible mais optionnel d'ajouter une virgule puis une chaîne de caractères (ceci n'est pas à savoir, mais quitte à utiliser des assertions, autant leur donner toute leur puissance).

La sémantique est la suivante : l'expression est évaluée au moment où la ligne est lue, si elle est vraie on passe à la suite, sinon une erreur d'assertion est déclenchée, et si la chaîne de caractère a été ajouée, c'est elle qui est imprimée en accompagnement de l'annonce de l'erreur.

On utilise habituellement des assertions en début de programme, et plus généralement là où on rédigerait une précondition, l'idée étant de vérifier que ces préconditions sont satisfaites, ou que les arguments d'une fonction respectent des contraintes associés à la fonction, en guise de protection contre les entrées douteuses de l'utilisateur. Les assertions interviendront donc désormais là où la documentation de code disait « on ne vérifiera pas que ... ».

En l'absence d'assertions, il n'est pas rare que le programme déclenche de lui-même une autre erreur. Celle-ci aura moins de chances d'être facile à lire et son origine s'en déduira moins bien.

TD 1 : Terminaison, correction et complexité

Pour les fonctions successives, prouver la terminaison et calculer la complexité en fonction des arguments, déterminer la spécification précise, insérer des préconditions et postconditions et prouver la correction. À la fin, créer une documentation de code contenant des tests pour chacune des fonctions et annoter la ligne de définition avec la signature.

```
from random import randint
from copy import deepcopy

def FisherYates(l):
    n = len(l)
    ll = deepcopy(l)
    for i in range(n-1, 0, -1):
        j = randint(0, i)
        ll[j], ll[i] = ll[i], ll[j]
    return ll

# ---------------------------------------

def r(s):
    rep = ""
    for i in range(len(s)):
        car = s[i]
        rep = car + rep
    return rep

# ---------------------------------------

def c(s, m):
    rep = 0
    for i in range(len(s)):
        if s[i:i+len(m)] == m:
            rep += 1
    return rep

# ---------------------------------------
```

```
def verif1(l):
    i = 0
    while i < len(l)-1 and l[i] <= l[i+1]:
        i += 1
    return i == len(l)-1

# ---------------------------------------

def verif2(l):
    n = len(l)
    if n <= 2:
        return True
    i = 2
    flag = True
    while i < len(l):
        if l[i] != l[0] + i*(l[1]-l[0]):
            flag = False
            i = len(l)
        i += 1
    return flag

# ---------------------------------------

def verif3(l, ll):
    for x in l:
        if x not in ll:
            return False
    for x in ll:
        if x not in l:
            return False
    return True
```

Chapitre 2

Représentation des nombres

2.0 Introduction aux bases numériques

Cette section, qui peut être un rappel de primaire pour certains, ne fait plus partie du programme mais présente des méthodes de calcul à maîtriser en vue de l'utilisation du binaire.

Définition

Soit b un entier naturel ≥ 2. On considère un ensemble C de b caractères, usuellement des chiffres à partir de 0, en complétant avec des lettres dans l'ordre alphabétique, donnés dans l'ordre croissant.
On représente un entier naturel en base b en écrivant des caractères de l'ensemble C en séquence, en précisant la valeur de b pour lever toute ambiguïté.
La notation proposée dans ce cours, qui n'est pas uniformisée, est $\overline{a_{n-1}a_{n-2}\dots a_1a_0}^b$, pour représenter le nombre $\sum_{i=0}^{n} a_i b^i$. Dans le nombre en question, le caractère dit de poids fort est le plus à gauche et le caractère dit de poids faible est le plus à droite.

Les bases usuelles sont 10 (base décimale, utilisée de façon naturelle actuellement), 2 (le binaire, fondement de l'informatique) et 16 (les nombres hexadécimaux, utilisés en informatique pour éviter d'avoir trop de chiffres).

On trouve également les bases 8 et 12 de manière cependant moins prononcée. Dans le reste de ce chapitre, ce sera du binaire en majorité écrasante et de l'hexadécimal quand ce sera vraiment approprié.

Toutes les opérations arithmétiques apprises au primaire se font de manière similaire dans n'importe quelle base. Par exemple, en base 8, $\overline{35}^8 + \overline{56}^8 = \overline{113}^8$ car $5 + 6$, qui correspond au nombre 11, s'écrit $\overline{13}^8$. On note donc 3 et on retient 1, de même qu'on a une retenue pour les « huitaines », qui devient la « soixante-quatraine ».

La représentation en base b s'étend aux entiers relatifs en précisant le signe.

Proposition

Soit b un entier naturel ≥ 2. On peut représenter un nombre rationnel en base b de manière exacte, c'est-à-dire avec un nombre fini de caractères, si et seulement si le nombre en question est le quotient d'un entier relatif par une puissance de b. Une autre formulation : le nombre en question a pour fraction la plus simplifiée $\frac{p}{q}$, où tous les diviseurs premiers de q sont des diviseurs de b. La représentation comporte alors éventuellement une virgule, et on écrit $\overline{a_{n-1}a_{n-2}\dots a_1a_0,a_{-1}a_{-2}\dots a_{-m}}^b = \sum_{i=-m}^{n-1} a_i b^i$.

Remarque : En base 10, on retrouve la notion de nombres décimaux. Bien entendu, quelle que soit la valeur de b, un nombre irrationnel aura toujours un nombre infini de caractères après la virgule dans sa représentation en base b.

Exemple : Le nombre $\frac{13}{3}$ s'écrit de manière exacte dans toute base multiple de 3.

Proposition

[Pour la culture] Quelle que soit la valeur de b, tout nombre rationnel r a une écriture en base b ultimement périodique, c'est-à-dire qu'à partir d'un certain rang fini, un même motif se répète.

Cette proposition se prouve simplement à l'aide du petit théorème de Fermat quand b est premier (sinon, c'est de l'arithmétique avancée). On peut l'illustrer en considérant les fractions $\frac{1}{p}$ en base b. Si p divise b, alors l'écriture de $\frac{1}{p}$ est exacte : $\frac{1}{p} = \overline{0,a}^b$, où a est le caractère correspondant à l'entier $\frac{b}{p}$, la période étant alors de 1 et le motif étant 0. Sinon, le petit théorème de Fermat dit que b^{p-1} est congru à 1 modulo p, donc diviser 1 par p laisse entrevoir une répétition après $p-1$ étapes. La période est donc $p-1$ ou un de ses diviseurs.

2.1 Représentation des entiers

2.1.1 Entiers naturels

Les informations dans les ordinateurs peuvent être vues comme des signaux électriques, et chaque unité de mémoire peut alors être dans deux états.

Il faut garder à l'esprit qu'aucun nombre n'est stocké en tant que tel, mais représenté à l'aide d'unités de mémoire dont l'interprétation dépend du contexte.

Ainsi, un ordinateur travaille avec des nombres constitués de 0 et de 1, donc en binaire, et on appelle « *bits* » (pour "binary digits", soit chiffres binaires) les caractères de base du binaire quand ils sont utilisés en informatique.

On regroupe éventuellement les bits par paquets de 4 (pour former un caractère hexadécimal) ou de 8 (des *octets*) dans un souci de concision.

Important : En pratique, la limite de la mémoire d'un ordinateur empêche évidemment d'écrire n'importe quel entier naturel, et on parlera d'entiers sur n bits, souvent 32 ou 64.

Par exemple, les dates sont usuellement stockées dans l'ordinateur à l'aide de ce qu'on appelle "timestamp". Cette horloge compte les secondes écoulées depuis le premier janvier 1970 à minuit UTC (pour les systèmes Unix), ce qui causera des problèmes le 19 janvier 2038 pour les systèmes sur 32 bits (qui ont de toute façon déjà presque disparu actuellement). Ceci nous amène à parler de *dépassement arithmétique*.

Les opérations arithmétiques, en binaire comme en décimal et dans toutes les bases, font occasionnellement apparaître des retenues. Que se passe-t-il quand la retenue apparaît ou se propage au-delà du dernier caractère disponible ? Elle est tout simplement perdue. Ainsi, dans la représentation des entiers naturels sur 32 bits, $2^{31} + 2^{31}$ donne 0. C'est ici que réside le dépassement arithmétique.

L'intérêt de fixer à un certain nombre de bits l'écriture de tous les nombres est la facilitation de l'allocation de la mémoire, par la certitude que si une valeur commence sur une certaine case mémoire, elle occupe une plage qui s'en déduit.

En Python, le type correspondant à cette représentation n'est pas disponible dans la bibliothèque standard mais dans `numpy`. On appelle cela les entiers non signés, soit en anglais *unsigned int*, avec les types `uint8`, `uint32`, etc. Un premier contact avec le type `uint8` a eu lieu au TP 7, car les intensités dans les trois couleurs d'un pixel peuvent être encodées par un entier entre 0 et 255, ce qui forme un choix alternatif par rapport à l'utilisation d'un réel entre 0 et 1.

2.1.2 Entiers relatifs

Naïvement, on peut penser à réserver un bit dans la représentation d'un entier naturel pour préciser le signe. Cette méthode a l'avantage de la simplicité, mais plusieurs inconvénients, dont l'existence de deux versions de zéro, font qu'en pratique elle est laissée de côté.

Une autre façon de faire pourrait être de considérer que puisqu'on peut écrire les nombres de 0 à $2^n - 1$ sur n bits, on soustrait 2^{n-1} à tous les nombres représentés. L'inconvénient majeur réside cette fois dans les opérations, qui n'ont rien d'évident.

On emploie donc pour représenter un entier relatif la notation dite en *complément à deux*. Il s'agit de distinguer les cas suivant que l'entier soit positif (auquel cas le premier bit est à zéro) ou strictement négatif (auquel cas le premier bit est à 1). Dans le premier cas, on écrit simplement l'entier que l'on veut représenter sur les $n - 1$ derniers bits, et dans le deuxième cas on écrit en fait l'entier plus 2^n sur les n bits.

La notation en complément à deux permet alors d'écrire sur n bits les entiers entre -2^{n-1} et $2^{n-1} - 1$. Pour n valant 32, cela correspond à la zone de $-2\,147\,483\,648$ à $2\,147\,483\,647$, et pour n valant 64, de $-9\,223\,372\,036\,854\,775\,808$ à $9\,223\,372\,036\,854\,775\,807$. Cela suffit pour les nombres utilisés dans la vie quotidienne, et pour de plus grands nombres Python fournit une solution qui ne nécessite pas de se poser les problèmes rencontrés en C.

Exercice

Il est utile de connaître la représentation des nombres les plus classiques. Elle est évidente pour les nombres positifs, retenir celles de -2^{n-1}, de -1 et d'autres opposés de puissances de 2 est incontournable. Commencer par déduire ces valeurs à partir de la description de l'encodage énoncée ci-avant.

Propriété : Dans la notation en complément à deux, pour calculer l'opposé d'un entier x, on remplace tous les 0 par des 1 et vice-versa, puis on ajoute 1 à la représentation. Ceci ne marche évidemment pas pour -2^{n-1}, qui s'écrit avec un 1 au début et des 0 partout ailleurs, car son opposé n'est pas représentable sur n bits avec cette notation. Par conséquent, l'opération proposée redonnerait nécessairement le même nombre, comme dans le cas de 0, évidemment.

Exercice

Puisque l'opposé est une opération involutive, c'est-à-dire que la répéter fait revenir au nombre de départ, on se rend compte que retirer 1 puis remplacer tous les 0 par des 1 et vice-versa donne aussi l'opposé. Prouver ceci, ainsi que la propriété ci-dessus.

Comme pour les entiers naturels, le problème du dépassement arithmétique demeure. Sur n bits, ajouter 1 à $2^{n-1}-1$ donne donc -2^{n-1} et non 2^{n-1}. Cela donne en particulier des calculs exacts modulo 2^n.

2.2 Entiers longs et entiers multi-précision de Python

Pour pallier le problème de dépassement arithmétique, lorsque des grands nombres sont requis, et pour éviter de faire des opérations sur des nombres comprenant trop de chiffres, une méthode revient à couper les nombres en paquets de $\lfloor \frac{n}{2} \rfloor - 1$ bits quand les entiers manipulés par le processeur sont sur n bits.

Ceci permet de ne pas avoir de dépassement arithmétique pour les multiplications, car deux nombres de $\lfloor \frac{n}{2} \rfloor - 1$ bits multipliés ensemble donnent encore un nombre de moins de n bits, en gardant à l'esprit qu'un bit doit être réservé au signe, d'où le -1 après la division.

Pour $n = 32$, les nombres sont donc découpés en paquets de 15, dont le premier paquet donne deux informations : le signe du nombre (premier bit) et le nombre de paquets de 15 bits (sur 15 bits). Les paquets de 15 sont stockés dans des tableaux de taille 16 (en répercutant les retenues aux bons endroits pour que le premier bit de chaque tableau soit à 0 à la fin de chaque calcul).

Le nombre le plus grand qui peut ainsi être représenté pour $n = 32$ est alors $2^{15\times(2^{15}-1)} - 1$, et son écriture tient sur environ 1 Mb. On signalera qu'il vaut mieux ne pas tester à faire afficher le nombre le plus grand qui peut être représenté lorsque $n = 64$.

Le type appelé “long” utilisé dans de nombreux langages a été fusionné avec le type “int" en Python, ce qui fait que nous échappons aux dépassements et la seule limite est la mémoire disponible.

Il est intéressant de souligner que, lorsque l'entier à manipuler est de taille supérieure à celle prévue par le processeur, les opérations de découpage et de recombinaison font intervenir pour les opérations arithmétiques des algorithmes également rencontrés pour les multiplications de polynômes de grand degré (voir l'algorithme de Karatsuba à ce sujet) et de matrices carrées de grande taille (voir l'algorhtme de Strassen, la plus simple des optimisations historiques).

En particulier, la complexité d'une multiplication ne peut plus décemment être considérée comme constante, mais évolue plutôt linéairement en la taille de l'entier. En tout cas, c'est ce que dirait le chronomètre en faisant tourner l'algorithme. Le calcul de complexité éludera ce souci en prenant par exemple la multiplication d'entiers comme unité de base.

2.3 Représentation des réels

Le nom donné en informatique à la représentation des nombres non entiers est la *virgule flottante* (le type correspondant est "float", appelés « flottants » en français).

Rappelons qu'aucun nombre irrationnel ne peut être représenté de manière exacte avec des caractères de n'importe quelle base. D'ailleurs, tous les nombres représentables de manière exacte en base 2 sont en particulier décimaux, tandis que dans l'autre sens, le nombre $0,2$ par exemple, aussi innocent soit-il, a un développement infini en base 2, à savoir $\overline{0,001100110011\ldots}^2$. Des conséquences de l'absence d'écriture exacte finie de certains nombres décimaux en base 2 sont présentées dans les TP 10 et 12.

Bien plus, même un nombre qui s'exprime de manière exacte en binaire n'est pas forcément représentable de manière informatique, puisque la mémoire est finie. Tout comme on peut représenter 2^{64} entiers sur 64 bits, cette restriction demeure évidemment pour les flottants.

Ainsi, on ne peut pas aller vers l'infiniment proche de zéro, et le plus grand nombre représentable sur 64 bits est limité à quelques centaines de chiffres, ce qui est néanmoins suffisant pour modéliser les grandeurs physiques. La troisième limitation est la précision : entre deux nombres représentables, il y a un écart correspondant environ à un milliardième de milliardième de la valeur de ces nombres.

La représentation des nombres réels, qui est habituellement approximative, se fonde sur l'écriture dite scientifique en binaire, normalement connue pour les nombres décimaux Rappelons ici si besoin que l'écriture scientifique revient à écrire un nombre non nul, éventuellement en tant qu'arrondi, sous la forme $x \times 10^k$, où $k \in \mathbb{Z}$ et $1 \leq x < 10$, et l'écriture est unique.

Grâce à la représentation à virgule flottante, les nombres très grands ou très petits peuvent s'écrire sans une surcharge de caractères peu représentatifs au niveau de la virgule. De toute façon, comme annoncé ci-avant, les limites de la mémoire imposent de procéder rapidement à un arrondi.

Selon la norme IEEE754, avec 64 bits, un nombre en virgule flottante est alors un produit $(-1)^s \times m \times 2^n$, où s est bit de signe, m est la *mantisse* $\overline{1,b_1b_2\ldots b_{52}}^2$ (puisque le 1 est systématique, il n'entre pas dans la représentation qui est donc sur 52 bits) et n est un entier relatif entre -1022 et 1023 écrit sur 11 bits et représenté comme $n + 1023$. Avec 32 bits, la taille de la mantisse passe à 23 bits et l'exposant est sur 8 bits.

La disposition des bits est la suivante : d'abord s, puis n, puis m.

On notera que pour comparer deux nombres écrits en virgule flottante, on regarde d'abord le signe, puis à même signe on regarde l'exposant, puis à même exposant on regarde la mantisse.

Les valeurs non utilisées pour les exposants correspondent aux mots de 11 bits 00000000000 et 11111111111. On les réserve pour des valeurs exceptionnelles, qui ne sont pas à connaître. On ne citera pour la culture que le zéro, qui existe en version positive et en version négative (suivant le premier bit), dont les bits de l'exposant et de la mantisse sont tous nuls, et les infinis, qui s'obtiennent quand tous les bits de l'exposant sont à 1 et ceux de la mantisse à 0.

2.4 Conséquences

Il est évident que sur 64 bits, même en ne considérant aucune valeur exceptionnelle, on ne pourrait encoder que 2^{64} valeurs, et les nombres supérieurs à 2^{1024} en valeur absolue (une valeur qui reste certes déraisonnable en sciences) ainsi que ceux qui sont plus proches de zéro que 2^{-1022} n'auraient pas de représentation possibles.

En pratique, on traite les dépassements arithmétiques (et les « soupassements arithmétiques », le deuxième cas) de différentes façons, précisément à l'aide des valeurs exceptionnelles (d'où l'intérêt d'avoir un zéro positif et un zéro négatif, au passage). Le déclenchement d'erreurs à l'aide de valeurs exceptionnelles non évoquées est une façon de faire, peut-être plus efficace que l'utilisation systématique de zéros ou d'infinis seulement.

Dans l'intervalle où les réels sont représentables, la question de l'arrondi se traite de la même façon que pour les nombres usuels : si le premier bit dépassant la taille de la mantisse devrait être un 0, on arrondit le dernier bit par défaut, sinon par excès avec propagation éventuelle de retenue, à l'exception notable du cas où seul un bit à 1 dépasse la taille de la mantisse, ce qui est analogue à un arrondi d'un demi-entier à l'entier le plus proche.

Le fait d'arrondir des valeurs, quand bien même est-il nécessaire, favorise une perte d'informations lorsque des opérations sont effectuées entre des réels de valeurs très éloignées. Par conséquent, en tenant compte des priorités opératoires (et quand la priorité est la même, en effectuant les opérations de gauche à droite), deux expressions donnant dans la réalité le même résultat donneront parfois sur l'ordinateur des résultats différents.

Ce qu'il faut alors garder à l'esprit, c'est que les tests d'égalité entre réels sont très peu pertinents et qu'on leur préfèrera une comparaison de type « la différence est suffisamment petite en valeur absolue ».

2.5 L'essentiel

Les deux enseignements de ce chapitre sont les suivants : quand on regarde dans les détails, on tombe toujours sur le binaire, et la représentation de l'information doit être aussi efficace que possible, y compris dans la clarté de cette représentation.

Il ne faut alors pas se tromper au niveau de l'interprétation d'une chaîne de bits, ce qui veut aussi dire qu'il ne faut pas confondre complément à deux et virgule flottante.

Une bonne méthode pour ne pas se tromper est simplement de tenter de faire une opération de base sur les représentations : addition de deux entiers, multiplication de deux réels. Si cela semble infernal, c'est peut-être que la représentation est erronée. Il reste le souci de l'encodage de l'exposant selon la norme IEEE754, qui n'est ni intuitif ni particulièrement optimal par rapport à d'autres choix possibles. Dans ce cas, la mémorisation peut s'imposer. De toute manière, la connaissance poussée de cet encodage ne sert qu'à comprendre d'où viennent les comportements imprévus comme présenté en TP.

En tout état de cause, si les confusions demeurent, autant ne retenir que la virgule flottante, car ce sont avec les flottants que la plupart des calculs effectifs sont faits.

Et bien entendu, la moralité qui sera rappelée à chaque occasion : les flottants portent en eux une imprécision qui peut être une cause d'erreurs à tout moment, car on ne pourra jamais encoder un ensemble infini dans un espace fini.

TD 2 : Représentation des nombres

Avant toute chose, il est recommandé de faire les exercices du cours.

Représentation des entiers

Exercice 1 : Représenter en complément à deux sur 16 bits les nombres 36000 et -42.

Exercice 2 : Quel nombre décimal s'écrit sur 8 bits en complément à deux 11010110 ? Et 11000011 ?

Exercice 3 : On considère un nombre dont la représentation en complément à deux commence par 0, comporte un nombre pair de bits et est un palindrome (c'est-à-dire qu'il se lit de la même façon dans les deux sens). Déterminer deux facteurs premiers du nombre en question. A-t-on une équivalence ?

Représentation des réels

Exercice 4 : Convertir en binaire 0,7 en s'arrêtant une fois la période trouvée [1] et exprimer ce nombre en virgule flottante sur 16 bits (pour ne pas écrire trop d'informations), avec 5 bits d'exposant et 10 bits de mantisse.

Exercice 5 : Même exercice avec $\frac{3}{7}$.

Au passage, utiliser la représentation en virgule flottante pour des rationnels introduit une perte d'information en raison des approximations. C'est pour cela qu'en Python, il est possible de représenter aussi les rationnels comme des couples d'entiers (numérateur et dénominateur), ce qui est géré par un module nommé `fractions`.

Exercice 6 : Caractériser les entiers qui ont une représentation exacte en virgule flottante sur 64 bits.

1. Pour rappel, d'après le cours, elle est de taille 1, 2 ou 4. Exercice supplémentaire : retrouver pourquoi.

Chapitre 3

Bases des graphes, plus courts chemins

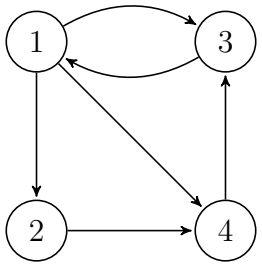
3.1 Définitions, notations et propriétés de base

Définition

Un graphe orienté est la donnée d'un ensemble fini S de sommets et d'un ensemble A d'arcs, qui sont des flèches (d'où l'adjectif « orienté ») reliant deux sommets, de sorte qu'il n'y ait jamais deux arcs qui ont le même sommet de départ et le même sommet d'arrivée. L'ensemble A est en fait un sous-ensemble du produit cartésien $S \times S$. Le graphe est alors décrit comme le couple (S, A).

Remarque : Dans un *graphe non orienté*, il y a des *arêtes* qui n'ont pas de sens précis. Un graphe non orienté pouvant aisément être simulé par un graphe orienté, nous considérons principalement ces derniers et le terme de « *graphe* » sans précision se réfèrera aux graphes orientés.

On représente habituellement un graphe ainsi :



Remarque : Rien n'interdit à un arc d'arriver à son sommet de départ. Il s'agit alors d'une *boucle*. C'est cependant plus rare avec des arêtes.

Le graphe ci-dessus est $(\{1;2;3;4\}, \{(1;2);(1;3);(1;4);(2;4);(3;1);(4;3)\})$.

Un graphe peut aussi être représenté par une *liste d'adjacence*, en donnant S et pour chaque élément s de S l'ensemble des sommets tels qu'il existe un arc de s à ces sommets.

Exemple : Toujours avec l'exemple ci-dessus, la représentation par liste d'adjacence est

$$(\{1;2;3;4\}, \{(1,\{2;3;4\});(2,\{4\});(3,\{1\});(4,\{3\})\}).$$

Pour la représentation par liste d'adjacence, plusieurs implémentations sont possibles. On remarquera notamment qu'il n'est pas nécessaire de renseigner à part la liste des sommets. Il est particulièrement intéressant d'utiliser un dictionnaire indexé par les sommets.

Enfin, un graphe peut être représenté par une *matrice d'adjacence*, qui est une matrice carrée de taille le nombre de sommets, avec un 1 (ou le booléen vrai) dans la cellule à la ligne i et la colonne j s'il existe un arc du i-ième sommet au j-ième sommet (en ordonnant les sommets au préalable), et un 0 (ou le booléen faux) sinon. Ainsi, le nombre de 1 est le nombre d'arcs.

Exemple : La matrice d'adjacence du graphe précédent est

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Définition

Un chemin (ou une chaîne dans un graphe non orienté) dans un graphe est une liste de sommets $s_1 s_2 \dots s_k$ tel que, pour tout $1 \leq i < k$, $(s_i; s_{i+1})$ est un arc.
On appelle s_i le prédécesseur de s_{i+1} et s_{i+1} le successeur de s_i dans le chemin. La longueur du chemin est $k-1$, c'est le nombre d'arcs traversés.

Remarque : Il n'est pas interdit qu'un chemin passe deux fois par le même sommet, mais il existe alors un chemin strictement plus court de mêmes *source* (s_1) et *destination* (s_k).

Exemple : Dans le graphe donné, 1 4 3 1 2 est un chemin de longueur 4.

On constate qu'il est possible de suivre les flèches pour parcourir ce chemin.

Définition

Un circuit (ou un cycle dans un graphe non orienté) dans un graphe est un chemin tel que $s_k = s_1$, c'est-à-dire dont la destination est aussi la source.

Exemple : Dans le graphe donné, 1 4 3 1 est un circuit de longueur 3.

Définition

Une composante connexe d'un graphe non orienté (S, A) est un sous-ensemble S' de S (ou pour certains auteurs le sous-graphe induit) tel que pour tous $s, t \in S'$ il existe une chaîne reliant s à t dans le graphe en ne passant que par des sommets de S'.

Remarque : Certains auteurs ajoutent une condition de maximalité pour cette propriété (sous laquelle aucun ensemble contenant S' ne doit vérifier la propriété ci-avant).

Définition

Un graphe non orienté est dit connexe si l'ensemble de ses sommets forme une composante connexe.

Définition

Une composante fortement connexe d'un graphe orienté (S, A) est un sous-ensemble S' de S (même remarque) tel que pour tout couple $(s, t) \in S' \times S'$ il existe un chemin de s à t dans le graphe en ne passant que par des sommets de S'.

Remarque : On retrouve la variante imposant la maximalité.

Définition

Un graphe orienté est dit fortement connexe si l'ensemble de ses sommets forme une composante fortement connexe.

Remarque : On peut aussi trouver la notion de graphe orienté connexe chez certains auteurs, signifiant en général que le graphe obtenu en ignorant l'orientation est connexe.

Définition

Le degré d'un sommet dans un graphe non orienté est le nombre d'arêtes touchant ce sommet. Le degré entrant (resp. sortant) d'un sommet dans un graphe orienté est le nombre d'arcs de destination (resp. source) ce sommet.

Définition

Le degré d'un graphe est le degré maximal d'un sommet dans un graphe. Cette notion s'emploie plutôt pour des graphes non orientés. Un graphe est dit régulier quand tous les sommets sont de même degré.

Définition

Un chemin hamiltonien est un chemin, nécessairement de longueur $\#S - 1$, passant une et une seule fois par chacun des sommets. Un circuit hamiltonien en est le prolongement par raccord, donc de longueur $\#S$.

Exemple : Dans le graphe donné, 1 2 4 3 est un chemin hamiltonien qui peut se prolonger en le circuit hamiltonien 1 2 4 3 1.

On retrouve le problème de détermination de chemins hamiltoniens dans certains casse-tête de jeux vidéo... Il faut noter que c'est un problème algorithmiquement difficile (NP-complet).

Définition

Un chemin eulérien est un chemin, nécessairement de longueur $\#A$, passant une et une seule fois par chacun des arcs. Un circuit eulérien est un chemin eulérien qui est en plus un circuit.

Exemple : L'intuition d'un circuit eulérien dans un graphe non orienté se retrouve dans les jeux qui consistent à dessiner une figure sans lever le crayon.

Proposition

Un graphe non orienté admet une chaîne eulérienne si, et seulement si, il est connexe et tous ses sommets sont de degré pair, sauf éventuellement deux. Un graphe orienté admet un chemin eulérien si, et seulement si, il est connexe et tous ses sommets sont de degré entrant égal au degré sortant, sauf éventuellement deux à condition que pour l'un la différence soit de 1 et pour l'autre de -1.

Proposition

Soit M^n la puissance usuelle n-ième de la matrice d'adjacence M d'un graphe (S, A). Alors la cellule à la ligne i et la colonne j de M^n contient le nombre de chemins distincts de longueur n du i-ième sommet du graphe au j-ième sommet du graphe.

Remarque : En particulier, on compte le nombre de circuits de longueur n sur la diagonale.

Définition

La fermeture transitive d'un graphe (S, A) est un graphe dont l'ensemble des arcs T est la plus petite relation transitive telle que $A \subseteq T \subseteq S \times S$. L'ensemble T est aussi l'ensemble des couples $(s; t)$ tels qu'il existe dans le graphe un chemin de s à t.

Exemple : La fermeture transitive du graphe donné est le graphe dit complet (chaque couple de sommets est relié par un arc). En effet, il existe un circuit hamiltonien dans le graphe, donc on peut aller à chaque autre sommet depuis chaque sommet.

Définition

La somme booléenne de deux entiers valant 0 ou 1 correspond au « OU » logique et le produit booléen correspond au « ET » logique. Ainsi, la somme booléenne de 1 et 1 sera 1. Les autres résultats correspondent à la somme et au produit classiques.

Remarque : La somme et le produit booléen peuvent être vus comme les lois de composition internes du semi-anneau {Vrai,Faux}.

Définition

Le produit booléen de deux matrices ne contenant que des 0 *et des* 1 *est le produit matriciel dans lequel les additions sont des sommes booléennes.*

Remarque : On peut aussi faire le produit booléen en calculant le produit usuel et en remplaçant toutes les valeurs non nulles par 1.

Une autre méthode, qui s'explique mieux visuellement, consiste à remplir les lignes du produit booléen en scannant les lignes de la matrice de droite correspondant aux colonnes où la matrice de gauche a un 1 dans la ligne étudiée.

Définition

La puissance booléenne n-ième d'une matrice carrée est sa puissance n-ième par le produit booléen.

Proposition

Soit $M^{[n]}$ la puissance booléenne n-ième de la matrice d'adjacence M d'un graphe (S, A). Alors la cellule à la ligne i et la colonne j de $M^{[n]}$ contient un 1 *si, et seulement si, il existe un chemin de longueur n du i-ième sommet du graphe au j-ième sommet du graphe.*

Proposition

Pour un entier $n \geq 1$, soit P la somme booléenne de I_n et de la matrice d'adjacence M d'un graphe (S, A) à n sommets. Alors la puissance booléenne $n-1$-ième de P est égale à la somme booléenne $M \bigoplus M^{[2]} \bigoplus \cdots \bigoplus M^{[n-1]}$, et c'est aussi la matrice d'adjacence de la fermeture transitive du graphe.

Pour s'en convaincre, il suffit de considérer que tous les coefficients binomiaux peuvent être remplacés par 1 dans la formule du binôme, et que I_n reste élément neutre du produit booléen des matrices.

Définition

Un graphe sans circuit est un graphe... dans lequel il n'y a pas de circuit. Dans un graphe sans circuit, le niveau d'un sommet est la longueur du plus long chemin terminant à ce sommet. Ainsi, si un sommet n'a pas de prédécesseur (et au moins un tel sommet existe, puisque le graphe n'a pas de circuit), il aura le niveau 0.

Remarque : On calcule le niveau de chaque sommet de manière récursive. Une fois les sommets de niveau 0 trouvés, on attribue a priori le niveau 1 aux successeurs des sommets de niveau 0, puis le niveau 2 aux successeurs des sommets de niveau 1, même s'ils ont déjà le niveau 1, puis on continue jusqu'à ce que rien ne soit modifié. Le niveau maximal possible est le nombre de sommets moins un.

Le processus termine car il n'y a pas de circuit.

Définition

Le niveau d'un graphe sans circuit est le plus grand niveau d'un de ses sommets.

Remarque : On peut modifier la représentation graphique d'un graphe sans circuit de sorte que les sommets de même niveau soient alignés horizontalement.

Remarque : Un arbre est un graphe sans circuit dont un seul sommet, appelé racine, est de niveau 0, et tel qu'il existe un et un seul chemin, appelé branche, de la racine à chaque autre sommet. Le nombre d'arcs d'un arbre est alors le nombre de sommets moins un.

Plus précisément, dans le cas non orienté, tout graphe connexe ayant une arête de moins que le nombre de sommets peut être vu comme un arbre en choisissant arbitrairement parmi les sommets une racine.

On peut remarquer une correspondance de vocabulaire entre les graphes et les arbres : sommet/nœud, prédécesseur/père, successeur/fils, chemin/branche, niveau d'un sommet/profondeur du sommet, niveau du graphe/hauteur de l'arbre, sommet sans successeur/feuille (et les autres sommets sont des nœuds internes), sommet sans prédécesseurs/racine, degré/arité.

Définition

Un chemin dans un graphe est dit optimal en longueur s'il n'existe pas de chemin de même source, de même destination et de longueur strictement moindre.

Remarque : Un tel chemin n'est pas forcément unique, mais dès qu'il existe un chemin entre un sommet s et un sommet t, il existe un chemin optimal en longueur entre s et t.

Proposition

Tout sous-chemin d'un sous-chemin optimal en longueur est optimal en longueur, mais raccorder deux chemins optimaux en longueur ne donne pas forcément un chemin optimal en longueur.

Remarque : Par convention, on pourra considérer que les chemins optimaux en longueur d'un sommet à lui-même sont les chemins vides, il y a alors forcément unicité. Et un tel chemin n'est pas la fusion de deux chemins optimaux en longueur, vers un autre sommet et dans l'autre sens.

Définition

Un graphe pondéré est un graphe dont les arcs sont étiquetés par un nombre, habituellement entier et potentiellement négatif, appelé le poids.

Il existe plusieurs conventions pour les graphes pondérés. Une possibilité est de donner une fonction de A vers $\mathbb{Z}$, ce qui exclut qu'il y ait deux arcs de même source et même destination avec un poids différent, une autre est de considérer A comme un sous-ensemble de $S \times \mathbb{Z} \times S$. En tout état de cause, les applications des graphes pondérés qui nous intéressent sont ordinairement des problèmes d'optimisation, pour lesquels une seule version d'un arc entre deux sommets sera pertinente. Mais si une ambigüité est possible, il faut préciser le poids de chaque arc quand on écrit un chemin dans un graphe pondéré.

Remarque : Suivant le contexte, le poids est appelé coût, durée, etc. Les graphes pondérés sont le plus généralement orientés, ce qui est cohérent avec les contextes où on les applique.

L'utilisation d'étiquettes sur les arcs d'un graphe, qu'il s'agisse d'entiers ou de données d'un autre type, est répandue en informatique. On citera notamment les automates, une extension directe de la structure de graphe présentée en option en deuxième année.

Par ailleurs, les sommets peuvent également porter une information autre que son numéro, ce qui permet de rassembler certains d'entre eux dans des classes d'équivalence, comme dans le problème de la coloration de graphes ou le domaine des graphes bipartis.

Définition

Le poids d'un chemin dans un graphe pondéré est la somme des poids des arcs qui forment ce chemin.

Définition

Un chemin dans un graphe pondéré est dit optimal en valeur (ou en poids) s'il n'existe pas de chemin de même source, de même destination et de poids strictement moindre.

Remarque : Il est possible qu'il n'y ait pas de chemin optimal en valeur lorsque le graphe admet au moins un circuit de poids négatif. Par ailleurs, le calcul de chemins optimaux en valeur est plus difficile lorsqu'il y a des arcs de poids négatif. Dans le cas contraire, nous verrons dans la section suivante qu'on peut utiliser l'algorithme de Dijkstra, qui est plus simple que l'algorithme de Bellman-Ford pour le cas général.

Remarque : Il n'y a aucune raison pour qu'un chemin optimal en longueur soit optimal en valeur, et vice-versa. On peut facilement trouver un contre-exemple, d'ailleurs : il suffit de prendre un chemin de 10 arcs de poids 1 d'un sommet à un autre et en parallèle un unique arc de poids 20.

3.2 Algorithmes sur les graphes

Les opérations élémentaires étant vues dans le TP 13, nous allons nous focaliser ici sur les algorithmes classiques sur des graphes.

Pour commencer, des structures de données qui serviront d'appui seront rapidement présentées sans entrer dans les détails de l'implémentation.

3.2.1 Structures de données utiles

Définition

Une pile est une structure de données dans laquelle des éléments figurent dans un ordre précis, de sorte que seul le dernier élément qui a été ajouté (on dit « empilé »), qu'on appelle le sommet de la pile, est accessible, plus précisément il peut être récupéré en le retirant (en le « dépilant ») de la pile.

On parle de structure de type LIFO, pour "Last In, First Out".

En Python, on peut réaliser une structure de pile en utilisant des listes qu'on s'interdit de manipuler autrement que par l'initialisation à `[]` et les méthodes `append` et `pop`.

Exercice

Dans ces conditions, comment accéder au i-ième élément d'une pile en partant du sommet ? Quelle est la complexité et de quoi a-t-on besoin ? En outre, comment peut-on simuler relativement simplement une liste entière à l'aide de piles ?

Définition

Une file est, à l'instar d'une pile, une structure de données dans laquelle des éléments figurent également dans un ordre précis, mais cette fois-ci seul le premier élément qui a été ajouté (on dit « enfilé »), qu'on appelle la tête de la file, est accessible, plus précisément il peut être récupéré en le retirant (en le « défilant ») de la file.

Remarque : Le dernier élément à avoir été enfilé porte également un nom : il s'agit de la *queue de la file*.

On parle de structure de type FIFO, pour "First In, First Out".

Le nom anglais de la file, y compris en informatique, est "queue", et c'est aussi le nom d'un module qui implémente efficacement la structure de file.

Pour une réalisation honnête mais de mauvaise complexité, on peut par exemple utiliser des listes en Python pour lesquelles on se limite à muter la liste par la méthode `append` et la méthode `pop` assortie du paramètre `0`, retirant l'élément à cet indice plutôt qu'à la fin. Cette dernière opération est alors en $\mathcal{O}(n)$, où n est le nombre d'éléments dans la file.

On préfèrera utiliser des solutions toutes prêtes en Python, précisément issues du module `queue`, contenant également un type pour des piles et un type pour des files de priorité.

Il existe aussi en Python un type nommé "deque", qu'on traduit en français par « dèque » ou « file à double entrée », dont le nom vient de **d**ouble-**e**nded `queue`, c'est-à-dire une file où les opérations d'ajout et de retrait sont disponibles (et optimisées) à la fois en tête et en queue.

Exercice

Lire la documentation du type `deque`, issu du module `collections`. À partir des informations trouvées, créer et manipuler une dèque.

Définition

Une file de priorité est une structure de données dans laquelle tout élément dispose d'une information appelée priorité, généralement sous la forme d'un entier naturel, et dans laquelle les défilements concernent à chaque fois un élément de priorité maximale (mais la relation d'ordre choisie peut entraîner que la priorité maximale correspond au plus petit entier), avec une gestion des égalités ne devant cependant pas laisser la place au hasard.

Les éléments d'une file de priorité peuvent voir leur priorité changer par des opérations sur la file. Certaines implémentations ne permettent pas d'augmenter la priorité, d'autres ne permettent pas de la diminuer. Ces restrictions peuvent éventuellement rendre la réalisation de la structure plus facile. Pour l'algorithme de Dijkstra vu en fin de chapitre, la priorité ne peut qu'augmenter, et la priorité sera en fait inversée car plus l'entier sera proche de 0 et plus la priorité sera considérée comme grande.

Remarque : En insérant les éléments avec comme priorité une valeur strictement inférieure à la plus petite priorité actuelle de la file (avec une valeur quelconque si la file est vide) et en s'interdisant toute modification de la priorité, on obtient une file. Remplacer « inférieure » par « supérieure » dans ce qui précède permet d'obtenir une pile.

3.2.2 Parcours de graphe

Le parcours de graphe correspond dans son application la plus intuitive à l'exploration d'un labyrinthe. Dans ce dernier cas, le but est simplement de chercher une sortie depuis un certain endroit, ce qui revient à déterminer l'existence d'un chemin dans le graphe correspondant, mais un parcours de graphe peut être effectué jusqu'à ce que l'exploration soit complète, suivant les besoins.

Deux parcours seront étudiés ici, et ont pour point commun de partir d'un sommet, de découvrir tous les sommets reliés par un arc (par principe, les algorithmes seront écrits pour des graphes orientés, et pour le cas de graphes non orientés l'adaptation est simple), de les mettre en attente dans une structure en garantissant qu'un sommet ne soit traité qu'une fois, puis de sélectionner pour continuer le parcours un des sommets en attente selon un principe qui sera propre au parcours.

Il s'avère que le sommet sélectionné sera celui qui sera directement disponible dans la structure adéquate pour l'algorithme choisi : une pile pour le parcours en profondeur, une file pour le parcours en largeur.

Parcours en profondeur

Le **parcours en profondeur** consiste à partir d'un sommet et à explorer d'abord un chemin en ne visitant que des sommets non encore visités, jusqu'à être bloqué et à revenir au sommet précédent pour tenter de découvrir d'autres sommets non encore visités, explorant ainsi un nouveau chemin, ce qu'on peut aussi voir comme le remplacement de la fin du chemin initial par autre chose.

Pour ce faire, tous les sommets accessibles depuis le sommet en cours d'étude sont placés dans une pile, et une structure annexe permet de vérifier avant d'empiler un sommet qu'il n'a pas encore été visité.

Le squelette de l'algorithme est :

```
fonction parc_prof(graphe, origine) { # graphe : liste d'adjacence préférablement
  ouverts <- [origine]; # plutôt une pile
  fermes <- [origine]; # plutôt un dictionnaire ou simplement un ensemble
  tant que non est_vide(ouverts) {
    s <- depiler(ouverts);
    imprimer(s); # dans le cas où on veut juste voir la liste des sommets
    pour tout t dans graphe[s] { # les voisins
      si non contient(fermes, t) { # simplification (classique : ni dans ouverts)
        empiler(ouverts, t);
        inserer(fermes, t); } } } }
```

Parcours en largeur

Le **parcours en largeur** consiste à partir d'un sommet et à « visiter » d'abord tous ses successeurs, avant de visiter tous les successeurs (non encore visités) des successeurs visités lors de la première étape, et ainsi de suite. Ce parcours est la base d'algorithmes de plus court chemin **en l'absence de pondération du graphe**.

Dans ce cas, les sommets découverts sont placés dans une file, en gardant la structure annexe pour vérifier qu'un sommet n'a pas encore été visité avant de l'enfiler.

Le squelette de l'algorithme est :

```
fonction parc_larg(graphe, origine) { # liste d'adjacence aussi
  ouverts <- [origine]; # plutôt une file
  fermes <- [origine]; # encore un dictionnaire ou ensemble
  tant que non est_vide(ouverts) {
    s <- defiler(ouverts);
    imprimer(s); # même remarque
    pour tout t dans graphe[s] {
      si non contient(fermes, t) {
        enfiler(ouverts, t);
        inserer(fermes, t); } } } }
```

La ressemblance est incroyable ! Il y a cependant une différence notable, car la structure de file n'est pas aussi simple à réaliser si on part de zéro. En compensation, le parcours en largeur permet, à quelques ajouts près, de déterminer la taille minimale d'un chemin depuis l'origine choisie et jusqu'à n'importe quel sommet rencontré, ce qui n'est pas faisable par un simple parcours en profondeur.

Applications des parcours

Considérons ici une version des parcours où la fonction renvoie `fermes`.

Proposition

Un graphe non orienté est connexe si, et seulement si, un algorithme de parcours depuis n'importe quel sommet renvoie un objet dont la taille est le nombre de sommets.

Pour la forte connexité d'un graphe orienté, ce n'est pas aussi simple, car un trajet retour n'est pas forcément garanti quand l'aller existe. Dans ce cas, une condition nécessaire et suffisante est de fixer n'importe quel sommet s et de vérifier que tous les sommets soient accessibles depuis s, et que s soit accessible depuis tous les autres sommets.

Pour ne pas faire autant de parcours qu'il y a de sommets, une astuce est de faire deux parcours depuis s, un dans le graphe tel quel, et un autre dans le graphe dont les arcs sont renversés.

Faisons un autre changement dans l'algorithme : on initialise désormais `fermes` au vide au lieu d'y mettre l'origine.

Proposition

Un graphe orienté possède un circuit, et seulement si, un algorithme de parcours depuis au moins un sommet renvoie un objet contenant ce sommet.

En effet, dans ce cas l'origine a pu être rejointe au cours du parcours, car elle n'était pas directement dans la liste `fermes`.

Cette fois tout de même, il faut commencer le parcours dans un sommet du circuit, car il est possible qu'aucun circuit ne soit accessible depuis une autre origine, et quand bien même il y en aurait au moins un d'accessible mais sans qu'aucun ne contienne l'origine, la méthode suggérée ci-avant ne détecterait rien.

Pour détecter un cycle dans un graphe non orienté, le principe habituel de créer deux arcs pour chaque arête fausserait les résultats car cela génèrerait des circuits ne pouvant pas compter comme des cycles.

Une suggestion est de s'interdire de prendre un arc correspondant à celui qu'on vient de prendre, mais dans le sens inverse, et donc de mémoriser à côté de chaque sommet le sommet précédent, à exclure de la liste des voisins dans la boucle « pour ».

3.2.3 Recherche du plus court chemin

Pour la recherche du plus court chemin dans un graphe pondéré, plusieurs cas se présentent, favorisant l'un ou l'autre des trois algorithmes présentés ici :

- Si on cherche le plus court chemin depuis un sommet fixé vers un autre sommet (voire tous les autres sommets) dans un graphe sans arc de poids strictement négatif, on privilégiera l'algorithme de Dijkstra.
- S'il y a des arcs de poids strictement négatif, on utilisera l'algorithme de Bellman-Ford pour limiter la complexité.
- Si on veut disposer de toutes les distances minimales d'un sommet à un autre, on utilisera l'algorithme de Floyd-Warshall.

Seul l'algorithme de Dijkstra est explicitement au programme, les deux autres sont donnés ici pour la culture. Ils étaient enseignés en option informatique dans l'ancien programme.

Algorithme de Dijkstra

L'algorithme de Dijkstra est en quelque sorte une version améliorée du parcours en largeur, où on choisit à tout moment pour sommet à traiter le sommet non encore traité dont la distance à l'origine est la plus petite.

Ceci implique qu'une fois un sommet traité, sa distance à l'origine ne peut plus diminuer, voilà pourquoi aucun arc ne doit avoir de poids strictement négatif.

Pour optimiser la complexité, les sommets non encore visités sont stockés dans une file de priorité. On peut alors facilement y récupérer le sommet de distance minimale, y ajouter un sommet, et baisser la priorité d'un sommet (augmenter n'est pas utile ici).

Ainsi, chaque arc sera traité au plus une fois, causant éventuellement une opération d'insertion ou de modification de priorité dans la file, et chaque sommet sera traité au plus une fois, causant son retrait de la file. Il n'y aura pas de mention supplémentaire de la complexité ici.

En particulier, on se sert de files de priorité comme boîtes noires sans se poser la question du fonctionnement interne pour récupérer l'élément de distance minimale à chaque étape.

Sans files de priorité, il est tout à fait imaginable de se contenter de stocker les éléments en attente dans une liste et de calculer la distance minimale à chaque étape par une boucle sur la liste.

Par construction, la distance à l'origine de sommets visités ne peut plus être modifiée, nous n'utilisons donc pas la variable `fermes` comme dans le parcours en largeur. L'algorithme ne termine alors pas s'il existe un circuit de poids strictement négatif (mais dans ces conditions il ne pourrait pas être correct).

Une version complète de cet algorithme mémoriserait aussi le prédécesseur de chaque sommet $t \neq s$ dans un chemin de poids minimal de s à t. Ceci utilise un tableau auxiliaire de taille $|S|$. En ne gardant que les arcs mémorisés, on obtient un arbre décrivant un chemin optimal de s à chaque sommet : le seul chemin restant disponible dans l'arbre.

Le squelette de l'algorithme est ici :

```
fonction dijkstra(graphe, origine) { # graphe : liste d'adjacence avec poids
  d <- dictionnaire indexé par S initialisé à l'infini;
  d[s] <- 0;
  ouverts <- [(origine, 0)]; # file de priorité
  tant que non est_vide(ouverts) {
    s <- defiler_valeur_min(ouverts); # nom explicite ici
    pour tout (t, poids_arc) dans graphe[s] {
      poids_t <- d[t];
      si d[s] + poids_arc < poids_t {
        d[t] <- d[s] + poids_arc;
        si poids_t = l'infini { ajouter_file(ouverts, (t, d[t])) }
        sinon { changer_priorite(ouverts, t, d[t]) } } } }
  retourner d; }
```

Algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall consiste à travailler sur une matrice carrée de taille $|S|$ dont la cellule (i, j) contient à la fin la distance entre le i-ième et le j-ième sommet. En fait, $|S|$ itérations sont effectuées, de sorte qu'à l'itération k le contenu de la cellule (i, j) est le poids minimal d'un chemin entre le i-ième sommet et le j-ième sommet, en ne passant que par des sommets de numéro inférieur ou égal à k. L'écriture est alors très simple :

```
fonction floyd_warshall(n, arcs) { # liste d'arcs avec poids
  dist <- matrice (n, n) initialisée à l'infini; # sommets : de 0 à n-1
  pour h entre 0 et n-1 { dist[h][h] <- 0 }
  pour tout ((s, t), poids) dans arcs { dist[s][t] <- poids }
  pour k entre 0 et n-1 {
    pour i entre 0 et n-1 {
      pour j entre 0 et n-1 {
        dist[i][j] <- min(dist[i][j], dist[i][k] + dist[k][j]); } } }
  retourner dist; }
```

À la fin, si une valeur diagonale est strictement négative, alors il existe un circuit de poids strictement négatif, et beaucoup de valeurs doivent être remplacées par moins l'infini si on veut la réponse exacte, autrement on peut déclencher une erreur.

Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford, dont la complexité est certes supérieure à celle de l'algorithme de Dijkstra, présente sur ce dernier l'avantage de fonctionner même s'il existe des arcs de poids strictement négatif.

En fait, l'algorithme peut détecter des circuits de poids strictement négatif et donner, en leur absence, un plus court chemin entre un sommet donné et n'importe quel autre sommet.

Il est également possible de calculer après aménagement du programme le prédécesseur dans le plus court chemin calculé.

Avec des circuits de poids strictement négatif, il faudrait faire comprendre quel circuit prendre et comment faire la jonction avec l'origine et la destination, ce qui est nettement plus délicat.

Le principe est de calculer les chemins de poids minimaux et de taille `k` d'un sommet particulier à tous les sommets pour `k` de `1` au nombre de sommets moins un.

Ensuite, si faire un tour de boucle supplémentaire améliore strictement l'un des poids, c'est qu'il y avait un circuit de poids strictement négatif (et des valeurs $-\infty$ à propager si l'algorithme était adapté).

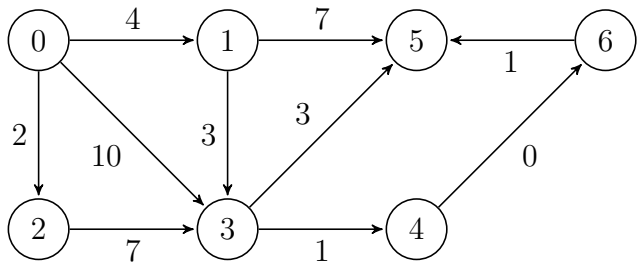
L'écriture en pseudo-code figure donc ci-après.

```
fonction bellman_ford(n, arcs, origine) { # liste d'arcs avec poids
  d <- tableau de taille n initialisé à l'infini;
  d[origine] = 0;
  pour i entre 0 et n {
    nv_d <- copie de d;
    pour ((s, t), poids) dans arcs { nv_d[t] <- min (nv_d[t], (d[s] + poids)); }
    si i = n et d != nv_d { retourner "Cycle de poids négatif"; }
    remplacer d par nv_d; }
  retourner d; }
```

TD 3 : Un graphe sommaire

L'objectif de ce TD est de se familiariser avec les notions vues en cours sur les graphes à partir d'un exemple simple.

Nous travaillerons sur ce graphe pondéré :



Pour les premières questions, la pondération des arcs sera ignorée.

Exercice 1 : Combien le graphe a-t-il de sommets, d'arcs ? Donner une expression de ce graphe selon les trois représentations vues en cours.

Exercice 2 : Donner un chemin de 0 à 6 dans le graphe. Ce graphe est-il fortement connexe ?

Exercice 3 : Trouver un chemin hamiltonien dans le graphe. Ce graphe admet-il des chemins eulériens ?

Exercice 4 : Dessiner la fermeture transitive du graphe. En déduire ses composantes fortement connexes.

Exercice 5 : Réaliser à la main un parcours en profondeur depuis le sommet 0 en donnant la priorité parmi les voisins d'un sommet en fonction du numéro associé (les petits en premier).

Exercice 6 : Réaliser à la main un parcours en largeur depuis le sommet 0 en donnant la priorité parmi les voisins d'un sommet en fonction du numéro associé (les petits en premier).

Désormais, on considèrera les poids des arcs.

Exercice 7 : Est-il possible de lancer l'algorithme de Dijkstra sur ce graphe et pourquoi ?

Exercice 8 : À l'aide d'un algorithme au choix, déterminer un chemin de poids minimal depuis le sommet 0 et vers chaque autre sommet, ainsi que le poids du chemin en question. En profiter pour restreindre le graphe à un arbre réalisant des chemins optimaux.

Deuxième partie

Travaux pratiques du premier semestre

TP 1 : Recherche d'éléments

Avant de commencer... la théorie

Le but de ce TP est de se remettre dans le bain en programmation par des algorithmes de base, et de découvrir ou redécouvrir les structures étudiées (listes et dictionnaires).

Un algorithme de recherche dans une structure, sans information particulière sur celle-ci, revient à parcourir la structure et déterminer si un des éléments rencontrés correspond à ce que l'on cherchait. En cas de succès, il est tout à fait envisageable d'arrêter le parcours.

Il s'agit d'un algorithme de type « il existe ». L'erreur à ne pas faire est de mettre au même niveau le succès et l'échec lors du parcours, c'est-à-dire d'enchaîner un `if` et un `else` comprenant chacun le mot-clé `return`.

Plus généralement, on retiendra que **dans une boucle, il faut toujours qu'il y ait une possibilité de ne pas rencontrer de `return` (ni de `break`)**. En effet, dans le cas contraire, il ne pourrait y avoir qu'un seul tour de boucle, ce qui rend la boucle elle-même inutile.

Manipulations à faire pour la prise en main

Recherche simple

Définir la fonction suivante :

```
def recherche(l, x):
    for element in l:
        if element == x:
            return True
    return False
```

Exécuter `recherche(range(10**7), 42)` puis `recherche(range(10**8), 42)`. Constater que la réponse est immédiate. [1]

Exécuter ensuite `recherche(range(10**7), -42)` puis `recherche(range(10**8), -42)`. Observer cette fois-ci que le temps mis est environ multiplié par dix entre la première et la deuxième instruction. Nous verrons au deuxième semestre le calcul de complexité et nous dirons que le coût de cette recherche est linéaire, dans la mesure où multiplier la taille de l'entrée par k multiplie le temps mis par k dans le pire des cas. Au contraire, la recherche ayant réussi avait un coût constant de quarante-trois comparaisons.

1. La notion de générateur n'est pas au programme, mais il peut être intéressant de savoir qu'un objet `range` n'est pas stocké dans la mémoire ni engendré totalement, d'où l'absence d'attente avant le démarrage. Comparer avec `recherche(list(range(10**8)), 42)`.

Structure de dictionnaire

La structure de *dictionnaire* est une structure optimisée permettant d'indexer, non pas par des entiers naturels consécutifs à partir de 0, mais par n'importe quelles valeurs, pas forcément d'un seul type, pourvu qu'elles soient immutables.

Les opérations élémentaires sont présentées et commentées ci-dessous.

```
dict = {} # Création d'un dictionnaire vide
dict2 = { "reponse" : 42, "question" : "???", 0 : False }
    # Création d'un dictionnaire avec trois entrées

dict[0] = 3 # Affectation
0 in dict # Test d'appartenance d'une clé : réponse True
3 in dict # False
dict[0] # 3
dict[3] # Erreur, 3 n'est pas une clé
dict[2] = "Deux"
1 in dict # False, on ne crée pas d'entrées intermédiaires !
len(dict) # Nombre d'éléments : 2

dict.keys() # Objet contenant 0 et 2, l'ordre n'est pas garanti
dict.values() # Objet contenant 3 et "Deux",
    # même remarque, les deux ordres peuvent par ailleurs différer
dict.items() # Objet contenant les couples (0, 3) et (2, "Deux")

for i in dict:
    print(i) # Imprime les clés

for i in dict:
    print(dict[i]) # Imprime les valeurs

for i in dict.values():
    print(i) # Autre version de ce qui précède

dict2 = dict.copy() # Écrase l'ancienne définition, et dict2 est indépendant de dict
dict3 = dict # dict3 est identique à dict
del dict[0] # Supprime une entrée
dict.pop(2) # S'évalue à "Deux", et le dictionnaire est désormais vide !
len(dict2) # Toujours 2
len(dict3) # Comme pour dict : 0
```

Nous ne parlerons pas de la façon dont les dictionnaires sont gérés en pratique, ni du temps de calcul effectif. Il s'agit pour le moment d'admettre que l'accès à une valeur est immédiat (ce qui est un léger mensonge).

Exercices

Exercice 1 : Écrire une fonction `compte(l, x)` déterminant le nombre d'indices où la valeur `x` est dans la liste `l`.

Exercice 2 : Écrire une fonction `premier(l, x)` déterminant le premier indice où la valeur `x` est dans la liste `l`. Si l'élément n'y figure pas, ne rien retourner.

Exercice 3 : Écrire une fonction `dernier(l, x)` déterminant le dernier indice où la valeur `x` est dans la liste `l`. Si l'élément n'y figure pas, ne rien retourner.

Exercice 4 : Écrire une fonction `occurrences(l, x)` déterminant la liste des indices où la valeur `x` est dans la liste `l`. Si l'élément n'y figure pas, la liste est donc vide.

Exercice 5 : Écrire une fonction `maximum(l)` déterminant le maximum de la liste de nombres `l`. Si la liste est vide, le comportement est au choix.

Exercice 6 : Écrire une fonction `localise_maximum(l)` déterminant l'indice du maximum de la liste de nombres `l`. Si la liste est vide, le comportement est au choix. Si le maximum apparait plusieurs fois, l'indice retourné est aussi au choix.

Exercice 7 : Écrire une fonction `maximumbis(l)` déterminant le deuxième plus grand élément (qui peut être égal au maximum) de la liste de nombres `l`. Si la liste est de taille inférieure ou égale à un, le comportement est au choix. Écrire aussi une fonction qui renvoie les deux plus grands éléments (le but est de s'entraîner sur une subtilité de `return` et sur la possibilité de ne faire qu'un parcours).

Exercice 8 : Écrire une fonction `moyenne(l)` déterminant la moyenne de la liste de nombres `l`.

Exercice 9 : Écrire une fonction `variance(l)` déterminant la variance de la liste de nombres `l`. Il est recommandé d'appeler la fonction `moyenne`, et surtout de ne pas le faire dans la boucle.

Exercice 10 : Écrire une fonction `detail(l)` retournant la liste des couples `(x, n)` pour tout élément `x` de la liste, où `n` est son nombre d'apparitions. On utilisera un dictionnaire.

Exercice 11 : Écrire une fonction `identiques(l, ll)` déterminant si les deux listes `l` et `ll` ont les mêmes éléments et avec pour chacun le même nombre d'occurrences.

TP 2 : Boucles imbriquées

Avant de commencer... la théorie

Dans ce TP, des structures plus complexes seront abordées, principalement des listes de listes, autrement dit des séquences dont les éléments sont eux-mêmes des séquences.

On devinera aisément que la boucle qui parcourt la séquence contiendra naturellement elle-même une boucle pour parcourir ses éléments.

Ce sera également l'occasion de présenter l'instruction `break`, dont l'utilisation est certes décriée (et que je n'aime d'ailleurs pas utiliser), mais qui est au programme. Reste à chacun de décider de continuer à programmer sans ou non, mais mieux vaut comprendre un programme qui l'utilise.

Des algorithmes variés sont au programme de ce TP : la recherche de motifs naïve[2], le tri à bulles, la recherche des deux valeurs les plus proches d'une liste.

Manipulations à faire pour la prise en main

Commençons par écrire un programme qui vérifie si tous les éléments d'une liste apparaissent au moins une fois dans l'autre liste, et vice-versa :

```
def verif(l, ll):
    for x in l:
        present = False # x n'est pas dans ll
        for y in ll:
            if x == y:
                present = True # si !
                break # passons à la suite
        if not present: # x n'est vraiment pas dans ll
            return False # fin du programme
    for x in ll:
        present = False
        for y in l:
            if x == y:
                present = True
                break
        if not present:
                return False
    return True # pas de False donc la réponse est oui
```

L'utilisation du mot-clé `break` permet d'arrêter la boucle la plus intérieure (`for y`) lorsqu'elle est rencontrée. Ici, si la variable `present` est mise à `True`, il n'est plus utile de terminer le parcours de la deuxième liste, car le test est validé. Un tel gain de temps est relativement intéressant, et permet d'éviter d'utiliser une boucle conditionnelle à la place de la boucle inconditionnelle.

2. problème qui est la base d'un des plus beaux chapitres d'option selon moi

On signale par ailleurs que l'opérateur `in` et sa négation permettent de rendre le code plus concis, mais en n'oubliant pas qu'une boucle est cachée par ce test d'appartenance (voir TP précédent) et que le nombre de comparaisons reste asymptotiquement le même.

```
def verifbis(l, ll):
    for x in l:
        if x not in ll:
            return False
    for x in ll:
        if x not in l:
            return False
    return True
```

Comme dans le TP précédent, exécuter `verif(range(2*10**3), range(2*10**3))` en estimant le temps mis, et `verif(range(10**4), range(10**4))` pour comparer. La multiplication de la taille de n'importe lequel des arguments par k multiplie le temps mis par k. Quand les deux sont multipliés par k, le temps est donc multiplié par k au carré. C'est l'idée de la complexité quadratique, qui peut bien entendu se retrouver quand il n'y a qu'un argument, sur lequel un double parcours est fait (voir les exercices).

Une consigne supplémentaire pour les exercices de ce TP : déterminer une propriété qui est vérifiée par chaque tour de la boucle intérieure. Pour l'exemple initial, cela peut être « `present` est vrai si, et seulement si, `x` est dans l'autre liste ». Ceci est la base des preuves de programmes, que nous verrons au semestre suivant.

Exercices

Exercice 1 : Reprendre les premiers exercices du TP précédent en travaillant sur les listes de listes. La localisation d'un élément donnera alors un couple (`i`, `j`) correspondant à l'indice `j` dans la liste à l'indice `i`.

Exercice 2 : Écrire une fonction `egalites(l, ll)` qui détermine le nombre de fois où un élément de la liste`l` est égal à un élément de la liste `ll`. Pour clarifier, si les deux listes sont uniquement des répétitions de la même valeur, la réponse sera le produit des tailles.

Exercice 3 : Écrire une fonction `tous_differents(l)` qui détermine si tous les éléments d'une liste `l` sont distincts. On se servira d'une double boucle et non pas d'un dictionnaire, malgré le fait que ce dernier choix soit bien plus efficace. La réponse par défaut sera `True` si la liste est vide.

Exercice 4 : Écrire une fonction `plus_proches(l)` qui détermine deux indices `i` et `j` différents tels qu'aucun couple d'éléments d'indices différents dans `l` n'ait un écart en valeur absolue strictement moindre que celui entre `l[i]` et `l[j]`. On se servira d'une double boucle et on réfléchira tout de même à une méthode plus efficace. [3] Si la liste est de taille inférieure ou égale à un, le comportement est au choix.

3. Il manque d'ailleurs le cours permettant de le confirmer !

Exercice 5 : Écrire une fonction `recherche_motif(s, m)` qui détermine si la chaîne de caractères `m` figure en entier et sans discontinuité dans `s`, et si oui à quel indice de `s` la première occurrence démarre. Si la réponse est négative, on retournera `-1`. Il s'agit d'une version basique de la méthode `find`.

Exercice 6 : Écrire une fonction `tri_bulles(l)` qui modifie par effet de bord la liste `l` de sorte qu'après l'exécution de la fonction cette liste soit triée par ordre croissant, en utilisant l'algorithme du tri à bulles. Celui-ci consiste à parcourir la liste `n-1` fois, où `n` est la taille de la liste, en échangeant à chaque parcours deux éléments voisins s'ils sont mal ordonnés. On réfléchira aux seuils des parcours afin de ne pas faire de tests inutiles.

Exercice 7 : Écrire une fonction `crible(n)` qui retourne la liste des nombres premiers inférieurs ou égaux à `n`, en utilisant la méthode du crible d'Ératosthène, par l'introduction d'une liste de booléens indiquant pour tout nombre jusqu'à `n` si ce nombre est premier.

TP 3 : Utilisation de modules et de bibliothèques

Le but de cette séance est de découvrir ou réviser la gestion des modules, qu'ils soient fournis ou créés par l'utilisateur.

En plus de cela, la compréhension des erreurs sera abordée, ainsi que la documentation de code et la fonction `help`.

Modules et bibliothèques

Si on veut simplifier à l'extrême, modules et bibliothèques représentent la même chose en Python : un fichier contenant du code Python et qu'on peut charger afin de l'utiliser sans avoir à réécrire son contenu. Ce contenu est principalement composé de fonctions, mais pas seulement. Ainsi, on peut charger la constante `pi` dans le module `math`, et il est également possible de récupérer dans des modules des déclarations de classes, ainsi que des objets des classes en question.

Les modules les plus classiques en prépa sont `math`, `random` et `numpy`, ce dernier étant a priori abordé dans l'enseignement d'autres matières, car il est très utile pour résoudre la plupart des problèmes pour lesquels l'outil informatique y sera sollicité. À ce sujet, d'éventuelles courbes obtenues seront classiquement tracées avec `matplotlib`, notamment par le sous-module `pyplot`. D'autres modules utiles pour la programmation en-dehors des études et qui seront utilisés sporadiquement : `time`, `os` et `sys`.

Pour information ou rappel, un module (ou une bibliothèque, donc) peut être importé en entier ou un sous-ensemble de son contenu peut être importé. La syntaxe de l'importation change, de même que celle des appels aux éléments importées. Les morceaux de code suivants produisent la même chose mais on ne peut pas échanger une ligne d'un morceau avec une ligne de l'autre sous peine de déclencher une erreur de nom :

```
import mon_module # tout le module est disponible
mon_module.ma_fonction(mon_module.ma_variable)

import mon_module as momo
# momo sera un alias, et mon_module sera inconnu
momo.ma_fonction(momo.ma_variable)

from mon_module import ma_fonction, ma_variable
# le reste du module est inaccessible
ma_fonction(ma_variable)

from mon_module import * # tout est importé
ma_fonction(ma_variable)
```

Il est peu recommandé d'utiliser la dernière syntaxe dans de gros programmes si le risque d'écrasement de noms définis par ailleurs est élevé. Par exemple, les modules `numpy` et `math` ont beaucoup de noms en commun, et ont tout intérêt à être importés par la première ou la deuxième syntaxe.

Dans le cadre de gros projets de programmation, diviser son code en plusieurs fichiers fait partie des réflexes de programmation à acquérir aussi vite que possible. Ce faisant, il faut pouvoir accéder aux fonctions et autres objets définis dans un autre fichier, et on importera ceux-ci tout comme on le ferait avec un module usuel. Pour ce faire, il faut que le dossier courant de la console soit le dossier où se situe le fichier à importer, et on écrira simplement une ligne d'importation associée au nom qui figure avant l'extension `.py` (qu'on a de toute façon intérêt à préciser). Si le dossier courant de la console n'est pas le bon, cela donnera une première occasion de découvrir les fonctions `chdir`, `listdir` et `getcwd` du module `os`.

Pour terminer, si d'aventure un module non standard doit être utilisé et que l'importation échoue, Python dispose d'un programme pour télécharger et installer de nouveaux modules, appelé `pip`, et fonctionnant en ligne de commande.

Erreurs

Erreurs de syntaxe

Les premières erreurs que nous évoquerons ici sont les erreurs de syntaxe. Elles sont détectables à de multiples titres : elles provoquent des messages explicites renvoyés par Python, un bon éditeur les signale par du surlignage rouge... et un examen minutieux du code suffit de toute façon, avec l'expérience.

La plupart du temps, il s'agit de parenthésage incorrect, de l'oubli (voire de l'ajout indu) d'un double point ou d'une mauvaise indentation.

Exercice 1 : Le code ci-dessous comporte six erreurs. Relever les six messages d'erreur de Python en les corrigeant dans l'ordre.

```
def sextuple erreur(n
print "Espace indue dans le nom, double-point manquant,
parenthèse ouverte, indentation oubliée, délimiteur simple
dans la chaine et print sans parenthèses"
```

On notera que les erreurs sont annoncées comme `SyntaxError`, sauf l'erreur d'indentation qui a son « code » à part.

Comprendre les messages d'erreur de Python

Dans cette section, des programmes contenant diverses erreurs seront écrits, et il s'agit d'analyser la réaction mécontente du moteur Python.

Exercice 2 : Relever les messages d'erreur de Python en exécutant les programmes ci-dessous et les expliquer.

(Attention, le principe n'est pas de corriger les programmes qui n'ont de toute façon pas d'intérêt mais de comprendre les messages d'erreur et de supprimer les morceaux de code ensuite.)

```
for i in range(input()):
    print(i)

"deux" * "quatre"

s = "bonjour"
s[0] = "B"
s[7]
ss[6]

def carre(n):
    return n * n
carre()

for i in 42:
    print("Il y en a qui ont essayé...")

l = [1, 2, 3]
for i in range(l):
    print("Ils ont eu des problèmes !")

len(42)

"abcdefghj".index("i")

x = 42
if x = 3:
    print("""En C ou en PHP, ça marcherait,
là c'est une autre erreur de syntaxe""")
    return "Et ceci provoque une autre erreur de syntaxe"
```

Documentation de code

Principe

En utilisant un environnement de développement intégré on est amené à voir apparaître des rectangles expliquant le principe d'une fonction dont on commence à écrire le nom (vive l'autocomplétion). C'est de la documentation de code, et ces « *docstrings* » peuvent bien entendu être écrits par n'importe qui : par exemple, au moment d'écrire une fonction, juste en-dessous de la ligne de définition, écrire une chaîne de caractères (généralement sur plusieurs lignes, et donc avec des triples guillemets comme délimiteurs) fera comprendre à Python qu'on a précisé un message pour l'utilisateur, et ceci sera imprimé dès lors qu'on demandera des informations à l'aide de la fonction `help`, qui prend en argument un nom de fonction (pas seulement, mais passons), que ce soit de la bibliothèque standard, d'un module ou d'une bibliothèque importée (en préfixant par le nom de celui-ci ou celle-ci, le cas échéant) ou définie par l'utilisateur.

Bien entendu, documenter une fonction fait partie des critères rendant un code agréable à lire, de même que des noms de variable et de fonction pertinents, des commentaires sur des passages délicats et une indentation rigoureuse (de toute façon Python ne laisse pas grandement le choix sur ce dernier point).

Pour aller encore plus loin, il est possible de faire des annotations au niveau des fonctions, qui contribuent encore plus à leur lisibilité. Il est possible d'en faire un réflexe pour les utilisateurs de Caml, et pour tout le monde en fait. [4]

La fonction `help`

Python dispose d'une fonction formidable, qui s'appelle `help`. Elle affiche la documentation de code associée à la fonction ou la méthode ou la classe ou le module à propos duquel on demande de l'aide.

Tester par exemple (plutôt dans la console, par principe) :

```
help(sum)
help([].append) # pour une méthode, il faut trouver un objet du bon type
help(42)
import random
help(random.randint)
help(random)
```

Exercice 3 : Étudier le manuel du module `random` ou des cinq fonctions mentionnées ci-après, et écrire avec `random`, `randint`, `choice`, `sample` et `shuffle` autant de fonctions qui permettent de faire un tirage sans remise de `n` éléments dans une liste `l`, avec `l` et `n` en argument.

Débug

Deux techniques pour détecter et corriger sans intervention extérieure un programme qui ne fonctionne pas sont présentées ci-après.

Baliser les programmes

Afin de comprendre le comportement de certains programmes, et entre autres remarquer pourquoi il ne termine pas ou ne fait pas ce qu'on attend de lui, faire un balisage peut être intéressant. C'est notamment le cas pour des programmes relativement longs, dont aucun exemple n'a à ce jour été écrit en TP.

L'idée est de choisir un endroit douteux dans une boucle et d'y forcer une impression d'un message explicite, permettant de constater le comportement de Python et les possibles déviations par rapport à ce qu'on attendait.

4. Les annotations ont été présentées dans le sujet de Centrale en 2017 : `https://www.concours-centrale-supelec.fr/CentraleSupelec/2017/Multi/sujets/2014-028.pdf`. Un autre lien utile à ce sujet : `http://sametmax2.com/les-annotations-en-python-3/`.

Ainsi, imaginons une simple boucle conditionnelle qui ne terminerait pas :

```
l = [1, 2, 3]
i = 0
while i < len(l):
    print(l[i])
    i =+ 1 # erreur ici : on voulait sans doute écrire += 1
```

Pour s'en rendre compte, il suffit d'ajouter le balisage en question et de voir que `i` vaut partout `1` :

```
l = [1, 2, 3]
i = 0
while i < len(l):
    print(l[i])
    i =+ 1
    print(i)
```

Un autre cas classique est de mettre `print("Coucou numéro XXX")` pour différents `XXX` dans tous les cas d'un test conditionnel afin de savoir si un des blocs n'est jamais visité.

Exercice 4 : S'engager à utiliser ce genre de méthodes à l'avenir afin de pouvoir corriger des bugs par soi-même.

Déboguer avec un IDE

Une autre façon de procéder est d'utiliser un débogueur intégré à l'éditeur. La plupart des environnements de développement intégrés permettent de placer des points d'arrêt dans du code.

Il s'agit en général de cliquer sur la zone à gauche d'une ligne afin de placer un tel point, et de lancer l'exécution une exécution pas à pas, qui est potentiellement déclenchée par une autre commande / un autre raccourci clavier.

Dans ce cas, l'exécution pausera à chaque fois qu'elle arrivera à une ligne marquée comme point d'arrêt.

À ce moment-là, l'utilisateur doit pouvoir, d'une manière ou d'une autre, inspecter la valeur des variables pertinentes et en déduire la raison pour laquelle le code ne fonctionne pas (il y a quand même un exercice cérébral, l'ordinateur ne déboguera pas tout à la place de l'humain).

Malheureusement, la mise à jour de ces variables n'est normalement pas visible quand le moteur est en pleine exécution, sinon un œil affuté pourrait bien plus profiter d'un tel aperçu de l'évolution des variables.

Exercice 5 : Déboguer des programmes au choix, qu'ils aient déjà été écrits dans des TP antérieurs ou qu'ils soient improvisés pour l'occasion.

TP 4 : Algorithmes dichotomiques

Avant de commencer... la théorie

Le premier contact avec la dichotomie est souvent le jeu du « C'est plus / c'est moins ». L'intuition, peut-être sans même savoir les tenants et les aboutissants, est de proposer une valeur la plus centrale possible afin de délimiter rapidement l'intervalle de recherche. En poussant encore plus loin, le jeu « Qui est-ce ? », pour les enfants à partir de six ans, permet déjà de chercher à optimiser les questions afin d'éliminer à coup sûr la moitié des possibilités.

Dans ces deux cas, et d'autres que nous verrons au cours du TP, le principe est le même : couper en deux (ce qui est l'étymologie grecque du terme), mais avec la propriété que les parts doivent être bien équilibrées.

Le nombre d'opérations pour une structure de taille comprise entre 2^k et 2^{k+1} sera alors de l'ordre de k, ce qu'on appelle une complexité logarithmique. C'est bien mieux que le traitement de l'ensemble de la structure, précisément parce que chaque opération exclut directement un certain pourcentage de la structure.

Pour formaliser, l'utilisation de la dichotomie revient à se placer dans un cadre où il faut déterminer une liste de n booléens [5]. Il y a 2^n listes possibles, et plutôt que de tester chacune on va déterminer par un seul test chacun des booléens de la liste. Cependant, il n'est pas garanti que le contexte permette de décomposer ainsi le problème...

Manipulations à faire pour la prise en main

La dichotomie classique, qui peut facilement être mise en relation avec le premier exemple, correspond à la recherche d'un élément dans un tableau trié. Voici une façon possible de l'écrire.

```
def recherche_dicho(sequence, element_a_chercher):
    n = len(sequence)
    if n == 0:
        raise ValueError("Introuvable")
    cro = sequence[0] < sequence[-1] # Booléen indiquant si la séquence est croissante
    ind_deb = 0
    ind_fin = n-1
    while (ind_deb <= ind_fin):
        ind_mil = (ind_deb + ind_fin)//2
        if sequence[ind_mil] == element_a_chercher:
            return ind_mil
        if (sequence[ind_mil] < element_a_chercher) == cro: # "si et seulement si"
            ind_deb = ind_mil+1 # ind_mil est de toute façon exclu désormais
        else:
            ind_fin = ind_mil-1 # idem
    raise ValueError("Introuvable")
```

5. avec le cours du deuxième semestre, j'aurais écrit déterminer un entier écrit en binaire sur n bits...

La fonction ci-avant détermine une position arbitraire de l'élément à chercher et fonctionne si la séquence est croissante ou décroissante.

Une consigne supplémentaire pour les exercices de ce TP : Comme il est très facile de laisser échapper un bug dans le code de la dichotomie, il s'agira de mettre en place des tests automatiques. Pour faire écho aux documentations de code introduites dans le TP précédent, le choix pédagogique ici est d'utiliser le module `doctest`.

Voici son fonctionnement sur un exemple très simple :

```
import doctest

def addition(x, y):
    """Additionne les deux arguments.
    >>> addition(2, 2)
    4
    >>> addition(6, -3)
    3
    >>> addition(0, 0)
    0
    """
    return x + y

def fonction_ratee(x):
    """
    Retourne la partie entière de l'argument.
    >>> fonction_ratee(4.5)
    4
    >>> fonction_ratee(-1.)
    -1
    >>> fonction_ratee(-2.2)
    -3
    """
    return int(x)

doctest.testmod()
```

Magnifique, n'est-ce pas ?

Exercices

Exercice 1 : Adapter ou réécrire le programme de recherche dichotomique pour déterminer la première position d'une séquence triée, qu'on pourra supposer croissante, où un élément figure. Le comportement est au choix s'il n'y figure pas. On prendra bien soin de ne pas faire de décalage une fois une occurrence arbitraire découverte, mais bien de chercher le seuil par dichotomie.

Exercice 2 : Écrire une fonction `recherche_zero_continue(f, a, b, eps)` qui détermine une approximation à `eps` près d'un antécédent de zéro par la fonction mathématique continue représentée par la fonction `f`, antécédent qui est garanti entre `a` et `b` dans la mesure où on fournit des valeurs pour lesquelles le signe donné par `f` est censé être différent (comportement au choix sinon).

Exercice 3 : Écrire une fonction `recherche_zero_monotone(f, a, b, eps, eta)` qui détermine une approximation à `eps` près de l'antécédent de zéro, s'il existe, par la fonction mathématique monotone représentée par la fonction `f`, antécédent qui serait forcément entre `a` et `b` dans la mesure où on fournit des valeurs pour lesquelles le signe donné par `f` est censé être différent (comportement au choix sinon). On admettra que l'antécédent est trouvé si son image par `f` est inférieure au seuil `eta`.

Pour les deux fonctions précédentes, il faut savoir que la précision des ordinateurs ne permet pas d'avoir une réponse exacte mais seulement une approximation (dans les deux cas), et même qu'il n'y a aucune garantie que l'annonce de la découverte d'un antécédent ou de son absence soit avérée (fonction monotone).

Exercice 4 : Écrire une fonction `multiplication(x, y)` qui détermine le produit de `x` et `y` en se limitant aux opérations d'addition (donc la multiplication par `2` est autorisée) et de quotient par `2`. On tâchera de ne pas faire un nombre linéaire d'additions.

Exercice 5 : Reprendre le même principe mais avec des opérateurs différents pour écrire la fonction `puissance(x, y)` en se limitant aux multiplications (donc l'élevation au carré est autorisée) et aux quotients par `2`.

Exercice 6 : Reprendre les exercices 1, 4 et 5 avec un algorithme naïf et, après avoir consulté la documentation du module `time` (notamment la fonction `time`), mesurer le temps effectivement mis par les versions optimisées et naïves.

Une version plus performante de l'estimation du temps mis par des programmes en Python utilise le module `timeit`. Il est bien entendu permis de s'en servir à la place du module `time` dans le dernier exercice.

TP 5 : Fonctions récursives

Avant de commencer... la théorie

Dans ce TP, nous présentons une alternative à l'utilisation de boucles en programmation, qui rejoint la notion mathématique de suite récurrente : la *récursivité*.

Il s'agit, lors de la définition d'une fonction, d'utiliser la fonction elle-même appelée sur un autre argument de sorte qu'un cas de base finisse par être atteint [6].

Par exemple, si on veut trouver la factorielle d'un nombre entier `n`, une version itérative, donc avec boucles, de la fonction calculera le produit $\Pi_{i=1}^{n} i$ et s'écrira :

```
def fact(n):
    rep = 1
    for i in range(1, n+1):
        rep *= i
    return rep
```

... et une version récursive définira une suite u_n telle que $u_n = nu_{n-1}$ et $u_0 = 1$:

```
def factrec(n):
    if n == 0:
        return 1
    else:
        return n * factrec(n-1)
```

Un appel à `factrec` se fera forcément avec un argument valant 1 de moins que l'argument actuel, de sorte que 0 finisse par être atteint dès lors que la fonction est appelée initialement sur un entier naturel.

En pratique, les appels récursifs sont stockés dans une pile de taille suffisante (environ mille en Python, modifiable par la fonction `setrecursionlimit` du module `sys`) pour ne pas causer de problème quand la fonction termine. [7]

On parle également de récursion, et précisément de récursion croisée, quand deux fonctions s'appellent mutuellement.

L'intérêt de la récursivité est de faciliter l'écriture des programmes, notamment lorsqu'il s'agit d'écrire une fonction où la récurrence est déjà visible, mais il n'en demeure pas moins vrai qu'on peut toujours remplacer une récursion par une boucle.

Attention à la syntaxe : il faut prendre garde à bien utiliser le mot-clé `return`, notamment si on a un style de programmation qui pousse à définir des fonctions locales (voir l'exemple ci-après).

6. C'est préférable...
7. À ce sujet, regarder ce que Python retourne dans la version récursive de factorielle quand on l'appelle sur `-1`.

Manipulations à faire pour la prise en main

La dichotomie, vue dans le TP précédent, peut être mise en œuvre de manière récursive, mais il faut prendre garde à ne pas ruiner son efficacité par des recopiages. Comparer les temps mis par les deux fonctions suivantes (ainsi qu'avec celle qui avait déjà été écrite).

```
def recherche_dicho_rec_nulle(sequence, element):
    # La fonction retourne désormais un booléen
    n = len(sequence)
    if n == 0:
        return False
    elif n == 1:
        return sequence[0] == element
    cro = sequence[0] < sequence[-1] # Booléen indiquant si la séquence est croissante
    ind_deb = 0
    ind_fin = n
    ind_mil = (ind_deb + ind_fin)//2
    if sequence[ind_mil] == element:
        return True
    if (sequence[ind_mil] < element) == cro:
        return recherche_dicho_rec_nulle(sequence[ind_mil+1:ind_fin], element)
    else:
        return recherche_dicho_rec_nulle(sequence[:ind_mil], element)

def recherche_dicho_rec_mieux(sequence, element):
    # La fonction retourne aussi un booléen
    n = len(sequence)
    if n == 0:
        return False
    cro = sequence[0] < sequence[-1]
    def aux(ind_deb, ind_fin):
        # sequence et element seront non locales, donc disponibles
        if ind_deb > ind_fin:
            return False
        ind_mil = (ind_deb + ind_fin)//2
        if sequence[ind_mil] == element:
            return True
        if (sequence[ind_mil] < element) == cro:
            return aux(ind_mil+1, ind_fin)
        else:
            return aux(ind_deb, ind_mil-1)
    return aux(0, n-1)
```

Le problème de la première version est que la tranche de liste récupérée dans les appels récursifs nécessite un recopiage d'où une complexité linéaire quand bien même le nombre d'appels reste logarithmique.

Enfin, voyons un cas de dépassement de la taille de la pile. Définir les fonctions (relativement inutiles) suivantes :

```
def pair(n): # On suppose que n est un entier
    n = abs(n)
    if n == 0:
        return True
    return impair(n-1)

def impair(n): # Même hypothèse
    n = abs(n)
    if n == 0:
        return False
    return pair(n-1)
```

Tester `pair(1000)` et constater le message d'erreur. En profiter pour déterminer (par dichotomie !) le seuil à partir duquel la pile d'appels récursifs de Python déborde.

Travail à faire pendant la séance

Exercice 1 : Écrire une version récursive de l'exponentiation rapide du TP précédent.

Exercice 2 : Écrire une fonction qui calcule le PGCD de deux entiers à l'aide de l'algorithme d'Euclide. Cet algorithme sera rappelé si besoin.

Exercice 3 : Écrire une fonction comptant le nombre de façons de tracer une ligne de longueur n (en argument) avec des segments de longueur 2 ou 3 (l'ordre est important pour les segments).

Exercice 4 : Écrire une fonction récursive prenant en entrée un entier naturel `n` et imprimant un « escalier » sur `n` lignes. Le rendu pour `n` valant `5` figure ci-après.

```
*
**
***
****
*****
```

Exercice 5 : Écrire une fonction récursive prenant en entrée un entier naturel `n` et imprimant un « triangle » sur `n` lignes. Le rendu pour `n` valant `3` figure ci-après. Après les étoiles, la ligne peut s'arrêter ou être complétée par des espaces pour avoir toujours le même nombre de caractères par ligne, au choix.

```
  *
 ***
*****
```

Exercice 6 : Écrire une fonction récursive prenant en entrée un entier naturel impair `n` et imprimant un « diamant » sur `n` lignes. Le rendu pour `n` valant `5` figure ci-après. Après les étoiles, la ligne peut s'arrêter ou être complétée par des espaces pour avoir toujours le même nombre de caractères par ligne, au choix.

```
  *
 ***
*****
 ***
  *
```

Exercice 7 : Écrire une fonction récursive imprimant les instructions pour résoudre le problème des tours de Hanoï.

Remarque : Ce problème consiste à déplacer une pile de n (en argument) anneaux de taille croissante d'un tas (matérialisé par un piquet) à un autre (parmi trois), les opérations élémentaires étant le déplacement d'un anneau du haut d'une pile sur le haut d'une autre pile, à condition qu'il soit plus petit que l'ancien sommet de la pile d'arrivée.

Le nombre optimal d'opérations élémentaires est $2^n - 1$.

Exercice 8 : Écrire une fonction récursive donnant la liste de toutes les permutations d'une liste en argument.

Exercice 9 : Écrire une fonction récursive donnant la liste de toutes les sous-listes d'une liste en argument. Une sous-liste de `l` correspond à une liste d'éléments apparaissant dans le même ordre mais non nécessairement consécutivement dans `l`.

Exercices pour réfléchir après la séance

Exercice 10 : En consultant la documentation du module `turtle` ou un exemple minimaliste, écrire une fonction récursive imprimant le flocon de von Koch après un nombre d'étapes donné en argument.

Exercice 11 : Chercher sur internet d'autres courbes fractales et les représenter à l'aide de fonctions récursives.

Exercice 12 : Écrire des programmes qui résolvent des problèmes classiques de « passage de rivière », notamment celui du loup, de la chèvre et du chou.

TP 6 : Algorithmes gloutons

Avant de commencer... la théorie

Pour certains problèmes en informatique, l'intuition d'une solution grossière, qui présente l'avantage de ne pas avoir besoin d'explorer toutes les possibilités, fournit la solution optimale. Pour d'autres, il est possible de prouver que la réponse obtenue est suffisamment proche de la solution optimale pour qu'on s'en contente, notamment si le calcul de la solution optimale n'est pas possible dans un délai raisonnable. Dans ce TP, nous allons donc aborder de tels algorithmes, dits *algorithmes gloutons*.

Lorsqu'un algorithme standard est face à plusieurs possibilités, il est censé toutes les tester afin de ne pas risquer de manquer la bonne. Ainsi, l'idée sera d'en tester une puis, si la solution n'a pas été trouvée entre temps, de revenir au point du choix et d'en tester une autre, et ainsi de suite. On appelle cela le retour sur trace, ou *backtracking*, une notion qu'on comprend aisément en pensant à l'exemple de l'exploration d'un labyrinthe.

Un algorithme glouton, quant à lui, dispose d'une relation de comparaison sur les choix, de sorte qu'il ne retienne que l'option considérée comme optimale, sans jamais revenir en arrière, à tort ou à raison.

Un exemple de la vie réelle pour fixer les idées : un déménageur qui doit remplir un camion avec des cartons peut tenter un nombre incommensurable de dispositions, mais sera a priori enclin à commencer par mettre les cartons les plus gros en premier, puis de boucher les trous avec les petits cartons. Cela ne marchera pas forcément, mais s'il y a assez de marge totale dans le camion on a toutes les raisons d'y croire.

Manipulations à faire pour la prise en main

Deux problèmes classiques serviront de fil rouge pour ce TP, illustrant deux cas d'application des algorithmes gloutons : le rendu de monnaie et le problème du voyageur de commerce ou TSP (*Travelling Salesman Problem*).

Des instances pour les problèmes de ce TP sont fournies à l'adresse `http://jdreichert.fr/Enseignement/CPGE/ITC%20SUP/tp6_base.py` et pourront être utilisées pour les tests.

Problème du rendu de monnaie

Le *problème du rendu de monnaie* consiste à déterminer le nombre minimal possible de pièces (ou billets) à rassembler, dans un système monétaire donné, pour produire une somme donnée.

Il peut être résolu de manière exhaustive : pour rendre `S` euros, on peut utiliser une pièce / un billet de `X` euros, avec `X` inférieur ou égal à `S`, puis rendre de manière optimale `S-X` euros ; un choix optimal de la valeur de `X` au regard des réponses pour les `S-X` donnera, après avoir ajouté un, un choix optimal pour `S`.

Le problème est que l'un des choix à prendre en compte sera finalement d'utiliser `100*S` pièces d'un centime, ce qu'on voudrait éviter.

Ainsi, un algorithme glouton considèrera que le choix favori est de rendre la pièce ou le billet de la plus grande valeur inférieure ou égale à la somme restante, et ce à chaque fois.

Il s'avère que cela donne la solution optimale en euros, mais par exemple pas dans un système où les valeurs seraient par exemple 1, 42 et 73 et la somme à obtenir serait 84. Pour rester réaliste, il existait autour du XIX^{e} siècle une unité monétaire où des valeurs de 2 et $2,5$ coexistaient, ce qui pouvait aussi mettre en échec l'algorithme glouton.

Écrire et exécuter ultérieurement en parallèle de l'algorithme glouton pour comparer les performances l'algorithme suivant pour le problème du rendu de monnaie, avec des arguments arbitraires :

```
def rendu(l, s): # l est une liste d'entiers, quitte à considérer des centimes
    reponses = [None] * (s+1)
    reponses[0] = 0
    def aux(valeur):
        if reponses[valeur] != None:
            return reponses[valeur]
        opt = s+1 # impossible de faire plus
        for x in l:
            if x <= valeur:
                test = aux(valeur-x)
                if test < opt:
                    opt = test
        reponses[valeur] = opt+1
        return opt+1
    return aux(s)
```

Problème du voyageur de commerce

Le *problème du voyageur de commerce* consiste à déterminer le trajet minimal en distance pour visiter un ensemble de villes, dont on fournit toutes les distances deux à deux, à partir d'une ville particulière.

Remarque : certains auteurs imposent de revenir à la ville de départ, ce qui permet de considérer que le choix de celle-ci n'a pas d'importance.

Il s'agit donc de trouver une permutation optimale de l'ensemble des villes, et donc l'exploration exhaustive ne peut pas terminer en un temps acceptable si le nombre de villes dépasse la dizaine.

Ainsi, l'algorithme glouton, qui aura peu de chances de donner une solution optimale, présentera néanmoins un intérêt certain (en étant éventuellement moins efficace que d'autres algorithmes, comme par exemple des algorithmes génétiques ou d'autres optimisations, mais en compensant cela par une simplicité et une rapidité accrues).

Un algorithme glouton peut être mis en œuvre de deux façons intuitives : soit choisir à chaque étape la ville la plus proche de la ville courante, soit relier à chaque étape le couple de villes les plus proches parmi celles dont la jonction ne formerait pas un circuit omettant des villes, voire un circuit suivant la donnée du problème.

Écrire et exécuter ultérieurement en parallèle des algorithmes gloutons pour comparer les performances l'algorithme suivant pour le problème du voyageur de commerce, avec des matrices de taille jusqu'à environ douze en argument.

```
def TSP(distances):
    # distances est une matrice de nombres, et on fixe le départ au sommet 0
    # Pour ce problème, on cherchera à visiter toutes les villes sans revenir à 0
    n = len(distances)
    configs_attente = [([True] + [False] * (n-1), [0], 0)]
    config_optimale = None
    while configs_attente != []:
        (visitees, chemin, dist) = configs_attente.pop()
        ville = chemin[-1]
        pas_toutes_visitees = False
        for i in range(1, n): # 0 est exclu
            if not visitees[i]: # nouvelle ville
                visiteesbis = visitees[:] # copie
                visiteesbis[i] = True
                distbis = dist + distances[ville][i]
                cheminbis = chemin + [i]
                configs_attente.append((visiteesbis, cheminbis, distbis))
                pas_toutes_visitees = True
        if not pas_toutes_visitees:
            if config_optimale == None or config_optimale[2] > dist:
                config_optimale = (visitees, chemin, dist)
    return config_optimale[1:]
```

Pour aller plus loin, le problème du voyageur de commerce a été mis à l'honneur par le sujet d'informatique commune au concours Centrale en 2017. On y retrouve un algorithme glouton et un algorithme génétique.

Travail à faire pendant la séance

Exercice 1 : Écrire un algorithme glouton pour le problème du rendu de monnaie.

Exercice 2 : Écrire la première version de l'algorithme glouton pour le problème du voyageur de commerce.

Exercice 3 : Écrire la deuxième version de l'algorithme glouton pour le problème du voyageur de commerce.

Exercice 4 : Si ce n'est pas déjà fait, créer une instance de taille environ quatre pour laquelle les algorithmes gloutons ne donnent pas la solution optimale, éventuellement une instance par algorithme.

Exercice 5 : Écrire un algorithme glouton pour le problème de sélection des activités, détaillé ci-après. On ne mettra pas en œuvre d'algorithme de tri pour cet exercice.

Le problème de sélection des activités consiste à choisir dans une liste d'activités, chacune représentée par un temps de début et un temps de fin, une sous-liste d'activités ne se chevauchant pas et de la plus grande taille possible. On ne considère pas que deux activités se chevauchent si le temps de fin de l'une est égal au temps de début de l'autre, seulement s'il est strictement supérieur. Ici, l'algorithme glouton est optimal à condition de trouver le bon critère discriminant les choix d'activité...

Exercice 6 : Écrire un algorithme glouton pour le problème d'allocation des salles de cours, détaillé ci-après.

Le problème d'allocation des salles de cours, qu'on pourra rapprocher du problème précédent, consiste à sélectionner pour chaque cours, représenté par un temps de début et un temps de fin, une salle où il peut avoir lieu, sans que deux cours n'aient lieu en même temps dans une même salle, et en minimisant le nombre de salles nécessaires. Là aussi, on ne considère pas que deux cours ont lieu en même temps si le temps de fin de l'un est égal au temps de début de l'autre. Le problème classique en informatique correspondant est celui de la coloration de graphes (cette structure sera vue au deuxième semestre), et bien que pour ce problème l'algorithme glouton n'est pas optimal en général, il s'avère que c'est tout de même le cas dans les graphes obtenus en recensant les incompatibilités entre cours, en affectant à chaque cours la salle du plus petit identifiant possible.

Pour aller plus loin, ce problème a été mis à l'honneur par le sujet d'option informatique au concours Centrale en 2015.

Exercices pour réfléchir après la séance

Exercice 7 : Écrire un algorithme glouton pour le problème du sac à dos, détaillé ci-après.

Le *problème du sac à dos* consiste à sélectionner un sous-ensemble d'objets dont la valeur totale est la plus grande possible en respectant une limite de poids. Les objets seront donc représentés comme des couples (poids, valeur), et suivant l'énoncé il est possible de prendre au plus une occurrence ou autant qu'on veut de chaque objet. L'algorithme glouton prendra de préférence les objets encore prenables maximisant le rapport valeur / poids.

Exercice 8 : Écrire un algorithme donnant la solution optimale au problème du sac à dos, à ne tester que pour de petites entrées là aussi. Créer une instance de taille limitée pour laquelle l'algorithme glouton ne donne pas la solution optimale.

TP 7 : Matrices de pixels et d'images

Pour ce TP, les listes de listes pourront être remplacées par des tableaux de dimension deux pour ceux qui maîtrisent la bibliothèque ***numpy****.*

Avant de commencer... la théorie

Une image est un rectangle composé de ce qu'on appelle des *pixels* (*picture element*), mis en lumière lorsqu'on zoome suffisamment en tant que petits carrés.

La couleur d'un pixel est déterminée par trois valeurs dans le système RGB (synthèse additive des couleurs), qui sont respectivement les degrés des couleurs primaires rouge, verte et bleue. Ce codage peut se faire sous la forme d'un entier entre `0` et `255` ou, plus fréquemment, d'un nombre entre `0` et `1`.

Attention, si on utilise des entiers entre `0` et `255`, la fonction `imshow` échouera sur le type `int` mais donnera le bon résultat sur le type `np.uint8`, un type particulier de la bibliothèque `numpy`, que l'image manipulée dans le TP utilise.

Pour fixer les idées, le triplet `(0, 0, 0)` représentera le noir, le triplet `(maximum, 0, 0)` représentera le rouge, et le triplet `(maximum, maximum, maximum)` représentera le blanc.

Les manipulations d'images de ce TP seront en fait des manipulations avancées sur des listes de listes.

Pour un sujet sur la manipulation d'images, on pourra consulter l'épreuve de tronc commun du concours Centrale de 2020.

Manipulations à faire pour la prise en main

Récupérer la magnifique image d'un raton laveur par ces instructions. Les dimensions de l'image sont 768 par 1024.

```
import scipy.misc

image = scipy.misc.face()
l = image.tolist() # Si on préfère travailler sur des listes de listes.
```

Si on veut consulter l'image, on pourra par exemple écrire

```
import matplotlib.pyplot as plt

plt.imshow(image) # marche aussi avec l
plt.show()
```

Travail à faire pendant la séance

Exercice 1 : Écrire une fonction qui engendre une image aléatoire dont les dimensions sont en argument.

Exercice 2 : Écrire une fonction qui prend en entrée une image et retourne sa version pivotée d'un quart de tour vers la droite. Il s'agit de trouver la correspondance entre les indices dans la liste de listes de départ et dans celle d'arrivée.

Exercice 3 : Écrire une fonction qui prend en entrée une image et retourne sa version agrandie d'un facteur deux. Il s'agit de remplacer chaque pixel par un carré de quatre pixels.

Exercice 4 : Écrire une fonction qui prend en entrée une image et retourne sa version agrandie d'un facteur deux « lissée ». Ici, entre deux pixels, on va insérer un pixel dont la valeur est la moyenne des deux voisins. Les dimensions ne sont pas doublées mais à un près.

Exercice 5 : Écrire une fonction qui prend en entrée une image et un entier `n` et retourne la version réduite de l'image d'un facteur `n`. La réduction se fera en remplaçant tous les rectangles de taille `(n, n)` par le / un pixel central du rectangle.

Exercice 6 : Écrire une fonction qui prend en entrée une image et un entier `n` et retourne une version réduite de l'image d'un facteur `n` de meilleure qualité. La réduction se fera en remplaçant tous les rectangles de taille `(n, n)` par un pixel dont les nuances des trois couleurs sont les moyennes des nuances des trois couleurs de tous les pixels du rectangle.

Exercice 7 : Écrire une fonction qui prend en entrée une image et qui retourne la version en nuances de gris de cette image. La nuance de gris sera la moyenne des nuances de rouge, vert et bleu. Pour que l'image reste exploitable, on reportera trois fois la nuance de gris. Observer le résultat.

Exercice 8 : Écrire une fonction qui prend en entrée une image en nuances de gris ainsi qu'un seuil et qui retourne la version binarisée de l'image. Il s'agit de mettre en noir tous les pixels d'intensité inférieure au seuil et en blanc les autres.

Exercice 9 : Écrire une fonction qui prend en entrée une image et qui retourne la version floutée de l'image. Pour les trois couleurs, la nuance d'une case devient la moyenne de la nuance initiale et de celle des huit cases voisines (en gérant le cas des bords).

Exercices pour réfléchir après la séance

Exercice 10 : Écrire une fonction qui prend en entrée une image et retourne la version où les couleurs sont inversées (en utilisant la fonction qui à une nuance `x` associe la nuance `maximum - x`). Observer le résultat sur une image en couleurs ainsi que sur sa version en noir et blanc.

Exercice 11 : Écrire trois fonctions d'extraction des couleurs qui prennent en entrée une image et retournent la version où deux des couleurs, toujours les mêmes, sont mises à zéro pour tous les pixels. Observer le résultat.

Exercice 12 : Comparer le rendu des fonctions des exercices 3 et 4. Si on ne voit pas de différence, reprendre les exercices en remplaçant deux par un entier plus grand, le lissage utilisant alors des moyennes pondérées[8]. En fait non, le faire de toute façon avec l'entier en paramètre, tiens !

Exercice 13 : Rassembler réduction et agrandissement (dans cet ordre...) pour obtenir une fonction de pixellisation.

Exercice 14 : Écrire une fonction de détection de contours d'une image. On appliquera ce qu'on appelle le filtre de Sobel.

Cette dernière fonction, comme celle de l'exercice 9, correspond à une convolution d'un extrait de l'image avec un certain filtre.

Cette fois-ci, il s'agit de remplacer, pour le travail dans une direction, chaque pixel (en gérant également les bords) par la somme pondérée des pixels voisins dans un carré de côté trois, avec une pondération d'un pour les deux pixels des coins de droite, deux pour le pixel de droite, l'opposé pour les trois pixels de gauche et rien pour la même colonne.

Pour l'autre direction, on échange les travaux sur les lignes et sur les colonnes.

Enfin, on fait la moyenne quadratique des pixels obtenus.

8. dire « barycentres » me démange !

TP 8 : Tris

Avant de commencer... la théorie

Dans un TP précédent, nous avons abordé la dichotomie, qui apportait un avantage considérable en termes de temps de calcul, mais qui nécessitait des conditions particulières sur les données. Ainsi, une liste est plus pratique si elle est triée. Nous allons désormais étudier divers algorithmes de tri dans cette optique.

Les tris de ce TP seront toujours dans l'ordre croissant, en gardant à l'esprit qu'il est immédiat de passer à la version décroissante, voire d'ajouter un booléen en argument précisant dans quel ordre le tri doit se faire (c'est le cas pour la méthode `sort`, qui a un argument optionnel).

Comparaisons

La plupart des tris classiques agissent par comparaison, c'est-à-dire que l'on regardera deux à deux certains couples d'éléments, et on prendra une décision en fonction de l'élément qui est le plus grand des deux. En quelque sorte, parmi les $n!$ permutations d'une liste de n éléments distincts, une comparaison permet d'éliminer une moitié : celles qui ne donnent pas le même résultat pour la comparaison en question. Le meilleur tri par comparaison fait donc une sorte de dichotomie parmi l'ensemble des permutations candidates restantes, mais a priori jamais de manière explicite.

Les tris par comparaison les plus simples seront en temps quadratique, en comptant comme unité de temps la comparaison (ou l'affectation, intuitivement plus coûteuse). C'est moins bien que le tri rapide et que le tri fusion, mais ces derniers ne sont pas aussi aisés à comprendre.

Il existe également des tris n'agissant pas par comparaison. Ils peuvent être plus simples, voire plus efficaces, mais nécessitent souvent des conditions particulières. Nous verrons au cours de la séance le plus classique d'entre eux : le tri par dénombrement.

Tris en place

Un tri est dit en place si toute l'exécution de l'algorithme utilise une mémoire constante, matérialisée par exemple par les éléments sauvegardés dans l'optique d'un échange, les variables représentant les indices, etc. En particulier, un tri en place agit par effet de bord car il ne peut pas se permettre de construire une liste à retourner.

Attention : un tri qui construit la liste à retourner puis remplace dans la mémoire la liste en entrée par la liste à retourner aura certes agi par effet de bord, mais ne sera pas en place.

On peut écrire la plupart des tris en place ou non en place, sachant que la version en place n'est pas toujours aussi facile à écrire (surtout pour le tri rapide), voire à proscrire (cas du tri fusion). Parfois c'est étonnamment l'inverse qui se passe (le tri par sélection est plus agréable en place, et des bugs menacent si on ne prend pas assez garde dans la version non en place).

Toujours est-il qu'un tri en place sera plus rapide en raison des allocations de mémoire qu'on économise.

Stabilité

Un tri est dit stable si deux éléments égaux au regard de la fonction de comparaison se retrouvent dans le même ordre dans la version initiale et dans la version triée de la liste.

Cette propriété n'est pas fondamentale si la relation de comparaison ne peut pas considérer comme égaux des éléments qui peuvent être distingués, mais peut présenter un intérêt si le tri n'étudie par exemple qu'un élément de chaque n-uplet d'une liste.

On se posera la question de la stabilité de tous les tris écrits au cours de la séance.

Manipulations à faire pour la prise en main

Ouvrir avec Python le fichier `http://jdreichert.fr/Enseignement/CPGE/ITC%20SUP/liste.py` et stocker son contenu dans la variable `l`. Attention à ne pas ouvrir le fichier avec un éditeur de texte, il est composé d'une seule ligne et occasionnerait potentiellement un plantage. Ce sera l'occasion de réviser la façon d'importer du contenu. La liste du fichier est une liste croissante qui a subi quelques modifications, à savoir une dizaine d'échanges et de retraits d'éléments. Sa taille est de l'ordre de dix mille.

Créer aussi deux listes `l_4000` et `l_8000` qui sont des mélanges (utiliser `random.shuffle`) de `range(4000)` et `range(8000)`, respectivement. Ces listes ne seront pas à modifier sur l'ensemble du TP, donc les tris avec effet de bord devront agir sur des copies.

On comparera les temps d'exécution des tris pour voir apparaître le temps quadratique des tris par insertion et par sélection, en l'occurrence une multiplication par quatre pour passer de la première à la deuxième listes, grossièrement.

Écrire l'algorithme de *tri par insertion* en place et de *tri par sélection* en place ci-après et les tester.

Ces tris consistent à construire la liste triée par ajout successif d'un élément, soit en insérant à chaque fois au bon endroit dans la partie gauche de la liste le premier élément de la partie droite, soit en sélectionnant à chaque fois l'élément minimal parmi ceux qui n'ont pas encore été sélectionnés.

```
def selection_sort(l):
    for i in range(len(l)-1):
# si i = n-1, le minimum est le seul élément restant
        ind_min = i # où est le minimum a priori
        for j in range(i+1, len(l)):
            if l[j] < l[ind_min]: # mise à jour du minimum
                ind_min = j
        if ind_min != i:
# l'avantage du tri par sélection est le faible nombre d'échanges,
# on fait l'hypothèse qu'ils sont plus chers que les comparaisons
            l[i], l[ind_min] = l[ind_min], l[i]
```

```
def insertion_sort(l):
    for i in range(1, len(l)): # on parcourt l
        j = i-1 # position après laquelle
# on va finalement insérer le i-ième élément
        x = l[i] # mémorisation de l'élément
        while j >= 0 and l[j] > x: # recherche de la bonne position
            l[j+1] = l[j] # on décale le reste de la liste
            j -= 1
        l[j+1] = x # insertion proprement dite
```

Exercices

Exercice 1 : Écrire une version non en place des tris par insertion et par sélection. La fonction devra alors avoir une valeur de retour, puisque la liste en argument ne sera pas modifiée.

Exercice 2 : Implémenter le tri cocktail, expliqué ci-après.

Le classique mais très mauvais tri à bulles, dont le principe a été donné au TP 2, pèche par sa complexité, et seules sa compréhension relativement facile et son originalité font qu'il est enseigné. Nous allons ici broder autour. Vérifier si la liste est déjà triée peut se faire de manière très pratique dans le tri à bulles : on utilise un booléen déterminant si un échange a été effectué dans un parcours de la boucle principale. Si le booléen reste à `False`, c'est que la liste est déjà triée.

L'utilité est de ne pas perdre de temps quand la liste de départ est presque triée, dans la mesure où par exemple quelques éléments parmi les plus grands sont au début de la liste. Le principe du tri à bulles fait que de tels éléments seront vite envoyés à leur place, mais le souci est qu'au contraire des éléments parmi les plus petits sont peut-être à la fin de la liste, et ils devront être échangés au cours d'un nombre considérable de parcours de la boucle principale.

Le tri cocktail pallie ce souci en faisant des parcours de la liste dans les deux sens. Après chaque (double) parcours dans le tri cocktail, un élément de plus est à la bonne place aux deux extrémités.

Exercice 3 : Implémenter le tri fusion, expliqué ci-après.

Le *tri fusion* s'adapte à merveille à la récursivité : on prend la moitié gauche et la moitié droite de la liste, on les trie par le tri fusion, puis on les fusionne en prenant à chaque fois l'élément minimum parmi les deux éléments de tête. La fusion proprement dite transforme alors effectivement deux listes triées en leur union triée.

Exercice 4 : Implémenter le tri rapide, expliqué ci-après. Le tri sera en place si possible.

Comme le tri fusion, le *tri rapide* est classiquement récursif et applique le principe de « diviser pour régner ». Ici, les comparaisons se font au moment de créer les deux listes (de taille non contrôlée cette fois) sur lesquelles les appels récursifs doivent être exécutés, ce qui se fait autour d'un pivot qui sera un élément arbitraire (le premier, disons, même si ce choix peut occasionner des temps très longs quand la liste est déjà croissante ou décroissante).

Exercice 5 : Implémenter le tri par dénombrement, expliqué ci-après.

Le **tri par dénombrement** ou **tri comptage** (*counting sort*) permet de trier une liste dont les éléments proviennent d'un ensemble fini, et si possible de taille très petite devant la taille de la liste.

Par exemple, on voudrait trier un million de caractères de la table ASCII, ce qui s'assimile à trier un million de nombres entre 0 et 127.

Pour ce faire, on crée un tableau de taille 128 dont les éléments sont le nombre d'occurrences de chacun des nombres de 0 à 127, que l'on va compter en parcourant la liste. Une fois le parcours terminé, on constitue directement la version triée de la liste.

Exercice 6 : Implémenter une fonction de tri qui applique un algorithme le plus efficace possible sur la liste récupérée dans les manipulations initiales. Ce sera à qui fait le meilleur temps sans tricher !

Troisième partie

Travaux pratiques du deuxième semestre

TP 9 : Gestion de la mémoire, effets de bord.

Ce premier TP du deuxième semestre sera le dernier à avoir des manipulations de prise en main. Il s'agit d'observer des cas de mutations de variables par des fonctions, les fameux effets de bord évoqués dans le chapitre 1, mais aussi de comprendre un peu mieux comment les variables sont gérées.

Effets de bord

Commençons par détailler les effets de bord les plus fréquents :

```
l = [1, 2, 3]
l.append(4) # Et voilà, la liste a changé, l vaut [1, 2, 3, 4] !

print(42) # Quelque chose apparaît sur la console !

import matplotlib.pyplot as plt
plt.plot([2, 0, 2])
plt.show() # Une fenêtre graphique s'ouvre !

fichier = open("toto.txt", "w") # Un fichier est créé / écrasé !
fichier.write("Hello world!") # Et du texte s'y ajoute !
fichier.close() # ... par principe
```

Noter au passage que la manipulation de fichiers est un attendu du programme, mais dans la mesure où il n'y a pas vraiment d'évaluation possible en-dehors de projets éventuellement, aucune séance de cours ni de TP n'y sera consacrée. Il est néanmoins possible de trouver les informations dans la section 1.6 du chapitre 0.

Comme le montre l'exemple de la méthode `append`, une fonction peut prendre en argument une liste et la muter, ce qui peut aussi signifier qu'une variable globale peut être affectée par l'effet d'une fonction, comme le montrent les exemples à suivre.

Il est important d'avoir à l'esprit qu'une fonction ne peut muter qu'un objet mutable, et ne peut agir sur des variables globales non mutables que si on le signale expressément, par le mot-clé `global` suivi du nom de la variable, à déclarer en début de fonction par principe, en tout cas avant la première utilisation de la variable.

Si d'aventure on souhaitait modifier dans une sous-fonction d'une fonction `f` des variables créées dans cette même fonction `f` sans que la mutation soit globale, le mot-clé est alors remplacé par `nonlocal`, avec les mêmes conditions d'utilisation.

Pour se familiariser avec cette syntaxe (hors-programme et donc non exigible et à éviter si on peut s'en passer dans un programme), les derniers exemples laisseront entrevoir la différence suivant l'utilisation ou non des mots-clés en question.

Exercice 1 : Deviner avant d'exécuter les programmes la valeur des variables inspectées.

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

def mute1(l):
    l.append(0)

def mute2(l):
    l += [0]

def mute3(l):
    l = l + [0]

def mute4(): # Pas d'argument
    l.append(0)

def mute5():
    l += [0]

mute1(l) # Deviner ce que vaut l, imprimer et vérifier !
mute2(l) # Deviner ce que vaut l, imprimer et vérifier !
mute3(l) # Deviner ce que vaut l, imprimer et vérifier !
mute4() # Deviner ce que vaut l, imprimer et vérifier !
mute5() # Surprise ! (Même résultat avec l = l + [0] dans la fonction)

def mute10(x):
    x += 1

x = 41

def mute11(): # Ne pas mettre x en argument, sinon boum !
    global x
    x += 1

def mute12(): # x peut être en argument sauf si global
    x = 72
    def mute_ici():
        nonlocal x # Essayer sans, puis en remplaçant par global
        x += 1
    mute_ici()
    print("x intermédiaire :", x) # Deviner ce que vaut x

mute10(x) # Deviner ce que vaut x, imprimer et vérifier !
mute11() # Deviner ce que vaut x, imprimer et vérifier !
mute12() # Deviner ce que vaut x, imprimer et vérifier !
```

Copies de listes et mutations

Le nombre de façons de créer une copie d'une liste n'est limité que par l'imagination, mais nous allons présenter ici les plus classiques.

Exercice 2 : Deviner avant d'exécuter les programmes le contenu des listes `l2` à `l9` après les deux mutations sur `l`.

```
import copy

l = [0, [1, 2], [3, 4], 5]
l2 = l
l3 = l[:]
l4 = l + []
l5 = l * 1
l6 = [x for x in l]
l7 = l.copy()
l8 = copy.copy(l)
l9 = copy.deepcopy(l)
l[0] = 6
l[1][0] = 7
```

Exercice 3 : Écrire une fonction agissant comme `copy.deepcopy` et la tester avec le code précédent.

En pratique, la dépendance de deux variables se déduit des adresses mémoire respectives, qu'on obtient avec la fonction `id`. Consulter dans l'exemple précédent les valeurs de `id(l)`, `id(l2)` et toutes les autres, puis de `id(l[1])`, `id(l2[1])` et toutes les autres, puis de `id(l[1][0])`, `id(l2[1][0])` et toutes les autres.

Les adresses en mémoire des petites valeurs entières (d'après un test, de `-5` à `256` inclus) sont préparées en avance et consécutives, ce qui permet de gagner du temps quand des variables avec de telles valeurs sont créées.

Pour se faire une meilleure idée (mais dans tous les cas savoir ceci n'est pas nécessaire) :

```
x = 35646
print(id(x))
y = x
print(id(y)) # même id vu le y = x
z = 35646
print(id(z)) # pas le même id
xx = 64
print(id(xx))
zz = 64
print(id(zz)) # cette fois-ci c'est le même, car 64 a toujours le même identifiant
print(id(326467))
print(id(395654)) # parfois comme 326467, mémoire aussitôt libérée sans affectation
```

Ce qu'il ne faut pas faire !

L'expérience a montré que de nombreux bugs provenaient de la mutation d'un objet pendant son parcours. Autant que possible, il ne faudrait jamais muter et surtout pas redimensionner une liste dans une boucle inconditionnelle, faute de quoi des erreurs peuvent apparaître, mais aussi des comportements imprévus voire des soucis de terminaison.

Des exemples de codes à bannir :

```
l = [1, 2, 3]
for x in l:
    l.append(x) # boucle infinie

l = [1, 2, 3]
for i in range(len(x)):
    l.pop(i) # erreur d'indexation au dernier tour
# (il ne reste qu'un élément, on tente d'enlever le troisième)

l = [1, 2, 3]
for x in l:
    l.remove(x) # Objectif : essayer de comprendre ce qui s'est passé !
```

À noter cependant, une boucle `for` initialise sa variable à l'élément suivant de l'objet à parcourir à chaque tour, ainsi elle n'est pas perturbable si on manipule cette variable sans toucher à l'objet parcouru.

```
l = [1, 2, 3]
for i in range(len(l)):
    print(l[i])
    i += 2 # totalement ignoré

# Attention cependant :
l = [1, 2, 3]
for i in range(len(l)):
    i += 2 # impact cependant pour tout le tour actuel
    print(l[i]) # erreur d'indexation au deuxième tour
```

Un dernier exemple avancé et hors programme pour quelqu'un qui voudrait utiliser des fonctions avec variables optionnelles, il faut savoir que si une telle variable est mutable, sa valeur par défaut subit les mutations et les garde en mémoire.

```
def f(l = []):
    l.append(len(l))
    return l

f() # [0]
f() # [0, 1]
```

Exercice 4 : Écrire une fonction qui prend en entrée une liste et qui la mute de sorte que son contenu soit ajouté une deuxième fois à la fin.

Exercice 5 : Écrire une fonction qui prend en entrée une liste et qui la mute de sorte que chaque élément soit immédiatement suivi d'une copie de lui-même.

Exercice 6 : Écrire une fonction qui prend en entrée deux listes et qui les mute de sorte que la première devienne la fusion des deux et la deuxième se vide. Le comportement est au choix si les deux listes sont la même variable.

Exercice 7 : Écrire une fonction qui prend en entrée une liste `l` et un entier positif `n` et qui mute `l` de sorte qu'elle soit remplacée par une liste de listes contenant `n` copies **indépendantes** de `l`.

Attention pour ce dernier exercice, la liste `[[1, 2, 3]] * 3` contient trois copies **dépendantes** de `[1, 2, 3]`.

Pour finir, une erreur trop classique à éviter absolument : deviner et comprendre ce qui se passe en écrivant ceci :

```
l = []
l = l.append(0)
l = l.append(1)
```

... et un conseil répété ici (et sans doute répété jusqu'à ce qu'il soit appliqué) : si une fonction mute son argument, inutile de retourner la version mutée, le nom de variable est connu et donc le travail de la fonction ne peut pas se perdre même sans `return`. On pourra envisager d'écrire explicitement `return None` dans ce cas.

TP 10 : Limites de la pratique par rapport à la théorie

Python ne sait pas s'arrêter !

Exercice 1 : On considère la fonction de Morris ci-dessous. Prouver qu'elle est censée toujours retourner `1`. Constater qu'elle ne termine pas quand on appelle `morris(1, 0)` et chercher à expliquer pourquoi.

```
def morris(m, n):
    if m == 0:
        return 1
    else:
        return morris(m-1, morris(m, n))
```

Exercice 2 : On considère la fonction d'Ackermann ci-dessous. Prouver qu'elle termine. Déterminer par le calcul (avec une récurrence) la valeur retournée en fonction de `n` quand `m` vaut successivement les entiers de `0` à `3`. Deviner ce qui se passe au-delà et tester.

```
def ackermann(m, n):
    if m == 0:
        return n+1
    elif n == 0:
        return ackermann(m-1, 1)
    else:
        return ackermann(m-1, ackermann(m, n-1))
```

Python ne sait pas calculer !

Exercice 3 : Trouver trois valeurs `x`, `y` et `z` telles que le résultat de la commande Python `x + (y + z)` soit différent de celui de `(x + y) + z`. Trouver aussi deux valeurs `x` et `y` telles que le résultat de la commande Python `int(x/y)` soit différent de celui de `x // y`.

Exercice 4 : Consulter la documentation de la fonction `round`. Noter la convention surprenante pour l'arrondi à l'unité des nombres entiers plus un demi. Demander l'arrondi au dixième près des nombres `1.05`, `10.05`, `100.05` et `1000.05`.

Pour cet exercice, une clarification sera donnée dans un TP ultérieur.

Exercice 5 : Anticiper le résultat de `1 - 1./3 - 1./3 - 1./3` et de `1 - 0.2 - 0.2 - 0.2 - 0.2 - 0.2`[9]. Qu'en serait-il de `1 - 0.25 - 0.25 - 0.25 - 0.25` ?

Exercice 6 : Écrire une fonction qui retourne le discriminant d'un polynôme du second degré à partir des trois coefficients `a`, `b` et `c`. Écrire ensuite une fonction qui retourne l'ensemble des solutions réelles d'une équation polynomiale définie par les trois mêmes coefficients. Tester la fonction pour les fonctions $x^2 + 1.4x + 0.49$, $x^2 + 0.2x + 0.01$ et $x^2 + x + \frac{1}{4} + 10^{-20}$. Commenter.

9. Cela ne marcherait pas avec `1 - 5 * 0.2` !

L'intérêt de ne pas faire de tests d'égalité sur des réels se dessine. Malheureusement, au vu du troisième exemple, on ne peut pas non plus se permettre d'assimiler à zéro une valeur certes proche mais différente. Les logiciels de calcul formel tentent de pallier ce genre de problèmes que l'on rencontre lorsque l'on manipule des réels comme ici sans outils adaptés. D'ailleurs, demander le sinus de π donnera un argument supplémentaire pour se méfier.

Exercice 7 : Exécuter les deux lignes suivantes dans la console et les expliquer.

```
n = 2**61 - 2
n < n - 1.0
```

Python ne sait pas... qu'il ne doit pas s'arrêter !

Exercice 8 : Prouver que le code ci-dessous ne doit pas terminer. L'exécuter. Constater qu'il termine tout de même. Compter le nombre de tours de boucle utilisés en modifiant le code.

```
x = 1
while x != 0:
    x /= 2
```

Exercice 9 : Même exercice mais en écrivant un code où la valeur de départ de `x` est `2` et on fait tendre `x` vers `1`. Compter de nouveau le nombre de tours de boucle.

Exercice 10 : Encore la même chose avec `101` et `100`.

Bien entendu, les problèmes soulevés ici ne dépendent pas du langage de programmation a priori.

TP 11 : Un peu d'amusement avec les bases numériques

En parallèle du cours sur la représentation des nombres en informatique, ce TP propose dans un premier temps de travailler les calculs de conversion, et par la suite quelques exercices originaux autour des bases numériques.

Conversions

La technique présentée ici fonctionne pour passer de n'importe quelle base à n'importe quelle autre. Pour autant, le passage préalable par la base 10 est un détour rassurant que la majorité choisira.

Sachant que la conversion de la partie entière et de la partie fractionnaire peut se faire séparément, on sépare dans un souci de simplification le travail en deux étapes analogues, dont le résumé est de procéder à des multiplications par la base à un certain exposant jusqu'à avoir un nombre entre 1 (inclus) et la base (exclue), puis de tronquer la partie entière qui est un nouveau chiffre dans l'écriture en base b, en reportant les éventuels zéros rencontrés. Cela suit le principe mathématique qui permet d'isoler chaque chiffre d'un nombre, non nécessairement entier, en base 10.

Ainsi donc, soit l'entier x qu'on veut écrire en base b. Le chiffre des unités en base b de x est x modulo b, c'est-à-dire le reste dans la division euclidienne.

Une fois ce chiffre écrit, on considère à présent le quotient de x par b et on recommence jusqu'à ce que le nombre considéré devienne 0.

On remarquera que la base dans laquelle x est écrit n'est jamais mentionnée, il convient simplement de faire les divisions par b dans la base dans laquelle x est écrit si on ne veut pas le convertir d'abord en décimal.

Quant à la conversion de la partie fractionnaire d'un nombre, soit y entre 0 (inclus) et 1 (exclu). Ce nombre s'écrira $0,\ldots$ en base b. Le premier chiffre après la virgule, et tous ceux qui suivent jusqu'à épuisement (ou jusqu'à tomber sur zéro), s'obtient ainsi : on multiplie y par b, on reporte la partie entière puis on recommence en considérant la partie fractionnaire.

Exemple : On veut écrire 10π en binaire sur 16 bits, en écriture scientifique tant qu'à faire ; le passage à l'écriture en virgule flottante est une étape supplémentaire mais indépendante de cette technique.

On anticipe le nombre de chiffres à utiliser dans l'approximation de π en base $10\ldots$ au hasard 11.

Partons donc de $31,415926535$. La partie entière est 31, ce qui est congru à 1 modulo 2, donnant 1 pour bit des unités.

On continue avec le quotient, qui est 15. En pratique, on va avoir cinq fois de suite un reste de 1 dans les divisions euclidiennes effectuées, jusqu'à tomber sur 0, ce qui donne $\overline{11111}^2$ pour partie entière.

Passons à la partie fractionnaire, à savoir $0,415926535$. On va détailler les étapes :

- $0,415926535 \times 2 \simeq 0,83185307$ (on laisse tomber le dernier chiffre en raison de la perte de précision), on écrit le 0 ;
- $0,83185307 \times 2 \simeq 1,6637061$, on écrit le 1 et on le retire ;
- $0,6637061 \times 2 \simeq 1,327412$, on écrit le 1 et on le retire ;
- $0,327412 \times 2 \simeq 0,65482$, on écrit le 0 ;
- $0,65482 \times 2 \simeq 1,3096$, on écrit le 1 et on le retire ;
- $0,3096 \times 2 \simeq 0,619$, on écrit le 0 ;
- $0,619 \times 2 \simeq 1,23$, on écrit le 1 ;
- $0,23$ étant proche d'un quart tout en restant inférieur, on peut déduire pour accélérer qu'il y aura ensuite deux 0 et au moins trois 1, donc on arrête le travail ici et on écrit notre nombre final : $\overline{11111,011010100111\ldots}^2$ (sans décider de l'arrondi).

Exercice 1 : Écrire une fonction permettant de convertir n'importe quel nombre entier positif `n` en argument du décimal à une base aussi en argument, la fonction devant alors produire la liste des coefficients de l'expression du nombre `n` dans la base d'arrivée.

Exercice 2 : Écrire une fonction prenant en argument une liste telle que produite dans la question précédente ainsi que la base dans laquelle la liste a été obtenue et renvoyant une chaîne de caractères dont les caractères correspondent aux coefficients dans le même ordre, avec la correspondance `0` $\to$ `'0'`, ..., `9` $\to$ `'9'`, `10` $\to$ `'A'`, ..., `35` $\to$ `'Z'`. Si la base n'est pas entre `2` et `36`, la fonction doit retourner une erreur. Tester la fonction `int(s, b)` sur la chaîne `s` obtenue avec la base en question, qui devrait renvoyer le nombre en argument de la fonction de l'exercice précédent.

Exercice 3 : Écrire finalement une fonction agissant comme `int(s, b)`.

Bases exotiques

Exercice 4 : le négabinaire. Il s'agit de la base -2, qui a été expérimentée vers le milieu du vingtième siècle. Le principe est le même que dans une base positive, à ceci près que les coefficients ne vont pas de 0 à $b-1$, mais ici il s'agit de 0 et 1. Ainsi, on représente le nombre 3 par $\overline{111}^{-2}$, à savoir une fois $(-2)^2$ plus une fois -2 plus une fois 1. Représenter 42 en négabinaire, et déterminer quel nombre est représenté par n occurrences de 1 pour tout entier naturel n. Écrire ensuite une fonction de conversion du décimal vers le négabinaire et une fonction réciproque, pour les entiers.

Exercice 5 : le quater-imaginaire. Il s'agit cette fois de la base $2i$, avec les coefficients 0, 1, 2 et 3. On représente les nombres réels en mettant un 0 à toute position paire en partant de la droite (donc une fois ces zéros retirés on obtient en fait un analogue de la base -4, d'où l'utilisation de quatre coefficients et non deux), et les imaginaires purs en mettant un 0 à toute position impaire en partant de la droite. Ainsi le nombre i s'écrira $\overline{10,2}^{2i}$, à savoir une fois $2i$ plus deux fois $\frac{1}{2i}$, et le nombre -1 s'écrira $\overline{103}^{2i}$, soit une fois $(2i)^2$ plus trois fois 1. Représenter 42 en quater-imaginaire et déterminer quel nombre est représenté par $\overline{3210123}^{2i}$. Écrire ensuite une fonction de conversion du décimal vers le quater-imaginaire et une fonction réciproque, non limitées aux entiers cette fois.

Exercice 6 : la numération de Cauchy (voir aussi : le système d'Avizienis). Il s'agit de la base 10, dont les coefficients ne vont plus de 0 à 9 mais de -5 à 5. L'écriture est facilitée pour éviter les confusions et la lourdeur : un coefficient négatif (hors 0) sera écrit comme sa version positive surmontée d'une barre horizontale. Cette écriture a l'inconvénient de ne pas être unique (retirer -5 serait plus pratique, mais qui suis-je pour contredire ou critiquer Cauchy ?), par exemple on peut écrire 15 tel quel ou en tant que $2\overline{5}$ (deux fois 10 plus -5). Représenter 46472 avec la numération de Cauchy et déterminer quel est le plus petit nombre entier strictement positif représentable en quatre caractères (utiles) avec cette numération. Écrire ensuite une fonction de conversion du décimal vers la numération de Cauchy, en produisant une chaîne de caractères où les coefficients négatifs seront effectivement assortis d'un signe moins, et une fonction réciproque, là aussi non limitées aux entiers.

L'hydre de Lerne

Ce nom faisant référence à la mythologie grecque est associé aux suites de Goodstein, qui comme le nombre de têtes de la créature en question semble croître démesurément, mais finit par atteindre zéro, sans cautérisation des blessures mais par la magie d'une preuve de terminaison à venir.

Une suite faible de Goodstein, caractérisée entièrement par son premier élément, se construit ainsi : soit u_0 son premier élément. On écrit u_0 en binaire, ce qui correspond à une séquence de 0 et de 1 qu'on écrit s_0. Le terme suivant u_1 se calcule en considérant s_0 comme la représentation du nombre $u_1 + 1$ désormais en base 3, et la représentation s_1 du nombre u_1 est donc s_0 retranché d'une unité avec création et propagation éventuelles d'une retenue (en base 3 donc). Plus généralement, le terme u_n ayant une représentation notée s_n en base $n + 2$, on obtient s_{n+1} en retirant une unité à s_n désormais vu comme la représentation d'un nombre en base $n + 3$, ce nombre étant la valeur qu'on donne à u_{n+1}, et dans le cas où $u_n = 0$, on fait stationner la suite (pas de valeurs négatives).

Ainsi, si le nombre de départ est 42, ce qui s'écrit $2^5 + 2^3 + 2^1$, le terme suivant sera un de moins que $3^5 + 3^3 + 3^1$, soit au total 272, qui s'exprime $3^5 + 3^3 + 2 \times 3^0$, et le nombre encore après sera un de moins que $4^5 + 4^3 + 2 \times 4^0$, soit 1089.

Une suite de Goodstein reprend ce principe, mais l'écriture des exposants elle-même ne devra pas contenir de chiffres interdits dans la base courante. Le nombre 42 s'écrira alors $2^{2^{2^1}+1} + 2^{2^1+1} + 2^1$ et le terme suivant sera un de moins que $3^{3^{3^1}+1} + 3^{3^1+1} + 3^1$, soit… 22876792455044. Nous laisserons ces suites de côté pour ce TP.

Exercice 7 : Prouver qu'une suite faible de Goodstein converge vers 0.

Exercice 8 : Écrire une fonction qui prend en entrée un entier et qui calcule la suite faible de Goodstein associée. Comme les données sont a priori très lourdes, retourner l'ensemble des valeurs est une mauvaise idée. On pourra imprimer les valeurs successives si l'on souhaite, mais ce qui sera intéressant est de retourner le nombre d'étapes nécessaires pour atteindre zéro ainsi que la plus grande valeur rencontrée entre temps.

TP 12 : Représentation des nombres - manipulations

Le comportement de l'interpréteur Python dépendra souvent dans ce TP de l'architecture de la machine ainsi que de la version de Python utilisée. Lorsque vous lancez Python, la version est précisée dans la première ligne de l'interpréteur, et elle est rappelée lorsque la fonction `help()` est utilisée sans argument.

Depuis la version 3 de Python, les types `int` et `long` sont fusionnés, ce qui ne laisse pas observer de dépassement arithmétique. On peut cependant forcer l'apparition d'entiers, signés ou non, sur un nombre limité de bits par des fonctions de conversion issus du module `numpy`, par exemple.

Diverses informations, notamment les seuils pour la virgule flottante, sont visibles en demandant `float_info` (dans le module `sys`, donc il faut au préalable écrire `from sys import float_info` par exemple). Quelques-unes de ces informations sont relativement faciles à comprendre.

Il est possible de faire de l'arithmétique dans les bases les plus usuelles en informatique (soit 2 et 16) en préfixant les nombres par `0b` (forçant le binaire) ou `0x` (forçant l'hexadécimal). Cependant, la réponse est donnée en base 10 par défaut. On peut cependant effectuer des conversions à l'aide des fonctions `bin` et `hex`, qui retournent des chaînes de caractères.

Attention : il n'est alors pas possible de faire des opérations sur les chaînes obtenues, précisément parce qu'elles ne sont plus des entiers. En fait, additionner `'0b10'` et `'0b1'` donnera `'0b100b1'`, soit la concaténation des chaînes.

La fonction `int` avec deux paramètres est interdite dans ce TP, car le programme qu'on demande dans l'exercice 2 revient exactement à réécrire `int(a,0)`. Il reste autorisé de convertir directement une chaîne en entier ou de tronquer un flottant si vraiment c'est nécessaire.

Exercice 1 : Faire quelques calculs en binaire et en hexadécimal, jusqu'à pouvoir prédire le résultat et savoir quelles expressions renvoient une erreur.

Pour l'exercice précédent, quelques suggestions...

- `0b110 * 0b111`
- `bin(63) * bin(10)`
- `0b1011 * 0xBACA`
- `0x1011 * 0bBACA`

Exercice 2 : Écrire une fonction qui transforme une chaîne de caractères obtenue à l'aide de `bin` ou de `hex` en le nombre qu'elle représente.

Exercice 3 : Écrire une fonction qui convertit un nombre flottant en binaire. L'algorithme est présenté dans le TP précédent. Un argument de la fonction sera le nombre maximal de bits à reporter après la virgule.

Tester cette fonction pour les nombres `1.05`, `10.05`, `100.05` et `1000.05` en demandant 52 chiffres après la virgule, puis en regardant le 52^{e} bit après le premier 1 (qui est avant la virgule). Comprendre l'arrondi qui avait été fait jusque là.

TP 13 : Manipulations de graphes

Le but de ce TP est de créer des graphes en Python, en utilisant différents types suivant la représentation.

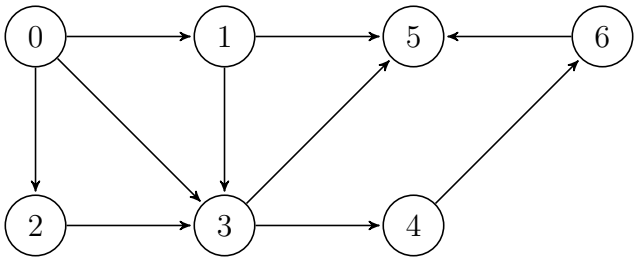
Il faut savoir que la programmation orientée objet, qui n'est pas au programme, permettrait de créer un type très pratique et efficace. Pour quelqu'un qui maîtriserait ce domaine par ailleurs, il est tout à fait acceptable de s'en servir à la place des suggestions de type ci-après.

Pour la représentation donnant explicitement les sommets et les arêtes, nous allons utiliser un couple `(S, A)`, où `S` et `A` seront des listes, `S` contenant des chaînes de caractères (ou des entiers naturels si possible dans l'ordre) et `A` contenant des couples d'éléments de `S`.

Pour la représentation par liste d'adjacence, nous allons utiliser un dictionnaire `g` indexé par des chaînes de caractères, chaque élément du dictionnaire étant une liste de chaînes de caractères. Si les sommets du graphe sont censés être les `n` premiers entiers naturels, le dictionnaire peut être remplacé par une liste de listes.

Pour la représentation par matrice d'adjacence, ce sera naturellement une liste de listes de booléens qui sera employée, ceci est moins alambiqué que d'utiliser un dictionnaire indexé par des chaînes de caractères, chaque élément étant lui-même un dictionnaire de booléens indexé par les mêmes chaînes de caractères.

Exercice 1 : Créer pour les trois représentations possibles une variable correspondant au graphe ci-dessous, version non pondérée du graphe du TD 3.



Bien entendu, ce graphe servira à tester les fonctions suivantes.

Exercice 2 : Écrire des fonctions transformant la représentation d'un graphe selon chaque type en la représentation du même graphe selon chaque autre type.

Exercice 3 : Écrire pour chaque représentation une fonction qui prend en entrée un graphe et retourne le même graphe avec un sommet en plus. Si les sommets sont censés être des chaînes de caractères, il faudra donner le nom du nouveau sommet en argument, et la fonction devra retourner une erreur si ce nom de sommet existe déjà.

On observe pour les exercices suivants l'intérêt d'avoir la liste des sommets dans la définition du graphe, afin de détecter l'existence en évitant le parcours d'une liste comportant un nombre quadratique d'éléments par rapport au nombre de sommets.

Exercice 4 : Écrire pour chaque représentation une fonction qui prend en entrée un graphe et retourne le même graphe avec un arc en plus entre deux sommets précisés. La fonction doit déclencher une erreur si les sommets n'existent pas encore ou si l'arc existe déjà.

Exercice 5 : Écrire pour chaque représentation une fonction qui prend en entrée un graphe et retourne le même graphe dont l'arc entre deux sommets précisés a été retiré. La fonction doit retourner une erreur si l'arc n'existe pas (à plus forte raison si les sommets n'existent pas).

Exercice 6 : Écrire pour chaque représentation une fonction qui prend en entrée un graphe et retourne le même graphe dont un sommet précisé a été retiré. La fonction doit retourner une erreur si le sommet n'existe pas.

Exercice 7 : Écrire pour la représentation au choix une fonction qui prend en entrée un graphe orienté et qui détermine s'il admet un chemin eulérien.

Exercice 8 : Écrire pour la représentation au choix une fonction qui prend en entrée un graphe orienté et une liste de sommets et qui détermine si cette liste correspond à un chemin hamiltonien.

Exercice 9 : Écrire pour la représentation au choix une fonction qui prend en entrée un graphe orienté et qui détermine s'il admet un chemin hamiltonien.

Pour ce dernier exercice, il ne faut pas s'inquiéter de la complexité, qui est a priori exponentielle.

TP 14 : Parcours de graphes, algorithme de Dijkstra

Les algorithmes au programme sur les graphes ont été écrits en pseudo-code délibérément pour donner l'occasion de les programmer soi-même en Python, occasion saisie dans ce TP.

Les types réalisant la structure de graphe peuvent différer grandement d'un exercice à l'autre. On se resservira du TP précédent.

Les algorithmes du programme

Exercice 1 : Choisir un ou plusieurs modules proposant une implémentation des structures de pile, de file et de file de priorité en Python. Lire la documentation sur internet ou avec la fonction `help`. Pour le choix du module, une recherche sur internet est également possible.

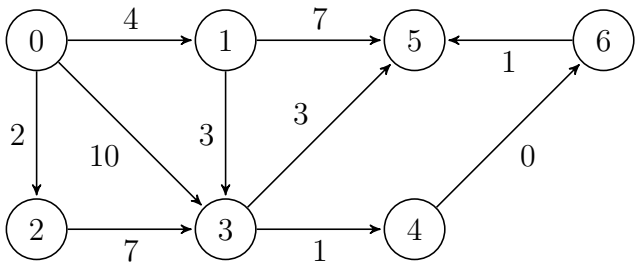
Le module retenu dans le premier exercice pour chaque structure sera à utiliser dans la suite du TP.

Exercice 2 : Écrire l'algorithme du parcours en profondeur en utilisant une pile.

Exercice 3 : Écrire l'algorithme du parcours en profondeur en utilisant une file.

Exercice 4 : Écrire l'algorithme de Dijkstra en utilisant une file de priorité.

Ces algorithmes pourront être testés sur le graphe du TD 3, en ne tenant compte de la pondération que pour l'exercice 4, dont la représentation graphique est rappelée ci-après :



Applications

Exercice 5 : Écrire pour la représentation au choix une fonction prenant en entrée un graphe non orienté, qu'on peut supposer orienté avec les arcs existant dans les deux sens et qui détermine s'il est connexe.

Exercice 6 : Écrire pour la représentation au choix une fonction prenant en entrée un graphe orienté et qui détermine s'il est fortement connexe.

Exercice 7 : Écrire pour la représentation au choix une fonction prenant en entrée un graphe orienté et qui détermine s'il admet un circuit.

Exercice 8 : Écrire pour la représentation au choix une fonction prenant en entrée un graphe non orienté et qui détermine s'il admet un cycle, en prenant bien garde à ne pas compter un éventuel cycle artificiel faisant un aller et retour par la même arête.

Un dernier exercice permet de présenter l'algorithme A*, introduisant à la notion d'heuristique au programme de deuxième année.

Comme on peut l'avoir vu sur des exemples, il n'y a pas forcément unicité du chemin le plus court ni du chemin de poids minimal (en cas d'existence en ce qui concerne ce dernier). En cas d'égalité, un critère majeur pour les algorithmes que nous avons écrit est l'ordre d'apparition des voisins dans la liste d'adjacence (structure principalement utilisée, mais ce serait analogue avec une autre). Changer l'ordre, donc changer l'entrée de l'algorithme en pratique, donnerait éventuellement un autre chemin de même poids à l'arrivée.

Nous allons ici considérer un ordre guidé par l'intuition, et même changer la priorité dans l'algorithme de recherche de plus court chemin, afin de laisser de côté des sommets de distance actuellement moindre mais moins susceptibles d'appartenir au plus court chemin. Pour ne pas complexifier les choses plus que nécessaires, on supposera les graphes non pondérés et les sommets comme des coordonnées du plan, sachant qu'un arc relie deux sommets si, et seulement si, ils sont à une unité de distance en considérant la distance euclidienne, c'est-à-dire que l'une des coordonnées est égale entre les deux sommets et l'autre diffère d'un.

Dans ce cas, si le sommet de départ est (x_0, y_0) et le sommet qu'on cherche à atteindre est (x_f, y_f), on attribuera à un sommet déjà découvert (x, y) dont la distance **a priori** est d un score de $d + |x_f - x| + |y_f - y|$, c'est-à-dire qu'on admet que si tout va bien on peut faire le reste du trajet en ligne droite. Parmi tous les sommets non encore traités, ceux de score minimal seront traités en priorité, même si la distance au sommet de départ est plus grande que d'autres sommets à traiter.

Une fois le sommet qu'on cherche à atteindre trouvé, l'algorithme s'arrête et admet avoir trouvé un chemin optimal.

L'algorithme ainsi décrit se compare ainsi à ceux du cours :

- Il va en moyenne bien plus vite car il ignore des chemins considérés comme défavorables.
- Il est plus compliqué à mettre en œuvre qu'un parcours en largeur, mais pas beaucoup plus une fois qu'on maîtrise l'algorithme de Dijkstra.
- Il peut ne pas retourner un chemin optimal s'il s'arrête selon les conditions annoncées.
- L'optimisation est anéantie sur un graphe pour lequel le seul chemin possible consiste à s'éloigner alors qu'il est possible de s'approcher de l'objectif sans pouvoir l'atteindre.

Exercice 9 : Implémenter l'algorithme A*.

TP 15 : Autres algorithmes sur les graphes

Dans ce dernier TP, quelques applications des graphes sont présentées avec des algorithmes à écrire. L'objectif est de comprendre que beaucoup de situations de la vie réelle se modélisent par des graphes, sachant qu'un informaticien réalisant ces modélisations aura éventuellement recours à des graphes avancés avec des étiquetages plus élaborés que ce qui est fait en tronc commun, voire en option en deuxième année.

L'ordre d'introduction des notions ne correspondant pas à la difficulté des exercices, il est recommandé de lire les sections normalement, mais de traiter la troisième avant la deuxième.

Le lièvre et la tortue

Considérons une fonction f d'un ensemble fini E, de taille notée k, dans lui-même, et supposons sans perte de généralité que E est l'ensemble des entiers naturels de 0 à $k-1$. Soit $(u_n)_{n\in\mathbb{N}}$ une suite récurrente définie par $u_0 \in E$ quelconque et $\forall n \in \mathbb{N}, u_{n+1} = f(u_n)$.

Il s'avère qu'il existe deux entiers $l < m \leq k$ tel que $u_l = u_m$, car les $k+1$ premières valeurs de la suite sont prises dans un ensemble de taille k et donc le principe des tiroirs s'applique. Il s'avère aussi que la sous-suite démarrant à l'indice l est périodique. La question est de détecter la taille de cette période.

La modélisation de la fonction par un graphe vient assez bien : l'ensemble S des sommets est E, et on relie par un arc tout sommet à son image par f. La question devient la détection d'un cycle dans le graphe, cycle qui soit de plus accessible depuis un sommet u_0 choisi.

Robert Floyd (co-auteur de l'algorithme de Floyd-Warshall) a suggéré un algorithme désormais connu sous le nom d'algorithme du lièvre et de la tortue. Il s'agit de calculer simultanément u_n et u_{2n}, en s'arrêtant dès qu'on a $u_n = u_{2n}$ avec $n > 0$. Dans ce cas, on aura découvert l'existence d'une période de taille n dans la suite. Ceci étant, atteindre le cycle peut avoir mis un certain temps, et il est possible que la taille de celui-ci ne soit pas la taille de la période détectée mais un diviseur strict, donc on continue le calcul jusqu'à la prochaine égalité, donnant cette fois-ci la vraie taille du cycle.

Exercice 1 : Implémenter cet algorithme.

Exercice 2 : Proposer un autre algorithme pour détecter un cycle, algorithme consistant par exemple à relever pour tout élément du chemin en construction le nombre de passages à l'aide d'un dictionnaire.

Exercice 3 : Comparer l'efficacité des deux algorithmes en termes de complexité en temps et en espace.

Couplages dans un graphe biparti

Un graphe biparti est un graphe (S, A), orienté ou non, tel que S puisse être partitionné en $S_1 \cup S_2$ de sorte que tout arc / toute arête relie forcément un sommet de S_1 et un sommet de S_2. On parlera ici pour simplifier des deux côtés du graphe.

Un graphe biparti est dit complet si tous les sommets de S_1 et tous les sommets de S_2 sont reliés, par une arête dans le cas non orienté ou par deux arcs, un pour chaque sens, dans le cas orienté.

Un couplage dans un graphe biparti dont les côtés sont de même taille est une liste de couples issus de $S_1 \times S_2$ telle que chaque élément de S_1 (resp. S_2) apparaisse comme premier (resp. deuxième) élément d'exactement un couple dans la liste.

Le problème des mariages stables, étant donné un graphe biparti muni d'une fonction qui à tout élément de S_i associe une permutation des éléments de S_{3-i} vue comme une liste ordonnée des préférences, consiste à rechercher un couplage du graphe tel qu'il n'existe pas de couple (s_1, s_2) instable, l'instabilité signifiant qu'il existe un autre couple (s'_1, s'_2) pour lequel s_1 préfère s'_2 à s_2 et s'_2 préfère s_1 à s'_1. La mise en situation est assez intuitive pour ne pas être détaillée.

Ici, créer un graphe biparti n'est pas nécessaire, mais si le nombre de sommets est assez limité cela permet une représentation alternative du problème, en étiquetant les arcs par un entier signalant la hiérarchie pour chaque sommet (on aura donc un graphe orienté biparti complet).

Pour résoudre le problème des mariages stables, David Gale et Lloyd Shapley ont proposé un algorithme en 1962, dont l'application à l'économie a valu en 2012 à Lloyd Shapley (David Gale étant décédé) de partager le prix de la Banque de Suède en sciences économiques (faussement appelé « Nobel ») avec Alvin Roth. Cet algorithme a été appliqué pour l'affectation dans les formations post-bac par APB et son successeur Parcoursup.

Le principe est le suivant : considérer tous les éléments de S_1 (choix arbitraire, il correspond dans les algorithmes mentionnés ci-avant aux établissements) comme disponibles a priori et prêts à proposer un couplage avec l'élément de S_2 (les étudiants) en tête de leur liste respective.

Dans chaque tour de boucle, tous les éléments s de S_1 qui ne sont pas dans un couple vont être appariés à l'élément t de S_2 en tête de leur liste, à condition que t ne soit pas dans un couple (le couple est formé) ou soit dans un couple (s', t) mais t préfère s à s' (s' est de nouveau disponible et retire t de sa liste puis le couple est formé), autrement s retire t de sa liste et reste disponible.

À noter : il est possible que dans un même tour de boucle un élément de S_1 soit mis dans un couple puis rendu à nouveau disponible.

Au fur et à mesure de l'exécution de l'algorithme, le nombre d'éléments de S_2 dans un couple croît, les préférences des éléments de S_2 dans un couple sont de plus en plus satisfaites (les étudiants ont un établissement qui leur plaît de plus en plus) et les préférences des éléments de S_1 sont de moins en moins satisfaites (les établissements progressent vers le fond de leurs listes d'admission).

Exercice 4 : Implémenter l'algorithme de Gale-Shapley.

Exercice 5 : Prouver la terminaison et la correction de cet algorithme.

Un autre problème de couplage est le problème d'affectation, que nous présenterons ici dans la version la plus simple possible.

Il s'agit de répartir n tâches parmi n agents (une version consiste à disposer d'au moins autant d'agents que de tâches), chaque agent ayant une réticence représentée par un entier (une autre interprétation peut être le temps qu'il met) à faire les tâches, et le but est que la répartition minimise la somme des valeurs associées à chaque couple agent / tâche sélectionné.

Pour résoudre ce problème, Harold Kuhn a proposé ce qu'il a appelé la méthode hongroise, en hommage aux mathématiciens Dénes Kőnig et Jenő Egerváry, dont les travaux constituent une base de l'algorithme présenté ci-après. Les travaux ultérieurs de James Munkres sur l'algorithme font que celui-ci est également connu sous le nom d'algorithme de Kuhn-Munkres.

La réalisation de l'algorithme étant plus parlante quand la structure de données d'appui est une matrice indexée en lignes par les agents et en colonnes par les tâches, c'est ce choix qui sera retenu.

Puisqu'il s'agit de sélectionner n éléments, un par ligne et un par colonne, afin de minimiser la somme totale, on peut supposer sans perte de généralité que le minimum de chaque ligne est 0.

En pratique, il s'agit de retrancher dans chaque ligne (resp. colonne) la valeur du minimum de la ligne (resp. colonne) à chaque cellule, ce qui se fait sans perte de généralité car une répartition qui minimise la somme continue de la minimiser lorsque tous les agents voient la valeur associée à une même tâche diminuée de la même constante, ainsi que lorsque pour un agent particulier la valeur associée à toutes les tâches subit également la même modification.

Par la suite, si une répartition permet de sélectionner n zéros, c'est gagné. Sinon il s'avère qu'on peut couvrir tous les zéros en traçant moins de n lignes et colonnes.

La transformation à appliquer est la suivante : une fois ces zéros couverts, prendre le minimum des éléments n'appartenant à aucune ligne ni colonne tracée, le soustraire dans toutes les **lignes non tracées** et l'additionner sur toutes les **colonnes tracées**.

Le minimum de la matrice est alors zéro une fois de plus, certains zéros ayant changé de place, et on retente de sélectionner n zéros, répétant l'autre opération sinon.

Il s'avère que cet algorithme termine, mais la preuve dépasse largement l'objectif de ce TP.

Exercice 6 : Implémenter l'algorithme hongrois.

Une fois de plus, un graphe biparti donne une représentation graphique du problème, mais ce graphe lui-même n'est pas manipulé par l'algorithme tel qu'exécuté par un humain. Quant à l'implémentation, elle fait généralement intervenir la recherche d'un couplage maximal.

Coloration de graphes

Une coloration d'un graphe non orienté (S, A) est une fonction $c : S \to \mathbb{N}$ telle que si $s, t \in S$ sont reliés par une arête, alors $c(s) \neq c(t)$. Cela signifie que deux sommets voisins ne peuvent pas avoir la même couleur.

Le problème de la coloration d'un graphe cherche à déterminer une coloration minimale, c'est-à-dire une fonction c telle que $c(S)$ soit du plus petit cardinal possible.

Une application intéressante consiste à considérer les sommets comme des activités, et la couleur comme le numéro d'une salle où l'activité doit avoir lieu, les sommets étant reliés par une arête si, et seulement si, les activités correspondantes ont lieu sur des plages de temps qui se chevauchent.

Une autre application consiste à considérer un graphe représentant une carte, où les sommets sont des entités géographiques et les arêtes relient deux sommets correspondant à des entités ayant une frontière en commun non réduite à un point.

Cela permet d'introduire la notion de graphes planaires, qui est essentiellement une notion géométrique : un graphe (a priori non orienté) est dit planaire s'il existe une représentation de ce graphe dans laquelle deux arêtes ne se croisent jamais, et bien évidemment les arêtes ne traversent pas les sommets.

En excluant les enclaves et autres particularités, un graphe représentant une carte est planaire, et le théorème des quatre couleurs, prouvé à l'aide d'un assistant de preuve français nommé Coq, stipule que tout graphe planaire admet une coloration optimale avec quatre couleurs ou moins.

Attention : la réciproque n'est pas vraie, car il existe des graphes coloriables avec deux couleurs mais non planaires. Par exemple, le graphe complet biparti à trois et trois sommets n'est pas planaire, mais il est coloriable avec deux couleurs comme tout graphe biparti.

Pour les deux exercices à suivre, on pourra admettre que les sommets sont les premiers entiers naturels.

Exercice 7 : Écrire un algorithme déterminant en fonction d'un graphe et d'un nombre de couleurs s'il existe une coloration de ce graphe avec ce nombre de couleurs, par exploration exhaustive. En cas de réponse positive, il s'agit de renvoyer la coloration.

Cet algorithme étant en temps exponentiel, nous allons chercher à aller plus vite.

Exercice 8 : Écrire un algorithme glouton pour la coloration d'un graphe.

Il s'avère que l'algorithme glouton ne donne pas forcément une coloration optimale, notamment car l'ordre des sommets n'est pas forcément contrôlable.

Ainsi, si on considère un graphe biparti avec des sommets nommés $A_1, \ldots, A_n$ à « gauche » et $B_1, \ldots, B_m$ à « droite » et des arêtes entre A_i et B_j pour tout $i \neq j$, commencer par tous les sommets d'un côté puis continuer par l'autre donnera un coloriage avec deux couleurs.

Au contraire, alterner entre les deux côtés par indice croissant donnera $m+1$ ou $n+1$ couleurs (la plus petite valeur, sachant que si $m = n = 1$ il n'y a pas d'arête et cela correspond au cas où $m = 2$ et $n = 0$ donc une couleur suffit).

Un graphe d'intervalles est un graphe non orienté dont les sommets correspondent à des intervalles de $\mathbb{R}$ et tel que deux sommets sont reliés par une arête si et seulement si les intervalles sont d'intersection non vides.

Dans de tels graphes, l'algorithme glouton donne une coloration optimale lorsque les sommets sont colorés par ordre croissant de début d'intervalle.

La coloration de graphes a fait l'objet du sujet d'option informatique du concours Centrale-Supélec en 2015, avec une grande partie sur les graphes d'intervalles. La première application de la coloration mentionnée ci-avant y est introduite.

Les graphes d'intervalles permettent de résoudre de nombreux problèmes difficiles sur les graphes en temps raisonnables, notamment le problème de recherche d'un chemin hamiltonien, qui a une solution si, et seulement si, la réunion de tous les intervalles est un intervalle (cas où les intervalles sont tous ouverts ou tous fermés, sinon c'est plus subtil), solution consistant à parcourir les intervalles dans l'ordre croissant de leur extrémité **droite**.

Lexique