Mémorisation – MPSI – Informatique

Lycée Vaugelas – Gédéon Légaut

Chapitre 1 – Python – Variables, affectation, entrées / sorties

1	Un nom de variable commence par ou et doit être .	une lettre / un underscore _ / pertinent
2	Qu'est-ce que le type d'une variable ?	La nature de l'information stockée dans la variable
3	Quels sont les 5 types de base en python ?	**str** = chaîne de caractère **int** = entier (integer) **float** = nombre réel **complex** = nombre complexe **boolean** = booléen (`True`/`False`)
4	Quelle est la fonction python qui renvoie le type d'un objet ?	La fonction `type()`
5	Quel est l'opérateur d'affectation ? Qu'est-ce qu'une affectation ? Y a-t-il un ordre ?	opérateur `=` variable = valeur ou expression (variable toujours à gauche)
6	Qu'est-ce qu'une expression ?	Ensemble de variables combinées à l'aide d'opérateurs

Une expression informatique est en réalité bien plus générale que la première ébauche donnée ici : une expression est tout ce qui a une valeur : elle peut contenir une expression conditionnelle (comme `x = 1+3 if 2>1 else 5`), une boucle, une fonction ...

7	Lors d'une affectation, le membre est évalué avant de faire . Conséquence pratique en python (tableau `t`)	de droite l'affectation `t[i], t[j] = t[j], t[i]`
8	Quelle fonction python permet de récupérer ce qui est entré par l'utilisateur au clavier ? Quelle fonction python permet d'afficher le contenu d'une variable ?	La fonction `input` La fonction `print`

9	Notations : puissance / valeur absolue
	quotient division entière / reste division entière
	quotient et reste / arrondi à n chiffres
		`**` ou `pow` / `abs`
		`//` / `%`
		`divmod` / `round`

10	Comment je trouve de l'aide sur une fonction ?	`help(nom_fonction)` dans l'interpréteur python
11	Exemple d'une double comparaison évaluée.	`x <= y < z`
12	Pour utiliser π ... (3 manières)	import math → `math.pi` from math import pi → `pi` from math import pi as Pi → `Pi`

Chapitre 2 – Python – Conditions

13	Quelle est la structure complète d'un bloc test en python ?	`if condition1 :`
		`  bloc1`
		`elif condition2 :`
		`  bloc2`
		`else :`
		`  bloc3`

14	Quels sont les opérateurs de comparaison `égal`, `différent` et `appartient` ?	`égal` : `==`
		`différent` : `!=`
		`appartient` : `in`

Chapitre 3 – Python – Boucles

Les boucles sont de type `for` ou `while`. L'instruction `range(a,b,p)` renvoie un itérable sur un ensemble d'entiers de `a` **inclu** à `b` **exclu** par pas de valeur `p`. Par défaut, si on ne les précise pas, `a=0` et `p=1`.

15	Structure d'une boucle `for`	`for variable in liste :` `  bloc`

16	Structure d'une boucle `while`
	
	Que faut-il vérifier dans la boucle `while` ?
		`while condition :`
		`  bloc`
		
		Que `condition` devienne *fausse* à un moment pour sortir de la boucle

17	L'instruction ____ fait sortir de la boucle, l'instruction ____ passe à l'itération suivante.	`break` `continue`

18	Que renvoie `list(range(5))` ?
	Que renvoie `list(range(2,5))` ?
	Que renvoie `list(range(10,2))` ?
	Que renvoie `list(range(4,10,2))` ?
		`[0, 1, 2, 3, 4]`
		`[2, 3, 4]`
		`[]`
		`[4, 6, 8]`

Chapitre 4 – Python – Chaîne de caractère

Une chaîne de caractère (type `string`) se déclare avec des guillements simples (`'exemple'`), des guillements doubles (`"exemple"`) ou avec des guillements doubles triples qui tient compte des retours à la ligne

```
"""Ceci est une chaîne de caractère
qui contient des éventuels retours à la ligne """
```

Si on a besoin de mettre une apostrophe dans une chaîne de caractère, on procède de la manière suivante : `"l'apostrophe"`. Ajouter une chaîne `s2` à la suite d'une chaîne `s1` est appelé *concaténation* : `'bon'+"jour"` → `'bonjour'` (l'ordre des chaînes est respecté, l'opérateur de concaténation est `+`).

19	Pourquoi dit-on qu'une chaîne est itérable ? L'index d'un caractère dans une chaîne `s` varie entre ____ et ____.	On peut la parcourir caractère par caractère. 0 / `len(s)-1`
20	`[Chaîne de caractère]` `s='azertyuiop'` Donner `s[2]` Donner `s[-2]` Donner `s[:2]` Donner `s[2:]` Donner `s[:-2]` Donner `s[2:4]`	`s[2]='e'` (3ème élément) `s[-2]='o'` (en partant de la fin) `s[:2]='az'` (3ème élément exclu) `s[2:]='ertyuiop'` (2ème élément inclu) `s[:-2]='azertyui'` `s[2-4]='er'` (indice de fin exclu)

Chapitre 5 – Python – Les fonctions

Une portion de code que l'on serait amené à écrire plusieurs fois est problématique à entretenir : si on a fait une erreur, il faut corriger de partout. Il est préférable d'avoir cette portion de code écrite **une seule fois** et pouvoir y accéder quand on a besoin : c'est un des rôles des fonctions.
Une fonction ne doit faire qu'une seule chose : la résolution d'un problème complexe est décomposée en sous-problèmes simples.

Une fonction doit être commentée : c'est le rôle de la `documentation string` ou `docstring` qui contient dans l'ordre

1. une description de la fonction
2. une description des paramètres et leur nature (pré-conditions)
3. une description de la nature de ce qui est renvoyé par la fonction (post-conditions)
4. une série de tests que l'on écrit en général avant le corps de la fonction pour s'assurer que la fonction fait bien ce qu'elle devrait. Ces tests, dits unitaires, sont réalisés uniquement si on appelle le module `doctest`.

Ce qui fait que le protopype complet d'une fonction est

```
import doctest
def nom_fonction(liste des arguments):
    """
    Description : ce que fait la fonction et comment elle le fait |
    Pré-conditions : nature des arguments                         |
    Post-conditions : nature de ce que renvoie la fonction        | docstring
    Tests unitaires introduits par >>> suivi d'une espace :       |
    >>> nom_fonction(arguments avec valeurs)                      |
    résultat attendu                                              |
    """
    # corps de la fonction

# Programme principal
doctest.testmod()        # exécution des test unitaires
```

En tapant directement `help(nom_fonction)` dans un interpréteur python, on a accès à la `docstring` de la fonction `nom_fonction` et c'est ainsi qu'est faite la documentation python. C'est un réflexe qu'il vous faudra acquérir.

On appelle **portée** d'une variable la zone du programme où on peut utiliser cette variable. Une variable déclarée dans une fonction n'est pas accessible à l'extérieur de la fonction, tandis qu'une variable déclarée dans le corps du programme principal est accessible dans la fonction.
On peut déclarer une fonction `aux` à l'intérieur d'une autre fonction : la fonction `aux` est appelée fonction auxiliaire.
On peut utiliser le mot réservé `lambda` pour définir une fonction en une ligne. S'utilise en général pour des fonctions simples.

21	Quelle est la structure d'une fonction en python ?	`def nom_fonction (paramètres) :`
		`  """ docstring """`
		`  ...`
		`  return résultat`

22	Que renvoie `help(nom_fonction)` en python ?	La « *docstring* »de la fonction `nom_fonction`
23	Définir la fonction $f(x) = x^3$ avec `lambda`	`f = lambda x: x**3`

Chapitre 6 – Les tableaux

Un tableau python est de type `list`. Un élément est accessible par son indice. Tout ce qui a été vu sur les chaînes de caractère est valable ici : `len(t)` renvoie le nombre d'éléments dans le tableau `t`. L'élément `i+1` du tableau est accessible via `t[i]` avec `0 <= i <= len(t)-1` .
L'indice d'un tableau commence à `0` et s'arrête donc à `len(t)-1`. Une erreur classique est d'essayer d'appeler `t[len(t)]`, ce qui lève l'erreur `IndexError: list index out of range`.
Un tableau est itérable : on peut le parcourir. (Vocabulaire : range → intervalle, index → indice)

L'instruction `t1=t` ne copie pas le tableau `t` : les variables `t1` et `t` pointent vers les mêmes cases mémoires. Dans ce cas, modifier `t1` revient aussi à modifier `t`. Si on veut travailler sur `t2` une copie indépendante de `t`, alors il y a deux possibilités : `t2=t[:]` ou `t2=copy.deepcopy(t)` du module `copy`. L'instruction `t2=t[:]` sera en général suffisante pour copier un tableau.
Cette instruction modifiée permet aussi de renvoyer le tableau `t` à l'envers : `t[::-1]` !

24	`[Tableau python]`
	Que renvoie `[1,2]+[3]` ?
	Déclarer un tableau `t` vide ?
	Déclarer un tableau t de 0 de taille `N` ?
	2 manières d'ajouter le contenu de la variable `i` au tableau `t`
		`[1,2,3]`
		`t = []`
		`t = [0]*N`
		`t.append(i)` ou `t+[i]`

25	`[Tableau python]`
	2 manières de parcourir un tableau `t` pour afficher les éléments de `t` ?
	Lien entre ces 2 manières ?
		`for x in t :`
		`  print(x)`
		`for i in range(len(t)) :`
		`  print(t[i])`
		`x` correspond à `t[i]`

26	`[Tableau python]` Créer par compréhension le tableau `t` des entiers pairs entre 0 et 10 inclu le tableau `t1` correspondant aux éléments de `t` élevés au cube	`t = [n for n in range(11) if n%2==0]` `t1 = [x**3 for x in t]`
27	Inverser le tableau `T` en python	`T[::-1]`

Chapitre 7 – Les dictionnaires

On accède à un élément d'un tableau par son indice (entier). Un dictionnaire (de type `dict`) est une généralisation de ce concept qui permet d'accéder à une valeur à partir, non pas d'un entier, mais d'une **clé** qui peut être un entier, une chaîne de caractère, ...
Un dictionnaire est donc un ensemble de couple **clé/valeur** et il est implémenté de manière à ce que l'accès à n'importe quelle valeur se fasse en temps constant, peu importe la taille du dictionnaire.

`d={cle1:valeur1, cle2:valeur2 }`

Les clés doivent être uniques. On ajoute un nouveau couple `cle3/valeur3` de la manière suivante : `d[cle3]=valeur3` ; on affiche la valeur associée à `cle2` avec `d[cle2]`.
Un dictionnaire implémente un certain nombre de méthodes :

- `d.keys()` renvoie un itérateur sur l'ensemble des clés du dictionnaire `d` (attention, l'ordre des clés n'est pas forcément préservé suivant la version de python utilisée)
- `d.values()` renvoie un itérateur sur l'ensemble des valeurs du dictionnaire `d`
- `d.items()` renvoie un itérateur sur l'ensemble des couples clé/valeur du dictionnaire `d`

28	2 manières d'afficher l'ensemble des couples clé/valeur du dictionnaire `d` ?	`for cle in d :`
		`  print(cle, d[cle])`
		`for cle, valeur in d.items() :`
		`  print(cle, valeur)`

Chapitre 8 – Algorithmes de recherche sur les tableaux (1)

29	Algorithme de recherche séquentielle de `valeur` dans le tableau `T`	`trouve = False`
		`for x in T :`
		`  if x == valeur :`
		`  trouve = True`

30	Algorithme de recherche du maximum d'un tableau `T` d'entier	`maximum = T[0]`
		`for x in T :`
		`  if x > maximum :`
		`  maximum = x`

31	Algorithme de recherche du second maximum d'un tableau `T` d'entier
	`max1 > max2`
		`max1, max2 = T[0], T[1]`
		`if max1 < max2 :`
		`  max1, max2 = max2, max1`
		`for x in T[2:] :`
		`  if x > max1 :`
		`  max1, max2 = x, max1`
		`  elif x > max2 :`
		`  max2 = x`

Une erreur fréquente consiste à mal initialiser la valeur `maximum` : il faut impérativement `maximum = T[0]` car sinon, il faut avoir des informations supplémentaires sur les données contenues dans le tableau pour que l'algorithme soit exact (exemple : si toutes les valeurs sont positives ou nulles, alors l'initialisation `maximum = -1` serait correct).

Chapitre 9 – Algorithmes de tri (1) : tri à bulle, tri par comptage

32	Qu'est-ce que le **tri à bulle** croissant d'un tableau d'entier ? Après le 1er parcours complet du tableau, où se trouve la valeur maximale du tableau ?	Parcours du tableau en échangeant systématiquement 2 éléments consécutifs s'ils sont mal triés Dans la dernière case du tableau

33	Quel est l'algorithme du tri à bulle ? (fonction `tri_bulle(T)`)
	indentation 2 carreaux
		`def tri_bulle(T) :`
		`  for i in range(len(T)-1, 0, -1) :`
		`  for j in range(i) :`
		`  if T[j+1]<T[j] :`
		`  T[j], T[j+1] = T[j+1], T[j]`

Tri par comptage : les notions de majorant et de maximum sont différentes, même si elles sont liées.

34	Le tri par comptage repose sur une certaine connaissance des données. Laquelle ?	Connaître le majorant strict `m` du tableau T $\forall x \in T, x < m$

35	[Tri par comptage]
	Quelles sont les 3 étapes du tri par comptage d'un tableau `T` d'entiers dont un majorant strict est `m` ?
		`# 1) initialise le tableau d'occurence`
		`Occur = [0]*m`
		`# 2) remplir le tableau d'occurence`
		`for i in T :`
		`  Occur[i] += 1`
		`# 3) le tri de T se fait par lecture`
		`#  du tableau d'occurence`

Chapitre 10 – Modules Matplotlib et Numpy

Les fonctions fournies par le module `numpy` sont dites vectorielles ou universelles car leur argument est un tableau et non une valeur.

36	Import des modules matplotlib et numpy
	
	Pour ajouter une courbe
	Pour afficher le graphique
		`import matplotlib as plt`
		`import numpy as np`
		
		`plt.plot(x, y)`
		`plt.show()`

37	[Numpy] Comment obtenir un tableau de valeurs comprises entre `a` inclu, `b` exclu, par pas de `c` ?
	
	Comment obtenir un tableau de `N` valeurs régulièrement réparties entre `a` et `b` inclus ?
		`np.arange(a,b,c)`
		
		`np.linspace(a,b,N)`
		linear spaced values

38	[Numpy] Comment obtenir un tableau de `N` zéros ?
	
	Comment obtenir un tableau de `N` cases non initialisées ?
		`np.zeros(N)`
		
		`np.empty(N)`

Chapitre 11 – Fonctions récursives

Une fonction récursive est une fonction qui s'appelle elle-même : ce qui est a priori une boucle infinie. Il faut donc à un moment sortir de cette boucle infinie : c'est le **cas de base**. Une fonction est dite **récursive terminale** si aucun traitement n'est effectué à la remontée des appels récursifs.
Cachée derrière la notion de fonction récursive, il y a la notion de pile (cf plus loin).

39	Une fonction récursive contient forcément 2 éléments. Lesquels ?	1) un **cas de base** pour terminer 2) un cas général où la fonction s'appelle elle-même de manière à rencontrer le cas de base à un moment donné

40	Fonction récursive naïve `fact(n)` de $n!$	`def fact(n) :`
		`  if n == 0 :`
		`  return 1`
		`  return n * fact(n-1)`

41	Fonction récursive terminale `fact(n)` de $n!$	`def fact(n, f=1) :`
		`  if n == 0 :`
		`  return f`
		`  return fact(n-1, n*f)`

42	Algorithme récursif terminal du pgcd de $a > 0$ et $b > 0$
	(fonction `pgcd(a,b)`)
		`def pgcd(a, b) :`
		`  if b == 0 :`
		`  return a`
		`  return pgcd (b, a%b)`

Chapitre 12 – Complexité algorithmique

43	Un ____ est un ensemble d'instructions permettant de résoudre un problème donné en ____ et un espace de stockage fini. Il peut être décrit par un organigramme ou ____.	algorithme un temps fini un pseudo-code
44	Qu'est-ce que la complexité spatiale d'un algorithme ?	Son besoin en mémoire
45	Qu'est-ce que la complexité temporelle d'un algorithme ? (taille des entrées n)	Le nombre $T(n)$ d'opérations élémentaires exécutées par le processeur
46	Qu'est-ce que la terminaison d'un algorithme ?	Le fait que l'exécution de toute instance de l'algorithme se termine en un temps fini.
47	Qu'est-ce que la correction d'un algorithme ?	Le fait que la sortie retournée est correcte (ou exacte).
48	Quelle notion est utilisée pour prouver la correction d'un algorithme itératif ?	Un invariant de boucle.
49	Dans une boucle ____, l'invariant de boucle est une propriété qui est vraie ____ l'entrée dans la boucle et qui reste vraie après ____ du corps de la boucle.	while avant chaque parcours
50	Dans une boucle ____, l'invariant de boucle est une propriété qui est vraie ____ du 1er parcours de la boucle et qui reste vraie après ____ du corps de la boucle.	for à la fin chaque parcours
51	`[Complexité algorithmique]` Symbole de majoration asymptotique ? Symbole d'équivalence asymptotique ?	$\Theta()$ $\sim$
52	`[Complexité algorithmique]` $(c > 1)$ Comment définir qu'un algorithme est 1) en temps constant, 2) en temps polynomial, 3) en temps exponentiel, 4) en temps factoriel ?	1) $\Theta(1)$ 2) $\Theta(n^c)$ 3) $\Theta(c^n)$ 4) $\Theta(n!)$

Chapitre 13 – Algorithmes de tri (2) : tri sélection et insertion

Le **tri par sélection** consiste

1. à chercher l'indice `mini` de la valeur minimale du sous-tableau `T[i..n]`
2. à permuter `T[i]` avec `T[mini]` : la valeur minimale du sous-tableau est mise en premier
3. à recommencer sur le sous-tableau `T[i+1..n]`

On commence à l'indice `i=0` et on arrête quand le tableau `T` a été parcouru en entier.

Le **tri par insertion** consiste, lui,

1. à mémoriser l'élément courant `x=T[i]`
2. à décaler vers la droite tous les éléments du sous-tableau `T[1..i-1]` qui sont plus grands que `x` en commençant par `T[i-1]`
3. à placer `x=T[i]` dans le trou laissé par le décalage

Il ne sert à rien de traiter le cas `i=0` car le sous-tableau `T[0]` est trié. On parcourt les valeurs `1<=i<n`.

53

[Tri sélection]
Fonction `minimum(T,j)`
renvoyant l'indice du minimum du tableau `T`
à partir de l'indice `j`

```
def minimum(T,j) :
    pos, mini = j, T[j]
    for k in range( j+1, len(T) ) :
        if T[k] < mini :
            pos, mini = k, T[k]
    return pos
```

54

Algorithme du tri par sélection
Fonction `tri_selection(T)`

```
def tri_selection(T) :
    for j in range( len(T)-1 ) :
        i = minimum(T, j)
        if i != j :
            permute(T, i, j)
```

55

Algorithme du tri par insertion
Fonction `tri_insertion(T)`

```
def tri_insertion (T) :
    for i in range( 1, len(T) ):
        x = T[i]
        j = i
        while j > 0 and T[j-1] > x :
            T[j] = T[j-1]
            j -= 1
        T[j] = x
```

Chapitre 14 – Recherche dichotomique

La recherche dichotomique opère sur un tableau **trié**. Recherchons la valeur `x=6.5` dans le tableau

```
        t = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
étape 1       g           m                d
étape 2                      g  m          d
étape 3                      g=m=d  → est-ce que la valeur recherchée est ici ?
étape 4                      d  g  → arrêt
```

On divise le tableau en 2 en calculant `m`. Est-ce que la valeur recherchée est située à l'indice `m` ? Non : comme le tableau est trié, alors on sait que la valeur recherchée, si elle est dans le tableau, est dans la partie située à droite de `m`, soit dans `t[m+1,d]`, on fixe donc `g=m+1` et on recommence (étape 2). Dans cet exemple, la valeur recherchée n'est pas dans `t` pour montrer le cas le plus défavorable. Si la valeur recherchée était dans le tableau, elle serait trouvée à l'étape 3. Ce n'est pas le cas, on arrive à l'étape 4 : on s'arrête au moment où l'indice gauche `g` devient **strictement** supérieur à l'indice droit `d`.

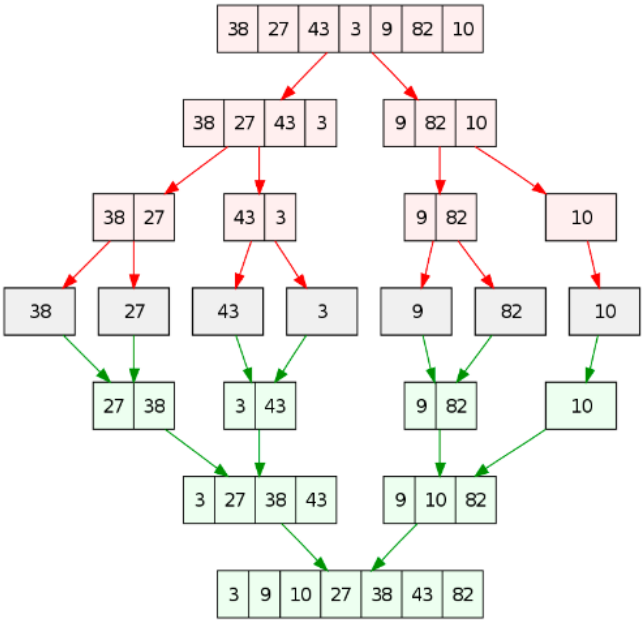
56	Quelle est la condition **fondamentale** pour pouvoir faire une recherche dichotomique ? Quelle est la complexité temporelle de la recherche dichotomique ?	La recherche dichotomique opère sur un tableau **trié** ! $\Theta(\ln n)$
57	Quelle est la version impérative de `recherche_dichotomique(t,x)` ? On recherche `x` dans le tableau `t` (On note `g,d` les indices gauche/droite du sous-tableau considéré)	

```python
def recherche_dichotomique(t, x) :
    g, d = 0, len(t)-1
    while g <= d : # piège <=
        m = (g+d)//2
        y = t[m]
        if y == x :
            return True
        if y > x : # piège <
            d = m - 1
        else :
            g = m + 1
    return False
```

Chapitre 15 – Algorithmes de tri (3) : tri fusion

On rappelle que l'on peut inverser un tableau `T` avec l'instruction `T[::-1]`.
Le **tri fusion** (en anglais *mergesort*) est un tri basé sur le paradigme `diviser pour régner`. Il consiste en

1. partager les éléments à trier en deux parties égales (ou au plus près)
2. trier les deux parties (ou ne rien faire si leur taille est de 1)
3. fusionner les deux moitiés triées



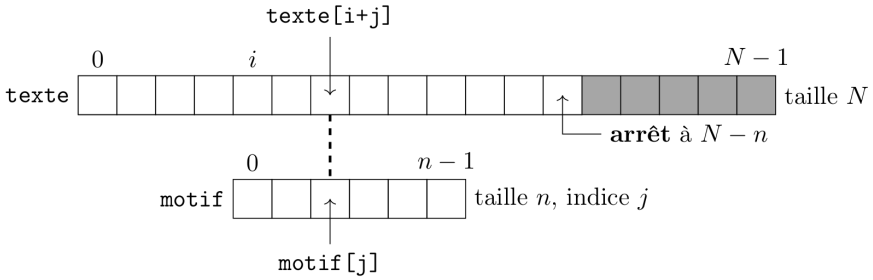
La fusion de deux tableaux se fait en temps linéaire, le nombre d'appels récursifs est logarithmique, la complexité temporelle est alors en $\Theta(n \ln n)$, ce qui est optimal pour un tri par comparaison.
Le tri fusion est particulièrement efficace pour trier des listes chaînées pour lesquelles on n'a pas d'accès aléatoire (ce qui signifie que l'on ne peut pas accéder en temps constant à un élément quelconque, contrairement aux tableaux).

58

Fonction tri fusion d'un tableau T
version récursive
(indentation 2 espaces)

```
def tri_fusion(t):
  n = len(t)
  if n > 1:
    i, j, m = 0, n - 1, n // 2
    z = tri_fusion(t[:m]) + \
        tri_fusion(t[m:])[::-1]
    for k in range(n):
      if z[i] <= z[j]:
        t[k] = z[i] ; i += 1
      else:
        t[k] = z[j] ; j -= 1
  return t
```

Chapitre 16 – Recherche textuelle naïve



Pour chaque caractère du texte dont l'indice est compris entre 0 et $N - n$ (soit $N - n + 1$ fois), on vérifie si le motif de taille n (soit n comparaisons) est trouvé : il y a donc $C(n, N) = n \times (N - n + 1)$ comparaisons.

$$\left.\frac{\partial C}{\partial n}\right|_N = N - 2n_{\max} + 1 = 0 \qquad \text{d'où} \qquad n_{\max} = \frac{N+1}{2}$$

$$C(n_{\max}, N) = (N+1)^2/4$$

d'où une complexité temporelle dans le pire des cas en $\mathcal{O}\big(N^2\big)$ lorsque la taille du motif est environ la moitié de celle du texte.

59

Fonction `recherche(motif, texte)` naïve d'un `motif` dans un `texte` avec une fonction auxiliaire `occurence()`

```
def recherche (motif, texte) :
    def occurrence() :
        for j in range(n) :
            if texte[i+j] != motif[j] :
                return False
        return True

    N = len(texte)
    n = len(motif)
    for i in range(N-n+1) :
        if occurence() :
            return True
    return False
```

Chapitre 17 – Type abstrait de donnée : piles et files

Un **type abstrait de donnée** est une structure permettant d'ordonner des données à l'aide d'un jeu d'opérations autorisées dont on ne se préoccupe pas de l'implémentation. On sait que ces opérations, mais on ne s'intéresse pas à savoir comment cela se passe dans la mémoire de l'ordinateur.
Les piles et les files sont des structures séquentielles et **non indexées** (on ne peut accéder à l'élément par son indice) de données.
Une structure de type LIFO (*Last In, First Out*) est une pile. Le nom évoque la pile d'assiette.
Une structure de type FIFO (*First In, First Out*) est une file. Le nom évoque la file d'attente.

60

Algorithme de parenthésage simple `parenthesage(mot)` avec une pile dont les fonctions sont `creer_pile()`, `empiler(motif, pile)`, `depiler(pile)` et le test `pile_vide(pile)`.

```
def parenthesage(mot) :
    pile = creer_pile()
    for c in mot :
        if c == '(' :
            empiler(1, pile)#CeQu'onVeut
        elif c == ')' :
            if pile_vide(pile) :
                return False
            else :
                depiler(pile)
        return pile_vide(pile)
```

Chapitre 18 – Opérateurs « bitwise »

Les opérateurs `bitwise` littéralement comprennent les bits : ils ne travaillent pas sur la valeur même d'un entier, mais bit à bit sur sa représentation binaire en complément à deux. Ce sont, entre autre, les opérateurs

«, «=, », »=, & (and) et | (or)

- `x « n` : on considère la représentation binaire de `x` et on décale tous les bits de `n` rangs vers la gauche **en ajoutant `n` zéros à droite**. C'est une manière simple de calculer $x \times 2^n$. Ex : `3 « 4` ($3_{10} = 11_2$) renvoie 48 (car $48_{10} = 110000_2$).
- `x » n` : on considère la représentation binaire de `x` et on décale tous les bits de `n` rangs vers la droite, ce qui revient à tronquer les `n` bits de poids faible. C'est une manière simple de calculer $x/2^n$. Ex : `48 » 4` renvoie 3 car $3_{10} = 11_2$.
- `x & y` : chaque bit du résultat est `1` si le bit du rang considéré est `1` dans `x` **et** dans `y`, `0` sinon. Si `x` et `y` n'ont pas le même nombre de bits, on ajoute des `0` à gauche du plus petit. On obtient le bit de poids faible de `x` en faisant `x & 1`.
- `x | y` : chaque bit du résultat est `1` si le bit du rang considéré est `1` dans `x` **ou** dans `y`, `0` sinon. Si `x` et `y` n'ont pas le même nombre de bits, on ajoute des `0` à gauche du plus petit.

61

`[Opérateurs bitwise]`

Comment obtenir le bit de poids faible de `x` ?	`x & 1`
Diviser par 2 un entier `x` ?	`x » 1`
Multiplier `x` par 2^n et affecter le résultat à `x` ?	`x «= n`

Chapitre 19 – Représentation d'un entier dans une base B

Soit B un entier naturel supérieur ou égal à 2 appelé **base**.

B	Bases courantes	symboles dans la base
2	binaire	0 1
10	décimale	0 1 2 3 4 5 6 7 8 9
16	hexadécimale	0 1 2 3 4 5 6 7 8 9 a b c d e f

Tout entier naturel n se décompose de manière **unique** selon (base B ou base 2)

$$n = \sum_{k=0}^{p} c_k B^k \qquad \text{avec} \quad 0 \leqslant c_k < B \quad \text{et} \quad c_p \neq 0 \tag{1}$$

$$n = \sum_{k=0}^{p} b_k 2^k \qquad \text{avec} \quad b_k \in \{0,1\} \quad \text{et} \quad b_p = 1 \tag{2}$$

Pour l'algorithme de conversion vers la base B où $10 \leqslant B$, il est utile de recourir à la variable

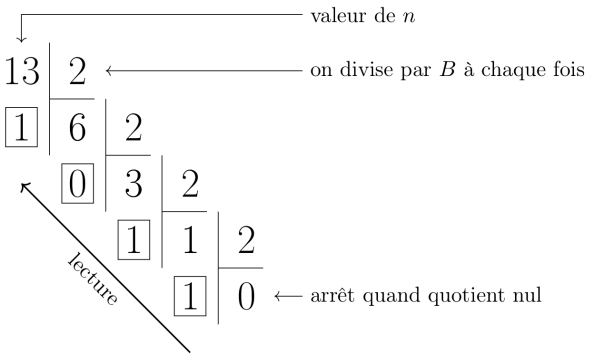
`chiffres = "0123456789abcdef...zABCDEF...Z"`

car `chiffres[i]` donne directement le coefficient c_k de (1)

62 | Sur N bits, on peut représenter toutes les valeurs comprises dans l'intervalle . | $0 \leqslant n \leqslant 2^N - 1$

63 | Quelle est la taille exacte N de la représentation de n en base B ? |

$$B^{N-1} \leqslant n < B^N$$
$$(N-1)\ln B \leqslant \ln n < N \ln B$$
$$\frac{\ln n}{\ln B} < N \leqslant \frac{\ln n}{\ln B} + 1$$
$$N = \left\lfloor \frac{\ln n}{\ln B} \right\rfloor + 1 = \lfloor \log_B n \rfloor + 1$$



$$(13)_{10} = (1101)_2 = (\mathrm{d})_{16}$$

64 | Algorithme de conversion de n en base $2 \leqslant B < 63$ par division euclidienne (version itérative) avec `chiffres="0123456789abcdef..zA..Z"`

```python
def conversion(n, B) :
    q = n
    st = ""
    while q != 0 :
        q,r = divmod(q,B)
        st = chiffres[r] + st
    return st
```

Chapitre 20 – Calcul de n à partir de sa représentation en base B

Soit le nombre n dont la représentation en base B est $(c_p, \ldots, c_2, c_1, c_0)$ appliqué à l'exemple `10101`.

$$n = c_4B^4 + c_3B^3 + c_2B^2 + c_1B^1 + c_0 \tag{3}$$
$$= (((c_4 \times B + c_3) \times B + c_2) \times B + c_1) \times B + c_0 \tag{4}$$

On peut faire le calcul de droite à gauche (de c_0 à c_p) via (3) ou de manière plus efficace de gauche à droite (de c_p à c_0, c'est le schéma de Hörner) via la relation (4), ce qui permet de gagner une multiplication par coefficient.

65

Algorithme de conversion **depuis** la base B
Fonction `depuis_base_B(ch, B)`
où `ch` représente $(c_p, \ldots, c_2, c_1, c_0)$
(restriction $2 \leqslant B \leqslant 9$)

```
def depuis_base_B(ch, B) :
    m = 1
    n = 0
    for i in range(len(ch)-1,-1,-1) :
        n += int(ch[i])*m
        m *= B
    return n
```

66

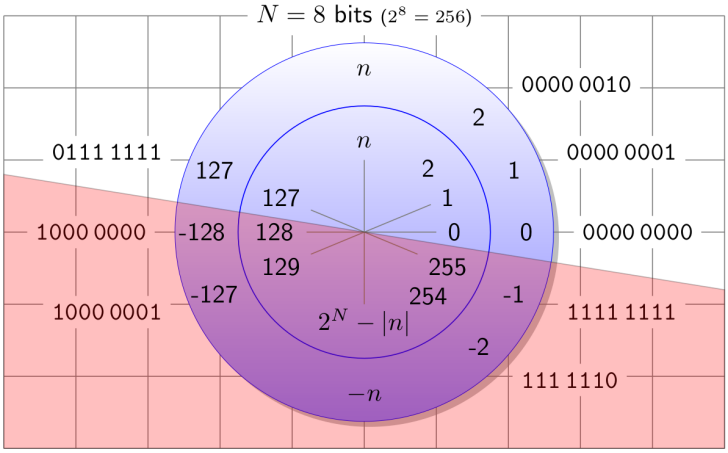
Algorithme de conversion **depuis** la base B
Fonction `depuis_base_B_horner(ch, B)`
où `ch` représente $(c_p, \ldots, c_2, c_1, c_0)$
(restriction $2 \leqslant B \leqslant 9$)

```
def depuis_base_B_horner(ch, B) :
    n = 0
    for i in range(len(ch)) :
        n = n*B + int(ch[i])
    return n
```

Chapitre 21 – Réprésentation des entiers relatifs

Les entiers relatifs sont représentés en complément à deux afin de préserver les algorithmes d'addition (une soustraction = 2 additions) et de multiplication (additions et décalages), sous certaines réserves (on oublie l'éventuelle retenue sur le bit de poids fort pour l'addition, ...).
Notation : $\sim n$ représente le complément à 1 de n (inverser tous les bits de n). Pour $n > 0$, la représentation binaire de $-n$ est alors $\sim n + 1$.

entier non signé	représentation binaire sur 8 bits	entier signé
0	0000 0000	0
1	0000 0001	1
2	0000 0010	2
	...	
127	0111 1111	127
128	1000 0000	-128
129	1000 0001	-127
	...	
254	1111 1110	-2
255	1111 1111	-1



Les exemples ci-dessus sont donnés sur $N = 8$ bits. Si la somme de deux entiers relatifs dépasse $127 = 2^{N-1} - 1$, il y a débordement et le résultat sera faux. De même si le résultat est inférieur à $-128 = -2^{N-1}$. Une telle situation se repère facilement : les 2 opérandes ont le même signe, mais le résultat est de signe opposé.

67

Que signifie `2c8` dans l'expression
`-5 = (1111 1011)`$_{2c8}$?

-5 est codé en complément à 2 sur 8 bits

68	On connaît la représentation d'un entier relatif n sur N bits. Comment trouver celle de $-n$?	Procédure du complément à 2 dans l'ordre : 1) inverser les N bits 2) ajouter 1 (oubli éventuelle retenue sur bit de poids fort)
69	Fonction `cplt_a_deux(n,N)` retournant une chaîne de caractère donnant la représentation binaire de n sur N bits	

```
def cplt_a_deux(n,N) :
    lim = 1 << (N-1) # 2**(N-1)
    if not (-lim <= n < lim) :
        return "entier hors intervalle"
    u = n if n >= 0 else n + (lim << 1)
    ch = ''              # n + 2**N
    for i in range(N) :
        ch = str(u & 1) + ch
        u >>= 1
    return ch
```

Chapitre 22 – Tri rapide

Le tri rapide (quiksort) est basé sur la stratégie `diviser pour régner` qui réduit la résolution d'un problème à plusieurs sous-problèmes de tailles inférieures. Sa complexité moyenne est $\Theta(n \ln n)$, ce qui est optimal pour un tri par comparaison. C'est l'un des tris les plus rapides en pratique sur les tableaux de grande taille. Le principe est de ramener le tri d'un tableau aux tris de deux tableaux de moindre taille. L'efficacité du tri repose sur la possibilité de partitionner rapidement, et en place, un tableau en deux tableaux sur lesquels portent les appels récursifs. Aucun travail n'est requis après les deux appels récursifs, contrairement au tri fusion. Le principe de ce tri repose sur 3 étapes :

1. on choisit un élément `p` du tableau appelé `pivot`
2. on réarrange le tableau (partitionnement) en 3 parties consécutives : les éléments inférieurs ou égaux à `p`, le pivot `p` et les éléments supérieurs à `p`
3. on trie récursivement les deux sous-tableaux des éléments inférieurs ou égaux à `p` et supérieurs à `p` : le tableau est alors trié

L'efficacité du tri rapide dépend du choix du pivot : si on le choisit aléatoirement, cela conduit au meilleur choix en moyenne. Ici, on choisit le premier élément du tableau à partionner par simplicité.

70 `[Tri rapide]`
Fonction `tri_rapide(t)` d'un tableau `t`
avec une fonction locale `tri(g,d)`
et une fonction `partition(t,g,d)`

```
def tri_rapide(t) :
    def tri(g, d) :
        if g < d :
            p = partition(t, g, d)
            tri (g, p-1)
            tri (p+1, d)
    tri(0, len(t)-1)
```

71 `[Tri rapide]`
Fonction `partition(t, g, d)`
(Le pivot est le 1er élément du tableau à partionner)

```
def partition(t, g, d) :
    p = g
    for i in range(g+1, d+1) :
        if t[i] <= t[g] :
            p += 1
            t[i], t[p] = t[p], t[i]
    t[g], t[p] = t[p], t[g]
    return p
```

Chapitre 23 – Algorithme glouton

Un algorithme **glouton** est un algorithme qui fait un **choix localement optimal** pour se ramener à un problème plus simple en espérant que cela conduise à une solution **globalement optimale**. On peut imager cette notion avec la notion de minimum : est-ce qu'un minimum local pour une fonction correspond forcément à son minimum global ? Suivant la fonction, ce sera le cas ou non !
Dans le cas d'un algorithme de complexité exponentielle, il devient très rapidement impossible de lister toutes les possibilités : dans ce cas, un choix glouton permet d'obtenir une solution rapidement : mais la solution est-elle optimale ?
Les problèmes typiques sont :

- le **rendu de monnaie** : on veut rendre la monnaie avec le moins de pièce possible. Le choix glouton conduit à une solution globalement optimale si le choix des pièces est bien fait.
- le **problème du sac à dos** : on remplit un sac à dos avec des objets de différentes valeurs. On a une contrainte (le poids maximal que peut contenir le sac à dos) et on veut que le sac à dos ait la plus valeur possible. Si on n'a pas le droit de casser un objet pour arriver exactement au poids maximal, alors la solution gloutonne n'est pas globalement optimale.
- l'**allocation de salle de cours** : quel est le nombre minimal de salle de cours, étant donné une liste de cours sur une certaine plage horaire ? En triant les cours par date de fin croissante, on peut démontrer que la solution gloutonne est globalement optimale.
- la **sélection d'activité** : dans une plage horaire donnée, quelle est la plus grande sélection d'activité possible ? En triant les activités par date de fin croissante, on peut démontrer que le choix glouton est globalement optimal.
  Dans l'exemple ci-dessous, le tri par fin de date croissant est réalisé ligne 3, après avoir copié (ligne 2) le tableau d'activité. `a[0]` représente le début d'une activité et `I[-1]` est la dernière activité sélectionnée donc `I[-1][1]` est la date de fin de la dernière activité sélectionnée.

72

Algorithme glouton de sélection d'activité (fonction `solution_optimale(E)`)
où `E = [(d1,f1), (d2,f2), ...]`
Chaque activité est représentée par un couple début/fin tel que $[d_i, f_i[$.

```
1 def selection_optimale(E) :
2   L = E[:] # copie
3   L.sort(key = lambda a : a[1])
4   I = []
5   for a in L :
6     if I == [] or a[0] >= I[-1][1] :
7       I.append(a)
8   return I
```

Chapitre 24 – Traitement d'image

Une image numérique est constitué de petits carrés de couleurs uniformes appelés **pixels**. Lorsque ces pixels sont visibles à l'oeil nu, on dit que l'image est pixellisée.
Le format PNG utilise le mode RGB (Red/Green/Blue) : chaque pixel est représenté par 3 octets. Le noir est `(0,0,0)` et le blanc est `(255,255,255)` : cela définit 16 millions de couleur. En niveau de gris, **les trois valeurs** sont identiques et l'information n'est stockée que sur un seul octet, soit 256 niveaux de gris.

```
1  from PIL import Image
2  import numpy as np
3  import matplotlib.pyplot as plt
4  im = Image.open("fichier.png")
5  print("Mode des couleurs de l'image", im.mode)
6  print("Format de l'image", im.format)
7  print("Taille de l'image", im.size) # renvoie le couple (largeur,hauteur)
8  print("Infos de l'image", im.info)
9  im.show() # pour afficher l'image
10 im.close() # ferme l'image
11
12 t = np.array(im) # image transformée en tableau numpy
13 print("Taille du tableau :", t.shape) # renvoie (hauteur,largeur,3)
14 print("Type des valeurs : ", t.dtype)
15 new_t = np.zeros((h,l,3), dtype='uint8') # couleur RGB
16 new_t = np.zeros((h,l), dtype='uint8') # N&B ou niveaux de gris
17
18 new_im = Image.fromarray(t) # crée une nouvelle image à partir de t
19 new_im.show() # pour l'afficher
20 new_im.save("new_fichier.png")
```

Lignes 15 et 16, on utilise des entiers non signés (`uint8`), on ne peut donc les ajouter directement : il faut les transformer en entier avant via `int()`.

Chapitre 25 – Les graphes

Un **graphe** est une **structure de donnée relationnelle** qui permet de représenter les relations entre les éléments d'un ensemble. Les éléments sont appelés **sommets** (en anglais *vertices*). Les sommets sont parfois appelés **noeids**. Il existe deux types de graphes :

- les graphes **orientés** où les relations s'appelent des **arcs**
- les graphes **non orientés** où les relations s'appelent des **arêtes**

Lorsqu'une relation (arête ou arc) existe entre deux sommets, on dit qu'ils sont **adjacents** ou **voisins**.
Un graphe G est donc représenté par la donnée de deux ensembles : l'ensembles des sommets S et l'ensemble des arêtes ou des arcs A : $G = (S, A)$. L'**ordre** du graphe est son nombre de sommets, sa **taille** est son nombre d'arête ou d'arc.
Le **degré** d'un sommet est son nombre d'arête ou son nombre d'arcs entrants et sortants.
On peut construire une bijection entre S et $[1, n]$ ou $[0, n-1]$: un sommet est représenté par un entier.

73

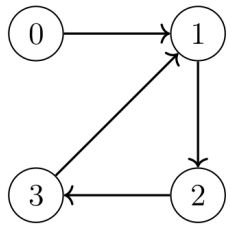
`[Graphes]` Un graphe est constitué de 2 choses. Lesquelles ?

Ordre d'un graphe ?
Taille d'un graphe ?

Un graphe est constitué d'un ensemble de sommet S et d'un ensemble d'arête ou d'arc $A : G = (S, A)$.
ordre = nombre de sommets
taille = nombre d'arête ou d'arc

74

`[Graphes]`



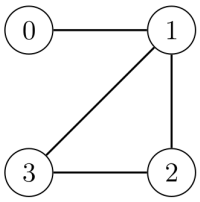
Ordre ?
Taille ?
Liste d'adjacence `la` ?
Matrice d'adjacence ?

Ordre = 4 sommets
Taille = 4 arcs
`la = [ [1], [2], [3], [1] ]`

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

75

`[Graphes]`



Ordre ?
Taille ?
Liste d'adjacence `la` ?
Matrice d'adjacence ?

Ordre = 4 sommets
Taille = 4 arêtes
`la = [ [1],[0,2,3],[1,3],[1,2] ]`

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

Chapitre 26 – Parcours d'un graphe en largeur

Parcourir ou énumérer un graphe consiste à énumérer l'ensemble des sommets accessibles à partir d'un sommet donné afin de réaliser un certain traitement sur ces sommets. La différence entre les différents parcours réside dans l'ordre dans lesquels les sommets sont visités.

Le **parcours en largeur** s'apparente à une exploration par distance croissante à partir d'un sommet particulier (le sommet source) : les voisins les plus proches sont visités en premier, avant de commencer à visiter les voisins des voisins.
Chaque sommet est considéré deux fois : il est d'abord **découvert** (ou marqué), puis **traité** (c'est-à-dire que l'on visite tous ces voisins). Il peut être rédecouvert avant d'avoir été traité, mais dans ce cas, on ne fait rien.
La structure de donnée adaptée au traitement des sommets est de type FIFO : il s'agit d'une file.
On utilise :

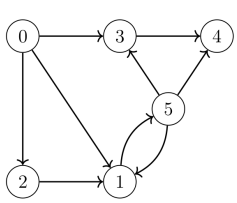
- un tableau `couleur` pour indiquer l'état de traitement d'un sommet dans le graphe : `BLANC` si le sommet n'a pas encore été découvert ou marqué, `GRIS` si le sommet a été découvert et placé dans la file d'attente, `NOIR` si le sommet a été traité (tous ses voisins) ont été visités et retiré de la file d'attente. Les sommets qui restent en `BLANC` ne sont pas accessibles au sommet source. La frontière d'exploration du graphe est représentée par les sommets `GRIS`.
- un tableau `distance` depuis le sommet source `s` : on ajoute 1 à la distance du sommet qui a permis de le découvrir et dont il est voisin
- un tableau `predecesseur` qui donne le prédécesseur de chaque sommet dans un plus court chemin depuis le sommet origine `s`, soit le sommet par lequel il a été visité

```
from queue import Queue
def parcours_largeur(g, s) :
    "g = liste d'adjacence, s = sommet"
    BLANC, GRIS, NOIR = 0, 1, 2
    n = len(g)
    couleur, distance, predecesseur = [BLANC]*n, [-1]*n, [-1]*n
    distance[s], couleur[s] = 0, GRIS
    file = Queue()
    file.put(s)
    while not file.empty() :
        u = file.get() # récupération du sommet à traiter
        for voisin in g[u] : # pour chaque voisin de u
            if couleur[voisin] == BLANC : # si voisin n'a pas encore été visité
                predecesseur[voisin] = u
                distance[voisin] = distance[u] + 1
                couleur[voisin] = GRIS
                file.put(voisin) # voisin est en attente de traitement dans la file
                couleur[u] = NOIR # sommet u défilé et traité
    return distance, predecesseur, couleur
```

Pour les questions de mémorisation, **on suppose qu'on parcourt les voisins d'un sommet par ordre croissant des indices** ! Le but n'est pas d'apprendre par coeur la prochaine ligne de mémorisation, mais d'avoir un exercice typique sur le parcours en largeur sur un exemple donné.

76

[Graphes : parcours en largeur]



Ordre ?
Taille ?
Ordre des sommets visités depuis 0 ?
Depuis 5 ?

Ordre = 6 sommets
Taille = 9 arcs
depuis le sommet 0 : `0,1,2,3,5,4`
depuis le sommet 5 : `5,1,3,4`
(depuis 5, les sommets ne sont pas tous visités)

77 [Graphes : parcours en largeur]
On dispose de la fonction `parcours_largeur(g,s)`.
Quel est **un** plus court chemin entre deux sommets `s1` et `s2` ?
	`def plus_court_chemin(g, s1, s2) :`
	`  predecesseur = \`
	`  parcours_largeur(g, s1)[1]`
	`  if predecesseur[s2] >= 0 : # init à -1`
	`  chemin = []`
	`  def remonte(s) : # fn auxiliaire`
	`  if s != s1 : # sinon remonté à s1`
	`  remonte(predecesseur[s])`
	`  chemin.append(s)`
	`  remonte(s2)`
	`  return chemin`

Chapitre 27 – Parcours en profondeur d'un graphe

Parcourir ou énumérer un graphe consiste à énumérer l'ensemble des sommets accessibles à partir d'un sommet donné afin de réaliser un certain traitement sur ces sommets. La différence entre les différents parcours réside dans l'ordre dans lesquels les sommets sont visités.

Le **parcours en profondeur** d'un graphe consiste à aller le plus loin possible en suivant les arêtes et à ne revenir en arrière que lorsqu'on ne peut plus avancer. Dès que l'on rencontre un voisin non exploré en remontant, on repart vers l'avant aussi loin que possible. Le parcours en profondeur est de **nature récursive** même si on peut en donner une version impérative.

```
1  def parcours_profondeur(g, s) :
2      """ parcours en profondeur du graphe g représenté par sa liste d'adjacence
3      :: param
4      g : liste adjacence
5      s : sommet de départ
6      :: sorties
7      predecesseur : tableau des prédécesseurs
8      couleur : tableau donnant la couleur d'un sommet
9      """
10     n = len(g)
11     BLANC, GRIS, NOIR = 0, 1, 2
12     predecesseur, couleur = [-1]*n, [BLANC]*n
13     def visiter(sommet) :
14         couleur[sommet] = GRIS
15         for voisin in g[sommet] :
16             if couleur[voisin] == BLANC :
17                 predecesseur[voisin] = sommet
18                 visiter(voisin)
19         couleur[sommet] = NOIR
20     visiter(s)
21     return predecesseur, couleur
```

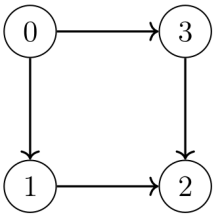
Si on souhaite parcourir **tous** les sommets du graphe, on remplace alors la ligne 20 par une itération sur tous les sommets de `g` BLANCS (et on supprimer le sommet de départ dans l'argument de la fonction) :

```
for u in range(n) :
     if couleur[u] == BLANC :
         visiter(u)
```

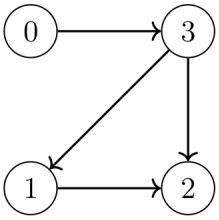
Le coloriage des sommets est fondamental pour éviter de tourner en rond dans un circuit : un sommet est `BLANC` initialement, `GRIS` en cours d'exploration et `NOIR` une fois que tous les voisins de ce sommet

ont été explorés. Ainsi, dans un graphe orienté, si on découvre à nouveau un sommet déjà découvert, il y a 2 possibilités :

- soit il est `GRIS` et il y a un circuit,
- soit il est `NOIR` et on a trouvé un autre chemin menant à ce sommet :



Depuis le sommet `0`, le parcours en profondeur est `0,1,2,3`. Le sommet `2` n'ayant pas de voisin, il est directement noté `NOIR`. Il est rencontré à nouveau lors du traitement du sommet `3` : il y a deux chemins arrivant au sommet `2`, mais pas de circuit.

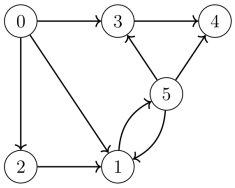


Depuis le sommet `0`, le parcours en profondeur est `0,3,1,2`. Le sommet `3` est rencontré comme voisin du sommet `0` : il est coloré en `GRIS`. Puis, lors de l'exploration du sommet `2`, il de nouveau rencontré alors qu'il est toujours `GRIS` : il y a bien un circuit.

On suppose que le graphe est implémenté par une liste d'adjacence. Et que les voisins d'un sommet sont explorés par numéro croissant.

78

`[Graphes : parcours en profondeur]`



Ordre ?
Taille ?
Ordre des sommets visités depuis 0 ?
Depuis 2 ?

Ordre = 6 sommets
Taille = 9 arcs
depuis le sommet 0 : `0,1,5,3,4,2`
depuis le sommet 2 : `2,1,5,3,4`

79

`[Graphes : parcours en profondeur]`

Le parcours récursif en profondeur d'un graphe (représenté par sa liste d'adjacence `g`) fait appel à une fonction auxiliaire `visiter(sommet)`. Quelle est-elle ?

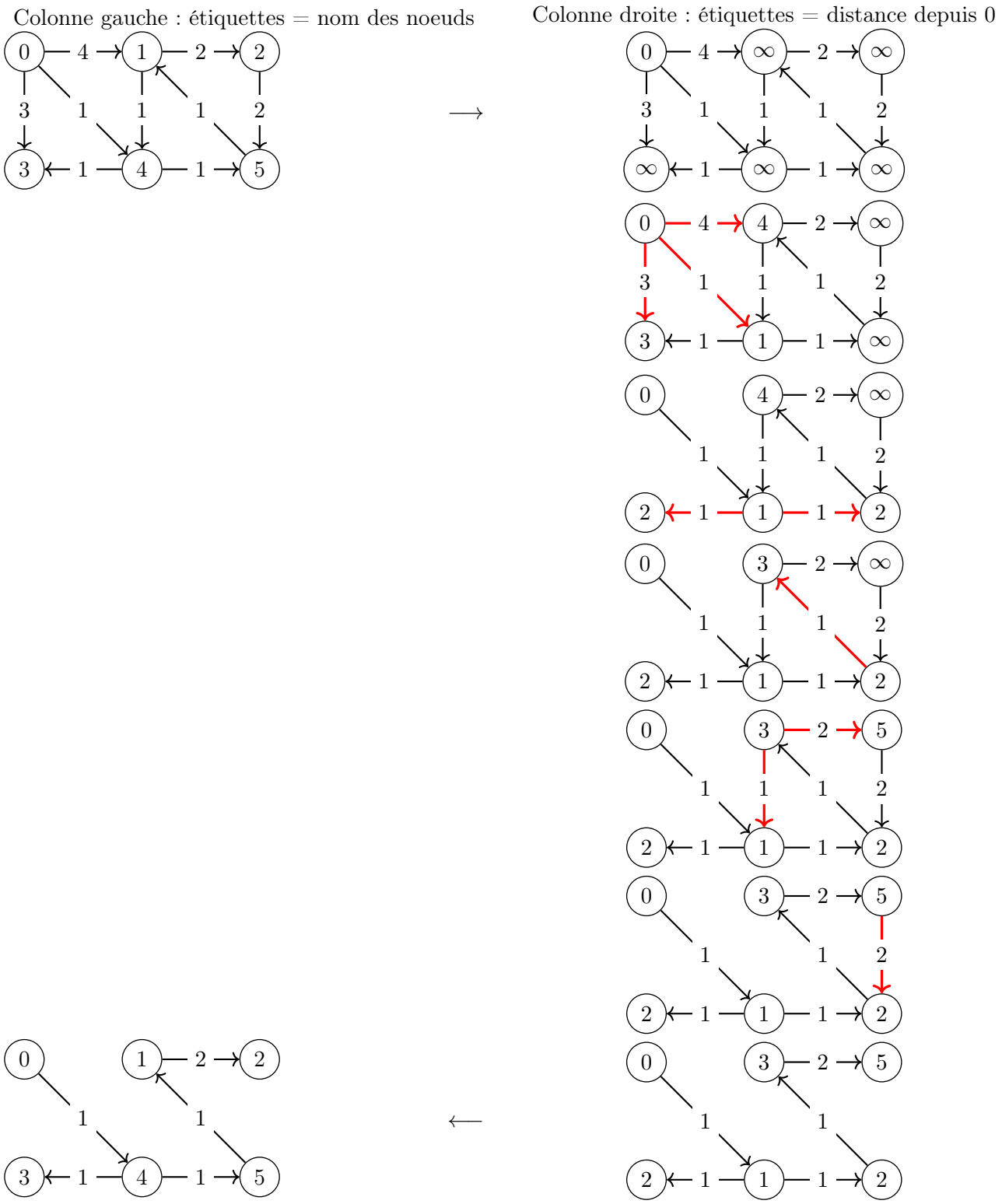
```
def visiter(sommet) :
  couleur[sommet] = GRIS
  for voisin in g[sommet] :
    if couleur[voisin] == BLANC :
      predecesseur[voisin] = sommet
      visiter(voisin)
  couleur[sommet] = NOIR
```

Chapitre 28 – Dijkstra : plus court chemin dans un graphe pondéré

L'algorithme de Dijkstra permet de déterminer **un** plus court chemin dans un graphe orienté pondéré (un poids est associé à chaque arête, comme la distance en km entre les sommets), depuis un sommet particulier vers **tous les autres sommets** (on ne connaît pas d'algorithme plus performant qui permette de s'arrêter à un sommet donné). Une contrainte forte de cet algorithme est qu'il n'est valable que pour des pondérations **positives ou nulles** ! Si ce n'est pas le cas, il faut considérer l'algorithme de Bellman-Ford.
Le graphe est donc décrit par une liste d'adjacence modifiée : on remplace chaque sommet par un tuple `(sommet, poids)`.
L'algorithme de Dijkstra est un algorithme **glouton** dont le résultat est **optimal** : il utilise un parcours en largeur (tous les voisins sont explorés à partir d'un sommet donné avant d'aller plus loin, mais la file du parcours en largeur est remplacée par une **file de priorité**, ce qui permet d'accéder au plus court chemin.
La file de priorité contient tous les sommets déjà visités (classés par ordre de distance au sommet source, la plus grande priorité correspondant à la distance la plus petite). Si un plus court chemin est découvert, le sommet est à nouveau enfilé avec une prorité plus grande (une distance plus petite). Ainsi, si un sommet est défilé, on est certain de ne pas pouvoir l'atteindre par un chemin plus court, il est définitivement traité et sa distance à la source est définitivement connue. On construit donc 2 tableaux de taille `n` :

- la longueur du plus court chemin de `i` (source) à tous les autres sommets `j`
- le prédécesseur de `j` dans le plus chemin de `i` à `j`

Cet algorithme explore donc les voisins par distance croissante.



Dans la colonne de droite, les arcs conduisant à des chemins plus longs ont été enlévés (alors qu'ils existent toujours évidemment). Les deux tableaux construits sont donc

- la longueur du plus court chemin de `i` (source) à tous les autres sommets `j`
  `distance = [0, 3, 5, 2, 1, 2]`
- le prédécesseur de `j` dans le plus chemin de `i` à `j`
  `predecesseur = [-1, 5, 1, 4, 0, 4]`

On peut donc trouver le plus court chemin entre le sommet 0 et le sommet 2 : on part du sommet 2 et on remonte jusqu'au sommet 0 par le tableau des prédécesseurs : `2→1→5→4→0`. Il suffit donc de prendre ce chemin à l'envers. Si le chemin a été stocké dans un tableau `chemin = [2,1,5,4,0]`, le chemin **depuis** le sommet 0 est donné par `chemin[::-1]`. Si on utilise une fonction récursive, il suffit

de stockée la valeur dans le tableau *après* l'appel récursif.
On remarque que lors de la visite du sommet 1, il faut visiter les sommets 2 et 4. Or le sommet 4 a déjà été visité, donc on ne pourra pas trouver de plus court chemin vers 4, il ne faut donc pas le considérer. Un moyen simple de le faire est d'utiliser un tableau `pasvu` initialisé à `True` pour chaque sommet.
Le tableau `distance` doit être initialisé avec des valeurs infinies : il suffit de mettre une valeur bien plus grande que la plus grande des distances possibles.
Le graphe donné en exemple est représenté par une liste d'adjacence (que vous devez savoir construire) composée des tuples `(sommet, poids)` :

```
g = [ [(1,4), (3,3), (4,1)], # sommet 0 (relié au sommet 1 avec un poids de 4, ...)
      [(2,2), (4,1)],        # sommet 1
      [(5,2)],               # sommet 2
      [],                    # sommet 3
      [(3,1), (5,1)],        # sommet 4
      [(1,1)] ]              # sommet 5

from queue import PriorityQueue
def Dijkstra(g, s) :
    n= len(g)
    filep = PriorityQueue()
    distance = [1000000]*n
    pasvu = [True]*n
    predecesseur = [-1]*n
    filep.put( (0,s) ) # couple (poids, sommet)
    distance[s] = 0
    while not filep.empty() :
        poids, sommet = filep.get()
        if pasvu[sommet] :
            pasvu[sommet] = False
            for (voisin, weigth) in g[sommet] :
                d = poids + weigth
                if d < distance[voisin] :
                    distance[voisin] = d
                    filep.put( (d, voisin) )
                    predecesseur[voisin] = sommet
    return distance, predecesseur
```

80

[Graphe : Dijkstra]
Quelle est la fonction récursive `plus_court_chemin(g,s1,s2)` lorsqu'on a la fonction `Dijkstra(g,s1)` qui renvoie les tableaux `distance` et `predecesseur` ?

```
def plus_court_chemin(g, s1, s2) :
  distance, predecesseur=Dijkstra(g,s1)
  # s2 est atteint
  if predecesseur[s2] >= 0 :
    chemin = []
    def remonte(s) :
      if s != s1 : # arrêt sur s1
        remonte(predecesseur[s])
      # après appel récursif, ordre ok
      chemin.append(s)
    remonte(s2)
    return chemin, distance[s2]
```

Chapitre 29 – Algorithme A*

L'algorithme A* (A étoile) est un algorithme de plus court chemin dans un graphe pondéré : il s'appuie sur la recherche *best-first* (le meilleur en premier) et le parcours en largeur dans la version de Dijkstra. Contrairement à l'algorithme de Dijkstra qui ne vise aucun sommet particulier, mais cherche le plus court chemin d'un sommet vers **tous** les autres sommets, l'algorithme A* se limite à un seul sommet cible.

La stratégie **gloutonne** best-first est une approche qui utilise une estimation de la distance entre un sommet et le sommet cible, de manière à explorer en priorité le voisin le plus proche de la cible (choix glouton local). Mais cette distance ne peut pas être connue exactement, puisqu'on ne connaît pas encore le plus court chemin (on est en train de le construire). Pour cela, on va utiliser une **heuristique** qui permet de faire un calcul approximatif afin d'orienter la recherche et la rendre pertinente.
La stratégie best-first est vouée à l'échec s'il y a un cul de sac ou un obstacle, mais elle oriente dans la bonne direction la recherche, ce que ne fait pas l'algorithme de Dijkstra. Elle est typiquement implémentée par une **file de priorité**.

L'algorithme A* est un algorithme de recherche du plus court chemin qui combine la stratégie de distance croissante depuis le sommet de départ (Dijkstra) et la stratégie de distance minimale à la cible (recherche best-first via une heuristique) et combine les deux informations our gérer une file de priorité pour l'ordre de visite des sommets. Ainsi, on n'explore pas la totalité du graphe si les sommets de départ et cible sont proches et si l'on stoppe la recherche dès que la cible est trouvée.

A* utilise une fonction f qui calcule le coût total estimé entre le sommet de départ et le sommet cible, passant par le sommet n :

$$f(n) = d(n) + h(n)$$

$d(n)$ (d comme Dijkstra) est le coût exact du plus court chemin entre le sommet de départ et le sommet n en cours de traitement, tandis que $h(n)$ (h comme heuristique) est le coût estimé entre le sommet n et le sommet cible. h doit avoir la même dimension que g. Les valeurs de $f(n)$ sont placées dans une file de priorité : c'est l'élément n dont la valeur $f(n)$ est la plus faible qui est sorti en premier. Le stockage de cette file peut être couteux en mémoire, c'est une des principales limitations de cet algorithme, qui est par ailleurs simple, il ne nécessite que peu de calcul : c'est un excellent compromis, efficace et très utilisé en navigation ou dans les jeux nécessitant de trouver un chemin contournant des obstacles.
Exemple : si d correspond à un temps de parcours, alors la fonction h (qui doit estimé un temps pour la somme $d(n) + h(n)$ ait du sens) peut estimer le temps nécessaire à vol d'oiseau pour aller du sommet n au sommet cible.

```
1  from queue import PriorityQueue
2  def Astar(g,s1,s2):
3      "g = liste d'adjacence, s1 = sommet source, s2 = sommet cible"
4      x2, y2 = Coords[s2] # un moyen d'avoir les coordonnées des sommets
5      def h(s):
6          ...
7          return ...
8      n = len(g)
9      distance, pasvu, predecesseur = [1000000] * n, [True] * n, [-1] * n
10     filep = PriorityQueue()
11     filep.put( (h(s1), s1) )
12     distance[s1] = 0
13     while not filep.empty():
14         _, s = filep.get()
15         if s == s2: break
16         if pasvu[s]:
17             pasvu[s] = False
18             for voisin, poids in g[s]:
```

```
19                  d = distance[s] + poids
20                  if d < distance[voisin]:
21                      distance[voisin] = d
22                      filep.put( (d + h(voisin), voisin) )
23                      predecesseur[voisin] = s
24      return distance, predecesseur, pasvu.count(False)
```

Le graphe `g` est représenté par une liste d'adjacence de la forme `[ [(sommet1,poids1), (sommet8,poids8)], ...]` qui signifie que le sommet `0` est relié aux sommets `1` et `8` avec différents poids.
Ligne 24, l'instruction `pasvu.count(False)` compte le nombre de fois que la valeur `False` est trouvée dans le tableau `pasvu`, soit le nombre de sommets visités entre le sommet de départ et le sommet cible.
Pour avoir le nombre de sommets visités lors du plus court chemin, il faut partir du sommet cible `s2` et remonter le tableau `predecesseur` jusqu'à `s1`.

Pour les questions de mémorisation, on se place dans un contexte de graphe de métro où les sommets sont les stations de métro.

81	`[Graphe : Algo A*]` Quelles informations a-t-on besoin de connaître sur les sommets du graphe ?	1) les temps de parcours/distances entre sommets 2) les coordonnées des sommets pour l'heuristique
82	`[Graphe : Algo A*]` Pourquoi avoir besoin des coordonnées du sommet cible (ligne 4) ?	Pour pouvoir utiliser l'heuristique (fonction `h`) entre le sommet en cours de traitement `voisin` et le sommet cible `s2`.
83	`[Graphe : Algo A*]` A quoi sert l'initialisation `distance = [1000000] * n` ? Limite ?	Normalement, le tableau `distance` est initialisé avec des valeurs infinies. Limite : on prend une valeur d'initialisation qui dépasse toutes les valeurs possibles.
84	Quest-ce qu'une file de priorité ?	Une structure de donnée qui défile en premier l'élément associé à la plus grande priorité.